Lecture Notes in Computer Science

802

S. Brookes M. Main A. Melton
M. Mislove D. Schmidt (Eds.)

Mathematical Foundations of Programming Semantics

9th International Conference
New Orleans, LA, USA, April 7-10, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Stephen Brookes
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, USA

Michael Main
Department of Computer Science, University of Colorado
Boulder, CO 80309, USA

Austin Melton
Department of Computer Science, Michigan Technological University
Houghton, MI 49931, USA

Michael Mislove
Department of Mathematics, Tulane University
New Orleans, LA 70118, USA

David Schmidt
Department of Computing and Information Sciences, Kansas State University
Manhattan, KS 66506, USA

CR Subject Classification (1991): D.3, F.1, F.3-4

ISBN 3-540-58027-1 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-58027-1 Springer-Verlag New York Berlin Heidelberg

CIP data applied for

Typesetting: Camera-ready by author
SPIN: 10131099 45/3140-543210 - Printed on acid-free paper

Preface

The Ninth International Conference on the Mathematical Foundations of Programming Semantics was held on the campus of Tulane University, New Orleans, Louisiana from April 7 to 10, 1993. The major goal of this conference series is to bring together computer scientists who work in programming semantics and mathematicians who work in areas which might impact programming semantics so that they may share ideas and discuss problems of mutual interest. By letting mathematicians see applications of their work to programming semantics and by letting computer scientists see their ideas and intuitions expressed in pure mathematics, the organizers have sought to improve communication among the researchers in these areas and to establish ties between related areas of research. With these goals in mind, the invited speakers for the conference were Peter Aczel (University of Manchester), Pierre-Louis Curien (LIENS, Paris), Albert Meyer (MIT), Dale Miller (University of Pennsylvania), Andrew Pitts (University of Cambridge), and Gordon Plotkin (University of Edinburgh).

In addition, there were contributed talks by twenty-eight researchers. Some of the contributed talks were presented in two special sessions. The first of these special sessions was devoted to Real-Time Concurrency and was organized by G.M. Reed and A.W. Roscoe (Oxford). The second was on Full Abstraction and was organized by Stephen Brookes (CMU). There was also an invited address by S. Tucker Taft (Intermetrics) on the Ada-9X project.

The Conference Chairpersons were Stephen Brookes and Michael Mislove. The Program Committee Chairpersons were Michael Main and Austin Melton. In addition to the Conference and Program Committee Chairpersons, the Program Committee consisted of Samson Abramsky, Bard Bloom, Matthew Hennessy, Gary Leavens, John Mitchell, Philip Mulry, Frank Oles, Ana Pasztor, Amir Pnueli, G.M. Reed, Edmund Robinson, A.W. Roscoe, Robert Tennent, Glynn Winskel, Steven Vickers, and Guo-Qiang Zhang. The editors wish to express their thanks to the other members of the Committee for their efforts in reviewing the papers submitted for presentation at the conference.

The conference was supported by funds from the Office of Naval Research; we wish to thank ONR for its continuing and generous support of the conference series.

Thanks are due to the many people who helped make the conference run so smoothly. These include John Maraist, Magnus Rothe and Han Zhang. We all owe a special thank you to Geralyn Caradona, Administrative Associate of the Mathematics Department of Tulane University, who managed to oversee virtually all of the small details of running the conference and allow the rest of us to concentrate on the meeting itself. Also we owe thanks to Kelly McLean of the Computer Science Department at Michigan Technological University for her efforts in collecting and organizing the papers for this proceedings.

March 1994

Stephen Brookes
Michael Main
Austin Melton
Michael Mislove
David Schmidt

Table of Contents

Final Universes of Processes[*]

Peter Aczel
Departments of Mathematics and Computer Science
Manchester University
Manchester M13 9PL, U.K.
email: petera@cs.man.ac.uk

February 16, 1994

Abstract

We describe the final universe approach to the characterisation of semantic universes and illustrate it by giving characterisations of the universes of CCS and CSP processes.
Keywords: Final Universe, Process, Coalgebra, Labelled Transition System, CCS, CSP.

1 Introduction

1.1 Process Algebra

In the last decade and a half there has been an explosion of work aimed at the development of a mathematical theory of concurrent processes. One major strand of this work may perhaps be put under the general title of 'process agebra'. The process algebra approach originated from the seminal ideas of Milner and Hoare. They have presented developments of their ideas in the books [5] and [6]. Other variants of process algebra have appeared in book form, ([4],[3]) and there have been hundreds of research papers on the topic.

[*]This research was partially supported by an SERC Senior Research Fellowship.

The aim of this paper is to describe a *fairly simple* approach to the characterisation of certain kinds of mathematical structure that seem to be fundamental to process algebra. I will illustrate the approach by applying it to the two versions of process algebra that appear in [5] and [6]; i.e. CCS and CSP. There are many variants of the ideas of CCS and CSP so it should be emphasised that it is the presentations in those books that will be used, although I shall find it convenient to use my own notation and definitions to some extent. These two versions of process algebra have played a central role in the subject and it seemed natural to apply the approach to these in the first instance. But I expect that the approach will be just as applicable to variations of them and to other versions of process algebra.

For a given abstract informal notion of process, the idea is to specify a universe of abstract processes, make that universe into a mathematical structure and characterise that structure (up to isomorphism). Of course we would like to find simple mathematical structures - well, as simple as allowed by the informal notions. And we would like to find simple characterisations of those structures.

1.2 The Final Universe approach to Semantics

There is a standard picture associated with formal semantics:-

$$\begin{tikzcd}\boxed{\textbf{SYNTAX}} \arrow[r, "\textbf{meaning}"] & \boxed{\textbf{SEMANTICS}}\end{tikzcd}$$

In the picture a syntactic universe is linked to a semantic universe by an arrow representing meaning. The picture may be viewed set theoretically, as two sets linked by a denotation function, or more abstractly as simply an arrow in a category, the two universes being represented as objects in the category. One natural development of this picture is the familiar initial algebra, compositional approach to syntax and semantics. In this approach the forms of expression of the syntax determine a category of algebras, with homomorphisms between them, in which the syntactic universe forms an **initial object**; i.e. an object I with the characterising property that for every object A in the category there is a unique map $I \rightarrow A$ in the category. Now in order to give a formal semantics to the syntax it suffices to represent the semantics as a semantic universe that is an algebra in the category. The meaning function between syntax and semantics is then the uniquely determined homomorphism.

This initial algebra approach is *syntax led*; i.e. the category chosen is dictated by the syntax. There is an alternative dual approach to syntax and semantics which is *semantics led*. Here it is the kind of semantics that is being considered that determines a category in which the semantic universe is now represented as a **final object**; i.e. an object F, with the characterising property that for every object A of the category there is a unique map $A \rightarrow F$. In order to use this semantic universe for a particular syntax it is necessary to represent the syntax as an object in the category and once this has been done the meaning map between syntax and semantics is determined as the unique map between them.

These two approaches use the dual category-theoretic notions of initial object and final object in a category and, of course, characterise objects up to isomorphism. It is worth noting that the characterisations are up to a unique isomorphism so that they give the mathematically most stringent kind of characterisation.

In discussing the final universe approach it will be useful to consider the two component notions that make up the notion of a final object. An object F in a category is **weakly final/strongly extensional** if for every object A there is at least/most one map $A \rightarrow F$. Clearly an object is final if and only if it is both weakly final and strongly extensional.

The paper [8] is a useful complement to the present paper. It is concerned with essentially the same idea of final semantics as presented here. A different, but possibly related topic is 'final algebra semantics' in the theory of abstract data types. See, for example [7] and the references cited there. Further investigation is needed to see what the relationships are, if any.

1.3 Outline

The rest of this paper is organised as follows. In the next section we review the general mathematical apparatus for final coalgebras that has been developed in [1] and [2]. There is one new result, Theorem 2.2 that plays a useful unifying role here. This theory is very general, and in section 3 we specialise to the key application for process algebra, labelled transition systems. This specialised theory is then applied in sections 4 and 5 to CCS and CSP and the paper ends with some final remarks in section 6. The central notion of section 4 is that of a τ-LTS, defined in definition 4.6. A final τ-LTS is used. in section 4.4, as the semantic universe for CCS that corresponds to the weak bisimulation congruence operational semantics of CCS. In section 4.5 it is shown how the CCS combinators can be defined on any final τ-LTS. This work shows that CCS can be given a non-syntactic axiomatic treatment. In section 4.6 we show that there is a denotational semantics for CCS, assigning a denotation in a final τ-LTS to each CCS agent. This semantics corresponds exactly to the operational weak bisimulation congruence semantics.

Several of the results in this paper are stated without proof. It is hoped that it should be a fairly routine matter for the reader to find the missing proofs for themselves.

2 Coalgebras and Classes

2.1 Coalgebras

We shall work with the following general notion that is dual to the more familiar notion of an algebra relative to an endofunctor. Given an endofunctor $F : \mathbf{C} \to \mathbf{C}$ on a category $\mathbf{C}$, a **coalgebra (for F)** is a pair (A, θ) such that $\theta : A \to FA$ is a map in the category $\mathbf{C}$. The coalgebras themselves form a category, where a map $(A, \theta) \to (A', \theta')$ is a map $f : A \to A'$ of $\mathbf{C}$ such that $(Ff)\theta = \theta' f$; i.e. the obvious square commutes.

2.2 Classes

Our examples of final universes of processes will be final objects in certain full subcategories of coalgebras for functors on the category of classes. So we shall want to work with classes. This could be avoided by using a universe of sets or else by using an inaccessible cardinal or making cardinality restrictions. But for us the most natural approach is to use classes. For the most part we use a standard axiomatic set theory approach to classes. But we will need to make use of the following principle:-

Quotient Existence Principle for Classes: If A is a class and $R \subseteq A \times A$ is an equivalence relation on A then there is a function $[-]_R$ defined on A such that for all $x, y \in A$

$$[x]_R = [y]_R \iff xRy.$$

We call $[-]_R$ a **quotient of A with respect to R** and call $[A]_R = \{[x]_R \mid x \in A\}$ a **quotient class of A with respect to R.** The subscript R will usually be omitted when there is no confusion.

This principle is an easy consequence of a global form of the axiom of choice, and this may be the simplest perspective for the reader to take. In fact the principle has really little to do with the axiom of choice and an alternative approach is to depart from the traditional approach to classes taken in axiomatic set theory and re-define the notion of class, so that according to the new notion a class is a pair of old classes, the second being an equivalence relation on the first. Then the Quotient Existence Principle becomes a triviality as the quotient class is simply obtained by changing the equivalence relation. Although not traditional, this approach would seem to be rather natural from the category-theoretic perspective.

2.3 Standard Functors

Let *Class* be the category of classes. This is a superlarge category. But we shall not worry here about making our use of this rigorous. From previous experience (e.g. see [1]) there is no serious problem with a careful handling of it. If A is a subclass of B then the identity map from A to B will be called an **inclusion map** $A \hookrightarrow B$. We say that a functor $F : Class \rightarrow Class$ **preserves inclusion maps** if, whenever $A \subseteq B$ then $FA \subseteq FB$ and $Fi : FA \hookrightarrow FB$, where $i : A \hookrightarrow B$. We also say that F is **set continuous** if for all classes A

$$FA = \bigcup\{Fa \mid a \in powA\},$$

where $powA$ is the class of all subsets of the class A. If F both preserves inclusion maps and is set continuous then we say that F is **standard**. Note that pow can be made into a standard functor by defining pow on a map $f : A \rightarrow B$ to be $powf : powA \rightarrow powB$, where

$$powf(x) = \{fy \mid y \in x\}$$

for all $x \in powA$. In fact most naturally-defined functors on the category of classes turn out to be standard.

A key result about the category of coalgebras for a standard functor is:

Theorem 2.1 (Final Coalgebra Theorem) *Every standard functor on the category of classes has a final coalgebra.*

A weaker result was first proved in [1], where it was assumed that the functor preserved weak pullbacks. A slightly stronger result was proved in [2], where it was only assumed that the functor was set-based, a better assumption from the category-theoretic purists point of view, but not much weaker than the assumption that the functor is standard.

An interesting new generalisation of the above theorem will be useful in this paper. Let F be a standard functor and let $\mathbf{C}$ be a full subcategory of the category of coalgebras for F. We say that $\mathbf{C}$ is **image-closed** if whenever $A \rightarrow B$ is a surjective coalgebra map, with A in $\mathbf{C}$, then B is also in $\mathbf{C}$. We also say that $\mathbf{C}$ is **union-closed** if whenever A is a coalgebra, such that every element of A is in some subcoalgebra of A that is in $\mathbf{C}$, then A is in $\mathbf{C}$.

Theorem 2.2 *Let F be a standard functor and let $\mathbf{C}$ be an image-closed and union-closed, full subcategory of the category of coalgebras for F. Then $\mathbf{C}$ has a final object.*

Proof Sketch: By the final coalgebra theorem F has a final coalgebra A. Let A' be the union of the subcoalgebras of A that are in

C. As **C** is union closed A' is in **C**. As A' is a subcoalgebra of the strongly extensional A, A' is strongly extensional. To show that A' is also a weakly final object of **C** let B be in **C**. Then there is a unique coalgebra map $B \to A$. This has a factorisation $B \to B' \hookrightarrow A$, where the map $B \to B'$ is surjective. As **C** is image-closed B' is in **C**. As B' is a subcoalgebra of A, it must be a subcoalgebra of A' so that we have a map $B \to B' \hookrightarrow A'$.

Note: If $\mathbf{C}_i$ is a full subcategory of the category of coalgebras for F, for $i \in I$, then we may form their intersection $\bigcap_{i \in I} \mathbf{C}_i$ as a full subcategory. If each $\mathbf{C}_i$ is image-closed (union-closed) then so is their intersection. This observation can be useful in applying the above theorem.

The following further weakening of the notion of a weakly final coalgebra will be useful. It was used in [2] when proving the Final Coalgebra Theorem. Given a standard functor on the category of classes, a coalgebra (A, θ) is **small** if A is a set. A coalgebra (A, θ) is **weakly complete** if, for every small coalgebra (A', θ') there is at least one coalgebra map $(A', \theta') \to (A, \theta)$. We have the following results from [2]:-

Proposition 2.3 *Every weakly complete strongly extensional coalgebra is final.*

Proposition 2.4 *For every coalgebra there is a surjective map from it onto a strongly extensional coalgebra.*

These two results together give a construction of a final object as a strongly extensional quotient of any weakly complete coalgebra.

3 Labelled Transition Systems

We assume given a fixed set *Act* of **atomic actions**. We define a **labelled transition system (LTS)** (relative to *Act*) to be a coalgebra for the standard functor *pow*$(Act \times -)$. We also define the notion of *LTS*-map in the obvious way by specialising the terminology for coalgebras to this particular functor. Let $\mathcal{A} = (A, \theta)$ be an *LTS*. The map $\theta : A \to pow(Act \times A)$ is called the **transition map** of $\mathcal{A}$. If $a, b \in A$ we write $a \xrightarrow{\alpha} b$ in $\mathcal{A}$, or just $a \xrightarrow{\alpha} b$ when there is no ambiguity, when $(\alpha, b) \in \theta(a)$. So an *LTS* associates with each $\alpha \in Act$ the transition relation $\xrightarrow{\alpha}$. Note that the transition map can be recovered from these transition relations by defining

$$\theta(a) = \{(\alpha, b) \mid a \xrightarrow{\alpha} b\}$$

for all $a \in A$.

We have the following results about *LTS*s that carry over from the results about coalgebras in general.

Theorem 3.1 *There is a final LTS.*

Theorem 3.2 *Any image-closed and union-closed, full subcategory of the category of LTSs has a final object.*

In chapter 8 of [1] I showed that any final *LTS* gives a universe for the *SCCS* processes, up to the strong bisimulation equivalence that is the natural one to consider for *SCCS*. There I also showed how the *SCCS* combinators could be defined as operations on any final *LTS*. It is equally clear that the same final *LTS* is also a universe for the *CCS* processes, again up to strong bisimulation equivalence.

3.1 Deterministic Processes

We can now briefly consider the simplest and most familiar notion of process. The deterministic processes form subuniverses of both the universes of *CCS* processes and of *CSP* processes. The universe of deterministic processes has a natural construction as an *LTS* of trace sets. We see below that the universe also has a natural characterisation as a final deterministic *LTS*.

Definition 3.3 *An LTS* $\mathcal{A} = (A, \theta)$ *is* **deterministic** *if, for all* $a \in A$ *the set of pairs* $\theta(a) \subseteq Act \times A$ *is (the graph of) a function; i.e. if* $a \xrightarrow{\alpha} a_1$ *and* $a \xrightarrow{\alpha} a_2$ *then* $a_1 = a_2$.

If $d\mathbf{C}$ is the full subcategory of the category of *LTS*s consisting of deterministic *LTS*s then it is not hard to see that $d\mathbf{C}$ is image-closed and union-closed and hence has a final object. This category $d\mathbf{C}$ may also be defined as the category of all the coalgebras for the standard functor $Map(Act, -)$ that associates with each class

X the class $Map(Act, X)$ of all partial functions $f : Act \rightharpoonup X$; i.e. functions, each defined on some subset of Act, with values in X.

As usual we let Act^* be the set of strings of elements of Act. A set $X \subseteq Act^*$ is a **trace-set** if it is non-empty and is prefix closed; i.e. if $\sigma\alpha \in X$, where $\sigma \in Act^*$ and $\alpha \in Act$ then $\sigma \in X$. Let TR be the set of trace-sets. We can make this into a deterministic LTS (TR, θ_{TR}), called the **trace-set**-LTS, by defining

$$\theta_{TR}(X) = \{\ (\alpha, \{\sigma \in Act^* \mid \alpha\sigma \in X\})\ \mid \alpha \in X \cap Act\},$$

for each $X \in TR$. Now given any LTS $\mathcal{A} = (A, \theta)$ we can define a function $tr : A \to TR$ by:

$$tr(a) = \{\sigma \in Act^* \mid a \xrightarrow{\sigma} a' \text{ for some } a'\},$$

for all $a \in A$, where, if $\sigma = \alpha_1 \cdots \alpha_n \in Act^*$ then

$$a \xrightarrow{\sigma} a' \iff a \xrightarrow{\alpha_1} a_1 \cdots a_{n-1} \xrightarrow{\alpha_n} a' \text{ for some } a_1, \ldots, a_{n-1}.$$

Theorem 3.4 *If $\mathcal{A}$ is a deterministic LTS then $tr : \mathcal{A} \to (TR, \theta_{TR})$ is the unique LTS map into the trace-set LTS. Hence the trace-set LTS is a final object of* $d\mathbf{C}$.

3.2 Bisimulation on an *LTS*

The notion of a bisimulation relation on an LTS is fundamental. Here we give a definition that exploits the brevity of relation algebra. If $\mathcal{R}_1, \mathcal{R}_2$ are relations then their relational composition $\mathcal{R}_1\mathcal{R}_2$ is defined to be the relation $\mathcal{R}$ where

$$a\mathcal{R}b \iff a\mathcal{R}_1 c \mathcal{R}_2 b \text{ for some } c.$$

Also the inverse $\mathcal{R}^{-1}$ of a relation $\mathcal{R}$ is given by

$$a\mathcal{R}^{-1}b \iff b\mathcal{R}a.$$

Now if $\mathcal{R}$ is a relation on A, where $\mathcal{A} = (A, \theta)$ is an LTS, then , for each $\alpha \in Act$, we can form the relational compositions $\mathcal{R} \xrightarrow{\alpha}$ and $\xrightarrow{\alpha} \mathcal{R}$ and, for $a \in A$, let

$${}^{\mathcal{R}}\theta(a) = \{(\alpha, x) \mid a\mathcal{R} \xrightarrow{\alpha} x\}$$

and

$$\theta^{\mathcal{R}}(a) = \{(\alpha, x) \mid a \xrightarrow{\alpha} \mathcal{R}x\}.$$

Definition 3.5 *$\mathcal{R}$ is a* **simulation on** *$\mathcal{A}$ if $^{\mathcal{R}}\theta(a) \subseteq \theta^{\mathcal{R}}(a)$ for all $a \in A$, and a* **bisimulation on** *$\mathcal{A}$ if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are simulations on $\mathcal{A}$.*

Proposition 3.6 *Let $\sim$ be an equivalence relation on A. Then $\sim$ is a bisimulation on $\mathcal{A}$ iff*

$$a_1 \sim a_2 \implies \theta^{\sim}(a_1) = \theta^{\sim}(a_2).$$

If $\sim$ is a bisimulation on $\mathcal{A}$ and $[-] : A \to [A]$ is a quotient of A with respect to $\sim$ then there is a unique map

$$[\theta] : [A] \to pow(Act \times [A])$$

such that $[-]$ is an LTS map $[-] : \mathcal{A} \to [\mathcal{A}]$, where $[\mathcal{A}]$ is the LTS $([A], [\theta])$.

Theorem 3.7 *For any LTS $\mathcal{A}$ there is a maximal bismulation, $\sim_{\mathcal{A}}$, on $\mathcal{A}$. Moreover $\sim_{\mathcal{A}}$ is an equivalence relation on A and is the maximal relation $\sim$ on A such that for all $a_1, a_2 \in A$*

$$a_1 \sim a_2 \iff \theta^{\sim}(a_1) = \theta^{\sim}(a_2).$$

A quotient $[-] : \mathcal{A} \to [\mathcal{A}]$ of an *LTS* $\mathcal{A}$ with respect to its maximal bisimulation will be called a **collapse** of $\mathcal{A}$.

Lemma 3.8 *An LTS is strongly extensional iff its maximal bisimulation is the equality relation on the LTS.*

Theorem 3.9 *Any collapse of an LTS is strongly extensional, so that a collapse of any weakly complete LTS is a final LTS.*

3.3 Coloured *LTS*s

There is a simple variation on the notion of an *LTS* that allows each process to have a **colour** so as to get a more intensional notion of process. This will be a useful tool in dealing with *CSP*. Suppose that we are given a set *Col* of **colours**.

Definition 3.10 *A* **coloured labelled transition system** (*CLTS*), *(relative to Act and Col) is a coalgebra for the functor $pow(Act \times -) \times Col$ on the category of classes.*

If $\mathcal{A} = (A, \phi)$ is a $CLTS$ then we can define maps $\theta : A \to pow(Act \times A)$ and $col : A \to Col$ so that for $a \in A$

$$\phi(a) = (\theta(a), col(a)).$$

So we get the underlying LTS (A, θ) of the $CLTS$ and a map that associates with each element a of A its **colour** $col(a)$. We can carry over notation and terminology from LTSs to $CLTS$s. In particular we call a $CLTS$ **deterministic** if its underlying LTS is deterministic.

4 CCS Processes

4.1 Review of CCS

Here we want to summarise the syntax and operational semantics of CCS, as it is presented in the book [6].

4.1.1 Syntax of CCS

We assume given a set Λ of **names**, with an associated disjoint set $\bar{\Lambda}$ **of conames**, one coname, $\bar{a}$ for each name a. Each name a forms a **complementary pair** with its coname $\bar{a}$. Names and conames in general will be called **labels** and, for any label l we will write $\bar{l}$ for its complement. So if $l = \bar{a}$ then $\bar{l} = a$ and in general, for any label l, $\bar{\bar{l}} = l$. Any set L of labels will also be called a **sort**, and for any sort L we let $\bar{L} = \{\bar{l} \mid l \in L\}$ and $L^{\#} = L \cup \bar{L}$.

We will need a special **silent action** τ. The labels, with the silent action form the set Act of **atomic actions**. So $Act = \Lambda^{\#} \cup \{\tau\}$. We call $f : Act \to Act$ a **relabelling map** if $f(\tau) = \tau$ and, for each label l, $f(l)$ is a label and $\overline{f(l)} = f(\bar{l})$.

We first define the class E_K of **agents (agent expressions)** of CCS, relative to a class K of **agent constants**. Given the class K, with typical element c, we specify the class E_K, with typical element e, in the following BNF style:-

$$e \quad ::= \quad c \mid \alpha.e \mid \sum_{i \in I} e_i \mid e_1|e_2 \mid e\backslash L \mid e[f]$$

Here I can be any set and e_i is an agent for each $i \in I$. Also L is a sort and f is a relabelling map. In order to give the CCS operational semantics to this language of agents it is necessary to have an assignment of a **defining equation** $c \overset{\text{def}}{=} e_c$ to each agent constant c. This can be specified by a function $\kappa : K \to E_K$, which associates with each constant c the agent $e_c = \kappa(c)$ that appears on the right hand side of the defining equation. We will call the pair $\mathcal{K} = (K, \kappa)$ a **system of constants** for CCS.

As envisioned in the book [6], constants with their defining equations can be introduced as needed. We can capture this idea of an open language with an expanding system of constants by using a fixed system of constants that is universal in the following sense.

Definition 4.1 *A system of constants $\mathcal{K} = (K, \kappa)$ for CCS is* **universal** *if, for every small LTS (I, ψ) there is $\pi : I \to K$ such that for all $i \in I$*

$$\kappa(\pi i) = \sum_{i \xrightarrow{\alpha} j} \alpha . \pi j.$$

The following fact is easy to prove.

Proposition 4.2 *There is a universal system of constants for CCS.*

4.1.2 Operational Semantics of CCS

The operational semantics of CCS is given by the following clauses of an inductive definition that is used to generate the labelled transition relation that has a transition relation $\xrightarrow{\alpha}$ for each $\alpha \in Act$.

$$\frac{e_c \xrightarrow{\alpha} e}{c \xrightarrow{\alpha} e} \qquad \alpha.e \xrightarrow{\alpha} e \qquad \frac{e_j \xrightarrow{\alpha} e}{\sum_{i\in I} e_i \xrightarrow{\alpha} e}\ (j \in I)$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1|e_2 \xrightarrow{\alpha} e_1'|e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{e_1|e_2 \xrightarrow{\alpha} e_1|e_2'}$$

$$\frac{e_1 \xrightarrow{l} e_1' \qquad e_2 \xrightarrow{\bar{l}} e_2'}{e_1|e_2 \xrightarrow{\tau} e_1'|e_2'}\ (l \in \Lambda^{\#})$$

$$\frac{e \xrightarrow{\alpha} e'}{e\backslash L \xrightarrow{\alpha} e'\backslash L}\ (\alpha \notin L^{\#}) \qquad \frac{e \xrightarrow{\alpha} e'}{e[f] \xrightarrow{f(\alpha)} e'[f]}$$

The above operational semantics can be reformulated as a recursive definition of the function $\theta : E_K \rightarrow pow(Act \times E_K)$, where

$$\theta(e) = \{(\alpha, e') \mid e \xrightarrow{\alpha} e'\}$$

for all $e \in E_K$. It is the 'least' function satisfying the following equations:-

$$\begin{array}{lcl}
\theta(c) & = & \theta(e_c), \\
\theta(\alpha.e) & = & \{(\alpha, e)\}, \\
\theta(\sum_{i\in I} e_i) & = & \bigcup_{i\in I} \theta(e_i), \\
\theta(e_1|e_2) & = & Q(e_1, e_2, \theta(e_1), \theta(e_2)) \\
\theta(e\backslash L) & = & \{(\alpha, e'\backslash L) \mid (\alpha, e') \in \theta(e)\ \&\ \alpha \notin L^{\#}\} \\
\theta(e[f]) & = & \{(f(\alpha), e'[f]) \mid (\alpha, e') \in \theta(e)\}
\end{array}$$

In the equation for $\theta(e_1|e_2)$ we have used an operation Q, where for $e_1, e_2 \in E_K$ and sets $X_1, X_2 \subseteq Act \times E_K$ the set $Q(e_1, e_2, X_1, X_2) =$

$$\{(\alpha, e_1'|e_2) \mid (\alpha, e_1') \in X_1\} \cup \{(\alpha, e_1|e_2') \mid (\alpha, e_2') \in X_2\}$$

$$\cup \{(\tau, e_1'|e_2') \mid (l, e_1') \in X_1\ \&\ (\bar{l}, e_2') \in X_2 \text{ for some label } l\}$$

The sense of 'least' intended here is such that, for any other function θ' satisfying the equations, $\theta(e) \subseteq \theta'(e)$ for all $e \in E_K$. Note that these equations can also be viewed as a compositional definition by structural recursion on the 'syntax' of E_K, combined with a least fixed point definition of a function on the set K of constants. More specifically, if $\psi : K \rightarrow pow(Act \times E_K)$ then we can define, by structural recursion a function θ_ψ using the equations above for θ, except that the first equation should be replaced by

$$\theta(c) = \psi(c).$$

Now let θ be θ_ψ, where ψ is the 'least' function such that $\psi(c) = \theta_\psi(c)$ for all constants $c \in K$.

Finally we can give the CCS construction of a final LTS.

Proposition 4.3 *If $\mathcal{K} = (K, \kappa)$ is a universal system of constants for CCS then $\mathcal{E}_\mathcal{K} = (E_K, \theta)$ is a weakly complete LTS, so that any collapse of it is a final LTS.*

4.2 Weak Bisimulation

We have seen how the syntax and operational semantics of CCS gives rise to an LTS $\mathcal{E}_\mathcal{K} = (E_K, \theta)$, relative to a system of constants $\mathcal{K}$. The maximal bisimulation relation on this LTS has been called **strong bisimulation equivalence**. It is a congruence with respect to the combinators of CCS, so that these combinators induce operations on any collapse of this LTS.

But this LTS does not incorporate any special treatment of the distinguished action τ to reflect the intended intuition that τ should not be externally observable. To capture this [6] introduces relations $\overset{\hat{\alpha}}{\Rightarrow}$ on E_K and uses them, instead of the relations $\xrightarrow{\alpha}$, to get a maximal bisimulation relation $\approx$, called **weak bisimulation equivalence.** As this equivalence relation turns out not to be a congruence it is used to define the main equivalence relation $\approx^c$, called **weak bisimulation congruence** because it is indeed a congruence with respect to all the CCS combinators. So it is possible to take any quotient class $[E_K]$ of E_K with respect to the congruence $\approx^c$ and have the combinators induce operations on $[E_K]$. One of the main aims of this paper has been to characterise an underlying *LTS* for this structure that determines these operations.

The definitions of $\approx$ and $\approx^c$ will make sense for any *LTS*, $\mathcal{A}$, provided we assume a distinguished atomic action $\tau \in Act$. As the latter relation is no longer sensibly called a congruence in general, we shall call it the **τ-bisimulation equivalence on $\mathcal{A}$** and write it $\approx_\tau$. We first let $\Rightarrow$ be the reflexive transitive closure $(\xrightarrow{\tau})^*$ of $\xrightarrow{\tau}$ and then, for each $\alpha \in Act$ we can define $\overset{\alpha}{\Rightarrow}$ to be the relational composition $\Rightarrow\xrightarrow{\alpha}\Rightarrow$. Also, for each $\alpha \in Act$, the relation $\overset{\hat{\alpha}}{\Rightarrow}$ is defined to be the relation $\overset{\alpha}{\Rightarrow}$, except that when $\alpha = \tau$ it is $\Rightarrow$, so that $\overset{\hat{\tau}}{\Rightarrow}$ is the reflexive closure of $\overset{\tau}{\Rightarrow}$.

Definition 4.4 *For $i = 0,1,2$, we define the LTS $\mathcal{A}_i = (A, \theta_i)$, where the map $\theta_i : A \rightarrow pow(Act \times A)$ is given by:*

$$\begin{aligned}\theta_0(a) &= \{(\alpha, a') \mid a \overset{\alpha}{\Rightarrow} a'\},\\ \theta_1(a) &= \{(\alpha, a') \in \theta_0(a) \mid a' \overset{\tau}{\Rightarrow} a'\},\\ \theta_2(a) &= \theta_0(a) \cup \{(\tau, a)\}.\end{aligned}$$

Note that θ_0 and θ_2 are the transition maps whose transition relations are $\overset{\alpha}{\Rightarrow}$ and $\overset{\hat{\alpha}}{\Rightarrow}$ for $\alpha \in Act$.

Definition 4.5 *Given an LTS $\mathcal{A}$ the relation $\approx$ of* **weak bisimulation equivalence on $\mathcal{A}$** *is the maximal bismulation on the LTS $\mathcal{A}_2$ and the relation $\approx_\tau$ of* **τ-bisimulation equivalence on $\mathcal{A}$** *is given by*

$$a_1 \approx_\tau a_2 \iff (\theta_0)^{\approx}(a_1) = (\theta_0)^{\approx}(a_2).$$

When $\mathcal{A}$ is $\mathcal{E}_{\mathcal{K}}$, for some CCS system of constants $\mathcal{K}$, then $\approx_\tau$ is the relation $\approx^c$ of weak bisimulation congruence.

In [6] weak bisimulation congruence is the fundamental equivalence relation for *CCS*. So it will be our main concern. In proposition 4.14 we will give a reformulation of the definition of $\approx_\tau$, on certain *LTS*s $\mathcal{A}$, as a maximal bisimulation on an associated *LTS* $\mathcal{A}_\tau$.

Definition 4.6 *Let $\mathcal{A}$ be an LTS. It is a τ-LTS if*

1. *If $a \xrightarrow{\alpha}\xrightarrow{\tau} b$ or $a \xrightarrow{\tau}\xrightarrow{\alpha} b$ then $a \xrightarrow{\alpha} b$,*
2. *If $a' \xrightarrow{\alpha} a$ then $a \xrightarrow{\alpha} a$.*

$\mathcal{A}$ is τ-**transitive** *if 1 holds and* **weakly** τ-**reflexive** *if 2 holds. If the following strengthening of 2 holds then $\mathcal{A}$ is* τ-**reflexive**.

- *$a \xrightarrow{\alpha} a$ for all $a \in A$.*

Note that, for any *LTS* $\mathcal{A}$, $\mathcal{A}_0, \mathcal{A}_1$ and $\mathcal{A}_2$ are all τ-transitive and that $\mathcal{A}_1$ and $\mathcal{A}_2$ are τ-*LTS*s, with $\mathcal{A}_2$ also τ-reflexive. The following result will be useful.

Lemma 4.7 *Let $\pi : \mathcal{B} \to \mathcal{A}$ be an LTS map. Then, for $i = 0, 1, 2$, π is also an LTS map $\pi : \mathcal{B}_i \to \mathcal{A}_i$, provided that when $i = 1$ the LTS $\mathcal{B}$ is weakly τ-reflexive.*

4.3 Three full subcategories of *LTS*s

It will be useful to focus on the full subcategories $\mathbf{C}_0, \mathbf{C}_1, \mathbf{C}_2$ of the category of *LTS*s. An *LTS* is in $\mathbf{C}_0$ if it is τ-transitive. If it is also τ-reflexive (weakly τ-reflexive) then it is in $\mathbf{C}_2$ ($\mathbf{C}_1$). Thus $\mathbf{C}_1$ is the full subcategory of τ-*LTS*s. These subcategories are easily observed to be image-closed and union-closed, so that we can apply theorem 3.2 to get the following result.

Theorem 4.8 *For $i = 0, 1, 2$ there is a final object of $\mathbf{C}_i$.*

Proposition 4.9 *Let $\mathcal{A}$ be any LTS. For $i = 0, 1, 2$,*

$$\mathcal{A} \text{ is in } \mathbf{C}_i \iff \mathcal{A}_i = \mathcal{A}.$$

Theorem 4.10 *If $\mathcal{A}$ is a weakly complete LTS then, for $i = 0, 1, 2$, $\mathcal{A}_i$ is a weakly complete object of $\mathbf{C}_i$ so that any collapse of $\mathcal{A}_i$ is a final object of $\mathbf{C}_i$.*

4.4 *LTS*s with a τ-prefix operation

Definition 4.11 *A* **τ-prefix operation** *on an LTS $\mathcal{A} = (A, \theta)$ is an assignment of an element $a^\tau \in A$ to each $a \in A$, such that*

$$\theta(a^\tau) = \theta(a) \cup \{(\tau, a)\}.$$

Note that $\mathcal{E}_\mathcal{K}$ always has the τ-prefix operation given by $a^\tau = \tau.a$. The following result is a familiar fact about weak bisimulation on $\mathcal{E}_\mathcal{K}$ and will be useful below.

Lemma 4.12 *Let $\mathcal{A} = (A, \theta)$ be an LTS with a τ-prefix operation. Then $a \approx a^\tau$ for all $a \in A$.*

Proof: It suffices to show that $\mathcal{R} = \{(a, b) \in A \times A \mid a^\tau = b \text{ or } a = b\}$ is a weak bisimulation relation on $\mathcal{A}$. For that it suffices to show that

(a) $a^\tau \overset{\hat{\alpha}}{\Rightarrow} x$ implies $a \overset{\hat{\alpha}}{\Rightarrow} \mathcal{R}x$,

(b) $a \overset{\hat{\alpha}}{\Rightarrow} x$ implies $a^\tau \overset{\hat{\alpha}}{\Rightarrow} \mathcal{R}x$.

For (a), let $a^\tau \overset{\hat{\alpha}}{\Rightarrow} x$. Then either $a^\tau \overset{\alpha}{\Rightarrow} x$ or else $\alpha = \tau$ and $a^\tau = x$. In the first case $a \overset{\hat{\alpha}}{\Rightarrow} x$ so that $a \overset{\hat{\alpha}}{\Rightarrow} x\mathcal{R}x$. In the second case $\alpha = \tau$ so that $a \overset{\hat{\alpha}}{\Rightarrow} a\mathcal{R}a^\tau = x$. In either case $a \overset{\hat{\alpha}}{\Rightarrow} \mathcal{R}x$.
For (b), if $a \overset{\hat{\alpha}}{\Rightarrow} x$ then $a^\tau \overset{\hat{\alpha}}{\Rightarrow} x\mathcal{R}x$.

Definition 4.13 *Let $\mathcal{A} = (A, \theta)$ be an LTS with a τ-prefix operation. Then we define the LTS $\mathcal{A}_\tau = (A, \theta_\tau)$, where θ_τ is given by*

$$\theta_\tau(a) = \{(\alpha, b^\tau) \mid a \overset{\alpha}{\Rightarrow} b\}$$

for $a \in A$.

Proposition 4.14 *The maximal bisimulation on $\mathcal{A}_\tau$ is the same as the τ-bisimulation equivalence relation $\approx_\tau$ on $\mathcal{A}$.*

Proof: We need to prove the following two results:

1. $\approx_\tau$ is a bisimulation on $\mathcal{A}_\tau$,

2. If $\mathcal{S}$ is a bisimulation on $\mathcal{A}_\tau$ then

$$a_1 \mathcal{S} a_2 \implies a_1 \approx_\tau a_2.$$

1. As $\approx_\tau$ is symmetric it suffices to show that

$a \approx_\tau \overset{\alpha}{\Rightarrow} x$ implies that there is z such that $a \overset{\alpha}{\Rightarrow} z$ and $z^\tau \approx_\tau x^\tau$.

By the claim below it suffices to show that if $a \approx_\tau \overset{\alpha}{\Rightarrow} x$ then $a \overset{\alpha}{\Rightarrow} \approx x$. So let $a \approx_\tau b$ and $b \overset{\alpha}{\Rightarrow} x$. Then $\theta_0^{\approx}(a) = \theta_0^{\approx}(b)$ and, as $b \overset{\alpha}{\Rightarrow} x \approx x$ so that $(\alpha, x) \in \theta_0^{\approx}(b)$, we get that $(\alpha, x) \in \theta_0^{\approx}(a)$ so that $a \overset{\alpha}{\Rightarrow} \approx x$.

Claim: $a_1 \approx a_2 \iff a_1^\tau \approx_\tau a_2^\tau$.
To prove this claim first observe that, for all $a, b \in A$,

$$a^\tau \overset{\alpha}{\Rightarrow} b \iff a \overset{\hat{\alpha}}{\Rightarrow} b,$$

so that for any $a \in A$

$$\theta_0^{\approx}(a^\tau) = \{(\alpha, x) \mid a^\tau \overset{\alpha}{\Rightarrow} \approx x\} = \{(\alpha, x) \mid a \overset{\hat{\alpha}}{\Rightarrow} \approx x\} = \theta_2^{\approx}(a).$$

Now, as $\approx$ is the maximal bisimulation on $\mathcal{A}_2$,

$$\begin{aligned} a \approx b &\iff \theta_2^{\approx}(a) = \theta_2^{\approx}(b) \\ &\iff \theta_0^{\approx}(a^\tau) = \theta_0^{\approx}(b^\tau) \\ &\iff a^\tau \approx_\tau b^\tau. \end{aligned}$$

2. Let $\mathcal{S}$ be a bisimulation on $\mathcal{A}^\tau$. We will successively prove the following assertions, ending with the desired one. Note that $\mathcal{S}^{-1}$ is also a bisimulation on $\mathcal{A}^\tau$ so that whatever we prove about $\mathcal{S}$ will also be true of $\mathcal{S}^{-1}$.

(i) $x\mathcal{S}y \overset{\alpha}{\Rightarrow} z$ implies $x \overset{\alpha}{\Rightarrow} \approx \mathcal{S} \approx z$,

(ii) $x\mathcal{S}y \overset{\hat{\alpha}}{\Rightarrow} z$ implies $x \overset{\hat{\alpha}}{\Rightarrow} \approx \mathcal{S} \approx z$,

(iii) $a_1 \approx \mathcal{S} \approx a_2$ implies $a_1 \approx a_2$,

(iv) $a_1 \mathcal{S} a_2$ implies $a_1 \approx a_2$,

(v) $x\mathcal{S}y \overset{\alpha}{\Rightarrow} z$ implies $x \overset{\alpha}{\Rightarrow} \approx z$,

(vi) $a_1 \mathcal{S} a_2$ implies $\theta_0^{\approx}(a_2) \subseteq \theta_0^{\approx}(a_1)$,

(vii) $a_1 \mathcal{S} a_2$ implies $a_1 \approx_\tau a_2$.

Proofs:

(i) Let $x\mathcal{S} \overset{\alpha}{\Rightarrow} z$. As $\mathcal{S}$ is a bisimulation on $\mathcal{A}_\tau$, $x \overset{\alpha}{\Rightarrow} u$ and $u^\tau \mathcal{S} z^\tau$ for some u. By the lemma $u \approx u^\tau$ and $z^\tau \approx z$ so that $x \overset{\alpha}{\Rightarrow} \approx \mathcal{S} \approx z$.

(ii) If $x\mathcal{S} \overset{\hat{\alpha}}{\Rightarrow} z$ then either $x\mathcal{S} \overset{\alpha}{\Rightarrow} z$, in which case we may use (i) and the fact that $\overset{\alpha}{\Rightarrow}$ is a subrelation of $\overset{\hat{\alpha}}{\Rightarrow}$, or else $\alpha = \tau$ and $x\mathcal{S}z$, in which case $x \overset{\hat{\alpha}}{\Rightarrow} x \approx x\mathcal{S}z \approx z$.

(iii) By (ii) for $\mathcal{S}$ and also for $\mathcal{S}^{-1}$ it easily follows that $\approx \mathcal{S} \approx$ is a weak bisimulation on $\mathcal{A}$. But $\approx$ is the maximal weak bisimulation on $\mathcal{A}$.

(iv) This is an immediate consequence of (iii) because $\mathcal{S}$ is a subrelation of $\approx \mathcal{S} \approx$.

(v) This follows from (i) and (iv).

(vi) If $a_1 \mathcal{S} a_2$ and $(\alpha, x) \in \theta_0^{\approx}(a_2)$ then $a_1 \mathcal{S} \overset{\alpha}{\Rightarrow} \approx x$ so that, by (v), $(\alpha, x) \in \theta_0^{\approx}(a_1)$.

(vii) By (vi) for both $\mathcal{S}$ and $\mathcal{S}^{-1}$ if $a_1 \mathcal{S} a_2$ then $\theta_0^{\approx}(a_2) = \theta_0^{\approx}(a_1)$; i.e. $a_1 \approx_\tau a_2$.

Note that when $\mathcal{A} = \mathcal{E}_\mathcal{K}$ we get a maximal bisimulation charac-teri:ation of weak bisimulation congruence and our interest is to give a characterisation of a collapse of $\mathcal{A}_\tau$, when $\mathcal{K}$ is a universal system of constants. We will give a cha.acterisation as a final τ-LTS.

Lemma 4.15 *Let* $[-] : \mathcal{A}_\tau \rightarrow [\mathcal{A}_\tau]$ *be a collapse of* $\mathcal{A}_\tau$, *where* $\mathcal{A}$ *is an LTS with a* τ*-prefix operation. Let* $\pi : \mathcal{B} \rightarrow \mathcal{A}$ *be an LTS map, where* $\mathcal{B}$ *is a* τ*-LTS. Then* $\pi' : \mathcal{B} \rightarrow [\mathcal{A}_\tau]$ *is also an LTS map, where for* $b \in B$,

$$\pi'(b) = [\pi(b)].$$

Theorem 4.16 *If* $\mathcal{A}$ *is a weakly complete LTS, with a* τ*-prefix operation, then any collapse of* $\mathcal{A}_\tau$ *is a final* τ*-LTS.*

4.5 Defining the *CCS* combinators on a final τ-*LTS*

We assume given a final τ-*LTS*, $\mathcal{P} = (P, \theta)$. Our purpose is to show how to define the combinators of *CCS* as operations on $\mathcal{P}$.

Definition 4.17 *We will call a subset Y of $Act \times P$ a* **τ-subset** *if*

1. $(\alpha, q) \in Y$ *and* $q \xrightarrow{\tau} r \implies (\alpha, r) \in Y$,
2. $(\tau, q) \in Y$ *and* $q \xrightarrow{\alpha} r \implies (\alpha, r) \in Y$,
3. $(\alpha, q) \in Y \implies q \xrightarrow{\tau} q$.

Note that $\theta(p)$ is always a τ-subset for any $p \in P$.

Proposition 4.18 *If Y is a τ-subset then there is a unique $p \in P$ such that $\theta(p) = Y$.*

Proof: Choose an object $* \notin P$ and let $P^* = P \cup \{*\}$. Extend θ to $\theta^* : P^* \to pow(Act \times P^*)$ by letting $\theta^*(*) = Y$. Then $\mathcal{P}^* = (P^*, \theta^*)$ is easily seen to be a τ-*LTS*. Let $\pi : \mathcal{P}^* \to \mathcal{P}$ be the unique *LTS* map. This exists because $\mathcal{P}$ is a final τ-*LTS*. Then the restriction of π to $\mathcal{P}$ is still an *LTS* map $\mathcal{P} \to \mathcal{P}$. But the identity map on P is the unique *LTS* map $\mathcal{P} \to \mathcal{P}$. So $\pi(q) = q$ for all $q \in P$.

Now let $p = \pi(*)$. As π is an *LTS* map

$$\begin{aligned} \theta(p) &= \theta(\pi(*)) \\ &= \{(\alpha, \pi(q)) \mid (\alpha, q) \in \theta^*(*)\} \\ &= \{(\alpha, q) \mid (\alpha, q) \in Y\} \\ &= Y. \end{aligned}$$

The uniqueness of this p follows from the uniqueness of π.
□

Definition 4.19 (Summation) *Given a family of elements $p_i \in P$, for $i \in I$, where I is a set, each $\theta(p_i)$ is a τ-subset and therefore so is the union $\bigcup_{i \in I} \theta(p_i)$. We define $\sum_{i \in I} p_i$ to be the unique $p \in P$ such that*

$$\theta(p) = \bigcup_{i \in I} \theta(p_i).$$

We define the τ-prefix operation on $\mathcal{P}$ using the next lemma.

Lemma 4.20 *If $p \in P$ then there is a unique $p' \in P$ such that*

$$\theta(p') = \theta(p) \cup \{(\tau, p')\}.$$

Proof: Let $p \in P$. Define $\mathcal{P}^* = (P^*, \theta^*)$ as in the proof of the previous lemma, except that we now let $\theta^*(*) = \theta(p) \cup \{(\tau, *)\}$. Observe that $\mathcal{P}^*$ is a τ-LTS, so that there is a unique LTS map $\pi : \mathcal{P}^* \to \mathcal{P}$. As before $\pi(q) = q$ for $q \in P$. Now let $p' = \pi(*)$. Then

$$\begin{aligned}\theta(p') &= \{(\alpha, \pi(q)) \mid (\alpha, q) \in \theta^*(*)\}\\ &= \{(\alpha, q) \mid (\alpha, q) \in \theta(p)\} \cup \{(\tau, p')\}\\ &= \theta(p) \cup \{(\tau, p')\}.\end{aligned}$$

As π is the unique LTS map $\mathcal{P}^* \to \mathcal{P}$ it is easy to see that p' must be the unique element of P such that

$$\theta(p') = \theta(p) \cup \{(\tau, p')\}.$$

□

Definition 4.21 *Given $p \in P$ we let $\tau.p$ be the unique p' given by the lemma. If l is a label then $Y = \{(l, q) \mid (\tau, q) \in \theta(\tau.q)\}$ is a τ-subset so that there is a unique $r \in P$ such that $\theta(r) = Y$. We let $l.p$ be this unique r. So we have defined $\alpha.p$ for any $\alpha \in Act$ and any $p \in P$.*

In order to define the CCS parallel composition on $\mathcal{P}$ we first need to define a labelled transition relation on $P \times P$. The relations $\xrightarrow{\alpha}$ on $P \times P$ are given by:-
If l is a label then let

$$(p, q) \xrightarrow{l} (p', q') \iff \text{either } (p \xrightarrow{l} p' \ \& \ q = q') \text{ or } (p = p' \ \& \ q \xrightarrow{l} q').$$

Also let

$$\begin{aligned}(p, q) \xrightarrow{\tau} (p', q') \iff \quad &\text{either } p \xrightarrow{l} p' \ \& \ q \xrightarrow{\bar{l}} q'\\ &\text{or } (p \xrightarrow{\tau} p' \ \& \ q = q') \text{ or } (p = p' \ \& \ q \xrightarrow{\tau} q').\end{aligned}$$

Having defined the $\xrightarrow{\alpha}$ relations on $P \times P$ we go on to define the relation $\overset{\alpha}{\Rightarrow}$, as usual, to be the relational composition $(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^*$

for each $\alpha \in Act$ and so can make $P \times P$ into an LTS $\mathcal{P}' = (P, \theta')$ where, for $(p,q) \in P \times P$,

$$\theta'(p,q) = \{(\alpha,(p',q')) \mid (p,q) \xrightarrow{\alpha} (p',q')\}.$$

Lemma 4.22 *$\mathcal{P}'$ is a τ-LTS.*

Definition 4.23 (Composition) *The composition operation, $-|-$: $P \times P \rightarrow P$ of CCS, is defined to be the unique LTS map $\mathcal{P}' \rightarrow \mathcal{P}$.*

Definition 4.24 (Restriction) *If L is a sort let $\mathcal{P}_L = (P, \theta_L)$ be the LTS where, for $p \in P$,*

$$\theta_L(p) = \{(\alpha, q) \in \theta(p) \mid \alpha \notin L^{\#}\}.$$

Then $\mathcal{P}_L$ is a τ-LTS. We let $-\backslash L : P \rightarrow P$ be the unique LTS map $\mathcal{P}_L \rightarrow \mathcal{P}$.

Definition 4.25 (Relabelling) *If f is a relabelling map let $\mathcal{P}_f = (P, \theta_f)$ where, for $p \in P$,*

$$\theta_f(p) = \{(f(\alpha), q) \mid (\alpha, q) \in \theta(p)\}.$$

Then $\mathcal{P}_f$ is a τ-LTS. We let $-[f] : P \rightarrow P$ be the unique LTS map $\mathcal{P}_f \rightarrow \mathcal{P}$.

4.6 A Denotational Semantics for CCS

We have seen how to define operations on a final τ-LTS $\mathcal{P}$, corresponding to the combinators of CCS. These can be used to give a denotational semantics for E_K; i.e we can associate a denotation $[[e]] \in P$ tc each $e \in E_K$. To take care of the possibly recursive definitions of the constants we first define $[\![-]\!]_\phi : E_K \rightarrow P$, given $\phi : K \rightarrow P$.

Definition 4.26 *Given $\phi : K \rightarrow P$ let $[\![-]\!]_\phi : E_K \rightarrow P$ be defined by structural recursion on the way agents in E_K are built up using the equations:-*

$$\begin{aligned}
[\![c]\!]_\phi &= \phi(c) \\
[\![\alpha.e]\!]_\phi &= \alpha.[\![e]\!]_\phi \\
[\![\textstyle\sum_{i\in I} e_i]\!]_\phi &= \textstyle\sum_{i\in I} [\![e_i]\!]_\phi \\
[\![e_1|e_2]\!]_\phi &= [\![e_1]\!]_\phi \mid [\![e_2]\!]_\phi \\
[\![e\backslash L]\!]_\phi &= [\![e]\!]_\phi\backslash L \\
[\![e[f]]\!]_\phi &= [\![e]\!]_\phi[f]
\end{aligned}$$

Now we can define $[\![e]\!] = [\![e]\!]_{\phi_0}$ where ϕ_0 is the 'least' map $\phi : K \to P$ such that $\phi(c) = [\![e_c]\!]_\phi$ for all constants $c \in K$. More precisely we have the following result and definition.

Theorem 4.27 *There is a unique map $\phi_0 : K \to P$ such that*

1. $\phi_0(c) = [\![e_c]\!]_{\phi_0}$ *for all constants $c \in K$.*
2. *If $\phi(c) = [\![e_c]\!]_\phi$ for all constants $c \in K$ then*

$$\theta(\phi_0(c)) \subseteq \theta(\phi(c)) \textit{ for all constants } c \in K.$$

Definition 4.28 *Let $[\![-]\!]$ be $[\![-]\!]_{\phi_0}$, where ϕ_0 is given by the theorem.*

Finally we spell out how this denotational semantics for CCS is related to the familiar operational semantics.

Theorem 4.29 $[\![-]\!]$ *is the (necessarily unique) LTS-map $(\mathcal{E}_K)_\tau \to \mathcal{P}$. More explicitly we get 1 and therefore 2 below.*

1. *For all $e \in E_K$*

$$\theta([\![e]\!]) = \{(\alpha, \theta([\![\tau.e']\!]) \mid e \overset{\alpha}{\Rightarrow} e'\},$$

2. *For all $e_1, e_2 \in E_K$*

$$e_1 \approx^c e_2 \iff [\![e_1]\!] = [\![e_2]\!].$$

5 *CSP* Processes

5.1 The notion of a *CSP*-system

We will give a characterisation of the CSP notion of process in terms of CSP-systems. In [5] event names play roughly the same role for CSP that atomic actions play in CCS. We shall assume given a fixed set $\mathcal{N}$ of **event names**. Earlier we formulated a **coloured** version of the notion of an LTS. We now specialise to the colours needed for the version of CSP presented in [5]. A feature of CSP is that processes can have varying alphabets, the alphabet of a process being the set of the event names that *ever make sense* for the process. This will be one aspect of CSP colouring. Another aspect will be the association of a refusals set to each process. Given an alphabet $\mathcal{L} \subseteq \mathcal{N}$, if $L \subseteq \mathcal{L}$ then a set $R \subseteq pow(\mathcal{L})$ is a **refusals set** for L. relative to the alphabet $\mathcal{L}$, if R is non-empty and

$$R = \{X \subseteq \mathcal{L} \mid fin(L \cap X) \subseteq R\},$$

where, for any set Y, $fin(Y)$ is the set of all finite subsets of Y. We define the set Col of colours for CSP to be the set of those triples $(\mathcal{L}, L, R)$ such that $L \subseteq \mathcal{L} \subseteq \mathcal{N}$ and R is a refusals set for L relative to $\mathcal{L}$. Now if $\mathcal{A} = (A, \phi)$ is a $CLTS$, with this set of colours, the map $col : A \to Col$ determines maps $alph, dom : A \to pow\mathcal{N}$ and $refus : A \to pow(pow(\mathcal{N}))$ so that for $a \in A$

$$col(a) = (alph(a), dom(a), refus(a)).$$

Here $alph(a)$ is the **alphabet of** a, $dom(a)$ is the **domain of** a and $refus(a)$ is the **refusals set of** a.

Definition 5.1 *A CSP-***system** *is a deterministic $CLTS$, using the sets $\mathcal{N}$ of atomic actions and Col of colours as above, such that if $a \xrightarrow{\alpha} a'$ then $\alpha \in dom(a)$ and $alph(a') = alph(a)$.*

Let $\mathbf{C}_{CSP}$ be the full subcategory of the category of all $CLTS$s, consisting of those that are CSP-systems. It is easy to check that

Proposition 5.2 $\mathbf{C}_{CSP}$ *is image and union closed and hence there is a final CSP-system.*

5.2 The CSP system of non-chaotic CSP processes

We give the mathematical definition of the notion of CSP process that is in [5], using our notation and terminology. We have incorporated an extra condition that is needed as we do not make the simplifying assumption, made in [5], that the alphabet of a process must be finite.

Definition 5.3 *A CSP* **process** *is a triple $(\mathcal{L}, F, D)$ where $\mathcal{L} \subseteq \mathcal{N}$, $F \subseteq \mathcal{L}^* \times pow(\mathcal{L})$ and $D \subseteq \mathcal{L}^*$ such that if $T = \{\sigma \in \mathcal{L}^* \mid (\sigma, X) \in F$ for some $X\}$ then*

1. *T is a trace-set.*

2. *For each $\sigma \in T$ the set $\{X \subseteq \mathcal{L} \mid (\sigma, X) \in F\}$ is a refusals set for $\{\alpha \in \mathcal{L} \mid \sigma\alpha \in T\}$ relative to $\mathcal{L}$.*

3. *For each $\sigma \in D$*

$$(\sigma, X) \in F \text{ for all } X \subseteq \mathcal{L} \text{ and } \sigma\alpha \in D \text{ for all } \alpha \in \mathcal{L}.$$

There are particular processes of CSP that play a singular role in the theory. For each alphabet $\mathcal{L} \subseteq \mathcal{N}$ there is the **chaotic process**

$$CHAOS_{\mathcal{L}} = (\mathcal{L}, \mathcal{L}^* \times pow(\mathcal{L}), \mathcal{L}^*).$$

These are processes to be avoided and any kind of divergence gives rise to one of them. For our purposes it will be convenient to leave them out of the universe that we shall characterise. They could easily be kept in by switching to the category of pointed classes (or sets) where we have been using the category of classes. There may even be conceptual reasons for feeling that they may be best not included, although the effect is to make some of the CSP combinators partial rather than total.

So we want to define a CSP system $\mathcal{P}^-_{CSP} = (P^-_{CSP}, \phi_{CSP})$, where P^-_{CSP} is the set of non-chaotic CSP processes. For $p = (\mathcal{L}, F, D)$ let

$$\phi_{CSP}(p) = (\theta_{CSP}(p), (\mathcal{L}, \{\alpha \in \mathcal{L} \mid (\alpha, \emptyset) \in F\}, \{X \mid (<, X) \in F\})).$$

Here

$$\theta_{CSP}(p) = \{(\alpha, p/\alpha) \mid \alpha \in \delta(p)\},$$

where

$$\delta(p) = \{\alpha \in \mathcal{L} \mid (\alpha, \emptyset) \in F \text{ and } \alpha \notin D\}$$

and, for $(\alpha, \emptyset) \in F$, $p/\alpha = (\mathcal{L}, F_\alpha, D_\alpha)$ where

$$F_\alpha = \{(\sigma, X) \mid (\alpha\sigma, X) \in F\} \text{ and } D_\alpha = \{\sigma \mid \alpha\sigma \in D\}.$$

It should be clear that $\mathcal{P}_{CSP}^{-}$ is a CSP system. In fact we have the following result.

Theorem 5.4 *$\mathcal{P}_{CSP}^{-}$ is a final CSP system.*

This final universe characterisation of CSP is unlikely to be the best. It focuses on the deterministic transition relations given by the *after* operations $-/\alpha$, where p/α is the process that behaves like p after p has engaged in α, provided that p can engage in α and p does not make any internal choices. A better comparison with CCS may be obtained by looking at the non-deterministic transition relations that combine the external after operation with internal choice. Also for comparison with CCS let us restrict attention to the processes having a fixed alphabet $\mathcal{L}$. Let $Act = \mathcal{L} \cup \{\tau\}$ where τ is not an event name. Let $P_{CSP}^{\mathcal{L}}$ be the set of *all* the processes of CSP with alphabet $\mathcal{L}$. In CSP the relation that expresses a purely internally determined transition is given by the following definition. If $p_i = (\mathcal{L}, F_i, D_i)$, for $i = 1, 2$, then

$$p_1 \sqsubseteq p_2 \iff F_1 \supseteq F_2 \text{ and } D_1 \supseteq D_2.$$

Note that this relation is a complete partial order with least element $CHAOS_{\mathcal{L}}$. When mixing internal with external transitions we get the following transition map $\psi_{CSP}^{\mathcal{L}}$ on $P_{CSP}^{\mathcal{L}}$. If $p = (\mathcal{L}, F, D) \in P_{CSP}^{\mathcal{L}}$ then

$$\psi_{CSP}^{\mathcal{L}}(p) = \{(\alpha, q) \mid ((\alpha, \emptyset) \in F \text{ and } p/\alpha \sqsubseteq q) \text{ or } (\alpha = \tau \text{ and } p \sqsubseteq q)\}.$$

It is easy to check that $\mathcal{P}_{CSP}^{\mathcal{L}} = (P_{CSP}^{\mathcal{L}}, \psi_{CSP}^{\mathcal{L}})$ is an LTS in $\mathbf{C}_2$ which can be embedded in any final object of $\mathbf{C}_2$ and hence of any final τ-LTS, the universe of CCS processes. We end by posing the problem: Find an elegant final universe characterisation of this LTS.

6 Conclusion

In this paper we have considered several final universe characterisations of universes of processes. We expect that other universes can be given similar treatments. We believe that the presentations of universes in this style will help to unify the subject. The notion of a final universe seems particularly appropriate for process algebra, as it captures in one idea several aspects. One aspect is the frequent use of a general scheme of mutual recursion for defining processes. This is captured by the weak finality property of a final universe. Another aspect of process algebra is its abstractness. Processes of process algebra are identified when they have the same abstract behaviour. This aspect is captured by the strong extensionality property of a final universe. A third aspect, that is really a combination of the previous two, is that the combinators of process algebra can be uniquely defined on the final universe and do not need to be explicitly featured in the mathematical structure that has been characterised. In this paper we have illustrated this point for CCS. The combinators of CSP could be given a similar treatment. But we have left this for another occasion where we would hope to have a better treatment of CSP than that presented here.

We end with a final remark about CCS. We have seen that the CCS combinators can be defined on any final τ-LTS $\mathcal{P}$. In fact, conversely, the transition relations associated with $\mathcal{P}$ can be defined in terms of binary sums and the prefix operations as follows.

$$p \xrightarrow{\alpha} q \iff p + \alpha.q = p \text{ and } \tau.q = q$$

An apparently simpler approach would be to modify this definition by leaving out the second conjunct $\tau.q = q$. But, as far as I can see, the resulting LTS would not be so easy to characterise. What is still left unanswered is the intuitive status of any notion of labelled transition on abstract processes. It would be pleasing if mathematically simple definitions could be linked to intuitively satisfying explanations of the computational ideas.

6 Conclusion

In this paper we have considered several final universe characterisations of universes of processes. We expect that other universes can be given similar treatments. We believe that the presentations of universes in this style will help to unify the subject. The notion of a final universe seems particularly appropriate for process algebra, as it captures in one idea several aspects. One aspect is the frequent use of a general scheme of mutual recursion for defining processes. This is captured by the weak finality property of a final universe. Another aspect of process algebra is its abstractness. Processes of process algebra are identified when they have the same abstract behaviour. This aspect is captured by the strong extensionality property of a final universe. A third aspect, that is really a combination of the previous two, is that the combinators of process algebra can be uniquely defined on the final universe and do not need to be explicitly featured in the mathematical structure that has been characterised. In this paper we have illustrated this point for CCS. The combinators of CSP could be given a similar treatment. But we have left this for another occasion where we would hope to have a better treatment of CSP than that presented here.

We end with a final remark about CCS. We have seen that the CCS combinators can be defined on any final τ-LTS $\mathcal{P}$. In fact, conversely, the transition relations associated with $\mathcal{P}$ can be defined in terms of binary sums and the prefix operations as follows.

$$p \xrightarrow{\alpha} q \iff p + \alpha.q = p \text{ and } \tau.q = q$$

An apparently simpler approach would be to modify this definition by leaving out the second conjunct $\tau.q = q$. But, as far as I can see, the resulting LTS would not be so easy to characterise. What is still left unanswered is the intuitive status of any notion of labelled transition on abstract processes. It would be pleasing if mathematically simple definitions could be linked to intuitively satisfying explanations of the computational ideas.

References

[1] P. Aczel, *Non-Well-Founded Sets*, CSLI Lecture Notes, No. 14, Stanford University, 1988.

[2] P. Aczel and Nax Mendler, *A Final Coalgebra Theorem*, in Springer Lecture Notes in Computer Science No. 389, Category Theory and Computer Science, edited by D.H. Pitt et al., pp 357-365.

[3] J.C.M. Baeten and W.P. Weijland, *Process Algebra*, Cambridge University Press, 1990.

[4] M. Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988.

[5] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[6] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[7] Lawrence S. Moss and Satish R. Thatte, *Generalisation of Final Algebra Semantics by Relativisation*, LNCS 442 (1990), 284-300.

[8] J.J.M.M. Rutten and D. Turi, *On the Foundations of Final Semantics: Non-standard sets, Metric Spaces, Partial Orders*, Technical Report CS-R9241, CWI (Centre for Mathematics and Computer Science), Amsterdam, 1992.

On the symmetry of sequentiality

Pierre-Louis Curien
CNRS - Ecole Normale Supérieure
45 rue d'Ulm 75230 Paris Cedex 05, curien@dmi.ens.fr

Abstract

We offer a symmetric account of sequentiality, by means of *symmetric algorithms*, which are pairs of sequential functions, mapping input data to output data, and output exploration trees to input exploration trees, respectively. We use the framework of *sequential data structures*, a reformulation of a class of Kahn-Plotkin's concrete data structures. In sequential data structures, data are constructed by alternating questions and answers. Sequential data structures and symmetric algorithms are the objects and morphisms of a symmetric monoidal closed category, which is also cartesian, and is such that the unit is terminal. Our category is a full subcategory of categories of games considered by Lamarche, and by Abramsky-Jagadeesan, respectively.

Following Lamarche, we construct a comonad corresponding to contraction. We define this comonad via an adjunction between the category of symmetric algorithms and the "old" cartesian closed category of sequential algorithms, defined in the late seventies by the author and Gérard Berry. Thus sequential algorithms model not only typed λ-calculus, but also intuitionistic affine logic, with connectives $\otimes$, 1, $\multimap$, $\times$, and $\top$.

This work, while finding its roots in the study of sequentiality, presents striking correspondences with game-theoretic concepts, introduced by Blass in the early seventies in a very different context. The aim of the present work is to offer a systematic connection between sequentiality and games. Also, the notion of symmetric algorithm appears to be new.

1. Introduction

In the last fifteen years, the semantic study of sequentiality has been associated with the full abstraction problem for sequential programming languages [CuMon, BCL, CuSur]. And indeed, a new result of Cartwright, Curien and Felleisen, reported in [CF, CuObs, CCF] is that the sequential model of Berry and Curien [BeCu1, CuMon] is fully abstract for SPCF, an extension of PCF with control operators. (PCF is a typed λ-calculus with recursion and arithmetic operations [Gun]).) In this paper, we address the relations between sequentiality and games. Game-theoretic interpretations of proofs have become a recent subject of interest, after Blass recently brought old work of his to the attention of the linear logic research community [Blass1, Blass2].

We proceed directly in the definition/theorem style, trying to explain concepts as they are introduced, both from the point of view of games and of sequentiality. The work presented here stands as a prefix to wider efforts aiming at developing interpretations of all or part of linear logic based on the ideas of sequential algorithms [Lam1, Lam2], and games [AJ2, HO].

In Section 2, we present a notion of sequential data structure, which is very close to the notions of game in the sense of Blass, or of Abramsky-Jagadeesan (which are themselves variants of Conway's, or Joyal's games [Con, Joy]), and to the notion of sequential domain found in [Lam1]. The definition of sequential data structure is an appealing reformulation of the notion of filiform concrete data structure [CuMon]. Concrete data structures, introduced by Kahn and Plotkin [KP], support a general definition of sequential function, which itself has served as a starting point to the author's semantic investigation of sequentiality. *Filiform* concrete data structures were recognized by the author as sufficient for the purpose of modelling a language like PCF. But we had not remarked the essential symmetry of this special class of concrete data structures, which turns out to be instrumental for an investigation of linear (actually, affine) sequentiality.

In Section 3, we define morphisms between sequential data structures, which we call affine sequential algorithms, in two different ways. First, as "programs" written in the style of the language CDS of Berry-Curien [CuMon, BeCu2]. Second, as pairs of two functions: an input-output behaviour, and a computation strategy, respectively. Such pairs (function, computation strategy) are already central in [BeCu1, CuMon], but these works present us with no real symmetry between the two components of the pairs, beyond the fact that a computation strategy is roughly "going from the output to the input". In the framework of sequential data structures, and, more importantly, in the special case of *affine* sequential algorithms, a computation strategy can be formulated as a (partial) function from output exploration trees to input exploration trees, and the pairs (function, computation strategy) can be axiomatized in such a way that the two functions have symmetric properties. At the best of our knowledge, this axiomatization appears for the first time here.

In Section 4, we define the composition of two affine algorithms as the pair of the compositions of their two components. There is also an operational definition of the composition of sequential algorithms, which goes back to [BeCu2, CuMon (Definition 3.5.5)], and has been elegantly reformulated by Abramsky and Jagadeesan as "parallel composition + hiding" [AJ2]. We show that we obtain a symmetric monoidal closed, cartesian category, where the unit of the tensor is also terminal (this is the categorical characterization of affinity).

In Section 5, we reintroduce Berry-Curien's sequential algorithms in the framework of sequential data structures, and define a left adjoint ! to the inclusion functor from the category of affine sequential algorithms to the category of sequential algorithms. By categorical reasoning, we deduce that the category of sequential algorithms can be recast as the CoKleisli category of the comonad ! induced by this adjunction. Altogether we have a model of affine linear logic.

On the side, following a suggestion of Streicher, we formulate yet another characterization of sequential algorithms, as sequential functions that propagate and *reflect* errors. Error-sensitive functions, also called observably sequential functions, are discussed at length in [CuObs, CCF]. The new feature here is error reflection. The explicit presence of error data in the domains allows us to witness computation strategies extensionally, and error reflection ensures that errors serve only that purpose.

In Section 6 we collect remarks and comparisons with related work. We briefly discuss ways of interpreting other connectives of linear logic.

Many of the proofs are only sketched, but the parts left out are mostly routine.

Notation *(Paths)*

Given a partial order $(X,\leq)$, we use the notation $x\uparrow y$ to denote the fact that $x\in X$ and $y\in Y$ are compatible, that is, have an upper bound. We use "glb" and "lub" as shorthands for "greatest lower bound" and "least upper bound".

Given an alphabet A, A* denotes the set of words, called here *paths* over A, that is of strings of symbols taken from A. String concatenation, and concatenation of a string and a letter of A, are denoted by simple juxtaposition: ww', aw, wa, etc... The empty string is written ε. A path different from ε is called a non-ε path. The paths are ordered by the prefix ordering: $w\leq w'$ if $w' = ww''$ for some w''. This preorder is such that every two paths are either comparable or incompatible. A path is called *non-repetitive* when each symbol of A occurs at most once in it. Given a subset B of A, for any word $w \in A^*$, we define $w\restriction B$ as follows:

$$\varepsilon\restriction B = \varepsilon,\ \ wd\restriction B = w\restriction B\ \text{ if } d\in A\backslash B,\ \ wd\restriction B = (w\restriction B)d\ \text{ if } d\in B$$

We shall need the following transformation "copycat" on words, which we define as follows:

$$\text{copycat}(\varepsilon) = \varepsilon,\ \text{copycat}(wd) = \text{copycat}(w)dd$$

Unless explicitly needed, we shall treat disjoint unions as ordinary unions.

2. Sequential data structures

The following definition appears also in [CCF].

Definition *(Sequential data structure)*

A sequential data structure structure (sds for short) M=(A,D,P) is given by:

- a set A of *addresses* a, a_1,...,
- a set D of *data* d, d_1,... (A and D are assumed disjoint),
- a collection P of non-ε alternating paths over A∪D that start with an address.

Thus, the paths are of the form $a_1d_1...d_{n-1}a_nd_n$ or of the form $a_1d_1...d_{n-1}a_nd_na_{n+1}$. Moreover, it is assumed that P is closed under non-ε prefixes. We shall loosely call the elements of P paths of M, or even simply paths.

We call *move* any element of A or of D. We use m to denote a move, and we say that m belongs to M, which will thus also serve sometimes to denote A∪D. A path ending with a datum is called a *response*, and a path ending with an address is called a *query*. We use p (or s, or t), q, and r to range over paths, queries, and responses, respectively. We denote by Q and R the sets of queries and responses, respectively.

A *strategy* (a tree in the terminology of [CCF], a state in the terminology of [KP, CuMon]) of M is a subset x of R that is closed under response prefixes and binary non-ε glb's[1]:

$$r_1,r_2\in x,\ r_1\wedge r_2\neq\varepsilon\ \Rightarrow\ r_1\wedge r_2\in x$$

A *counterstrategy* is a non-empty subset of Q that is closed under query prefixes and under binary glb's. We use x, y, ... and α, β, ... to range over strategies and counter-strategies, respectively.

Both sets of strategies and of counter-strategies are ordered by inclusion. They are denoted by D(M) and $D^{\perp}$(M), respectively. Notice that D(M) has always a minimum element (the empty strategy, written ∅ or ⊥), while $D^{\perp}$(M) has no minimum element in general. D(M) is called the sds domain generated by M. It is a Scott domain [GuSco], and more precisely a dI-domain [Be, BCL], whose compact elements are the finite strategies. We denote by D°(M) the set of finite strategies of M. The dI-domains enjoy the property that any compact element dominates only finitely many elements. This property, called property I, is essential in the theory of stable functions [Be] (see Section 3). $D^{\perp}$(M) enjoys the same properties (except for the existence of a minimum element), and we denote by $D^{\perp\circ}$(M) the set of finite counter-strategies.

Among the strategies are the sets of response prefixes of a response r. By abuse of notation we call still r the resulting strategy. It is easy to see that those r's are exactly the (non ⊥) prime[2] elements of D(M).

[1]Alternatively, as for example in [AJ2], we could have included the empty path in P, and have imposed strategies to be non-empty.

[2]We recall that an element p is prime when for any upper bounded $X\subseteq D(M)$, $(p\leq\vee X\Rightarrow\exists x\in X\ p\leq x)$.

We end this definition by fixing some terminology. Let x be a strategy.

- If qd∈x for some d, we say that q is *filled* in x, and we write q∈F(x).
- If r∈x and q=rc for some c, we say that q is *enabled* in x.
- If q is enabled but not filled in x, we say that q is *accessible* from x, and we write q∈A(x).

We define likewise r∈F(α) and r∈A(α) for a response r and a counterstrategy α. O

A more geometric reading of this definition is that an sds is a labelled forest, where the ancestor relation alternates addresses and data, and where the roots are labelled by addresses.

We give some examples of sds's (see also [CCF]). A flat domain is described as the set of strategies of an sds with a single address ? and a collection of paths of length not greater than 2. For example, in Figure 1, we represent the flat domain of natural numbers.

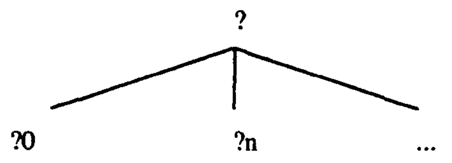


Figure 1: Flat domains of natural numbers

Figure 2 represents the cartesian product Bool^2 of the boolean domain with itself ("x" and "y" stand for the "coordinates" x and y, thinking of a function f(x,y) defined on this domain).

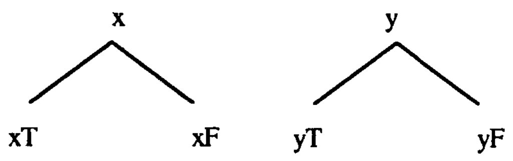


Figure 2: The sds Bool^2

The elements, say, (T,⊥) and (T,F), of Bool^2 are represented by the strategies {xT} and {xT,yF}, respectively.

A more sophisticated example is provided by the partial terms over a signature S such as $\{a^0,f^1,g^2\}$, where the superscripts are the arities. The partial terms over this signature are the strategies of the sds shown in Figure 3, whose addresses are the occurrences, and whose data are the function symbols.

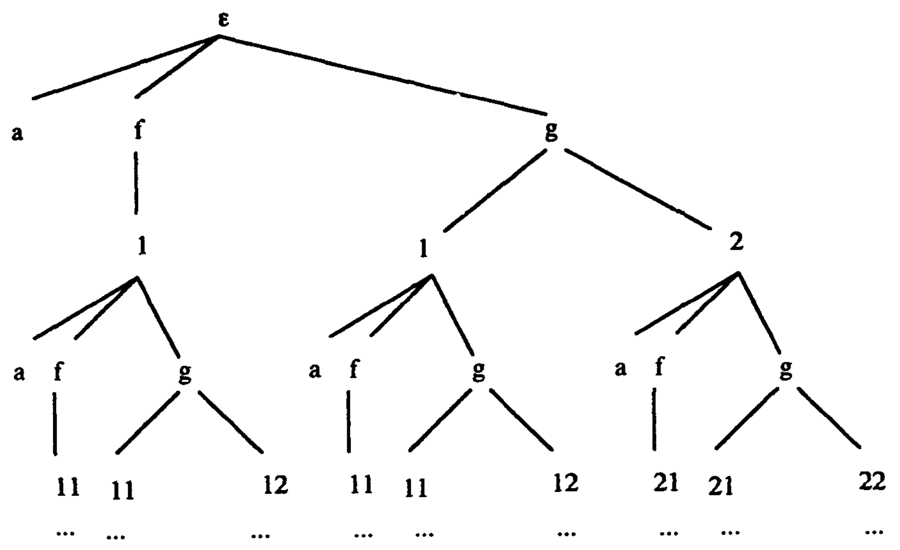


Figure 3: The sds of partial terms

In Figure 4, we show the strategy representing g(a,f(a)):

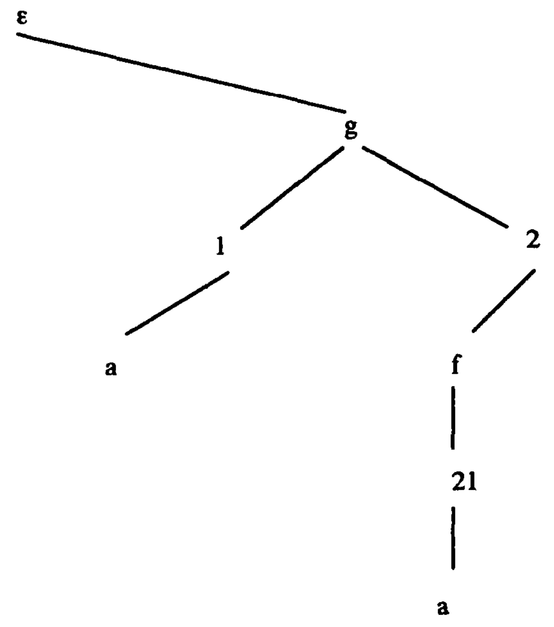


Figure 4: A strategy

and here is a counterstrategy:

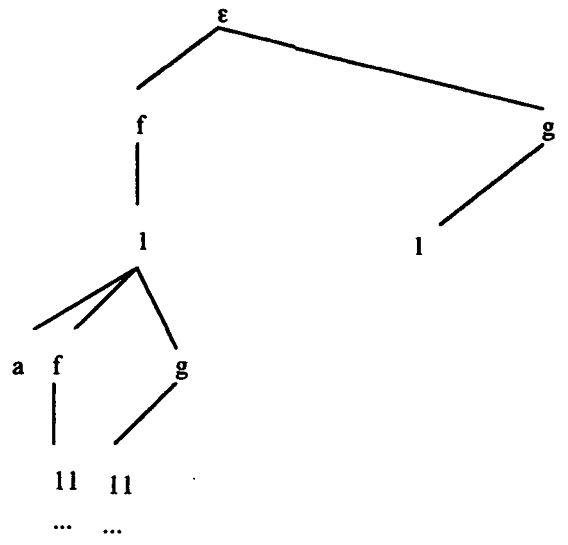


Figure 5: A counterstrategy

A counterstrategy can be read as an exploration tree, or a pattern. The root is investigated first; if the function symbol found at the root is g, then its left son is investigated next; otherwise, if the function symbol found at the root is f, then its son is investigated next, and the investigation goes further if the symbol found at node 1 is either f or g.

A more geometric reading of the definitions of sds, strategy and counterstrategy is the following:

- an sds is a forest,
- a strategy is a sub-forest which is allowed to branch only at data,
- a counterstrategy α is a non-empty sub-tree (if it contained c and d as paths of length 1, they should contain their glb, which is ε, contradicting $\alpha \subseteq P$) which is allowed to branch only at addresses.

The pairs address / datum, query / response, and strategy / counterstrategy give to sds's a large flavour of symmetry. These pairs are related to other important dualities in programming: Lamarche [Lam2], and Abramsky and Jagadeesan [AJ1] have pointed out the correspondence query - input (and response - output), and the correspondence strategy - constructor (and counterstrategy - destructor), respectively.

It is thus tempting to conceive of the counter-strategies of an sds M as the strategies of a dual structure whose addresses are the data of M and whose data are the addresses of M. This can be done in a number of ways. For example, Abramsky and Jagadeesan relax the condition that paths start with an address: in their framework, given a structure (A,D,P) (a game, in their terminology), (D,A,P) is another game, its dual, or its linear negation[3]. More restrictively, in his second work on sequentiality and games [Lam2], Lamarche considers structures of either of two polarities ∘ and •, which stand for "address" and "datum", or "input" and "output" (in the sense of Danos-Régnier [Dan, MR, Reg]). The sds's are exactly Lamarche's games of polarity •. The negation of an sds has polarity ∘. Lamarche represents the polarity • explicitly, by adding a root of that polarity to the forest representation of one of our sds's[4]. A generic sds viewed in this way is represented in Figure 6.

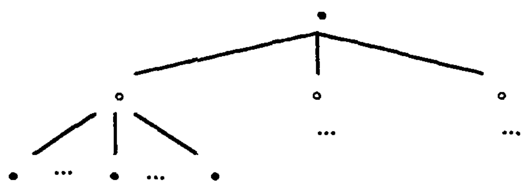


Figure 6: A game of polarity •

A representation of the negation of this generic sds is obtained by changing each label to its dual. Lamarche's notation presents some advantages. It enhances the symmetry of strategies and of counter-strategies. Strategies are non-empty subtrees that branch only at nodes labelled •, and counter-strategies are non-empty subtrees that branch only at nodes labelled ∘. Lamarche's convention can be fixed in notation by reformulating the definition of sds as follows: replace D by the disjoint union of D and of a distinguished element •, and require now strategies to be non-empty sets of paths. We shall call this a *Lamarchian sds*. We say that the data have polarity • and that the addresses have polarity ∘.

We now offer a reading of sds's as games. An sds can be considered as a game between two persons, the opponent and the player. The data are the player's moves, and the addresses are the opponent's moves. A player's strategy consists in having ready answers for (some of) the opponent's moves. Counter-strategies are just opponent's strategies. The following proposition makes the analogy more suggestive.

Proposition *(Play)*
Let M be an sds, x be a strategy and α be a counterstrategy of M, one of which is finite. We define $x|\alpha$ as the set of paths p which are such that all the response prefixes of p are in x and all the query prefixes of p are in α. Then $x|\alpha$ is totally ordered, and can be confused with its maximum element, which is uniquely characterized as follows:

[3]However, in [AJ2], the counter-strategies of (A,D,P) are not strategies of (D,A,P): in their framework, all the paths of a strategy start with an opponent's move. See also section 6.

[4]This root just corresponds to the empty path, missing in P (cf. footnote 1).

- $x|\alpha$ is the unique element of $x \cap A(\alpha)$ if $x|\alpha$ is a response,
- $x|\alpha$ is the unique element of $\alpha \cap A(x)$ if $x|\alpha$ is a query.

If $x|\alpha$ is a response, we say that x *wins* against α, and we denote this predicate $x \lhd \alpha$. If $x|\alpha$ is a query, we say that α *wins* against x, and we write $x \rhd \alpha$, thus $\rhd$ is the negation of $\lhd$. To stress the winner, we often write $x|^{\rhd}\alpha$ for $x|\alpha$ when α wins, and $x^{\lhd}|\alpha$ for $x|\alpha$ when x wins.

Proof: Suppose that $p_1, p_2 \in x|\alpha$. We show that p_1 and p_2 are comparable, by contradiction. Thus suppose $p_1 \wedge p_2 < p_1$ and $p_1 \wedge p_2 < p_2$. Let q_1 be the largest query prefix of p_1, let r_1 be the largest prefix of p_1 which is a response or ε, and let q_2 and r_2 be defined similarly. We show:

$$p_1 \wedge p_2 = q_1 \wedge q_2 = r_1 \wedge r_2$$

One direction follows by the monotonicity of $\wedge$: $q_1 \wedge q_2 \leq p_1 \wedge p_2$. For the other direction, we remark that by the maximality of q_1, $p_1 \wedge p_2 < p_1$ implies $p_1 \wedge p_2 \leq q_1$; and, similarly, we deduce $p_1 \wedge p_2 \leq q_2$, which completes the proof of $p_1 \wedge p_2 = q_1 \wedge q_2$. The equality $p_1 \wedge p_2 = r_1 \wedge r_2$ is proved similarly. But by definition of a strategy and of a counterstrategy, $q_1 \wedge q_2$ is a query, and $r_1 \wedge r_2$ is either a response or ε. Thus the equalities just proven imply that $p_1 \wedge p_2$ is of both odd and even length: contradiction. Thus $x|\alpha$ is totally ordered. It has a maximum element, since the finiteness of x or α implies the finiteness of $x|\alpha$.

To prove the rest of the statement, we first observe that $x \cap A(\alpha) \subseteq x|\alpha$ and $\alpha \cap A(x) \subseteq x|\alpha$, by definition of $x|\alpha$. We next show that $x \cap A(\alpha)$ and $\alpha \cap A(x)$ have at most one element. If $p_1, p_2 \in x \cap A(\alpha)$, then by the first part of the statement p_1 and p_2 are comparable, say $p_1 \leq p_2$. But if $p_2 \in A(\alpha)$ and $p_1 < p_2$, then $p_1 \in F(\alpha)$, contradicting the assumption $p_1 \in A(\alpha)$. Hence $p_1 = p_2$. The proof is similar for $\alpha \cap A(x)$. Finally, if $x|\alpha$ viewed as a path is a response, then $x|\alpha \in x$, $x|\alpha$ is enabled in α, and the maximality of $x|\alpha$ implies that $x|\alpha$ is not filled in α. Hence $x|\alpha \in x \cap A(\alpha)$, which by what precedes can be rephrased as $x \cap A(\alpha) = \{x|\alpha\}$. □

The path $x|\alpha$ formalizes the interplay between the player with strategy x and the opponent with strategy α. If $x|\alpha$ is a response, then the player wins since he made the last move, and if $x|\alpha$ is a query, then the opponent wins. Here is a game-theoretic reading of $x|\alpha$. At the beginning the opponent makes a move a: its strategy determines that move uniquely. Then either the player is unable to move (x contains no path of the form ad), or his strategy determines a unique move. The play goes on until one of x or α does not have the provision to answer its opponent's move. As an example, if x and α are the strategy and counterstrategy of the sds of partial terms which we have drawn in Figure 4 and Figure 5, then $x|\alpha$ is the path shown in Figure 7, and the player wins.

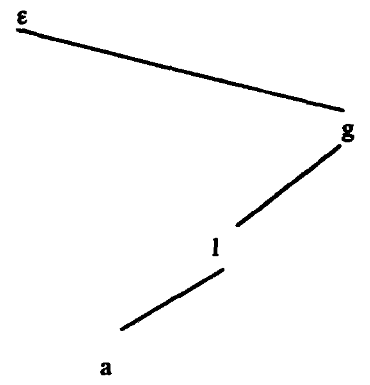


Figure 7: A play

The result x|α of the interplay between x and α is reminiscent of the models of linear logic based on linear algebra proposed by Lafont and Streicher. In this rough comparison, x is a "vector", and α is a "form". But the "evaluation" x|α does not take its value in the reference "field", but rather either in the vector space or in its dual. Thus in a sense, the games considered here or in the works of Blass, Abramsky-Jagadeesan and Lamarche are richer than the models considered in [LS].

Precise connections between sequential data structures and Kahn-Plotkin's concrete data structures are given in [CCF, appendix]. We only briefly recall the definition of concrete data structure. A concrete data structure is specified by four components:

- a set C of *cells,*
- a set V of *values,*
- a subset $E \subseteq C \times V$ of *events,* and
- a relation $\vdash$ of *enabling* between finite collections of events and cells.

The *states* of a concrete data structure are the subsets x of E that are *consistent* and *safe*, that is, such that $(c,v_1),(c,v_2) \in x$ implies $v_1=v_2$, and such that if $(c,v) \in x$, then x contains an enabling of c. Sequential data structures correspond to the *filiform* concrete data structures, in which each enabling relates at most one event with a cell. In a filiform cds, there is a hidden symmetry between an event and an enabling: (c,v) represents "v after c", and $(c_1,v_1) \vdash c$ represents "c after v_1". Strategies correspond to states: safety is built-in an sds, and the consistency is the condition of closure under binary glb's.

In [AJ2, HO], and also in [Lam2], attention is restricted to winning strategies: in a winning strategy the player has ready answers against *any* strategy of the opponent. Winning strategies provide a notion of totality that is important to get completeness results. We briefly come back to winning strategies in the last section.

We end the section with a few elementary lemmas.

Lemma *($x \lhd \alpha$)*
Let M be an sds, x be a strategy and α be a counterstrategy of M. The following properties hold:

(1) If $x \lhd \alpha$, then $(x^{\lhd}|\alpha) \lhd \alpha$.
(2) If $x \lhd \alpha$ and $x \le y$, then $y \lhd \alpha$ and $x^{\lhd}|\alpha = y^{\lhd}|\alpha$.
(3) If $x \rhd \alpha$ and $y \le x$, then $y \rhd \alpha$.

Similar implications hold with the assumptions $x \rhd \alpha$, $x \rhd \alpha$ and $\alpha \le \beta$, $x \lhd \alpha$ and $\alpha \ge \beta$, respectively.

Proof: The properties (1) and (2) follow obviously from the characterization of $x^{\lhd}\alpha$ as the unique element of $x \cap A(\alpha)$. Property (3) is a consequence of (2) by contraposition.
□

Lemma *($\lhd$-F)*
Let M be an sds, x be a strategy and q be a query of M. The following implications hold:

(1) $q \in F(x) \Rightarrow x \lhd q$[5]
(2) $q \in A(x) \Rightarrow x \rhd q$
(3) $q \in F(x)$, $y \le x$ and $y \lhd q \Rightarrow q \in F(y)$

Similar implications hold with a counterstrategy and a response of M.

Proof: If $q \in F(x)$, then $qd \in x$ for some d, hence $qd \in x \cap A(q)$, which means $x \lhd q$. If $q \in A(x)$, then $q \in q \cap A(x)$, which means $x \rhd q$. If $q \in F(x)$, $y \le x$ and $y \lhd q$, let $q_1 d_1$ be the unique element of $y \cap A(q)$. In particular, $q_1 \le q$. Suppose $q_1 < q$: then $q_1 d_1 \wedge qd = q_1$. But since $q_1 d_1$, $qd \in x$, their glb cannot be a query, by definition of a strategy: contradiction. □

Lemma *($A\wedge$)*
Let $M = (A,D,P)$ be an sds, x be a strategy and let $q \in A(x)$. Then, for any $r \in x$, $q \wedge r$ is ε or is a response, and thus, for any $qd \in P$, $x \cup \{qd\}$ is a strategy. Similarly, if α is a counterstrategy and $r \in A(\alpha)$, then, for any $q \in \alpha$, $r \wedge q$ is a query.

Proof: Let $q = r_1 a$. We claim:

$q \wedge r \le r_1$

Suppose $q \wedge r \not\le r_1$. Then $q \wedge r = q$ since $q \wedge r \le q = r_1 a$. Hence $q < r$, contradicting $q \in A(x)$: this proves the claim, which in turn implies $q \wedge r = r_1 \wedge r$. The conclusion follows, since by definition of a strategy $r_1 \wedge r$ is ε or is a response. □

[5] The converse is not true: we may have $x^{\lhd}q = q_1 d_1$ and $q_1 d_2 < q$, with $d_1 \ne d_2$.

3. Affine sequential algorithms

We now turn to morphisms between sequential data structures. We first recall Kahn-Plotkin's definition of a sequential function, which we formulate here in the framework of sequential data structures.

Definition *(Sequential function)*

Let M and M' be two sds's. A function $f:D(M)\rightarrow D(M')$ is called *sequential* if it is continuous and if for any pair $(x,\alpha') \in D^{\circ}(M)\times D^{\perp\circ}(M')$ such that $f(x)\rhd\alpha'$, but $f(z)\lhd\alpha'$ for some $z\geq x$, there exists $\alpha\in D^{\perp\circ}(M)$, called *sequentiality index* of f at (x,α'), such that $x\rhd\alpha$ and for any $y\geq x$, $f(y)\lhd\alpha'$ implies $y\lhd\alpha$. It is an easy exercise in the framework of sds's to show that we obtain an equivalent definition replacing α' by $q'\in Q'$, α by $q\in Q$, $f(x)\rhd\alpha'$ by $q'\in A(f(x))$, and $x\rhd\alpha$ by $q\in A(x)$. It is in this form that the definition was first given. O

The definition is illustrated in Figure 8. A sequentiality index represents an unavoidable computation.

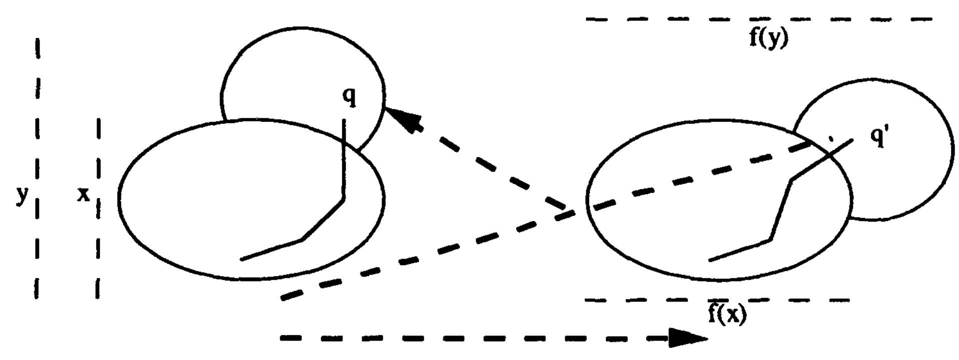


Figure 8: Sequential function

In [BeCu1, CuMon] we have shown that sequential functions do not form a cartesian closed category. The basic idea behind Berry-Curien's sequential algorithms is to assign to each pair (x,q') such as in Figure 8 a choice of a sequentiality index, as suggested by the upper fat dashed arrow of Figure 8. We can best capture this idea with examples, taken from [BeCu2, CuMon]. Consider the following function left_or, which has a unique algorithm (also called left_or) associated with it. Its input domain is the sds $Bool^2$ of Figure 2, and we take the following representation for its output domain:

Bool = ({?},{T,F},{?,?T,?F}).

The unique sequentiality index at (Ø,?) is x; at {xT}, left_or outputs T, that is, ?T ∈ left_or({xT}); at {xF}, the unique sequentiality index of left_or is y; finally, ?T ∈ left_or({xF,yT}) and ?F ∈ left_or({xF,yF}). This can be summarized as a "program":

```
left_or =
    request ? valof x
        is T output T
        if F valof y
            is T output T
            is F output F
```

In contrast, there are two *different* algorithms computing the strict version strict_or of the disjunction function (strict_or has *two* sequentiality indices at $(\varnothing,?)$):

```
left_strict_or =                    right_strict_or =
    request ? valof x                   request ? valof y
        is T valof y                        is T valof x
            is T output T                       is T output T
            is F output T                       is F output T
        is F valof y                        is F valof x
            is T output T                       is T output T
            is F output F                       is F output F
```

These examples should serve as a guide to the following definition.

Definition *(Affine exponent sds)*
Let $M \equiv (A,D,P)$ and $M' \equiv (A',D',P')$ be two sds's. We define the sds $M \multimap M' \equiv (A'',D'',P'')$ as follows:

- A'' is the disjoint union of A' and D,
- D'' is the disjoint union of D' and A,
- P'' consists of the alternating paths s starting with an $a' \in A'$ which are such that:
  - $s \upharpoonright M' \in P'$ and $(s \upharpoonright M = \varepsilon$ or $s \upharpoonright M \in P)$,
  - P'' contains no path of the form saa'.

We call *affine sequential algorithms* (or affine algorithms) from M to M' the strategies of $M \multimap M'$. The identity sequential algorithm $id_M \in D(M \multimap M)$ is defined as follows (recall the function copycat from Section 1):

- $id_M = \{copycat(r) \mid r$ is a response of $M\}$. ○

Remark: The constraints of the definition also impose that P'' contains no path of the form sd'd. Suppose it does. Then, since $sd'd \upharpoonright M \in P$, s contains a prefix s_1a such that $sd'd \upharpoonright M = (s_1a \upharpoonright M)d$. Let m be the move following s_1a in sd': m cannot belong to D since $sd'd \upharpoonright M = (s_1a \upharpoonright M)d$, and cannot belong to A' by the definition of $M \multimap M'$.

A generic strategy of $M \multimap M'$ is drawn in Figure 9, with the tags "request" and "is" for the disjoint components of A'', and the tags "output" and "valof" for the disjoint components of D''.

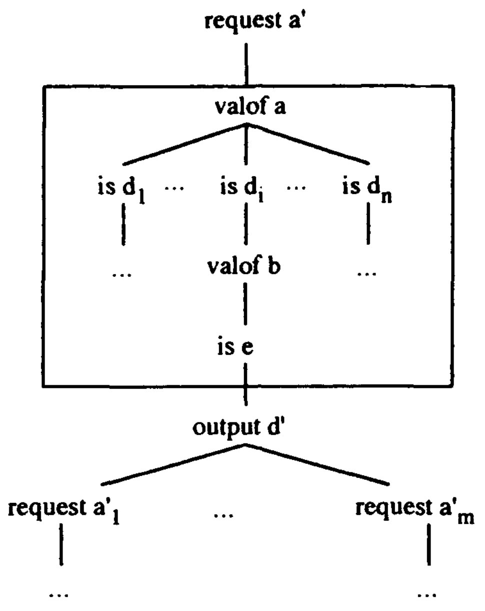
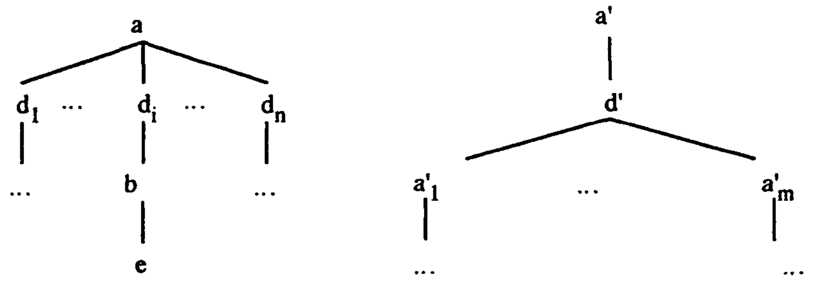


Figure 9: Generic affine algorithm

The constraint "no saa' " can be more informally formulated as follows: a "valof a" which is not an end point of the algorithm must be followed by an "is d". This constraint is the essence of sequential computation. Thinking of "valof a" as a call to a subroutine, the principal routine cannot proceed further until it receives the result "is d" from the subroutine.

We have framed a portion of the algorithm that is only concerned with the exploration of the input. If the tags are removed, this portion reads as a counterstrategy of M, and the rest of the drawing constitutes a strategy of M':



Thus an affine sequential algorithm appears as a "combination" of output strategies and of input exploration trees.

Remark: Our convention that unions are always disjoint is violated in the formula defining id_M. More appropriately, we should have used the tags: thus, say a path ad of M becomes a path "request a valof a is d output d". Later in this section we shall give a game-theoretic interpretation of id_M.

Although the programs for left_or, left_strict_or and right_strict_or have served as a motivation for Definition *Affine exponent sds,* they are *not* examples of affine sequential algorithms, but only of sequential algorithms, which we shall recall in Section 5. Take for example the "path" ?xFyTT in left_or: its projection xFyT on the input sds is not a path of that sds, but rather a sequentialization of two paths xF and yT. This is what makes the difference between affine and general sequential algorithms. An algorithm asks successive queries to its input, and proceeds only when it gets responses to these queries. An affine algorithm is required to ask these queries in a monotonous way: each new query must be an extension of the previous one. The "unit" of resource consumption is thus a sequence of queries/responses that can be arbitrarily large as long as it builds a path of the input sds. The disjunction algorithms are not affine, because they may have to ask successively the queries x and y, which are not related by the prefix ordering.

Our definition of affine exponent is the same as that given by Abramsky-Jagadeesan. It is equivalent to that given by Lamarche in [Lam1], restricted to what we call here Lamarchian sds's. According to Lamarche:

- The moves of the linear exponent are pairs (m,m') of moves m in M and m' in M' whose polarities are *not* in the combination (∘,•).

- The moves of polarity ∘ and those of polarity are as indicated by Table 1:

$\multimap$	●	○
●	●	○
○		●

Table 1: Polarities for $\multimap$

- One moves only on one side at a time: if (m,m') is a move, then its successor is a move of the form (n,m') or (m,n').

- As in our definition, it is required that the two projections of a path of $M \multimap M'$ are paths in M and M', respectively.

In Figure 10 we represent the generic algorithm of Figure 9, viewed as a Lamarchian one.

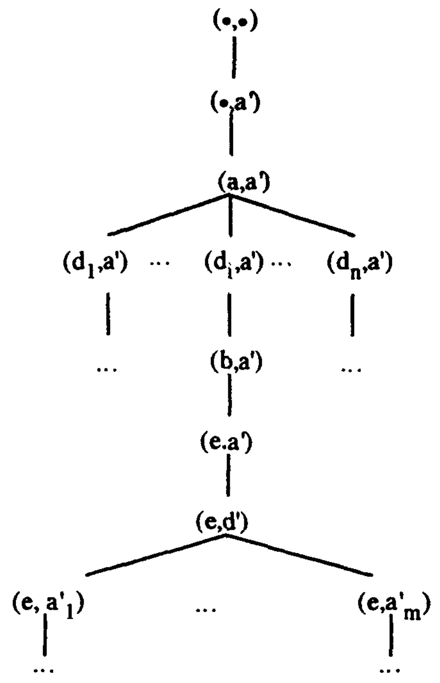


Figure 10: A generic Lamarchian algorithm

Lamarche's table of polarities elegantly captures the constraints of our definition: the first move after the root (•,•) must have the form (∘,•) or (•,∘), since only one component moves at a time. But the combination (∘,•) is forbidden; hence an algorithm starts with a "request a' ". For the same reason, a (∘,∘) move can only followed by a (•,∘) move, and this enforces the constraint that a "valof a" can only be followed by an "is d".

Table 1 is helpful in designing the tensor product of two sds's. Let us briefly anticipate Section 4. By simple logical manipulations, we get the following table of polarities for the tensor:

$\otimes$	•	∘
•	•	∘
∘	∘	

Table 2: Polarities for $\otimes$

(This table is obtained through the encoding of M$\otimes$M' as $(M\multimap M'^{\perp})^{\perp}$.). It is directly suggestive of a game-theoretic interpretation: M and M' can be thought of as two distinct boards, on which two persons, the opponent and the player, can play. The opponent has the ∘ moves on both boards, and the player has the • moves on both boards. The table indicates that only the opponent has the freedom to play his next move on the board of his choice. Indeed, an opponent's move is either a (∘,•) or a (•,∘) move; if it is a (∘,•) move, this indicates that the opponent last played on the first board and that the player has to move next on the same board. A similar analysis can be done for a (•,∘) move. In contrast, a player's move is a (•,•) move, and can be followed by either sort of opponent's move, which indicates that the opponent can play on either game.

With this interpretation in mind, we can read the identity algorithm id_M as a "copy-cat" counterstrategy, as it is called in [AJ2]. We first look at $M \multimap M$ as $(M \otimes M^{\perp})^{\perp}$. Hence we can describe id_M as a counterstrategy of $M \otimes M^{\perp}$. It is convenient to think of the player of $M \otimes M^{\perp}$ as a team of two players - one on the board M, the other on the board $M^{\perp}$ -, who play against the opponent. Following Lafont [LS] (see also [AJ2]), we call these two players Karpov and Kasparov. We also consider M and $M^{\perp}$ as two copies Left and Right of the same board, with the following distribution among the participants, as illustrated in Figure 11.

- Karpov plays black on Left,
- Kasparov plays white on Right,
- the opponent plays either white on Left or black on Right.

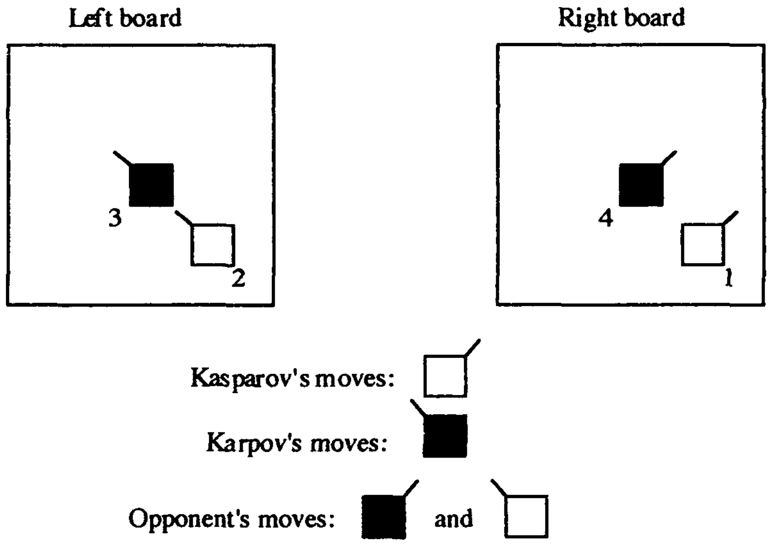


Figure 11: The identity algorithm

Table 2 forces Kasparov to move first (request a). The opponent immediately copies this move on Left (valof a), leaving to Karpov the task of finding a black move on Left as a response. If Karpov has succeeded (is d), the opponent immediately copies the move on Right (output d), and symmetrically leaves to Kasparov the task of thinking about the appropriate next move. It is clear that with this courageous strategy, the opponent is winning... The positions 1 through 4 in Figure 11 correspond to the four successive steps "request a", "valof a", "is d", and "output d".

We come back to affine sequential algorithms. We state a key technical property.

Lemma *(Injectivity)*

(1) For any affine sequential algorithm ϕ, the map $s \mapsto (s\upharpoonright M, s\upharpoonright M')$ is an order-isomorphism from ϕ to its image, ordered componentwise by the prefix ordering.

(2) If two elements s_1 and s_2 of ϕ are such that $(s_1\upharpoonright M)\wedge(s_2\upharpoonright M)$ is either ε or is a response, and if $(s_1\upharpoonright M')$ and $(s_2\upharpoonright M')$ are comparable, then s_1 and s_2 are comparable.

(3) If two elements s_1 and s_2 of ϕ are such that $(s_1\upharpoonright M')\wedge(s_2\upharpoonright M')$ is a query, and if $(s_1\upharpoonright M)$ and $(s_2\upharpoonright M)$ are comparable, then s_1 and s_2 are comparable.

Proof: We first show that the first part of the statement is implied by the second (or the third). It is obvious that $s \mapsto (s\upharpoonright M, s\upharpoonright M')$ is monotonous. Suppose that $s_1\upharpoonright M \leq s\upharpoonright M$, $s_1\upharpoonright M' \leq s\upharpoonright M'$, and $s_1 \not\leq s$. Then $s \leq s_1$ by the second part of the statement, and by monotonicity $s\upharpoonright M \leq s_1\upharpoonright M$, $s\upharpoonright M' \leq s_1\upharpoonright M'$. Hence $s_1\upharpoonright M = s\upharpoonright M$, $s_1\upharpoonright M' = s\upharpoonright M'$, and $s=s_1$ follows, since $s<s_1$ would imply either $s\upharpoonright M < s_1\upharpoonright M$ or $s\upharpoonright M' < s_1\upharpoonright M'$.

We now prove the second part of the statement. Let $t = s_1\wedge s_2$, which is ε or is a response, since ϕ is a strategy. It $t=s_1$, then $s_1 \leq s_2$. Similarly, if $t=s_2$ then $s_2 \leq s_1$. Thus we may assume for the rest of the proof that $t<s_1$ and $t<s_2$. If t has the form t_1a, then $t<s_1$ and $t<s_2$ imply that $t_1ad_1 \leq s_1$ and $t_1ad_2 \leq s_2$ for some d_1 and d_2, which must be different since $t_1a = s_1\wedge s_2$: but then $(s_1\upharpoonright M)\wedge(s_2\upharpoonright M)$ is a query, contradicting the assumption. If t is ε or has the form t_1d', then $t<s_1$ and $t<s_2$ imply that $ta'_1 < s_1$ and $ta'_2 < s_2$ for some a'_1 and a'_2, which must be different since $t = s_1\wedge s_2$: this contradicts the assumption that $(s_1\upharpoonright M')$ and $(s_2\upharpoonright M')$ are comparable.

The third part of the statement is proved similarly. □

Remark: The following observation is useful: any pair $(s\upharpoonright M, s\upharpoonright M')$ in the image of ϕ under the mapping $s \mapsto (s\upharpoonright M, s\upharpoonright M')$ is either a pair of responses or a pair of queries. It is a pair of responses if and only if s ends with a datum d'; it is a pair of queries if and only if s ends with an address a.

The definition of an affine algorithm as a strategy of $M \multimap M'$ is not denotational in character. It is clearly suited to the proof of existence of an internal homset in the category of affine algorithms, which will be carried out in the next section, but one would wish a more abstract functional description of the morphisms of our category. Fortunately, there is one, which we state after some preliminaries.

First, we call a function $f: D(M) \rightarrow D(M')$ *prime-continuous* when it is monotonous and satisfies the following condition:

- if $r' \in f(x)$, then there exists $r \in x$ such that $r' \in f(r)$.

It is easy to see that, equivalently, a prime-continuous function can be defined as a continuous function preserving lubs of pairs of compatible elements. These definitions apply also to (partial) functions g: $D^{\perp}(M') \rightarrow D^{\perp}(M)$. (By a monotonous partial function g, we mean that if $\alpha \leq \beta$ and $g(\alpha)$ is defined, then $g(\beta)$ is also defined and $g(\alpha) \leq g(\beta)$.)

The *trace* of a continuous function f: $D(M) \rightarrow D(M')$ is the relation Trace(f) $\subseteq D^{\circ}(M) \times D^{\circ}(M')$ consisting of the pairs (x,x') such that $x' \leq f(x)$ and $x' \not\leq f(y)$ for $y<x$. The trace of g: $D^{\perp}(M') \rightarrow D^{\perp}(M)$ is defined likewise. The functions that we shall consider will always be *stable*, which for, say, f: $D(M) \rightarrow D(M')$ means:

- if $(x_1,x') \in$ Trace(f), $(x_2,x') \in$ Trace(f), and $x_1 \uparrow x_2$, then $x_1=x_2$.

Equivalently, and more abstractly, stability is the preservation of glbs of pairs of compatible elements. Another equivalent definition of stability is in terms of minima:

Notation *(M(f,x,x'))*
Let M and M' be sds's, and let f be a continuous function from D(M) to D(M'). If $x \in D(M)$, $x' \in D^{\circ}(M')$, and $x' \leq f(x)$, then we denote by M(f,x,x') the minimum $y \leq x$, if it exists, such that $x' \leq f(y)$.

If f is stable, then M(f,x,x') exists: a minimal y can be found by continuity and well foundedness, and the uniqueness follows from stability. Clearly, (M(f,x,x'),x') $\in$ Trace(f). Conversely, the existence of all the M(f,x,x')'s implies the preservation of glbs of pairs of compatible elements [Be, CHL, CuMon]. Also, one proves easily that sequentiality implies stability (see [CuMon]).

We call *affine* a stable and prime-continuous (partial) function. The interest of this combination of preservation properties lies in the following lemma.

Lemma *(Trace composition)*
Let f and g be two composable affine functions. The composition of f and g is itself affine, and its trace is the relation composition of the traces of f and g.

Proof: Let, say, f: $D(M) \rightarrow D(M')$ and g: $D(M') \rightarrow D(M'')$. The first part of the statement is obvious using the characterization of prime-continuity and stability by lub and meet preservation properties. We show Trace(g $\circ$ f) $\subseteq$ Trace(g) $\circ$ Trace(f), without using stability. Let (r,r'') $\in$ Trace(g $\circ$ f). By prime-continuity, there exists r' such that $r' \leq f(r)$ and (r',r'') $\in$ Trace(g). We show (r,r') $\in$ Trace(f). If $r' \leq f(r_0)$ for some $r_0<r$, then $r'' \leq g(r') \leq g(f(r_0))$, contradicting (r,r'') $\in$ Trace(g $\circ$ f). Finally, we show Trace(g) $\circ$ Trace(f) $\subseteq$ Trace(g $\circ$ f), making use of the stability of g. Let (r,r') $\in$ Trace(f) and (r',r'') $\in$ Trace(g). Since $r' \leq f(r)$ and $r'' \leq g(r')$, we have $r'' \leq g(f(r))$. Assume $r'' \leq g(f(r_0))$ for some $r_0<r$, and let r'_0 be such that $r'_0 \leq f(r_0)$ and $(r'_0,r'') \in$ Trace(g). Then $r'_0 \uparrow r'$ implies $r'_0=r'$. But then $r'=r'_0 \leq f(r_0)$ contradicts (r,r') $\in$ Trace(f). □

We now formulate our symmetric definition of affine sequential algorithm. It relies on a notation which is similar to the notation $M(f,x,x')$.

Notation *($M(f,x,\alpha')$)*
Let M and M' be sds's, and let f be a continuous function from $D(M)$ to $D(M')$. If $x \in D(M)$, $\alpha' \in D^{\perp\circ}(M')$, and $f(x) \lhd \alpha'$, then we denote by $M(f,x,\alpha')$ the minimum $y \leq x$, if it exists, such that $f(y) \lhd \alpha'$.

This notation coincides with the notation $M(f,x,r')$ when $\alpha' = q'$ and $r' = q'd' \in f(x)$. (This is proved thanks to statement (3) of Lemma $\lhd$-F.)

Definition *(Symmetric algorithm)*
Let $M = (A,D,P)$ and $M' = (A',D',P')$ be two sds's. A symmetric algorithm from M to M' is a pair (f: $D(M) \rightarrow D(M')$, g: $D^{\perp}(M') \rightarrow D^{\perp}(M)$) of a function and a partial function that are both continuous and satisfy the following axioms:

(L) $\quad x \in D(M) \wedge \alpha' \in D^{\perp\circ}(M') \wedge f(x) \lhd \alpha' \Rightarrow x \lhd g(\alpha') \wedge (M(f,x,\alpha') = x^{\lhd}|g(\alpha'))$

(R) $\quad \alpha' \in D^{\perp}(M') \wedge x \in D^{\circ}(M) \wedge x \rhd g(\alpha') \Rightarrow f(x) \rhd \alpha' \wedge (M(g,\alpha',x) = f(x)|^{\rhd}\alpha')$

We set as a convention that for any x, and any α' such that $g(\alpha')$ is undefined:

$x \lhd g(\alpha')$ and $x^{\lhd}|g(\alpha') = \varnothing$

To see that this convention is natural, recall that in a Lamarchian sds, every path starts with the initial • move. When $g(\alpha')$ is undefined, this initial move is the final move of the interplay, and thus x wins. With this convention, the conclusion of (L) is simply $M(f,x,\alpha') = \varnothing$ when $g(\alpha')$ is undefined. In contrast, when we write $x \rhd g(\alpha')$ as in (R), we assume that $g(\alpha')$ is defined.

The collection of symmetric algorithms is ordered componentwise by the pointwise ordering:

$(f_1,g_1) \leq (f_2,g_2)$ iff $\forall x\ f_1(x) \leq f_2(x)$ and $\forall \alpha\ g_1(\alpha) \leq g_2(\alpha)$ (if $g_1(\alpha)$ is defined) ○

These axioms enable us, knowing f and g, to reconstruct the traces of f and g. Definition *Symmetric algorithm* is strikingly compact with respect to the definition of abstract algorithm found in [BeCu1, CuMon] (see also BuEhr]) and reformulated in Section 5. It implies that the two functions f and g are prime-continuous and sequential. Moreover, g allows to compute the sequentiality indices of f, and f allows to compute the sequentiality indices of g.

Proposition *(Symmetry and sequentiality)*
Let f and g be as in the previous definition. Then f and g are prime-continuous and sequential, and they satisfy the following two axioms:

(LS) If $x \in D(M)$, $\alpha' \in D^{\perp\circ}(M')$, $f(x) \triangleright \alpha'$ and if $f(y) \triangleleft \alpha'$ for some $y > x$, then $x \triangleright g(\alpha')$ and $x |^{\triangleright} g(\alpha')$ is a sequentiality index of f at (x,α').

(RS) If $\alpha' \in D^{\perp}(M')$, $x \in D^{\circ}(M)$, $x \triangleleft g(\alpha')$ and if $x \triangleright g(\beta')$ for some $\beta' > \alpha'$, then $f(x) \triangleleft \alpha'$ and $f(x)^{\triangleleft}|\alpha'$ is a sequentiality index of g at (α',x).

Proof: The prime-continuity of f follows from Axiom (L), since this axiom implies that any element $M(f,x,\alpha')$ is a prime element. Precisely, suppose $q'd' \in f(x)$. Then $f(x) \triangleleft q'$. By (L), $x \triangleleft g(q')$ and $f(r) \triangleleft q'$, where $r = x^{\triangleleft}|g(q')$. Let $q'_1 d'_1 = f(r)^{\triangleleft}|q'$, and suppose $q'_1 < q'$. On one hand $q'_1 d'_1 \in A(q')$ implies $q'_1 d'_1 \not\leq q'$. On the other hand, since $q'_1 d'_1 \in f(r)$ and $r \leq x$, we have $q'_1 d'_1 \in f(x)$, and $q'd'$, $q'_1 d'_1 \in f(x)$ imply $q'_1 d'_1 \leq q'$: contradiction. Hence $q'_1 = q'$, and moreover $d'_1 = d'$ since $q'd'$, $q'd'_1 \in f(x)$. We have proved $f(r)^{\triangleleft}|q' = q'd'$, and a fortiori $q'd' \in f(r)$.

We now prove that Axiom (L) implies property (LS) (which itself implies the sequentiality of f). Suppose $x \in D(M)$, $\alpha' \in D^{\perp\circ}(M')$, $f(x) \triangleright \alpha'$, and $f(y) \triangleleft \alpha'$ for some $y > x$. Let $r_1 = y^{\triangleleft}|g(\alpha')$. By (L), we have $f(r_1) \triangleleft \alpha'$, which implies $r_1 \notin x$ since $f(x) \triangleright \alpha'$. Let r be the largest response prefix of r_1 contained in x, and let ra be such that $ra < r_1$. We claim:

$$x|g(\alpha') = ra$$

From $r_1 \in A(g(\alpha'))$ and $ra < r_1$, we get $ra \in g(\alpha')$. We have $r \in x$ by construction, thus ra is enabled in x. If ra is filled in x, it must be filled with the same datum d in x and r_1, contradicting the maximality of r. Hence $ra \in g(\alpha') \cap A(x)$, which proves the claim. The proof of (LS) is completed by observing that $ra < r_1$, $r_1 \leq y$ imply $ra \in F(y)$. □

Thus the two components f and g of a symmetric algorithm (f,g) are sequential and prime-continuous. A fortiori, they are stable, hence, affine, which entails that they are actually strongly sequential. Strong sequentiality means that at every (x,q'), there is at most one sequentiality index) (cf. [CuMon, Exercise 2.4.11.3 (second edition)])[6]. One can show that any affine function f is the first component of some affine algorithm (f,g) (a similar theorem is shown in [CuMon, Proposition 2.5.6]).

A familiar feature of stability is not apparent in Definition *Symmetric algorithm:* the order is not defined as Berry's stable ordering. But the stable ordering is a derived property. We recall a definition of the stable ordering. Let $f_1, f_2 \in D(M) \to D(M')$. We write:

$$f_1 \leq_s f_2 \text{ when } \forall x\ \forall y \leq x\ \ f_1(y) = f_2(y) \wedge f_1(x)$$

[6]Conversely, there are strongly sequential functions that are not affine: left_or is an example.

Proposition *(Stable ordering)*
If $(f_1,g_1) \leq (f_2,g_2)$ (cf. Definition *Symmetric algorithm),* then $f_1 \leq_s f_2$ and $g_1 \leq_s g_2$.

Proof: We only prove $f_1 \leq_s f_2$, the proof being symmetrical for g_1 and g_2. Consider x and y≤x, and assume $q'd' \in f_2(y) \wedge f_1(x)$. Suppose $q'd' \notin f_1(y)$. We can take q'd' minimal with this property, thus we can assume $q' \in A(f_1(y))$, which implies $f_1(y) \triangleright q'$ by Lemma ◁-F. Since in an sds domain the glbs of compatible elements are set intersections, we have $q'd' \in f_2(y)$ and $q'd' \in f_1(x)$. By (LS) applied to (f_1,g_1), we have: $y \triangleright g_1(q')$, and $r_1a = y|^{\triangleright} g_1(q')$ is a sequentiality index of f_1 at (y,q'). Since $g_1 \leq_e g_2$ (where $\leq_e$ denotes the pointwise ordering), we have $r_1a \in g_2(q')$. Hence by (R) applied to (f_2,g_2) we get $f_2(r_1) \triangleright q'$. By (LS) applied to (f_2,g_2), $r_1 \triangleright g_2(q')$ and $r_1|^{\triangleright} g_2(q')$ is a sequentiality index of f_2 at (r_1,q'). Since $r_1a \in g_2(q')$, we have $r_1|^{\triangleright} g_2(q') = r_1a$. Now:

- by sequentiality $f_2(y) \triangleleft q'$ implies $y \triangleleft r_1a$;
- by definition of r_1a, $r_1a \in A(y)$, hence $y \triangleright r_1a$ by Lemma ◁-F.

This contradiction proves $q'd' \in f_1(y)$, and $f_1 \leq_s f_2$. □

We have to show the equivalence between the concrete and the denotational presentations of our morphisms.

Definition *(From concrete to symmetric)*
Let M and M' be two sds's. Given an affine sequential algorithm $\phi \in D(M \multimap M')$, we define a symmetric algorithm (f,g) as follows:

- $f(x) = \{r' \mid r'=s{\restriction}M'$ and $s{\restriction}M \in x$, for some $s \in \phi\}$,
- $g(\alpha') = \{q \mid q=s{\restriction}M$ and $s{\restriction}M' \in \alpha'$ for some $s \in \phi\}$.

By convention, if for some α' the right-hand side of the definition of g is empty, we interpret this definitional equality as saying that g(α') is undefined. O

The traces of f and g have an easy characterization, as the following lemma shows.

Lemma *(Trace)*
Let (f,g) be constructed from an affine sequential algorithm φ as above. Then:

- $\text{Trace}(f) = \{(r,r') \mid r=s{\restriction}M$ and $r'=s{\restriction}M'$, for some $s \in \phi\}$,
- $\text{Trace}(g) = \{(q',q) \mid q'=s{\restriction}M'$ and $q=s{\restriction}M$, for some $s \in \phi\}$.

Proof: If $r=s{\restriction}M$ and $q'd'=r'=s{\restriction}M'$, for some $s \in \phi$, then a fortiori $s{\restriction}M \leq r$, thus $r' \in f(r)$. Suppose that $r' \in f(r_1)$ for some $r_1<r$. Let $s_1 \in \phi$ be such that $r'=s_1{\restriction}M'$ and $s_1{\restriction}M \leq r_1$. Thus $(s_1{\restriction}M,s_1{\restriction}M') < (s{\restriction}M,s{\restriction}M')$, which by Lemma *Injectivity* implies $s_1<s$. But by the definition of $M \multimap M'$, $r'=s{\restriction}M'$ implies that s ends with d', and hence $s_1{\restriction}M' < s{\restriction}M'$, contradicting $r'=s_1{\restriction}M'$. Thus $(r,r') \in \text{Trace}(f)$. Reciprocally, if $(r,r') \in$

Trace(f), then let s ∈ a be such that r'=s↾M' and s↾M ≤ r. Then, by minimality of r, we must have s↾M = r. The proof of the second equality is similar. □

The definition of the function f computed by ϕ is so compact that it may hide the underlying operational semantics. The application of ϕ to a strategy x of M involves an interplay between ϕ and x that is very similar to the situation described in Proposition *Play*. We have already suggested pictorially that an affine sequential algorithm "contains" input counter-strategies. Figure 12, taken from [CuMon (second edition)], illustrates the interplay between ϕ and x. In this figure, the bold oriented path represents the flow of control. The relation with Figures 4, 5, and 7 is as follows: the counterstrategy "valof a ..." is matched against x, resulting in the path ad_ibe.

This is reminiscent of Girard's geometry of interaction. We refer to [AJ1, AJ2, Lam2] for some more precise connections.

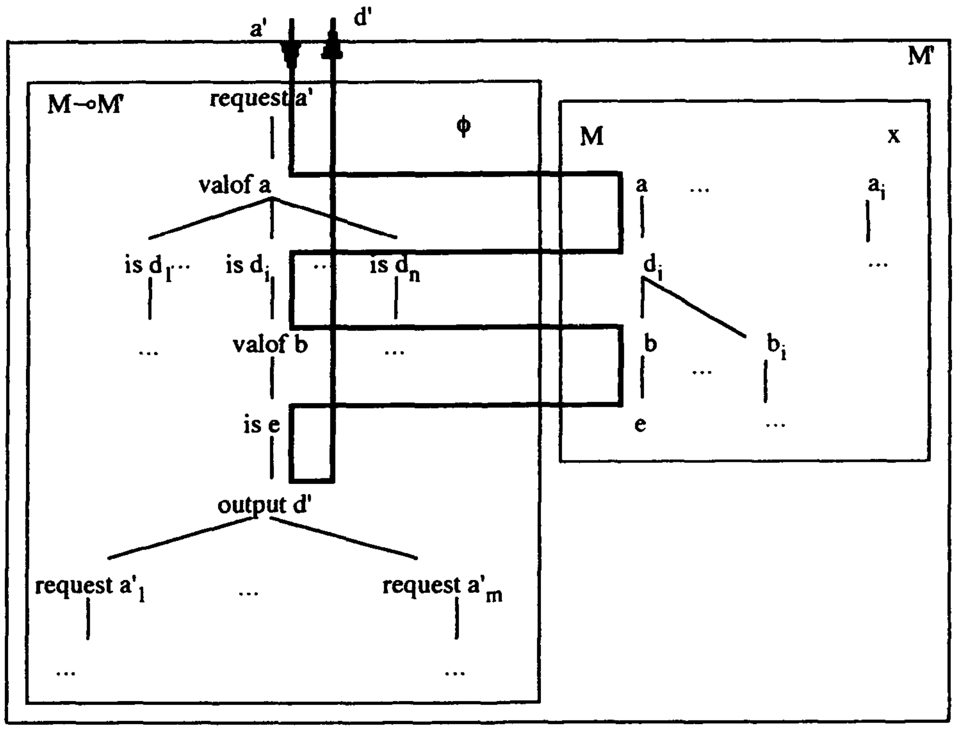


Figure 12: Application

Conversely, given a symmetric algorithm (f,g), we construct a strategy ϕ of $M \multimap M'$ inductively, as suggested by Figure 13. The construction of a path of ϕ is carried out as an experiment. The experimenter is free to give addresses of $M \multimap M'$, and the specification (f,g) provides corresponding data. At the beginning, the experimenter gives an a'_1. If a'_1 is filled in $f(\emptyset)$, the next datum on the experimentation path is an "output" instruction. If $g(a'_1)$ contains some path a of length 1, the next datum is a "valof" instruction. The axioms of symmetric algorithms guarantee that these two situations are exclusive. Indeed, if a'_1 is filled in $f(\emptyset)$, then $f(\emptyset) \lhd a'_1$, which by (L) implies $\emptyset \lhd g(a'_1)$, and if $g(a'_1)$ contains a path of length 1, then $\emptyset \rhd g(a'_1)$. It is easy to see that this argument applies all along the path constructed in Figure 13. More precisely:

- An "output" instruction is indicated by f until an address a' is placed by the experimenter for which g indicates a "valof" instruction. Then the next address placed by the experimenter must be a d_1 (cf. the definition of affine sequential algorithm).
- The function g keeps the hand on the shown path until an address d is placed by the experimenter for which f indicates an "output" instruction.

The argument to which the function f or g is applied at each stage is the projection of the path constructed so far on the appropriate sds (M for f, M' for g). It is easily seen by construction that, collecting together all these experimentation paths, we obtain a strategy of $M \multimap M'$.

The following definition formalizes the construction.

Definition *(From symmetric to concrete)*
Let M and M' be two sds's. Given a symmetric algorithm (f,g) from M to M', we construct an affine algorithm $\phi \in D(M \multimap M')$ as follows. We build the paths s of ϕ by induction on the length of s:

- if $s \in \phi$, if $s{\upharpoonright}M$ and $s{\upharpoonright}M'$ are responses, and if $q' = (s{\upharpoonright}M')a'$ for some a', then
  $sa'a \in \phi$ if $(s{\upharpoonright}M)a \in g(q')$
  $sa'd' \in \phi$ if $q'd' \in f(s{\upharpoonright}M)$

- if $s \in \phi$, if $s{\upharpoonright}M$ and $s{\upharpoonright}M'$ are queries, and if $r = (s{\upharpoonright}M)d$ for some d, then
  $sda \in \phi$ if $ra \in g(q')$
  $sdd' \in \phi$ if $q'd' \in f(r)$ O

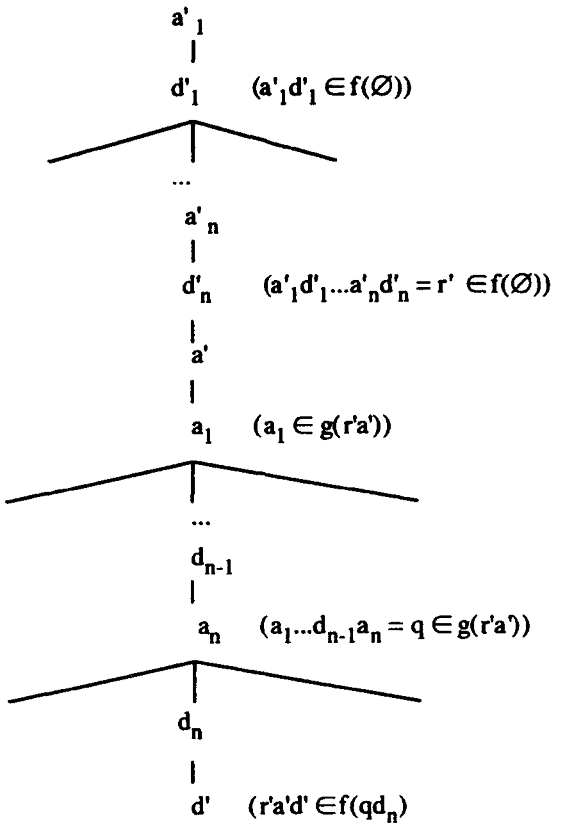

a'_1
d'_1 $(a'_1d'_1 \in f(\varnothing))$
...
a'_n
d'_n $(a'_1d'_1 \ldots a'_nd'_n = r' \in f(\varnothing))$
a'
a_1 $(a_1 \in g(r'a'))$
...
d_{n-1}
a_n $(a_1 \ldots d_{n-1}a_n = q \in g(r'a'))$
d_n
d' $(r'a'd' \in f(qd_n))$

Figure 13: From symmetric to concrete

This construction defines an inverse to the map $s \mapsto (s\restriction M, s\restriction M')$ considered in Lemma *Injectivity*.

The above transformations are inverse order-isomorphisms.

Theorem *(Symmetric/affine)*
Given M and M', the above transformations define order-isomorphisms between $D(M \multimap M')$, ordered by inclusion, and the set of symmetric algorithms from M to M', ordered pointwise componentwise.

Proof: We limit ourselves to check that (f,g) constructed as in Definition *From concrete to symmetric* from an affine algorithm ϕ satisfies (L). If $x \in D(M)$, $\alpha' \in D^{\perp\circ}(M')$ and $f(x) \lhd \alpha'$, let $q'd' \equiv f(x)^{\lhd}|\alpha'$, and let $s \in \phi$ be such that $q'd' = s\restriction M'$ and $s\restriction M \in x$. Then s ends with d'. We claim:

(i) $s\restriction M = M(f,x,\alpha')$
(ii) $s\restriction M = x^{\lhd}|g(\alpha')$

We first prove (ii). Since $s{\restriction}M \in x$, we are left to show $s{\restriction}M \in A(g(\alpha'))$. Since $q'd' = f(x)^{\triangleleft}|\alpha'$, we have $q'd' \in A(\alpha')$, hence $q' \in \alpha'$. We first show that $s{\restriction}M$ is enabled in $g(\alpha')$. Let $s{\restriction}M = qd$, and let s_1 be the least prefix of s such that $s_1{\restriction}M = q$. We claim:

$s_1{\restriction}M' \in \alpha'$

By the definition of s_1, and since s ends with d', s_1 is a strict prefix of s and $s_1{\restriction}M' < s{\restriction}M'$. Hence $s_1{\restriction}M' \leq q'$, which implies the claim. Since $s_1{\restriction}M = q$, the claim implies $q \in g(\alpha')$ by definition of g, and that $s{\restriction}M$ is enabled in $g(\alpha')$. Suppose now that $s{\restriction}M$ is filled in $g(\alpha')$. Then there exist a and $s_2 \in \phi$ such that $(s{\restriction}M)a = s_2{\restriction}M$ and $s_2{\restriction}M' \in \alpha'$. By Lemma $A\wedge$, we can apply Lemma *Injectivity* (part 3) to s and s_2. Thus s and s_2 are comparable. But since $(s{\restriction}M)a = s_2{\restriction}M$ we cannot have $s_2 \leq s$, and since $s_2{\restriction}M' \in \alpha'$ and $s{\restriction}M' \in A(\alpha')$ we cannot have $s \leq s_2$: contradiction. This completes the proof of (ii).

We now prove (i). By definition of f, we have $s{\restriction}M' \in f(s{\restriction}M)$, hence $f(s{\restriction}M) \triangleleft \alpha'$. Suppose now that $y \leq x$ and $f(y) \triangleleft \alpha'$. By Lemma $x \triangleleft \alpha$, $f(y)^{\triangleleft}|\alpha' = f(x)^{\triangleleft}|\alpha'$, thus $q'd' \in f(y)$. Let $s_3 \in \phi$ be such that $q'd' = s_3{\restriction}M'$ and $s_3{\restriction}M \in y$. By Lemma *Injectivity* (part 2), s and s_3 are comparable. Since s ends with d' and since $s_3{\restriction}M' = s{\restriction}M'$, s_3 cannot be a proper prefix of s. Thus $s \leq s_3$, and this entails $s{\restriction}M \in y$ since $s{\restriction}M \leq s_3{\restriction}M$ and $s_3{\restriction}M \in y$. This completes the proof of (i). □

4. A symmetric monoidal closed category

We now turn sequential data structures and symmetric algorithms into a category by adding a notion of composition. The formulation of the morphisms as symmetric algorithms allows us to define composition in a straightforward way.

Definition *(Denotational composition)*
Let M, M' and M'' be sds's, and let (f,g) and (f',g') be symmetric algorithms from M to M' and from M' to M''. We define their composition (f'',g'') from M to M'' as follows:

- $f'' = f' \circ f$ and $g'' = g \circ g'$.

Proposition *(Denotational composition is well-defined)*
The pair (f'',g'') in Definition *Denotational composition* indeed defines a symmetric algorithm.

Proof: We only check Axiom (L). Suppose $f'(f(x)) \triangleleft \alpha''$. By (L) applied to (f',g'), we have $f(x) \triangleleft g'(\alpha'')$ and $M(f',f(x),\alpha'') = f(x)^{\triangleleft}|g'(\alpha'')$. Since $f(x) \triangleleft g'(\alpha'')$, by (L) applied to (f,g), we get $x \triangleleft g(g'(\alpha''))$ and $M(f,x,g'(\alpha'')) = x^{\triangleleft}|g(g'(\alpha''))$. We have to prove $M(f' \circ f,x,\alpha'') = x^{\triangleleft}|g(g'(\alpha''))$. We set $r = x^{\triangleleft}|g(g'(\alpha''))$. Since $M(f,x,g'(\alpha'')) = r$, we have $f(r) \triangleleft g'(\alpha'')$. We claim:

$f'(f(r)) \triangleleft \alpha''$

Suppose the contrary, that is, $f'(f(r)) \triangleright \alpha''$. Then, by (LS) applied to (f',g') at $(f(r),\alpha'')$, we have $f(r) \triangleright g'(\alpha'')$, which contradicts our previous deduction that

$f(r) \lhd g'(\alpha'')$. Hence the claim holds. We are left to prove that any $y \leq x$ such that $f'(f(y)) \lhd \alpha''$ is such that $y \geq r$. Since $M(f,x,g'(\alpha'')) = r$, this second claim can be rephrased as:

$f(y) \lhd g'(\alpha'')$

We set $r' = f(x) \lhd g'(\alpha'')$. Since $f(y) \leq f(x)$ and since $M(f',f(x),\alpha'') = r'$, we have $r' \leq f(y)$. But $r' \lhd g'(\alpha'')$ by definition of r' and by property (1) of Lemma $x \lhd \alpha$, and the second claim follows by property (2) of Lemma $x \lhd \alpha$. □

Definition *(AFFALGO)*
The category $\mathbf{AFFALGO_{sds}}$ (**AFFALGO** for short) is defined as follows. Its objects are the sequential data stuctures and its morphisms are the affine sequential algorithms. If $a \in D(A \multimap A')$ and $a' \in D(A' \multimap A'')$, if (f,g) and (f'g') are the symmetric algorithms associated with ϕ and ϕ', respectively, then $\phi' \circ \phi$ is the affine sequential algorithm ϕ'' associated with $(f' \circ f, g \circ g')$. The identity morphisms are those associated with the pairs (id,id). O

Less formally, we shall indifferently look at morphisms as affine sequential algorithms or as symmetric algorithms.

A closely related way of looking at the composition of affine algorithms, which is adopted in [Lam1], is to define the composition of algorithms as a relation composition.

Proposition *(Relation composition)*
Let (f,g) be a symmetric algorithm. We define Trace(f,g) as follows:

- $\text{Trace}(f,g) = \text{Trace}(f) \cup \{(q,q') \mid (q',q) \in \text{Trace}(g)\}$.

The following holds for any (f,g) and (f',g') as in Definition *Denotational composition:*

- $\text{Trace}((f',g') \circ (f,g)) = \text{Trace}(f',g') \circ \text{Trace}(f,g)$

Proof: Immediate consequence of Lemma *Trace composition.* □

Abramsky and Jagadeesan give a different definition of composition, which is operational in flavour. This definition requires a notation.

Notation
Let $M = (A,D,P)$, $M' = (A',D',P')$ and $M'' = (A'',D'',P'')$ be three sds's. We let $\mathcal{L}(M,M',M'')$ denote the set of words in $((A \cup D) \cup (A' \cup D') \cup (A'' \cup D''))^*$ such that two consecutive symbols are not such that one is in $A \cup D$ and the other is in $A'' \cup D''$.

Proposition *(Hiding)*
Let $\phi \in D(M \multimap M')$, $\phi' \in D(M' \multimap M'')$. Then:

$$\phi' \circ \phi = \{s \restriction M \cup M'' \mid s \in \mathcal{L}(M,M',M''),\ s \restriction M \cup M' \in \phi \text{ and } s \restriction M' \cup M'' \in \phi'\}$$

Proof : We refer to [AJ2] for a proof that the right-hand side defines a strategy of $M \multimap M''$. Then it is enough to check:

$$\{(s \restriction M, s \restriction M'') \mid s \in \mathcal{L}(M,M',M''),\ s \restriction M \cup M' \in \phi \text{ and } s \restriction M' \cup M'' \in \phi'\} =$$
$$\{(p,p'') \mid p = s_1 \restriction M,\ s_1 \restriction M' = s_2 \restriction M' \text{ and } p'' = s_2 \restriction M'' \text{ for some } s_1 \in \phi,\ s_2 \in \phi'\}$$

Obviously, the left-hand side is included in the right-hand side, taking $s_1 = s \restriction M \cup M'$ and $s_2 = s \restriction M' \cup M''$. For the other direction we construct s from s_1 and s_2 by replacing every a'd' in s_2 by the corresponding portion $a'a_1d_1 \ldots a_nd_nd'$ of s_1. It is clear by construction that $s \in \mathcal{L}(M,M',M'')$. □

This alternative definition of composition is convenient to establish the symmetric monoidal structure of the category **AFFALGO**. It is closely related to the operational semantics of composition in the language CDS0 [CuMon, Definition 3.5.5].

The definition of tensor product is "dictated" by the equation $A \otimes B = (A \multimap B^{\perp})^{\perp}$, as suggested at the end of Section 3.

Definition *(Tensor product)*
Let M = (A,D,P) and M' = (A',D',P') be two sds's. We define the sds $M \otimes M'$ = (A'',D'',P'') as follows:

- A'' is the disjoint union of A and A',
- D'' is the disjoint union of D and D',
- P'' consists of the alternating non-ε paths which are such that:
  - $(s \restriction M = \varepsilon$ or $s \restriction M \in P)$ and $(s \restriction M' = \varepsilon$ or $s \restriction M' \in P')$
  - P'' contains no path of the form sad'. O

As for Definition *Affine exponent sds*, the second constraint implies that P'' contains no path of the form sa'd.

In order to define a symmetric monoidal structure, we need to turn ⊗ into a functor. We follow [AJ2].

Definition *(Tensor product continued)*
Let M_1, M_2, M'_1 and M'_2 be four sds's, and let $\phi_1 \in D(M_1 \multimap M'_1)$ and $\phi_2 \in D(M_2 \multimap M'_2)$. We define $\phi_1 \otimes \phi_2 \in D((M_1 \otimes M_2) \multimap (M'_1 \otimes M'_2))$ as follows. It consists of the paths of $M_1 \otimes M_2 \multimap M'_1 \otimes M'_2$ whose projections on $M_1 \cup M'_1$ and on $M_2 \cup M'_2$ are in a_1 and in a_2, respectively. O

Proposition *(Tensor product functor)*
The above definitions indeed define a functor which, together with the empty sds $(\emptyset,\emptyset,\emptyset)$ as unit, makes **AFFALGO** a symmetric monoidal category.

Proof: We check the preservation of composition. Let $\phi_1 \in D(M_1 \multimap M'_1)$, $\phi_2 \in D(M_2 \multimap M'_2)$, $\phi'_1 \in D(M'_1 \multimap M''_1)$ and $\phi'_2 \in D(M'_2 \multimap M''_2)$. An element of $(\phi'_1 \otimes \phi'_2) \circ (\phi_1 \otimes \phi_2)$ has the form $s\upharpoonright M_1 \cup M_2 \cup M''_1 \cup M''_2$, where

$$s\upharpoonright M_1 \cup M_2 \cup M'_1 \cup M'_2 \in \phi_1 \otimes \phi_2 \text{ and } s\upharpoonright M'_1 \cup M'_2 \cup M''_1 \cup M''_2 \in \phi'_1 \otimes \phi'_2$$

which is the same as

$$s\upharpoonright M_1 \cup M'_1 \in \phi_1,\ s\upharpoonright M_2 \cup M'_2 \in \phi_2,\ s\upharpoonright M'_1 \cup M''_1 \in \phi'_1 \text{ and } s\upharpoonright M'_2 \cup M''_2 \in \phi'_2$$

Hence

$$s\upharpoonright M_1 \cup M''_1 \in \phi'_1 \circ \phi_1 \text{ and } s\upharpoonright M_2 \cup M''_2 \in \phi'_2 \circ \phi_2$$

and thus $s\upharpoonright M_1 \cup M_2 \cup M''_1 \cup M''_2 \in (\phi'_1 \circ \phi_1) \otimes (\phi'_2 \circ \phi_2)$.

The symmetric monoidal structure is obvious and strict, with the convention that disjoint unions are ordinary unions. A more standard treatment (as adopted in [CCF]) consists in building such unions with the help of tags 1 and 2 for the left and right components. In this case, coherent isomorphisms arise: for example $(x,1)$ in $X \cup (Y \cup Z)$ corresponds to $((x,1),1)$ in $(X \cup Y) \cup Z$. □

Proposition *(Monoidal closed)*
The category **AFFALGO** is symmetric monoidal closed.

Proof: With our convention about disjoint unions, $D((M \otimes M') \multimap M'')$ and $D(M \multimap (M' \multimap M''))$ coincide. Our convention stands in the way to give a rigorous justification of the naturality condition. Loosely, given $\phi \in D(M_1 \multimap M)$, in order to turn a path s whose projection on $M_1 \cup (M' \multimap M'')$ is in a composition $M_1 \to M \to (M' \multimap M'')$ into a path whose projection on $(M_1 \otimes M') \cup M''$ is in the corresponding composition $(M_1 \otimes M') \to (M \otimes M') \to M''$, we replace every M' portion a'd' of s by a'a'd'd' (cf. the description of the copy-cat strategy). □

The category **AFFALGO** is also cartesian. It is easily checked that the empty sds $(\emptyset,\emptyset,\emptyset)$ is a terminal object, and that the following data yield binary products.

Definition *(Product)*
Let $M = (A,D,P)$ and $M' = (A',D',P')$ be two sds's. We define the sds $M \times M' = (A'',D'',P'')$ as follows:

- A'' is the disjoint union of A and A',
- D'' is the disjoint union of D and D',
- P'' is the disjoint union of P and P'.

It is easily seen that D(M×M') is the set-theoretical product of D(M) and D(M'), and that $D^{\perp}$(M×M') is the disjoint union of $D^{\perp}$(M) and $D^{\perp}$(M'). The projections and pairing functions are as follows:

- (Fst,Inl), where Fst is the set-theoretical first projection and Inl is the set-theoretical injection from $D^{\perp}$(M) into $D^{\perp}$(M×M'),

- (Snd,Inr) (similarly),

- if (f,g) ∈ D(M⊸M') and (f',g') ∈ D(M⊸M''), then <(f,g),(f',g')> is defined as (<f,f'>,[g,g']), where < , > and [,] denote the set-theoretical pairing and copairing. ○

In **AFFALGO**, the empty sds (∅,∅,∅) is both the unit of the tensor and a terminal object. It is this property which makes **AFFALGO** a model of affine logic. Indeed, the equations

tensor unit ≡ 1 = ⊤ ≡ terminal object

allow us to construct projections from the tensor product to its components, as follows:

$A \otimes B \rightarrow A \otimes \top \rightarrow A$

Moreover, the assumption that the terminal object is a *multiplicative unit corresponds to* the following proof transformations:

- naturality: for example, the proof (⊢Γ,A and ⊢Δ,$A^{\perp}$ implies ⊢Γ,Δ implies ⊢Γ,Δ,B) by cut and weakening is equivalent to the proof obtained by first weakening ⊢Γ,A into ⊢Γ,A,B, and then applying cut;

- the logical inference rule (⊢Γ implies ⊢Γ,⊥) for ⊥ (the negation of 1) is an instance of weakening;

- if Π is a proof of ⊢Γ,0 (O is the negation of ⊤), then Π is equivalent to the proof obtained by first cutting Π with ⊢⊤,⊥ (an instance of the axiom ⊢⊤,Γ for ⊤), then cutting with ⊢1 (the axiom for 1), and finally weakening (the successive conclusions are ⊢Γ,⊥, ⊢Γ, and ⊢Γ,0).

5. Sequential algorithms

In order to obtain a model of λ-calculus, we must construct a comonad accounting for the possible "duplication" of arguments. We have already suggested a meaning for the "unit of consumption" of inputs considered as resources. It appears most convenient in our setting to define this comonad via an adjunction. We construct a left adjoint to the inclusion functor of the category **AFFALGO** into the category **ALGO** of Berry-Curien's sequential algorithms [BeCu1, CuMon]. To this aim, we give yet another equivalent characterization of the morphisms of **AFFALGO**.

Definition *(Affine abstract algorithms)*
Let $M \equiv (A,D,P)$ and $M' \equiv (A',D',P')$ be two sds's. Recall that $D^{\circ}(M)$ denotes the set of finite strategies of M. An affine abstract algorithm χ from M to M' is a partial function from $D^{\circ}(M) \times Q'$ to $Q \cup R'$ which satisfies the following axioms:

(A1) $\chi(x,q') = q \Rightarrow q \in A(x)$
$\chi(x,q') = r' \Rightarrow r' = q'd'$ for some d'
(A2) $\chi(x,q') = q,\ q' \leq q'_1,\ x \leq y$ and $q \notin F(y) \Rightarrow \chi(y,q'_1) = q$
$\chi(x,q') = r'$ and $x \leq y \Rightarrow \chi(y,q') = r'$
(A3) $\chi(x,q')$ defined, $y \leq x$ and $q'_1 \leq q' \Rightarrow \chi(y,q'_1)$ defined

(AFF) $\chi(x,q')$ defined $\Rightarrow \chi(x,q') = \chi(r,q')$ for some $r \in x$

The composition of two affine abstract algorithms is defined as follows. Let χ be as above, and let χ' be an affine abstract algorithm from M' to an sds M''. Then the composition χ'' of χ and χ' is defined by:

- $\chi''(x,q'') = q''d''$ if $\chi'((\chi.x),q'') = q''d''$
- $\chi''(x,q'') = q$ if $\chi'((\chi.x),q'') = q'$ and $\chi(x,q') = q$

where $\chi.x \equiv \{q'd' \mid \chi(x,q') = q'd'\}$. O

We leave to the reader the abstract definition of the identity algorithm. The next proposition states that affine abstract algorithms are the same objects as symmetric algorithms.

Proposition *(Symmetric/abstract)*
There are order-isomorphisms between the sets of affine abstract algorithms and of symmetric algorithms, that preserve the composition of algorithms.

Proof: We construct the inverse mappings by going from symmetric algorithms to affine abstract algorithms , and from affine abstract algorithms to affine algorithms. To close the circle, we use the transformation of affine algorithms into symmetric algorithms justified in Theorem *(Symmetric/affine)*. We call 1, 2, and 3 these transformations:

$$\begin{tikzcd}
 & \text{symmetric } (f,g) \arrow[dr, "1"] & \\
\text{strategy } \phi \arrow[ur, "3"] & & \text{abstract } \chi \arrow[ll, "2"]
\end{tikzcd}$$

Transformation 1: Let (f,g) be a symmetric algorithm. We build a function χ as follows:

- $\chi(x,q') = q'd'$ iff $q'd' \in f(x)$,
- $\chi(x,q') = x|^{\triangleright}g(q')$ iff $x \triangleright g(q')$.

We check the axioms of Definition *Affine abstract algorithms*.

(A1): If $\chi(x,q')=q$, then by definition $x \triangleright g(q')$ and $q = x|^{\triangleright}g(q') \in A(x)$. The second part of (A1) is built-in in the definition of χ.

(A2): If $\chi(x,q') = q$, $q' \leq q'_1$, $x \leq y$ and $q \notin F(y)$, then $x|g(q') = y|g(q'_1)$, hence $\chi(y,q'_1) = \chi(x,q')$. The second part of (A2) is obvious by the monotonicity of f.

(A3): It is enough to prove separately that if $\chi(x,q')$ is defined and $q'_1d'_1 < q'$, then $\chi(x,q'_1)$ is defined, and that if $\chi(x,q')$ is defined and $y<x$, then $\chi(y,q')$ is defined. We thus concentrate first on $q'_1d'_1 < q'$. If $q'd' \in f(x)$, then a fortiori $q'_1d'_1 \in f(x)$, hence $\chi(x,q'_1)$ is defined. If $x \triangleright g(q')$, let $q'_2 = M(g,q',x)$. In particular, $x \triangleright g(q'_2)$. We distinguish two cases:

- $q'_1 \geq q'_2$: Then $x \triangleright g(q'_1)$ and hence $\chi(x,q'_1) = \chi(x,q'_2) = \chi(x,q')$ is defined.
- $q'_1 < q'_2$: By (R) we have $q'_2 = f(x)|^{\triangleright}q'$: this entails $q'_1d'_1 \in f(x)$ and $\chi(x,q'_1) = q'_1d'_1$.

In either case, $\chi(x,q'_1)$ is defined.

Now we consider $y<x$. If $q'd' \in f(x)$, then $f(x) \triangleleft q'$. Let $r = M(f,x,q')$. We distinguish two cases:

- If $r \leq y$, then $f(y) \triangleleft q'$. Since $f(y) \leq f(x)$ implies $f(y)^{\triangleleft}|q' = f(x)^{\triangleleft}|q' = q'd'$, we have $q'd' \in f(y)$: hence $\chi(y,q')$ is defined.
- If $r \not\leq y$, then $f(y) \triangleright q'$, hence, by (LS), $y \triangleright g(q')$, and $\chi(y,q')$ is defined.

In either case, $\chi(y,q')$ is defined. If $x \triangleright g(q')$, then, by property (3) of Lemma $x \triangleleft \alpha$, $y \leq x$ implies $y \triangleright g(q')$, and hence $\chi(y,q')$ is defined. This completes the proof of (A3).

(AFF): If $\chi(x,q') = ra$, then $ra = x|^{\triangleright}g(q')$ by definition of χ, that is, $ra \in A(x) \wedge g(q')$. Then also $ra \in A(r) \wedge g(q')$, hence $\chi(r,q') = ra$. The second part of (AFF) follows from the prime continuity of f.

Transformation 2: We construct an affine algorithm ϕ out of an affine abstract algorithm χ from M to M'. We build the paths s of ϕ by induction on the length of s (cf. Definition *From symmetric to concrete*):

- if $s \in \phi$, if $s \upharpoonright M$ and $s \upharpoonright M'$ are responses, and if $q' = (s \upharpoonright M')a'$ for some a' and $\chi(s \upharpoonright M, q')$ is defined, then

  $sa'a \in \phi$ if $\chi(s \upharpoonright M, q') = (s \upharpoonright M)a$

  $sa'd' \in \phi$ if $\chi(s \upharpoonright M, q') = q'd'$

- if $s \in \phi$, if $s \upharpoonright M$ and $s \upharpoonright M'$ are queries, and if $r = (s \upharpoonright M)d$ for some d and $\chi(r, s \upharpoonright M')$ is defined, then

  $sda \in \phi$ if $\chi(r, s \upharpoonright M') = ra$

  $sdd' \in \phi$ if $\chi(r, s \upharpoonright M') = (s \upharpoonright M')d'$

We omit the verification that this indeed defines an affine algorithm, and the proof that 1 ; 2 ; 3 and 2 ; 3 ; 1 are identity transformations. □

Berry-Curien's original sequential algorithms are obtained by withdrawing (AFF).

Definition *(Abstract algorithms)*
Let M and M' be as in Definition *Affine abstract algorithms*. An abstract algorithm is a partial function ψ from $D^{\circ}(M) \times Q'$ to $Q \cup R'$ which satisifes the axioms (A1), (A2) and (A3). The composition of abstract algorithms is defined exactly as the composition of affine abstract algorithms.

This definition is *mutatis mutandis* the one appearing in [CuMon, Definition 2.5.4], and is equivalent to it. The only difference lies in the fact that in [CuMon],we require that if ψ is defined at (x,q'), then q' is enabled from $\psi.x$. This limitation can be removed when the concrete data structures are *sequential* (see [CuMon, Definition 2.1.10, and (in the second edition) exercise 2.4.5.1]), as it is the case for sequential data structures.

Theorem *(CCC)*
The category $\mathbf{ALGO_{sds}}$ (**ALGO** for short) of sequential data structures and (abstract) sequential algorithms is cartesian closed.

Proof: The proof can be found in [BeCu1, CuMon], in the setting of concrete data structures. As we have seen, the product already exists in the subcategory of affine sequential algorithms, and it is easily verified that it is still a product in the category of sequential algorithms. The exponent $M \rightarrow M'$ is most readily described, not as an sds, but as a filiform concrete data structure $(C'', V'', E'', \vdash)$, defined as follows:

- $C'' = D^{\circ}(M) \times Q'$,
- $V'' = Q \cup R'$,
- $((x,q'),q) \in E''$ iff $q \in A(x)$,
  $((x,q'),r') \in E''$ iff $r' = q'd'$ for some d',
- $((x,q'),q) \vdash (x_1,q')$ iff $x_1 = x \cup \{qd\}$ for some d,
  $((x,q'),d') \vdash (x,q'_1)$ iff $q'_1 = q'd'a'$ for some a'.

Notice that in this description a cell (x_1,q'_1) may have two enablings: a "valof" enabling $((x,q_1),q)$ or an "output" enabling $((x_1,q'),d')$. In order to turn this description of $M \rightarrow M'$ into an sds, one has to "split" the cell (x_1,q'_1) in as many ways as there are to enable it. We indicate in the appendix of [CCF] how to do this. Another less direct way of constructing $M \rightarrow M'$ as an sds is to rely on the next theorem, and to define $M \rightarrow M'$ as $!M \multimap M'$. □

Our presentation makes it clear that the category of sds's and affine algorithms is included in the category of sds's and sequential algorithms. The following piece of categorical reasoning then tells us what to do.

Proposition *(CoKleisli and inclusion)*
Let **C** and **C'** be two categories having the same class of objects, and such that for each pair of objects A and B, the homset **C**(A,B) is included in **C'**(A,B). If the inclusion functor $\subseteq$: **C**$\rightarrow$**C'** has a left adjoint !, then **C'** is isomorphic to the CoKleisli category associated with the comonad $! \circ \subseteq$: **C**$\rightarrow$**C** (! for short). □

Hence, in order to define a model of intuitionistic affine logic, we only have to construct a left adjoint !: **ALGO**$\rightarrow$**AFFALGO** to the inclusion functor. Another piece of categorical folklore tells us that we only need to construct ! on objects, and to build appropriate natural bijections. The following definition agrees with that given in [Lam1].

Definition *(Exponential)*
Let M be an sds. We define $!M = (Q,R,P_!)$, where we recall that Q and R are the sets of queries and responses of M, and where $P_!$ is the collection of alternating non-ε and non-repetitive paths σ over $Q \cup R$ which satisfy the following conditions:

- every prefix $\sigma_1 r$ of σ, where $r = q_1 d$ for some d, is such that σ_1 ends with q_1,
- every prefix $\sigma_1 q$ of σ, where $q = r_1 a$ for some a, is such that some prefix of σ_1 ends with r_1.

Such paths are called *path sequences.* ○

It is easily seen that the collection of response prefixes of a response ρ of !M forms a finite strategy of M. Hence the prime elements of the domain !M represent the finite strategies of M. This is reminiscent of the coherent semantics of linear logic, where the tokens of the exponential !D are the finite cliques of D [GirLin]. But notice here that the same "clique" x gives in general rise to as many "tokens" of the exponential as there are ways to sequentialize x.

Notation

Given a response ρ of !M, we denote by $\|\rho\|$ the collection of response prefixes of ρ.

Theorem *(Adjunction)*

Let M and M' be two sds's. There is an order-isomorphism $\zeta_{M,M'}$ between the set of abstract algorithms from M to M' and the collection of affine abstract algorithms from !M to M'. This bijection is natural in M', that is, for any abstract algorithm ϕ: M→M', and for any affine abstract algorithm ψ: M'→M'', we have:

$$\zeta_{M,M''}(\psi \circ \phi) = \psi \circ \zeta_{M,M'}(\phi)$$

Thus ! and the collection of bijections $\zeta_{M,M'}$ define a left adjoint to the inclusion functor.

Proof: We only provide the definitions of the inverse mappings. Let ψ be an abstract algorithm from M to M'. We construct an affine abstract algorithm χ as follows:

- $\chi(\rho q r,q')$ can be defined only if $\chi(\rho,q')$ is defined:
  - if $\chi(\rho,q') = r'$, then $\chi(\rho q r,q') = r'$,
  - if $\chi(\rho,q') = q_1 \neq q$, then $\chi(\rho q r,q') = q_1$,
  - if $\chi(\rho,q') = q$, then $\chi(\rho q r,q') = \phi(\|\rho q r\|,q')$.

We set $\zeta(\psi) = \chi$. Conversely, let χ be an affine abstract algorithm from !M to M'. We construct an abstract algorithm ψ as follows. We need to keep track of the order of exploration of the input: $\psi(x,q')$ is defined when we can successfully build a path

$$q_1 r_1 \ldots q_n \quad \text{or} \quad q_1 r_1 \ldots q_n r_n$$

of !M such that

$$\chi(\emptyset,q') = q_1,\ r_1 = q_1 d_1 \in x, \ldots, \chi(q_1 r_1 \ldots,q') = q_1 r_1 \ldots q_n \text{ and}$$
$$q_n \in A(x) \text{ or}$$
$$r_n = q_n d_n \in x \text{ and } \chi(q_1 r_1 \ldots q_n r_n,q') = r'$$

and we set $\psi(x,q') = q_n$ or $\psi(x,q') = r'$ in the respective cases. □

Still by categorical reasoning, the conjunction of Theorem *CCC* and of Theorem *Adjunction* provides natural isomorphisms between $(!M)\otimes(!M')$ and $!(M\times M')$. We thus have all the ingredients for a semantics of affine intuitionisitic logic, with connectives $\otimes$, 1, $\multimap$, $\times$, and $\top$.

Digression: sequential algorithms and errors

We end this section with a digression. We have described (affine) sequential algorithms as pairs of two functions formalizing both an input-output behaviour and a computation strategy. It turns out that an enlargement of the domains with error elements allows us to capture the computation strategy as part of the input-output behaviour. This key observation, due to Cartwright and Felleisen [CF], has been integrated in the framework of concrete data structures and of sequential data structures in [CuObs, CCF], where larger categories of (sequential and) error-sensitive functions are considered.

We owe to Streicher the observation that sequential algorithms can be recast as those error-sensitive functions that are also error-reflecting. We briefly explain these notions and justify this claim.

Notation

We suppose that a non-empty set of error elements is given, which is disjoint from any set of addresses used in any sds. We call this set Err, and use e to range over it. The following definition is taken from [CCF].

Definition *(Observable strategy)*

Let M be an sds. An *observable response* of M is either a response of M, or a path of the form qe where q is a query of M and where $e \in \mathrm{Err}$. An observable strategy of M is a set of observable responses that is closed under response prefixes and non-ε glb's. If x is an observable strategy, F(x) denotes the set of queries q such that $q* \in x$, for some $* \in D \cup \mathrm{Err}$, and A(x) is as in Definition *Sequential data structure*. The set of observable strategies of M is written $D^{\mathrm{Err}}(M)$. O

The following statement formalizes Streicher's suggestion.

Proposition *(Errors)*

Let M and M' be two sds's. There is an order-isomorphism between $D(M \multimap M')$ and the set of *error-sensitive* and *error-reflecting* continuous functions $h: D^{\mathrm{Err}}(M) \to D^{\mathrm{Err}}(M')$, which are defined as follows.

- Error-sensitivity: for any x and q' such that $q' \in A(h(x))$ and $q' \in F(h(z))$ for some $z > x$, there exists $q \in A(x)$ such that

  - for any $y > x$, $q' \in F(h(y))$ implies $q \in F(y)$, and
  - $h(x \cup \{qe\}) = h(x) \cup \{q'e\}$, for any $e \in \mathrm{Err}$.

  Such a q is called the sequentiality index of h at (x,q') (the second condition implies the uniqueness of q).

- Error-reflection: for any q', e, and y, if $q'e \in h(y)$, then for some $x < y$, h has a sequentiality index q at (x,q'), and $x \cup \{qe\} \leq y$.

These order-isomorphisms preserve the composition of morphisms of **ALGO**, thus error-sensitive and error-reflecting continuous functions are just an alternative way of presenting the morphisms of the category **ALGO**.

Proof hint: In [CuObs] and [CCF, corollary 6.21] we have shown that the error-sensitive functions are in order-isomorphic correspondence with the observable strategies of $M \multimap M'$. We just have to show that the inverse functions map strategies to error-sensitive and error-reflecting continuous functions and vice-versa, which is almost immediate. □

We believe that the previous statement should have a more abstract meaning than the a priori ad hoc nature of a set of errors could induce one to believe. Errors allow us to "reverse the flow of information" by transfering the output-directed information contained by the component g of a (symmetric) algorithm (f,g) into the input-output function f of the algorithm.

6. Further remarks

In this section we include miscellaneous remarks. First we exhibit an intriguing self-adjunction. Consider the following construction. Let $M = (A,D,P)$ be an sds. We define $M^{\dagger} = (D \cup \{\circ\}, A, \{\circ p \mid p \in P\})$. It is easily seen that this operation on objects extends to a functor †: **AFFALGO** $\rightarrow$ **AFFALGO**^{op}. It is also useful to observe:

- $D(^{\dagger}(M)) = D^{\perp}(M) \cup \{\circ\}$,
- $D^{\perp}(^{\dagger}(M)) = D(M)$.

Proposition *(Self-adjoint)*
The functor †: **AFFALGO** $\rightarrow$ **AFFALGO**^{op} is left adjoint to itself.

Proof: Let M and M' be two sds's. A morphism from $^{\dagger}(M')$ to M in **AFFALGO**^{op} is a morphism from M to $^{\dagger}(M')$ of **AFFALGO**, that is,

- a pair of a function from D(M) to $D(^{\dagger}(M'))$ and a partial function from $D^{\perp}(^{\dagger}(M'))$ to $D^{\perp}(M)$,

which amounts to:

- a pair of a partial function from D(M) to $D^{\perp}(M')$ and a partial function from D(M') to $D^{\perp}(M)$.

which in turn can be presented as:

- a pair of a partial function from $D^{\perp}(^{\dagger}(M))$ to $D^{\perp}(M')$ and a function from D(M') to $D(^{\dagger}(M))$,

that is, a morphism from M' to $^{\dagger}(M)$ in **AFFALGO**. □

The construction $\uparrow$ is linked with the separated sum construction on concrete data structures [BeCu1, CuMon]. Specifically we define:

$$M + M' = \uparrow(\uparrow(M) \times \uparrow(M'))$$

The resulting (Lamarchian) sds is represented in Figure 14. It is such that D(M+M') is the separated sum of D(M) and D(M') [PloD].

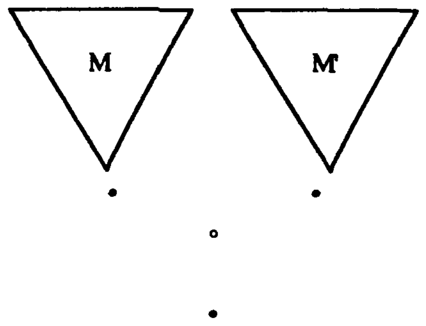


Figure 14: Separated sum

The above self-adjunction property was pointed out to the author by Hyland, who also noticed:

> The category of games considered in [AJ2] can be obtained out of the category of sds's and (winning) affine sequential algorithms by an instance of Chu's construction [Barr].

This general construction allows to get a *-autonomous category out of a cartesian, monoidal closed category $\mathcal{C}$, and is parametrized by a distinguished object of $\mathcal{C}$. We briefly describe it in the instance which interests us here, where the distinguished object is the terminal object 1 of $\mathcal{C}$ [illegible] category Chu($\mathcal{C}$,1), Chu($\mathcal{C}$) for short, has as objects pairs (M^+,M^-) of two objec[illegible] and as morphisms between (M^+,M^-) and (M'^+,M'^-) pairs $(a \in \mathcal{C}(M^+,M'^+)\,,\, b \in \mathcal{C}(M'^-,M^-))$. It is easy to check that the following yields a monoidal closed structure on Chu($\mathcal{C}$):

- $(M^+,M^-)\otimes(M'^+,M'^-) = (M^+\otimes M'^+\,,\,(M^+\multimap M'^-)\times(M'^+\multimap M^-))$,
- the unit is (I,1), where I is the unit of the tensor and 1 is terminal in $\mathcal{C}$,
- $(M^+,M^-)\multimap(M'^+,M'^-) = ((M^+\multimap M'^+)\times(M'^-\multimap M^-)\,,\,M^+\otimes M'^-)$.

Moreover, taking (1,I) to be the interpretation of $\perp$, we obtain a *-autonomous structure, where $(M^+,M^-)^\perp$ is (M^-,M^+).

It is natural to recover $\mathcal{C}$ as the full subcategory of objects of the form (M,1), and

$\mathcal{C}^{op}$ as the full subcategory of objects of the form (1,M). Let us now briefly consider the homsets in Chu($\mathcal{C}$) corresponding to the various combinations of polarities. We have:

- Chu($\mathcal{C}$)((M,1),(M',1)) $\simeq$ $\mathcal{C}$(M,M'),
- Chu($\mathcal{C}$)((M,1),(1,M')) is a singleton,
- Chu($\mathcal{C}$)((1,M),(M',1)) $\simeq$ $\mathcal{C}$(1,M)$\times\mathcal{C}$(1,M'),
- Chu($\mathcal{C}$)(1,M),(1,M')) $\simeq$ $\mathcal{C}$(M',M).

A more precise formulation of Hyland's observation is that the category of games G of [AJ2] can be obtained as Chu(G^+), where G^+ is a category of winning affine sequential algorithms.

A notable difference between the affine intuitionistic model considered here and the models in [AJ2,Lam1] is that the latter do not validate weakening. The arguments (and the categories considered) are different in [AJ2] and [Lam1].

- Abramsky-Jagadeesan: By De Morgan laws, finding a winning strategy in $A\otimes B\multimap A$ amounts to find a winning strategy in $(A\multimap A)\parr B^\perp$ (where $\parr$ is the dual of $\otimes$). Let A be an sds, and let B be the game (of polarity $\circ$) consisting of an empty set of addresses and a single datum: B = ($\emptyset$,{d},{d}). Both $A\multimap A$ and $B^\perp$ are sds's M and M'. Let us examine the definition of $\parr$ in G:

$$-\ (M^+,M^-)\parr(M'^+,M'^-) = ((M^-\multimap M'^+)\times(M'^-\multimap M^+)\ ,\ M^-\otimes M'^-).$$

When specialized to sds's (M,1) and (M',1), it amounts to:

$$(M,1)\parr(M',1) = (M'\times M,1)$$

(cf. Chu($\mathcal{C}$)((1,M),(M',1)) above). In other words, for this combination of polarities, the $\parr$ is ... the product.

While the copycat strategy, which is winning in $A\multimap A$, is also a strategy in $(A\multimap A)\parr B^\perp$, it is not winning in $(A\multimap A)\parr B^\perp$. In fact, there is no winning strategy in $(A\multimap A)\parr B^\perp$. (That is, there is no winning strategy in $A\otimes B\multimap A$, and weakening thus fails.) To see this, recall that winning means: winning against any counterstrategy. Consider the counterstrategy consisting of the move d in $B^\perp$ by the opponent. The player's first move has to be in $B^\perp$ since the initial move in $A\multimap A$ is an opponent's move. But since the player has no move in $B^\perp$, he is stuck.

-Lamarche: Unlike us, and unlike [AJ2], Lamarche accepts both ε paths and empty strategies in his formalization of games and strategies: as a consequence, for him, the terminal object is the empty game. On the other hand, the unit is the empty sds, that is, in Lamarche's terminology, the game consisting of only one move, by the player. Thus the strong form of weakening provided by the coincidence terminal/unit fails in Lamarche's model [Lam1].

We end this discussion of weakening by considering (a variant of) Lamarche's polarized constructions, where, as we did in Section 2, we assume that strategies are non-empty. We first suggest how a category $\odot$ whose objects are either sds's $M^\bullet$ or games M° of polarity $\circ$ could be built. Its homsets could be defined by cases as follows (the linear negation $()^\perp$ is defined by reversing the polarities of all the nodes of the game represented as a tree):

- $\odot(M^\bullet,M'^\bullet)$ = **AFFALGO**$(M^\bullet,M'^\bullet)$,
- $\odot(M^\bullet,M'^\circ)$ is the collection of strategies of $(M^\bullet)^\perp \parr M'^\circ$, where $\parr$ is defined on games $\circ$ dually to the tensor product of **AFFALGO**,
- $\odot(M^\circ,M'^\bullet)$ is empty,
- $\odot(M^\circ,M'^\circ)$ = **AFFALGO**$((M'^\circ)^\perp,(M^\circ)^\perp)$.

These definitions are dictated by De Morgan laws and by Table 1 (cf. Section 3):

Notice that this tentative category $\odot$ looks quite different from **G** in the mixed situations of morphisms between two objects of different polarities. We do not pursue here an investigation of $\odot$, but we limit ourselves to observing that weakening fails in $\odot$ for yet another reason. As a counterpart of $\odot(M^\circ,M'^\bullet) = \emptyset$, we have that the $\parr$ of two sds's is not defined, so that a fortiori there is no strategy in $(A\multimap A)\parr B^\perp$, whatever B of polarity $\circ$ is.

One lesson of the above discussion is that while the current game-theoretic semantics of (fragments of) linear logic all roughly agree on the intuitionistic affine fragment, they seem to be hard to compare outside this fragment. And indeed, the models of [Lam2] on one side, and of [AJ2, HO] on the other side, lead to quite different completeness results:

- Lamarche characterizes the winning strategies which are meanings of proofs, in a fragment of linear logic that contains the additive connectives, via conditions that are reminiscent of the trip conditions in proof nets;

- Abramsky and Jagadeesan define a notion of history-free strategy, and show that any winning and history-free strategy is the meaning of a unique cut-free proof of the multiplicative fragment of linear logic augmented with the MIX rule [Gir]. Hyland and Ong add a fairness constraint to the games, and show the same result with respect to the multiplicative fragment of linear logic (without the MIX rule).

We end with a question. We wonder whether game-theoretic semantics can be defined at a more abstract level. A step in this direction was already taken by Bucciarelli and Ehrhard [BuEhr]. They have generalized the notion of sequential algorithm in a setting of so-called sequential structures (X_*,X^*), where X_* and X^* are two partial orders, formalizing the idea of a space of data (or "points") and a space of questions (or "opens"). A sequential structure is endowed with a predicate, call it ANSWER, over $X_* \times X^*$. If $(x,\gamma) \in$ ANSWER, we say that x answers question γ. This is reminiscent of the winning predicate $\lhd$. But in [BuEhr], as in [LS], the predicate ANSWER is not refined to a notion of result of an interplay, as done here. We wonder whether we could define an abstract category of games (X_*,X^*), equipped with a function I mapping the elements of $X_* \times X^*$ to the set of primes of X_* or X^*. Any sds M gives rise to such a structure $(D(M), D^{\perp}(M))$, with I as defined by Proposition *Play*. What seems needed to carry out this program is a good denotational interpretation of the counter-strategies of an exponent sds.

Acknowledgements
I had useful discussions with Martin Abadi, Samson Abramsky, Antonio Bucciarelli, Thomas Ehrhard, Martin Hyland, Radha Jagadeesan, François Lamarche, Luke Ong, and Thomas Streicher. The material of the present paper has been developed mostly independently from the work of Abramsky-Jagadeesan [AJ2], and has been influenced by the ideas of Lamarche [Lam1]. Lamarche should be credited for the affine decomposition of the exponent in the category of sequential algorithms, which he discovered in the autumn of 1991 in Paris. I should also thank him for comments on an earlier, less elegant, axiomatization of symmetric algorithms.

References

[AJ1] S. Abramsky, R. Jagadeesan, New foundations for the geometry of interaction, in Proc. of Seventh Annual Symposium on Logic in Computer Science, Santa-Cruz (1992).

[AJ2] S. Abramsky, R. Jagadeesan, Games and full completeness for multiplicative linear logic, Technical report DoC 92/24, Imperial College (1992).

[Barr] M. Barr, *-Autonomous categories and linear logic, Mathematical Structures in Computer Science 1 (1991).

[Be] G. Berry, Stable models of typed lambda-calculi, in Proc. 5th Int. Coll. on Automata, Languages and Programming, Lect. Notes in Comp. Sci. 62, 72-89, Springer (1978).

[BeCu1] G. Berry, P.-L. Curien, Sequential algorithms on concrete data structures, Theoretical Computer Science 20, 265-321 (1982).

[BeCu2] G. Berry, P.-L. Curien, Theory and practice of sequential algorithms, in Algebraic Methods in Semantics, J. Reynolds and M. Nivat eds, Cambridge University Press, 35-88 (1985).

[BCL] G. Berry, P.-L. Curien, J.-J. Lévy, Full abstraction of sequential languages: the state of the art, in Algebraic Methods in Semantics, J. Reynolds and M. Nivat eds, Cambridge University Press, 89-131 (1985).

[Blass1] A. Blass, Degrees of indeterminacy of games, Fundamenta Mathematicae LXXVII, 151-166 (1972).

[Blass2] A. Blass, A game semantics for linear logic, Annals of Pure and Applied Logic 56, 183-220 (1992).

[BuEhr] A. Bucciarelli, T. Ehrhard, A theory of sequentiality, to appear in Theoretical Computer Science.

[CF] R. Cartwright, M. Felleisen, Observable sequentiality and full abstraction, in Proc. 19th ACM Symposium on Principles of Programming Languages, Albuquerque (1992).

[CCF] R. Cartwright, P.-L. Curien, M. Felleisen, Fully abstract models of observably sequential languages, to appear in Information and Computation.

[Con] J.H. Conway, On numbers and games, London Mathematical Society Monographs, vol. 6, Academic Press (1976).

[CuMon] P.-L. Curien, Categorical combinators, sequential algorithms and functional programming, Pitman (1986), revised edition, Birkhaüser (1993).

[CuObs] P.-L. Curien, Observable algorithms on concrete data structures, in Proc. Seventh Annual Symposium on Logic in Computer Science, Santa-Cruz (1992).

[Dan] V. Danos, Une application de la logique linéaire à l'étude des processus de normalisation (principalement du λ-calcul), Thèse de Doctorat, Université Paris VII (1990).

[GirLin] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1), 1-102 (1987).

[GirGI] J.-Y. Girard, Towards a geometry of interaction, in Categories in Computer Science and Logic, J.W. Gray and A. Scedrov eds, Contemporary Mathematics 92, 69-108 (1989).

[Gun] C. Gunter, Semantics of Programming Languages, Structures and techniques, MIT Press (1992).

[GuSco] C. Gunter, D. Scott, Semantic domains, Chapter in Handbook of Theoretical Computer Science, Vol. B, J. van Leeuwen ed., MIT Press/Elsevier (1990).

[HO] J.M.E. Hyland, C.-H. L. Ong, Fair games and full completeness for multiplicative linear logic without the MIX-rule, manuscript (1993).

[Joy] A. Joyal, Remarques sur la théorie des jeux à deux personnes, Gazette des Sciences Mathématiques du Québec 1(4) (1977).

[KP] G. Kahn, G.D. Plotkin, Concrete Domains, in Boehm Festschrift, Special Volume of Theoretical Computer Science, to appear (1993).

[KCF] R. Kanneganti, R. Cartwright, M. Felleisen, SPCF: its model, calculus, and computational power, REX Workshop on Semantics and Concurrency, Lecture Notes in Comput. Sci. 526, 131-151, Springer (1992).

[LS] Y. Lafont, T. Streicher, Games semantics for linear logic, in Proc. Sixth Annual Symposium on Logic in Computer Science, Amsterdam (1991).

[Lam1]] F. Lamarche, Sequentiality, games and linear logic, manuscript (1992).

[Lam2] F. Lamarche, Games, additives and correctness criteria, manuscript (1992).

[MR] P. Malacaria, V. Régnier, Some results on the interpretation of λ-calculus in operator algebras, in Proc. Sixth Annual Symposium on Logic in Computer Science, Amsterdam (1991).

[PloD] G.D. Plotkin, The category of complete partial orders: a tool for making meanings, lecture notes, Universita di Pisa (1978); extended, University of Edinburgh (1981).

[Reg] L. Régnier, Lambda-calcul et réseaux, Thèse de Doctorat, Université Paris VII (1992).

Computational Adequacy via 'Mixed' Inductive Definitions

Andrew M. Pitts*

University of Cambridge Computer Laboratory,
Pembroke Street, Cambridge CB2 3QG, England

Abstract. For programming languages whose denotational semantics uses fixed points of domain constructors of mixed variance, proofs of correspondence between operational and denotational semantics (or between two different denotational semantics) often depend upon the existence of relations specified as the fixed point of non-monotonic operators. This paper describes a new approach to constructing such relations which avoids having to delve into the detailed construction of the recursively defined domains themselves. The method is introduced by example, by considering the proof of computational adequacy of a denotational semantics for expression evaluation in a simple, untyped functional programming language.

1 Introduction

It is well known that various domain constructors can be extended to act on *relations* on domains. For example, given binary relations R and S on domains D and E, there is a binary relation $R \to S$ on the domain of continuous functions $D \to E$ given by: $(f, g) \in (R \to S)$ if and only if for all $(x, y) \in R$, $(f(x), g(y)) \in S$. The utility of such constructions on relations can be seen in the various applications of 'logical relations' techniques in denotational semantics, pioneered by Milne [6], Plotkin [10, 11] and Reynolds [12]. For applications to programming language semantics, undoubtedly the most important domain-construction technique is that of solving recursive domain equations. In general, the body of a domain equation may involve not only positive, but also negative occurrences of the defined domain. Traditionally, the construction of the action on relations of such a recursively defined domain constructor has involved delving into the quite heavy technical machinery used to establish the existence of the domain itself. In [9] the author described a more elementary method of construction, inspired by Freyd's recent categorical analysis of recursive types [1, 2, 3]. It makes use of mixed inductive/co-inductive definitions. Apart from this, only quite straightforward domain-theoretic techniques are needed—namely fixed point induction and the fact that the identity function on a recursively defined domain is the

* Research supported by UK SERC grant GR/G53279, CEC ESPRIT project CLICS-II and CEC SCIENCE project PL910296

least fixed point of a certain continuous functional canonically associated with the domain equation.

In this paper, we illustrate the use of this new method of construction of relations on recursively defined domains by example. We consider a specific application where such relations are needed—namely the proof of correspondence between the denotational and operational semantics of a functional programming language. Recall that a denotational semantics is called 'computationally adequate' for an operationally defined expression evaluator provided any expression evaluates to canonical form just in case its denotation is not the bottom element of the corresponding semantic domain. This property is important since, combined with compositionality of the denotational semantics, it implies that observational equivalence of programming language expressions may be established via equality of denotations. See Meyer [5] for a discussion of this property. Proofs of computational adequacy are non-trivial when the denotational semantics of the programming language involves solving recursive domain equations $X = \Phi(X)$ in which X occurs negatively (and maybe also positively) in the domain constructor $\Phi(X)$. We consider a very simple example of this — an untyped lambda calculus — in order not to obscure the novelty of our approach with language-related details.

The computational adequacy property is reviewed in Sect. 2, where we recall how it can be established via the existence of a certain recursively specified relation of 'formal approximation' between domain elements and programs. Our new method of construction of the formal approximation relation $\lhd$ is given in Sect. 3. The method involves three steps:

- First, the negative and positive occurrences of $\lhd$ in the body of its recursive specification $\lhd = \phi(\lhd)$ are replaced by fresh variables $\lhd^-$ and $\lhd^+$ respectively. This results in a new operator $\psi(\lhd^-, \lhd^+)$ which is monotonic in $\lhd^+$, anti-monotonic in $\lhd^-$, and from which the original operator ϕ can be obtained by diagonalizing. (This separation of variables is a key feature of Freyd's recent analysis of recursive types.)
- Secondly, the new operator ψ is used to give simultaneous inductive definitions of positive and negative versions of the formal approximation relation.
- Lastly, these positive and negative versions are proved equal, and so by construction constitute the required relation. The proof of equality is a simple fixed point induction argument. It makes use of a key property of recursively defined domains, namely that they are 'minimal invariants' for their associated domain constructor: see Definition 2.

Finally in Sect. 4 we indicate an important aspect of the above method of construction, namely that it not only produces a suitable relation, but also characterizes it via a 'universal property' (in the category-theoretic sense). It is this universal property which gives rise to the reasoning principles established in [8, 9].

2 Computational adequacy

In this section we review the standard approach to proving computational adequacy, using a very simple untyped functional programming language $\mathcal{L}$ to illustrate what is involved. $\mathcal{L}$ is an untyped version of Plotkin's call-by-name PCF [10]. Its expressions are given by:

$M ::= x$	variables
$\mid \underline{n}$	numerals
$\mid suc(M)$	successor
$\mid pred(M)$	predecessor
$\mid \mathit{if}\ M = 0\ \mathit{then}\ M\ \mathit{else}\ M$	conditional
$\mid \lambda x.M$	function abstraction
$\mid MM$	function application

where x runs over a fixed, infinite set of *variables*, and n runs over the set of integers, $\mathbb{Z}$. Function abstraction is the only variable-binding construct (occurrences of x in M are bound in $\lambda x.M$). We denote by $M[M'/x]$ the result of substituting an expression M' for all free occurrences of x in M (subject to the usual conventions about renaming bound variables if necessary to avoid variable capture).

Let *Prog* ('programs') denote the collection of closed expressions in $\mathcal{L}$, i.e. those with no free variables. We denote by *Val* ('values') the subset of *Prog* consisting of all *canonical forms*, which here means all closed expressions that are either numerals $\underline{n}$ or function abstractions $\lambda x.M$. An operational semantics for $\mathcal{L}$ can be given via an *evaluation relation*

$$P \Downarrow V \qquad (P \in Prog,\ V \in Val)$$

which is the subset of $Prog \times Val$ inductively defined by the rules in Table 1. The last rule embodies the non-strict, or 'call-by-name' scheme for evaluating function applications.

Table 1. Rules for evaluating programs in $\mathcal{L}$.

$$\frac{}{V \Downarrow V} \qquad \frac{P \Downarrow \underline{n}}{suc(P) \Downarrow \underline{n+1}} \qquad \frac{P \Downarrow \underline{n+1}}{pred(P) \Downarrow \underline{n}}$$

$$\frac{P \Downarrow \underline{0} \qquad Q \Downarrow V}{(\mathit{if}\ P = 0\ \mathit{then}\ Q\ \mathit{else}\ R) \Downarrow V} \qquad \frac{P \Downarrow \underline{n} \qquad R \Downarrow V}{(\mathit{if}\ P = 0\ \mathit{then}\ Q\ \mathit{else}\ R) \Downarrow V}\ (n \neq 0)$$

$$\frac{P \Downarrow \lambda x.M \qquad M[Q/x] \Downarrow V}{PQ \Downarrow V}$$

Denotational semantics for expressions in $\mathcal{L}$ can be given using a solution to the domain equation

$$D \cong (\mathbb{Z} + (D \to D))_\perp \ . \tag{1}$$

Here we can take 'domain' to mean a partially ordered set with a least element $\perp$ and possessing least upper bounds $\bigsqcup_{i<\omega} d_i$ of all countable chains $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$. The domain on the right-hand side of (1) is the lift of the disjoint union of the set of integers $\mathbb{Z}$ (discretely ordered) with the domain of continuous functions $D \to D$ (ordered pointwise). Thus a domain D is a solution to (1) if it comes equipped with continuous functions

$$\begin{aligned} \mathrm{num} &: \mathbb{Z} \longrightarrow D \\ \mathrm{fun} &: (D \to D) \longrightarrow D \end{aligned}$$

which combine to give an order isomorphism between the disjoint union $\mathbb{Z} + (D \to D)$ and $\{d \in D \mid d \neq \perp\}$. Given such a D, one can assign to each $\mathcal{L}$-expression M and each environment ρ (a finite partial function from the set of variables to D) whose domain of definition contains the free variables of M, an element

$$[\![M]\!]\rho \in D \ .$$

The definition of $[\![M]\!]\rho$ is by induction on the structure of M and is quite standard; for the record, we give the clauses of the definition in Table 2. The clause for $\lambda x.M$ uses the notation $\rho[x \mapsto d]$ to indicate the environment mapping x to d and otherwise acting like ρ.

If an environment ρ' extends ρ, then $[\![M]\!]\rho' = [\![M]\!]\rho$. In particular for programs $P \in \mathit{Prog}$, i.e. for closed expressions, $[\![P]\!]\rho$ is an element of D which is independent of ρ, and which we write simply as $[\![P]\!]$. The following property can be established by induction on the derivation of the evaluation $P \Downarrow V$.

Proposition 1 (Soundness). *If $P \Downarrow V$ then $[\![P]\!] = [\![V]\!]$.*

Of course one cannot expect the converse of this soundness property to hold, since function abstractions are canonical forms whether or not the body of the abstraction is fully evaluated. For example $[\![\lambda x.\mathit{suc}(\underline{0})]\!] = [\![\lambda x.\underline{1}]\!]$, but $\lambda x.\mathit{suc}(\underline{0}) \Downarrow \lambda x.\underline{1}$ does not hold. However, if $[\![P]\!] = [\![V]\!]$, then since (from Table 2) the denotations of canonical forms are non-bottom elements of D, one at least has that $[\![P]\!] \neq \perp$. D is called *computationally adequate* if for all programs P, $[\![P]\!] \neq \perp$ holds (if and) only if $P \Downarrow V$ holds for some canonical form V. The point of this property is that it permits observational equivalence of $\mathcal{L}$-expressions to be established via equality of denotations: see Meyer [5].

Whilst the soundness property of Proposition 1 holds for any domain D which is a solution for the domain equation (1), computational adequacy only holds if D is a suitably minimal solution. One way of expressing this minimality, essentially due to D. Scott, is as follows.

Table 2. Denotations of $\mathcal{L}$-expressions.

$$[\![x]\!]\rho = \rho(x)$$

$$[\![\underline{n}]\!]\rho = \mathrm{num}(n)$$

$$[\![suc(M)]\!]\rho = \begin{cases} \mathrm{num}(n+1) & \text{if } [\![M]\!]\rho = \mathrm{num}(n) \\ \perp & \text{otherwise} \end{cases}$$

$$[\![pred(M)]\!]\rho = \begin{cases} \mathrm{num}(n-1) & \text{if } [\![M]\!]\rho = \mathrm{num}(n) \\ \perp & \text{otherwise} \end{cases}$$

$$[\![if\ M = 0\ then\ M'\ else\ M'']\!]\rho = \begin{cases} [\![M']\!]\rho & \text{if } [\![M]\!]\rho = \mathrm{num}(0) \\ [\![M'']\!]\rho & \text{if } [\![M]\!]\rho = \mathrm{num}(n) \text{ and } n \neq 0 \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\lambda x.M]\!]\rho = \mathrm{fun}(\lambda d \in D.[\![M]\!]\rho[x \mapsto d])$$

$$[\![MM']\!]\rho = \begin{cases} f([\![M']\!]\rho) & \text{if } [\![M]\!]\rho = \mathrm{fun}(f) \\ \perp & \text{otherwise} \end{cases}$$

Definition 2 (Minimal invariant property). Let $\Phi(-) \overset{def}{=} (\mathbb{Z}+(-)\rightarrow(-))_\perp$. An *invariant* for Φ is a domain D equipped with an order isomorphism $i : D \cong \Phi(D)$. Such an invariant is *minimal* if the identity function $id_D \in (D \rightarrow D)$ is the least fixed point of the continuous function $\delta_\Phi : (D \rightarrow D) \longrightarrow (D \rightarrow D)$ which maps $e \in (D \rightarrow D)$ to $i^{-1}\Phi(e)i$. Here $\Phi(e) : \Phi(D) \longrightarrow \Phi(D)$ is the function which is the identity on $\perp$ and integers, and acts on functions by pre- and post-composing with e. Thus if the isomorphism i is described in terms of functions num : $\mathbb{Z} \longrightarrow D$ and fun : $(D \rightarrow D) \longrightarrow D$ as above, then

$$\delta_\Phi(e)(d) = \begin{cases} \mathrm{num}(n) & \text{if } d = \mathrm{num}(n) \\ \mathrm{fun}(e \circ f \circ e) & \text{if } d = \mathrm{fun}(f) \\ \perp & \text{if } d = \perp \end{cases} \tag{2}$$

for all $e \in (D \rightarrow D)$ and all $d \in D$.

Theorem 3 (Computational Adequacy). *If (D, i) is a minimal invariant for $(\mathbb{Z} + (-)\rightarrow(-))_\perp$, then the denotational semantics of $\mathcal{L}$ in D is computationally adequate, i.e. for all $P \in Prog$*

$$\exists V(P \Downarrow V) \Leftrightarrow [\![P]\!] \neq \perp \ .$$

The statement of this theorem appears more general than corresponding results in the literature, which refer to the computational adequacy of a particular domain. However it is not really so general, since one can show that

- the minimal invariant property characterizes solutions to domain equations uniquely up to isomorphism; and
- the solutions to domain equations $D \cong \Phi(D)$ (for a wide class of domain constructors $\Phi(-)$) constructed via any of the several methods available in the literature (such as via colimits of embedding-projection pairs: see [4, Sect. 10.1]; or via Scott's 'information systems': see [15, Chap. 12]) yield minimal invariants. Indeed, the minimal invariant property amounts to the fact, familiar frcm the 'local' characterization of colimits of chains of embeddings [14, Theorem 2], that any element d of a recursively defined domain rec$X.\Phi(X)$ can be expressed as the least upper bound of a chain of projections of the element:

$$d = \bigsqcup_{i<\omega} \pi_i(d), \qquad \text{where } \begin{cases} \pi_0(d) = \bot \\ \pi_{i+1}(d) = \delta_\Phi(\pi_i)(d) \end{cases} .$$

However, it seems a step forward to have an abstract criterion on solutions of domain equations that suffices for computational adequacy. Moreover, the key construction needed in the new proof of Theorem 3 which we give in the next section, relies directly upon the minimal invariant property of D rather than upon any particular concrete construction of the domain.

The classical method for proving Theorem 3 is an adaptation by Milne [6] and Plotkin [10, 11] of Tait's use of 'computability' predicates in normalization proofs. It relies upon the construction of a binary relation between domain elements and programs with the following properties.

Definition 4. Let D be a solution to (1). A *formal approximation* relation is a binary relation $\lhd \subseteq D \times Prog$ satisfying:

1. For all $d \in D$ and $P \in Prog$, $d \lhd P$ if and only if
   either $d = \bot$,
   or $d = \text{num}(n)$ for some n such that $P \Downarrow \underline{n}$,
   or $d = \text{fun}(f)$ and $P \Downarrow \lambda x.M$ for some f and $\lambda x.M$ such that for all d', P',
   if $d' \lhd P'$ then $f(d') \lhd M[P'/x]$.
2. If $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$ is a chain in D with $d_i \lhd P$ for all i, then $\bigsqcup_{i<\omega} d_i \lhd P$.

Given such a formal approximation relation, for any expression M, any environment ρ whose domain of definition $dom(\rho) = \{x_1, \ldots, x_n\}$ contains the free variables in M, and any programs $P_1, \ldots, P_n$, it is easy to prove by induction on the structure of M that

$$\rho(x_1) \lhd P_1 \wedge \cdots \wedge \rho(x_n) \lhd P_n \Rightarrow [\![M]\!]\rho \lhd M[P_1/x_1, \ldots, P_n/x_n] .$$

In particular, in case $n = 0$ we obtain for all programs P that

$$[\![P]\!] \lhd P .$$

Hence if $[\![P]\!] \neq \bot$, then by the properties of $\lhd$ in part 1 of Definition 4, it follows that $P \Downarrow V$ for some V, as required for computational adequacy.

Therefore, to complete the proof of Theorem 3 we need to demonstrate that when D is a minimal invariant for $(\mathbb{Z} + (-) \to (-))_\bot$, there exists a relation $\lhd$ as in Definition 4.

3 A new construction of the relation $\lhd$

Let us begin by pointing out why the existence of a relation $\lhd$ as in Definition 4 is problematic. One can formulate the problem as one of solving a certain fixed point equation. Let $\mathcal{R}el$ be the set of all binary relations $R \subseteq D \times \mathit{Prog}$ which contain $\{\bot\} \times \mathit{Prog}$ and which satisfy condition 2 of Definition 4. In other words a binary relation R is in $\mathcal{R}el$ if and only if for each $P \in \mathit{Prog}$, $\{d \mid (d, P) \in R\}$ is an *admissible* subset of the domain D, i.e. chain-complete and containing $\bot$. Define an operation $\phi : \mathcal{R}el \longrightarrow \mathcal{R}el$ by:

$$\begin{aligned}\phi(R) \overset{def}{=} \{(d,P) \mid d = \bot \vee \exists n(d = \mathrm{num}(n) \wedge P \Downarrow \underline{n}) \vee\\ \exists f, \lambda x.M(d = \mathrm{fun}(f) \wedge P \Downarrow \lambda x.M \wedge\\ \forall (d', P') \in R.(f(d'), M[P'/x]) \in R)\}\ .\end{aligned}$$

Then a formal approximation relation is precisely an element $\lhd \in \mathcal{R}el$ satisfying $\lhd = \phi(\lhd)$. It is easy to see that $\mathcal{R}el$ is closed under taking intersections of binary relations, and hence it is a complete lattice when ordered by inclusion, $\subseteq$. However, ϕ is not a monotonic operation for $\subseteq$ (since the definition of $\phi(R)$ contains a negative as well as a positive occurrence of R), so we cannot appeal to the familiar Tarski fixed point theorem to construct a fixed point for ϕ.

In the literature, two methods can be found for constructing relations on recursively defined domains with certain non-monotonic fixed point properties. One method, due to Milne, Plotkin and Reynolds, makes use of Scott's construction of a recursively defined domain $D \cong \Phi(D)$ as the colimit of a chain of embedding-projections $D_0 \longrightarrow D_1 \longrightarrow \cdots$, where the domain D_n is obtained by iterating the domain constructor $\Phi(-)$ n times, starting with the trivial domain $\{\bot\}$. Then $\lhd$ can be constructed as an inverse limit of relations $\lhd_n \subseteq D_n \times \mathit{Prog}$ built up by iterating an appropriate action of $\Phi(-)$ on relations; see [12].

A second method, essentially due to Martin-Löf, applies only to Scott domains (precluding the use of constructors like the Plotkin powerdomain) and makes use of their presentation in terms of 'information systems' [13]. This method hinges upon the fact that for each program P, $\{d \mid d \lhd P\}$ is in fact a Scott-closed subset of D. Hence it suffices to construct the relation $\lhd$ only for compact elements of D, since $d \lhd P$ holds if and only if $a \lhd P$ holds for all compact a with $a \sqsubseteq d$. Information systems provide a formal language for compact elements of (recursively defined) Scott domains, and $a \lhd P$ (a compact) can be defined by a well-founded induction on the size of (a formal representation of) a. See [15, Sect. 13.4].

Here we present a third method, which is more abstract than the above two in that it relies upon the 'minimal invariant' property of Definition 2 rather than either of the techniques for giving concrete constructions of recursively defined domains mentioned above. To begin with, following Freyd's recent work on recursive types [1, 2, 3], we separate the positive and negative occurrences of R in the definition of $\phi(R)$. Thus given two relations $R^-, R^+ \in \mathcal{R}el$, define:

$$\begin{aligned}\psi(R^-,R^+) \overset{def}{=} \{(d,P) \mid\ & d = \bot \vee \exists n(d = \mathrm{num}(n) \wedge P \Downarrow \underline{n}) \vee \\ & \exists f, \lambda x.M(d = \mathrm{fun}(f) \wedge P \Downarrow \lambda x.M \wedge \\ & \quad \forall (d',P') \in R^-.(f(d'), M[P'/x]) \in R^+)\} \ .\end{aligned}$$

Clearly ψ determines a monotonic function

$$\psi : \mathcal{R}el^{op} \times \mathcal{R}el \longrightarrow \mathcal{R}el$$

where $\mathcal{R}el$ is partially ordered via $\subseteq$ and where $\mathcal{R}el^{op}$ has the opposite ordering. Furthermore, ϕ can be recovered from ψ by diagonalizing:

$$\phi(R) = \psi(R,R) \ . \tag{3}$$

We remarked above that $\mathcal{R}el$ is a complete lattice, with infima given by set-theoretic intersection. Hence $\mathcal{R}el^{op} \times \mathcal{R}el$ is also a complete lattice. We obtain a monotonic operator

$$\psi^{\S} : \mathcal{R}el^{op} \times \mathcal{R}el \longrightarrow \mathcal{R}el^{op} \times \mathcal{R}el$$

on this complete lattice by 'symmetrizing' ψ:

$$\psi^{\S}(R^-,R^+) \overset{def}{=} (\psi(R^+,R^-), \psi(R^-,R^+)) \ .$$

Now we can apply Tarski's fixed point theorem to obtain the least fixed point of $\psi^{\S}$, which we will denote by $(\lhd^-, \lhd^+)$. Thus $\lhd^-$ and $\lhd^+$ are given by simultaneous, inductive definitions. Using the fact that infima in $\mathcal{R}el$ are given by intersection, together with the definition of $\psi^{\S}$, these relations can be described explicitly as follows:

$$\begin{aligned}\lhd^+ &\overset{def}{=} \bigcap\{R^+ \in \mathcal{R}el \mid \exists R^- \in \mathcal{R}el(R^- \subseteq \psi(R^+,R^-) \wedge \psi(R^-,R^+) \subseteq R^+)\} \\ \lhd^- &\overset{def}{=} \bigcap\{S \in \mathcal{R}el \mid \forall R^-, R^+ \in \mathcal{R}el. \\ &\qquad (R^- \subseteq \psi(R^+,R^-) \wedge \psi(R^-,R^+) \subseteq R^+ \Rightarrow R^- \subseteq S)\} \ .\end{aligned}$$

All we need to know about $(\lhd^-, \lhd^+)$ is that it is the least *pre-fixed* point of $\psi^{\S}$. Writing out this least pre-fixed point property for $\psi^{\S}$ on $\mathcal{R}el^{op} \times \mathcal{R}el$ purely in terms of ψ and $\mathcal{R}el$, we obtain the following characteristic properties of $\lhd^-, \lhd^+$ which have a mixed inductive/co-inductive flavour.

Lemma 5. *1.* $\lhd^- = \psi(\lhd^+, \lhd^-)$ *and* $\psi(\lhd^-, \lhd^+) = \lhd^+$.
2. For all $R^-, R^+ \in \mathcal{R}el$, *if*

$$R^- \subseteq \psi(R^+,R^-) \quad \textit{and} \quad \psi(R^-,R^+) \subseteq R^+$$

then

$$R^- \subseteq \lhd^- \quad \textit{and} \quad \lhd^+ \subseteq R^+ \ .$$

Theorem 6 (Existence of $\lhd$). *When D is a minimal invariant for the domain constructor $(\mathbb{Z} + (-) \rightarrow (-))_\perp$, the relations $\lhd^-$ and $\lhd^+$ are equal, and yield a formal approximation relation as in Definition 4.*

Proof. First note that by (3) and part 1 of Lemma 5, if $\lhd^- = \lhd^+$ then this relation is a fixed point for the operation ϕ and hence has the properties required by Definition 4.

We split the equality $\lhd^- = \lhd^+$ into two inclusions. The inclusion $\lhd^+ \subseteq \lhd^-$ follows immediately from Lemma 5, since by clause 1 we may take $R^- = \lhd^+$ and $R_+ = \lhd^-$ in clause 2. So it remains to prove that $\lhd^- \subseteq \lhd^+$.

It is only at this point that we need the minimal invariant property of D. Recall that it says that id_D is the least fixed point of the continuous function $\delta_\Phi : (D \rightarrow D) \longrightarrow (D \rightarrow D)$ defined in (2). We introduce the following piece of notation: given $R, S \in \mathcal{R}el$ and a continuous function $e \in (D \rightarrow D)$, write

$$e : R \subset S$$

to mean that for all $(d, P) \in R$, $(e(d), P) \in S$. From the definition of δ_Φ, it is straightforward to verify that

$$e : R \subset S \Rightarrow \delta_\Phi(e) : \psi(S, R) \subset \psi(R, S) \ .$$

So taking $R = \lhd^-$ and $S = \lhd^+$ and using part 1 of Lemma 5, we have that δ_Φ maps the set

$$\{e \in D \rightarrow D \mid e : \lhd^- \subset \lhd^+\} \tag{4}$$

into itself. Clearly this subset of $D \rightarrow D$ is chain-closed and contains $\perp$, because of the admissibility condition elements of $\mathcal{R}el$ satisfy. Hence by the familiar fixed point induction principle (see [15, Sect. 10.2] for example), id_D, being the least fixed point of δ_Φ, lies in the subset (4). Thus $id_D : \lhd^- \subset \lhd^+$, which is just to say that $\lhd^- \subseteq \lhd^+$. □

4 Further development

The method of construction of $\lhd$ we have given in this paper can be used quite generally to construct recursively specified relations on recursively defined domains without having to delve into the details of the construction of the domain. Moreover, the construction applies to many different notions of 'relation' on a domain. (Here for example, a relation on D has meant a subset of $D \times$ *Prog.*) The construction can be phrased in terms of an abstract notion of 'relational structure' on a category of domains and of the 'action' of domain constructors on relations, due to O'Hearn and Tennent [7]. This general form of the construction is described in [9, Sect. 5]. That paper treats the case of *unary* relational structures, but the method generalizes easily to n-ary relations. For example, we believe that the recursively specified relation between two recursively defined domains employed by Reynolds to relate a direct and a continuation semantics of an untyped functional language in [12] can be constructed by applying our method to a suitable binary relational structure.

As pointed out in [9], the method of construction not only provides a simpler construction of certain relations, but also *characterizes* these relations uniquely via a 'universal property'. For instance, by virtue of Lemma 5 (and the fact that $\lhd^{-}=\lhd=\lhd^{+}$), the formal approximation relation $\lhd$ is a 'mixed' fixed point in the sense of the following definition.

Definition 7 (Mixed fixed point). Let $(\mathcal{R}, \leq)$ be a partially ordered set and let $\psi : \mathcal{R}^{\mathrm{op}} \times \mathcal{R} \longrightarrow \mathcal{R}$ be a monotonic function. Then $M \in \mathcal{R}$ is a *mixed fixed point* for ψ if

$$M = \psi(M, M) \tag{5}$$

and

$$\forall R, S \in \mathcal{R}(R \leq \psi(S, R) \wedge \psi(R, S) \leq S \Rightarrow R \leq M \leq S) \ . \tag{6}$$

Note that the mixed fixed point of ψ is unique if it exists. Indeed, if $R \in \mathcal{R}$ satisfies $R = \psi(R, R)$, then (6) implies that $R \leq M \leq R$, i.e. $R = M$.

It is not hard to see that conditions (5) and (6) are equivalent to saying that (M, M) is the least pre-fixed point of the monotonic operator

$$(R, S) \mapsto (\psi(S, R), \psi(R, S))$$

on $\mathcal{R}^{\mathrm{op}} \times \mathcal{R}$; or to saying that (M, M) is the greatest post-fixed point of that operator. In fact Definition 7 is the special case for monotonic functions of the condition on functors of mixed variance formulated by Freyd in his work on 'algebraically compact' categories [2, 3]. One can summarize the results in [9, Sect. 5] as establishing that the algebraic compactness property of the category of domains and strict continuous functions is inherited by categories of 'domains equipped with relations' (for a very general notion of relation). As the rest of that paper demonstrates, from the mixed fixed point property of recursively defined relations it is possible to derive a number of induction and co-induction [8] principles for reasoning about the properties of recursively defined domains.

References

1. P. J. Freyd. Recursive Types Reduced to Inductive Types. In *Proc. 5th Annual Symp. on Logic in Computer Science, Philadelphia, 1990* (IEEE Computer Society Press, Washington, 1990), pp 498–508.
2. P. J. Freyd. Algebraically Complete Categories. In A. Carboni *et al* (eds), *Proc. 1990 Como Category Theory Conference*, Lecture Notes in Math. Vol. 1488 (Springer-Verlag, Berlin, 1991), pp 95–104.
3. P. J. Freyd. Remarks on algebraically compact categories. In M. P. Fourman, P. T. Johnstone and A. M. Pitts (eds), *Applications of Categories in Computer Science*, L.M.S. Lecture Note Series 177 (Cambridge University Press, 1992), pp 95–106.
4. C. A. Gunter. *Semantics of Programming Languages. Structures and Techniques.* (MIT Press, 1992.)

5. A. R. Meyer. Semantical Paradigms: Notes for an Invited Lecture. In *Proc. 3rd Annual Symp. on Logic in Computer Science, Edinburgh, 1988* (IEEE Computer Society Press, Washington, 1988), pp 236–255.
6. R. E. Milne. *The formal semantics of computer languages and their implementations*, Ph.D. Thesis, Univ. Cambridge, 1973.
7. P. W. O'Hearn and R. D. Tennent. Relational Parametricity and Local Variables. In *Conf. Record 20th Symp. on Principles of Programming Languages, Charleston, 1993* (ACM, New York, 1993), pp 171–184.
8. A. M. Pitts. A Co-induction Principle for Recursively Defined Domains, *Theoretical Computer Science*, to appear. (Available as Univ. Cambridge Computer Laboratory Tech. Rept. No. 252, April 1992.)
9. A. M. Pitts. Relational Properties of Recursively Defined Domains. In: *Proc. 8th Annual Symp. on Logic in Computer Science, Montréal, 1993* (IEEE Computer Soc. Press, Washington, 1993), pp 86–97.
10. G. D. Plotkin. LCF Considered as a Programming Language, *Theoretical Computer Science* 5(1977) 223–255.
11. G. D. Plotkin. Lectures on Predomains and Partial Functions. Notes for a course at CSLI, Stanford University, 1985.
12. J. C. Reynolds. On the Relation between Direct and Continuation Semantics. In J. Loeckx (ed.), *2nd Int. Colloq. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 14 (Springer-Verlag, Berlin, 1974), pp 141–156.
13. D. S. Scott. Domains for Denotational Semantics. In M. Nielsen and E. M. Schmidt (eds), *Proc. 9th Int. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 140 (Springer, Berlin, 1982), pp 577–613.
14. M. B. Smyth and G. D. Plotkin. The Category-Theoretic Solution of Recursive Domain Equations, *SIAM J. Computing* 11(1982) 761–783.
15. G. Winskel. *The Formal Semantics of Programming Languages. An Introduction.* (MIT Press, 1993.)

A Structural Co-induction Theorem

Jan Rutten

CWI
P.O. Box 4079, 1009 AB Amsterdam
(Jan.Rutten@cwi.nl)

Abstract. The Structural Induction Theorem (Lehmann and Smyth, 1981; Plotkin, 1981) characterises initial F-algebras of locally continuous functors F on the category of cpo's with strict and continuous maps. Here a dual of that theorem is presented, giving a number of equivalent characterisations of final coalgebras of such functors. In particular, final coalgebras are order strongly-extensional (sometimes called internal full abstractness): the order is the union of all (ordered) F-bisimulations. (Since the initial fixed point for locally continuous functors is also final, both theorems apply.) Further, a similar co-induction theorem is given for a category of complete metric spaces and locally contracting functors.

1 Introduction

Consider a preorder $(P, \leq)$ and a monotone function $f : P \rightarrow P$. An element $q \in P$ is a post-fixed point of f (also called f-consistent) if $q \leq f(q)$. If the collection of post-fixed points of f has a largest element, then this is also the greatest fixed point of f. Defining p as the greatest post-fixed point of f is sometimes called *a co-inductive definition.* (A typical example is a complete lattice $(P, \subseteq)$ and a monotone function f, which by Tarski's fixed-point theorem has a greatest (post-)fixed point.) Being the greatest post-fixed point can also be used as a proof method: in order to establish $q \leq p$, for $q \in P$, it is sufficient to prove $q \leq f(q)$. This fact is sometimes called a *co-induction principle.*

A familiar example in computer science is the co-inductive definition of the bisimilarity relation on a labelled transition system. It is defined as the greatest fixed point of a monotone function on the lattice of relations on the states of this transition system (see [Mil89]). An example of the above co-induction proof principle can be found in [MT91], where it is used to prove the consistency of the static and the dynamic semantics of a simple functional programming language with recursive functions.

By generalizing preorders to categories $\mathcal{C}$ and monotone functions to functors $F : \mathcal{C} \rightarrow \mathcal{C}$, a co-induction principle can be obtained for recursive data types, which are often defined as fixed points. Post-fixed points of F are F-coalgebras (A, α), and consist of an object A in $\mathcal{C}$ together with an arrow $\alpha : A \rightarrow F(A)$ (generalizing $\leq$). These F-coalgebras form again a category, as the post-fixed points of a monotonic function form a preorder. Arrows between two F-coalgebras (A, α) and (B, β) are arrows $f : A \rightarrow B$ (in $\mathcal{C}$) such that $\beta \circ f = F(f) \circ \alpha$. A greatest post-fixed point for a functor F is a *final* F-coalgebra (A, α): for any other F-coalgebra (B, β) there exists a unique arrow $f : (B, \beta) \to (A, \alpha)$. If (A, α) is a final F-coalgebra then A is a fixed point of F (i.e., α is an isomorphism).

As will become apparent, the richer structure of categories allows for a number of different formulations of a co-induction principle for final coalgebras of functors. For instance, let (A, α) and (B, β) be F-coalgebras, and suppose that (A, α) is final. The following can be easily proved. For any $\pi : (A, \alpha) \to (B, \beta)$: if π is epi then π is an isomorphism (cf. [Smy92]). Note that this generalizes the fact that for an ordered set $(P, \leq)$ and a monotone function $f : P \to P$: if $p, q \in P$, with p the greatest post-fixed point of f and $q \geq p$, then $q \leq f(q)$ implies $p = q$—another formulation of the co-induction principle mentioned above.

In particular, *locally continuous* (endo-)functors on the category of complete partial orders will be investigated. These functors are well-known to have an initial F-algebra (see [SP82]), which is at the same time a final F-coalgebra. A structural co-induction theorem will be proved, giving a number of equivalent characterizations for such final F-coalgebras. Maybe the most surprising and interesting one is the equivalence between finality and so-called *order strong-extensionality*, stating that two elements are ordered if and only if they are related by a so-called *ordered bisimulation*. Order-bisimulations generalize the F-bisimulations of [AM89], which at their turn are categorical abstractions of the notion of bisimulation of [Par81, Mil89]. In the present paper, the definition of ordered bisimulation from [Fio93] is used, which generalizes the original definition from [RT93] by the use of lax-homomorphisms.

The co-induction theorem (Section 5) is presented as and named after a dualization of the *structural induction theorem* of [Plo81] (but see also [LS81]), which is repeated here in the Appendix. Part of this dualization is fairly straightforward; order strong-extensionality, however, does not arise as the dual of the structural induction principle for ω-inductive sets (clause (3) of the induction theorem), nor do the corresponding parts of the proof. Note that because initial algebras of locally continuous functors are also final, both the induction and the co-induction theorem apply to them.

In Section 6, the co-induction theorem is used to extend the final semantics approach of [RT93] (initiated in [Acz88]) to the ordered case: the unique arrow from a coalgebra to a final coalgebra is shown to preserve and reflect the bisimulation order. The paper is concluded by proving, in Section 7, a slightly adapted version of the co-induction theorem for a category of *metric spaces* and locally contracting functors, in very much the same way. This last result is illustrated by the description of a metric hyperuniverse.

2 Preliminaries

Let C be a category and $F : C \to C$ be a functor from C to C. An *F-coalgebra* is a pair (A, α), consisting of an object A and an arrow $\alpha : A \to F(A)$ in C. It is dual to the notion of *F-algebra*: an *F-algebra* is a pair (A, α), consisting of an object A and an arrow $\alpha : F(A) \to A$ in C.

For instance, any preorder $(P, \leq)$ is a category (with an arrow between two elements iff they are order related) and post-fixed points of monotone functions $f : P \longrightarrow P$ are examples of f-coalgebras.

The collection of *F-coalgebras* constitutes a category by taking as arrows between coalgebras (A, α) and (B, β) those arrows $f : A \longrightarrow B$ in $\mathcal{C}$ such that $\beta \circ f = F(f) \circ \alpha$; that is, the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "\alpha"'] \arrow[dr, phantom, "*"] & B \arrow[d, "\beta"] \\
F(A) \arrow[r, "F(f)"'] & F(B)
\end{tikzcd}$$

Such an arrow f from (A, α) to (B, β) is called a homomorphism of F-coalgebras.

For example, a graph $(N, \longrightarrow)$, consisting of a set N of nodes and a collection $\longrightarrow$ of (directed) arcs between nodes can be regarded as a coalgebra of the (covariant) powerset functor $\mathcal{P}$ on the category *Set* of sets as follows: define *child* : $N \longrightarrow \mathcal{P}(N)$ by, for all $n \in N$, $child(n) \equiv \{m \mid n \longrightarrow m\}$. Arrows between graphs (as coalgebras) are those mappings between the sets of nodes that respect the child relation.

Definition 1. An object A in $\mathcal{C}$ is called *final* if for any other object B in $\mathcal{C}$ there exists a unique arrow from B to A. It is the dual notion of initial object (unique arrow *from* the object). Final and initial objects are unique up to isomorphism. □

The following is standard (see, e.g., [SP82]).

Proposition 2. *Every final F-coalgebra (A, α) is a fixed point of F (that is, α is an isomorphism).* □

3 Coalgebras in $CPO_\perp$

Let $CPO_\perp$ be the category with complete partial orders $(D, \sqsubseteq_D)$ as objects and strict and continuous functions as arrows. For any two cpo's D and E, the set hom(D, E) of arrows between D and E is itself a cpo, with the usual order: for all $f, g \in \text{hom}(D, E)$,

$$f \leq g \equiv \forall x \in D,\ f(x) \sqsubseteq_E g(x).$$

Moreover composition of arrows is continuous with respect to this ordering. Therefore the category $CPO_\perp$ is called an order-enriched (or **O**-) category ([SP82]).

The structure on hom sets can be used to characterize a class of functors.

Definition 3. A functor $F : CPO_\perp \to CPO_\perp$ is *locally continuous* if, for any two objects $D, E \in CPO_\perp$, the mapping

$$F_{D,E} : \hom(D,E) \to \hom(F(D), F(E))$$

is continuous. Similarly, F is *locally monotonic* if $F_{D,E}$ is monotonic. □

Next we recall the definition of the subcategory CPO^E of $CPO_\perp$. If D and D' are cpo's and $\mu^e : D \to D'$ and $\mu^p : D' \to D$ are arrows in $CPO_\perp$ then $\langle \mu^e, \mu^p \rangle$ is called an *embedding-projection* pair from D to D' provided that

$$\mu^p \circ \mu^e = id_D \text{ and } \mu^e \circ \mu^p \leq_{\hom(D',D')} id_{D'}.$$

Note that the one half of such a projection pair determines the other. Let CPO^E denote the subcategory of $CPO_\perp$ that has cpo's as objects and embedding-projection pairs as arrows. Note that also CPO^E is an order-enriched category. The following theorem is standard (see [SP82]).

Theorem 4. *Every $F : CPO_\perp \to CPO_\perp$ that is locally continuous can be extended to a functor $F^E : CPO^E \to CPO^E$ that is ω-continuous (preserving colimits of ω-chains): on objects F^E is identical to F; and on arrows, F^E is given by*

$$F^E(\langle \mu^e, \mu^p \rangle) \equiv \langle F(\mu^e), F(\mu^p) \rangle.$$

A fixed point of F is obtained by constructing an initial F^E-algebra D in CPO^E as the colimit of the ω-chain $(D_n, \alpha_n)_n$, given by $D_0 \equiv \{\perp\}$, the trivial embedding $\alpha_0 : D_0 \to F(D_0)$, and for all $n \geq 0$, $D_{n+1} \equiv F(D_n)$, $\alpha_{n+1} \equiv F(\alpha_n)$. □

This fixed point D is an initial F^E-algebra (D, i^{-1}) in the category CPO^E. Moreover, it can also be seen to be an initial F-algebra in $CPO_\perp$: the fact that D is a colimit (of its defining chain) in CPO^E implies, by a little exercise (Exercise 4.17 from [Plo81]—to be precise), that it is a colimit in $CPO_\perp$ as well; then the 'Basic Lemma', from [SP82], immediately yields the result. By the so-called "limit-colimit coincidence" for O-categories, which is extensively discussed in [SP82], the dual of these facts also holds: Let CPO^P be defined as $(CPO^E)^{op}$, the opposite category of CPO^E. Thus arrows in CPO^P are mappings μ^p for which there exists a (unique) μ^e such that $\langle \mu^e, \mu^p \rangle$ is an embedding-projection pair. The fact that (D, i^{-1}) is an initial F^E-algebra (in CPO^E) implies that (D, i) is a final F^P-coalgebra in CPO^P. (Here F^P is defined analogously to F^E.) Again, (D, i) is a final F-coalgebra in $CPO_\perp$ as well, which can be shown by dualizing the little argument above. Summarizing, we have the following.

Theorem 5. *Let $F : CPO_\perp \to CPO_\perp$ be a locally continuous functor and let (D, i^{-1}) be the (in CPO^E) initial F^E-algebra as described above. Then (D, i) is a final F^P-coalgebra in CPO^P as well as a final F-coalgebra in $CPO_\perp$.* □

4 Ordered F-bisimulation

In [AM89], a categorical generalization of the notion of *bisimulation* of [Par81, Mil89] has been given in terms of coalgebras of functors on a category of classes. In [RT93], this definition is extended to functors F on arbitrary categories, yielding the notion of F-bisimulation. The order on hom sets in the category $CPO_\perp$ makes the following generalization of that definition possible. Let for the rest of this section $F : CPO_\perp \to CPO_\perp$ be a functor.

Definition 6. Let (A, α) be an F-coalgebra and R a relation on A with projections $\pi_1, \pi_2 : R \to A$. (That is, $R \subseteq A \times A$ is a cpo $(R, \sqsubseteq_R)$ such that the inclusion function $i : R \to A \times A$ is continuous.) Then R is called an *ordered F-bisimulation* on (A, α) if there exists an arrow $\beta : R \to F(R)$ such that

$$\begin{tikzcd}
R \arrow[r, "\pi_1"] \arrow[d, "\beta"'] \arrow[dr, phantom, "\geq"] & A \arrow[d, "\alpha"] \arrow[dr, phantom, "*"] & R \arrow[l, "\pi_2"'] \arrow[d, "\beta"] \\
F(R) \arrow[r, "F(\pi_1)"'] & F(A) & F(R) \arrow[l, "F(\pi_2)"]
\end{tikzcd}$$

That is, π_2 is a homomorphism of coalgebras (satisfying $F(\pi_2) \circ \beta = \alpha \circ \pi_2$), and π_1 is a so-called *lax-homomorphism: it satisfies $F(\pi_1) \circ \beta \geq \alpha \circ \pi_1$.* □

The above definition is from [Fio93] and generalizes an earlier definition of ordered bisimulation given in [RT93], which required the existence of two coalgebra mappings $\beta_1, \beta_2 : R \to F(R)$ such that $\beta_1 \leq \beta_2$ and both π_1 and π_2 are coalgebra homomorphisms. The latter can be seen to be a special instance of the definition given above by taking $\beta \equiv \beta_2$. (Cf. the notion of simulation in [Pit92]; see also [Pit93], where proof principles that combine induction and co-induction are studied.)

The following definition generalizes the notion of *strong extensionality* used in [Acz88] (in the context of non-well-founded set theory). It is sometimes called *internal full abstractness* (cf. [Abr91]).

Definition 7. Let (A, α) be an F-coalgebra, and let $\sqsubseteq_A$ be the order on A. Let $\sqsubseteq^F \subseteq A \times A$ be defined by

$$\sqsubseteq^F \equiv \bigcup \{ R \subseteq A \times A \mid R \text{ is an ordered } F\text{-bisimulation on } (A, \alpha) \}.$$

Elements $a, b \in A$ with $a \sqsubseteq^F b$ are called (ordered) F-bisimilar. Now (A, α) is called *order strongly-extensional* if, for all $a, b \in A$,

$$a \sqsubseteq_A b \Leftrightarrow a \sqsubseteq^F b.$$

□

Example 1. A *deterministic partial transition system* is a pair $(S, \rightarrow)$ consisting of a set S of states and a transition relation $\rightarrow \subseteq S \times S$ that is a partial function. We assume that S contains a minimal element $\perp_S$ and is otherwise discretely ordered. Furthermore we assume that $\{s \in S \mid \perp_S \rightarrow s\} = \emptyset$.

Such transition systems can be represented as coalgebras of the functor $(\cdot)_\perp : CPO_\perp \rightarrow CPO_\perp$, which maps a cpo D to its lifted version $(D)_\perp$ by extending D with a new minimal element $\perp_{\text{new}}$. For $(S, \rightarrow)$, define $\alpha : S \rightarrow (S)_\perp$, for $s \in S$, by

$$\alpha(s) = \begin{cases} s' & \text{if } s \rightarrow s' \\ \perp_{\text{new}} & \text{otherwise.} \end{cases}$$

An ordered $(\cdot)_\perp$-bisimulation (R, β) on (S, α),

$$\begin{tikzcd}
R \arrow[r, "\pi_1"] \arrow[d, "\beta"'] \arrow[dr, phantom, "\geq"] & S \arrow[d, "\alpha"] \arrow[dr, phantom, "*"] & R \arrow[l, "\pi_2"'] \arrow[d, "\beta"] \\
(R)_\perp \arrow[r, "(\pi_1)_\perp"'] & (S)_\perp & (R)_\perp \arrow[l, "(\pi_2)_\perp"]
\end{tikzcd}$$

satisfies for all $s, t \in S$ with $s \, R \, t$, and for all $s' \in S$,

$$\text{if } s \rightarrow s' \text{ then } \exists t' \in S,\ t \rightarrow t' \text{and } s' \, R \, t'.$$

Two states s and t in S are bisimilar whenever the number of subsequent transition steps that can be taken from t is at least as big as the number of steps that are possible starting from s. If β would be such that also π_1 is a coalgebra homomorphism, then two states are bisimilar if they can take the same number of steps. $\square$

Example 2. A *nondeterministic transition system with divergence* is a triple

$$(S, \rightarrow, \uparrow)$$

consisting of a set S of states, a transition relation $\rightarrow \subseteq S \times S$, and a divergence set $\uparrow \subseteq S$. (This is the—for simplicity—unlabelled version of the transition systems with divergence considered in [Abr91].) One should think of states s in $\uparrow$ (notation: $s \uparrow$) as having the possibility of divergence. Similarly $s \downarrow$ is used to indicate that s converges, that is, s not in $\uparrow$.

As above, we assume that S has a minimal element $\perp_S$, satisfying now $\{s \in S \mid \perp_S \rightarrow s\} = \emptyset = \{s \in S \mid s \rightarrow \perp_S\}$ (so $\perp_S$ is not involved in any transitions) and in addition $\perp_S \uparrow$. We shall only consider transition systems that are *finitely branching*, i.e., for all $s \in S$, the set $\{s' \in S \mid s \rightarrow s'\}$ is finite.

Transition systems with divergence can be represented as coalgebras of the functor $\mathcal{P} : CPO_\perp \rightarrow CPO_\perp$, which takes a cpo D to the Plotkin powerdomain of its lifted version $(D)_\perp$, extended (as in [Abr91]) with the empty set. In the ordering of $\mathcal{P}(D)$, the empty set is greater than the bottom element $\{\bot_{\text{new}}\}$, and incomparable to all other elements; non-empty sets $X, Y \in \mathcal{P}(D)$ are ordered as usual by the Egli-Milner order. For $(S, \rightarrow, \uparrow)$ define $\alpha : S \rightarrow \mathcal{P}(S)$ by, for all $s \in S$,

$$\alpha(s) \equiv \{s' \in S \mid s \rightarrow s'\} \cup \{\bot_{\text{new}} \in (S)_\bot \mid s \uparrow\}.$$

An ordered $\mathcal{P}$-bisimulation (R, β) on (S, α),

$$\begin{tikzcd}
R \arrow[r, "\pi_1"] \arrow[d, "\beta"'] \arrow[dr, phantom, "\geq_1"] & S \arrow[d, "\alpha"] \arrow[dr, phantom, "*_2"] & R \arrow[l, "\pi_2"'] \arrow[d, "\beta"] \\
\mathcal{P}(R) \arrow[r, "\mathcal{P}(\pi_1)"'] & \mathcal{P}(S) & \mathcal{P}(R) \arrow[l, "\mathcal{P}(\pi_2)"]
\end{tikzcd}$$

satisfies for all $s, t \in S$ with sRt, and for all $s', t' \in S$,

if $s \rightarrow s'$ then $\exists t' \in S,\ t \rightarrow t'$ and $s'Rt'$;

if $s \downarrow$ then ($t \downarrow$ and if $t \rightarrow t'$ then $\exists s' \in S,\ s \rightarrow s'$ and $s'Rt'$).

(Relations satisfying these two conditions are called partial bisimulations in [Abr91].) For suppose sRt and $s \rightarrow s'$. By the definition of α, $s' \in \alpha(s) = \alpha \circ \pi_1(s,t)$, and because of $\geq_1$, also $s' \in \mathcal{P}(\pi_1)(\beta((s,t)))$. Thus there exists $t' \in S$ with $(s', t') \in \beta((s,t))$, satisfying $s'Rt'$; $*_2$ implies $t' \in \alpha(t)$ whence $t \rightarrow t'$.

Next suppose $s \downarrow$. Thus $\bot_{\text{new}} \notin \alpha(s)$ and hence $\bot_{\text{new}} \notin \mathcal{P}(\pi_1)(\beta((s,t)))$, by $\geq_1$ and the definition of the Egli-Milner order. By the definition of $\mathcal{P}(\pi_1)$ it follows that $\bot_{\text{new}} \notin \beta((s,t))$ (since for any $X \subseteq (S)_\bot$, $\mathcal{P}(\pi_1)(X)$ contains $\bot_{\text{new}}$ iff X does). Thus by $*_2$, $\alpha(t)$ does not contain $\bot_{\text{new}}$, that is, $t \downarrow$. Further suppose $t \rightarrow t'$. By $*_2$, there is $s' \in S$ with $(s', t') \in \beta((s,t))$. By $\geq_1$ and the fact that $\alpha(s)$ does not contain $\bot_{\text{new}}$ (nor $\bot_S$), it follows that $s' \in \alpha(s)$, thus $s \rightarrow s'$.

Conversely, any relation $R \subseteq S \times S$ (not involving $\bot_S$) satisfying the two above conditions can be turned into a $\mathcal{P}$-coalgebra (T, β) by defining

$$T \equiv R \cup (\{\bot_S\} \times S)$$

and $\beta : T \rightarrow \mathcal{P}(T)$ by, for all sTt,

$$\begin{aligned}
\beta((s,t)) \equiv\ & \{(s', t') \in T \mid s \rightarrow s' \text{ and } t \rightarrow t' \text{ and } s'Rt'\} \\
\cup\ & \{(\bot_S, t') \in T \mid s \uparrow \text{ and } t \rightarrow t'\} \\
\cup\ & \{\bot_{\text{new}} \in (T)_\bot \mid s \uparrow \text{ and } t \uparrow\}
\end{aligned}$$

It is left to the reader to verify that (T, β) is an ordered $\mathcal{P}$-bisimulation. □

5 A structural co-induction theorem

Next we formulate and prove the main theorem of this paper. (The definitions of some of the categorical and order-theoretic notions used here, can be found in the Appendix.)

Theorem 8. *Let $F : CPO_\perp \to CPO_\perp$ be a locally continuous functor. Let (A, α) be an F-coalgebra. Then of the following six statements, (1), (2), (2'), (4) and (5) are equivalent and all imply (3). If F moreover weakly preserves ordered kernel pairs then all statements are equivalent.*

1. *(A, α) is a final F-coalgebra.*
2. *α is epi; and for any F-coalgebra (B, β) and coalgebra homomorphism $e : (A, \alpha) \to (B, \beta)$: if e is epi then it is an isomorphism:*

$$\begin{tikzcd} A \arrow[r, "e"] \arrow[d, "\alpha"'] \arrow[dr, phantom, "*"] & B \arrow[d, "\beta"] \\ F(A) \arrow[r, "F(e)"'] & F(B) \end{tikzcd}$$

2'. *As 2., but with* epi *replaced by* dense-epi, *twice.*

3. *α is dense-epi and (A, α) is order strongly-extensional; that is, if $\sqsubseteq_A$ is the order on A then*

$$\sqsubseteq_A = \bigcup\{R \subseteq A \times A \mid \ R \textit{ is an ordered } F\textit{-bisimulation on } (A, \alpha)\ \}.$$

4. *α is an isomorphism and $1_A = \mu h.\ \alpha^{-1} \circ F(h) \circ \alpha$ (the least fixed point).*
5. *(A, α) is maximally-final: it is a final F-coalgebra and for any F-coalgebra (B, β) the unique coalgebra homomorphism $e : (A, \alpha) \to (B, \beta)$ is maximal among the lax-homomorphisms between (A, α) and (B, β); that is, for any $f : B \to A$, if $\alpha \circ f \le F(f) \circ \beta$ then $f \le e$.*

Schematically:

$$1 \Leftrightarrow 2 \Leftrightarrow 2' \Leftrightarrow 4 \Leftrightarrow 5 \Rightarrow 3,$$

$$3 + F \textit{ weakly preserves ordered kernel pairs } \Rightarrow 2'.$$

Proof:
(1) $\Rightarrow$ (2): By Proposition 2, α is an isomorphism and hence epi. Consider an epi $e : A \to B$ and suppose $e : (A, \alpha) \to (B, \beta)$ is a coalgebra homomorphism. Since (A, α) is final there exists a unique $h : (B, \beta) \to (A, \alpha)$. Thus both 1_A and $h \circ e$ are arrows from (A, α) to itself. By finality $h \circ e = 1_A$. From

$$\begin{aligned} (e \circ h) \circ e &= e \circ (h \circ e) \\ &= e \circ 1_A \\ &= 1_B \circ e \end{aligned}$$

and the fact that e is epi, it follows that $e \circ h = 1_B$.

(2) $\Rightarrow$ (1) : First we observe that α is an isomorphism, which follows from applying (2) to the following diagram (note that here the fact is used that α is epi):

$$\begin{tikzcd}
A \arrow[r, "\alpha"] \arrow[d, "\alpha"'] \arrow[dr, phantom, "*"] & F(A) \arrow[d, "F(\alpha)"] \\
F(A) \arrow[r, "F(\alpha)"'] & F(F(A))
\end{tikzcd}$$

Let (D, i^{-1}) be the initial F-algebra from Theorem 4. We saw (Theorem 5) that (D, i) is a final F^P-coalgebra (in CPO^P). Since α is an isomorphism it is also a projection, hence there exists a projection $e : A \to D$, which (by the construction of D) is also an arrow of coalgebras $e : (A, \alpha) \to (D, i)$. Now every projection is epi and by applying (2), e can be seen to be an isomorphism. Because (D, i) is a final F-coalgebra in $CPO_\perp$—again by Theorem 5—and (A, α) and (D, i) are isomorphic coalgebras, it follows that also (A, α) is a final F-coalgebra.

(1) $\Leftrightarrow$ (2′) : Inspection of the above two implications tells us that their proofs remain valid when epi is replaced by dense-epi.

(1) $\Rightarrow$ (4) : The finality of (A, α) implies that α is an isomorphism. Since F is locally continuous the function $\lambda h.\alpha^{-1} \circ F(h) \circ \alpha$ is continuous. Define $g \equiv \mu h.\ \alpha^{-1} \circ F(h) \circ \alpha$. It is immediate that $\alpha \circ g = F(g) \circ \alpha$, thus $g : (A, \alpha) \to (A, \alpha)$. By finality, $g = 1_A$.

(4) $\Rightarrow$ (2) : Since α is an isomorphism it is also epi. Consider an epi $e : (A, \alpha) \to (B, \beta)$. We prove that e is an isomorphism. Let $g \equiv \mu h.\ \alpha^{-1} \circ F(h) \circ \beta$. Then $\alpha \circ g = F(g) \circ \beta$, and we have the following diagram:

$$\begin{tikzcd}
B \arrow[r, "g"] \arrow[d, "\beta"'] \arrow[dr, phantom, "*"] & A \arrow[r, "e"] \arrow[d, "\alpha"] \arrow[dr, phantom, "*"] & B \arrow[d, "\beta"] \\
F(B) \arrow[r, "F(g)"'] & F(A) \arrow[r, "F(e)"'] & F(B)
\end{tikzcd}$$

Next we show that $g \circ e = 1_A$ from which it follows—as in the proof of "(1) $\Rightarrow$ (2)"—that $e \circ g = 1_B$, using the fact that e is epi. First we prove $g \circ e \leq 1_A$, using the fixed-point definition of g:

- $(\lambda b \in B.\ \perp_A) \circ e = \lambda a \in A.\ \perp_A \leq 1_A$.
- Suppose $g \circ e \leq 1_A$, then

$$\begin{aligned}
\alpha^{-1} \circ F(g) \circ \beta \circ e &= \alpha^{-1} \circ F(g) \circ F(e) \circ \alpha \\
&= \alpha^{-1} \circ F(g \circ e) \circ \alpha \\
&\leq 1_A
\end{aligned}$$

since, by assumption, $g \circ e \leq 1_A$, and the facts that α is an isomorphism and F is locally (continuous and hence) monotonic.

Next we shall use $1_A = \mu h.\ \alpha^{-1} \circ F(h) \circ \alpha$ from (4) to prove $1_A \leq g \circ e$:

- $\lambda a \in A.\ \bot_A \leq g \circ e$.
- Suppose $h \leq g \circ e$. Then

$$\begin{aligned}\alpha^{-1} \circ F(h) \circ \alpha \leq\ & (\text{since } F \text{ is locally monotonic})\\ & \alpha^{-1} \circ F(g \circ e) \circ \alpha\\ =\ & \alpha^{-1} \circ F(g) \circ F(e) \circ \alpha\\ =\ & \alpha^{-1} \circ F(g) \circ \beta \circ e\\ =\ & \alpha^{-1} \circ \alpha \circ g \circ e\\ =\ & g \circ e.\end{aligned}$$

(1) $\Rightarrow$ (5) : Let $f : (B, \beta) \to (A, \alpha)$ be a lax-homomorphism. By Proposition 2, α is an isomorphism. Define a sequence of functions from B to A inductively by

$$\begin{aligned} e_0 &\equiv f,\\ e_{n+1} &\equiv \alpha^{-1} \circ F(e_n) \circ \beta.\end{aligned}$$

Then $(e_n)_n$ is a chain ($f \leq \alpha^{-1} \circ F(f) \circ \beta$ because f is a lax-homomorphism) and its least upperbound e satisfies

$$\begin{aligned} e &= \bigsqcup e_n\\ &= \bigsqcup \alpha^{-1} \circ F(e_n) \circ \beta\\ &= (\text{by local continuity of } F)\\ &\quad \alpha^{-1} \circ F(\bigsqcup e_n) \circ \beta\\ &= \alpha^{-1} \circ F(e) \circ \beta.\end{aligned}$$

Hence e is the unique coalgebra homomorphism from (B, β) to (A, α). It follows from the definition of e that $f \leq e$.
(5) $\Rightarrow$ (1) : trivial.
(4) $\Rightarrow$ (3) : The fact that α is an isomorphism implies that it is dense-epi. We have to show that

$$\sqsubseteq_A = \bigcup \{ R \subseteq A \times A \mid R \text{ is an ordered } F\text{-bisimulation on } (A, \alpha) \}.$$

The inclusion from left to right follows from the fact that $\sqsubseteq_A$ is an ordered F-bisimulation on (A, α): First observe that $\sqsubseteq_A$, with the inherited order from $A \times A$, is a cpo. Next define $\Delta : A \to \sqsubseteq_A$ by, for all $a \in A$, $\Delta(a) \equiv < a, a >$ and $\beta : \sqsubseteq_A \to F(\sqsubseteq_A)$ by

$$\beta \equiv F(\Delta) \circ \alpha \circ \pi_2.$$

Then $(\sqsubseteq_A, \beta)$ is an ordered F-bisimulation on (A, α):

$$\begin{tikzcd}
\sqsubseteq_A \arrow[r, shift left, "\pi_1"] \arrow[d, "\beta"'] \arrow[dr, phantom, "\geq"] & A \arrow[l, shift left, "\Delta"] \arrow[r, shift right, "\Delta"'] \arrow[d, "\alpha"] \arrow[dr, phantom, "*"] & \sqsubseteq_A \arrow[l, shift right, "\pi_2"'] \arrow[d, "\beta"] \\
F(\sqsubseteq_A) \arrow[r, shift right, "F(\pi_1)"'] & F(A) \arrow[l, shift right, "F(\Delta)"'] \arrow[r, shift left, "F(\Delta)"] & F(\sqsubseteq_A) \arrow[l, shift left, "F(\pi_2)"]
\end{tikzcd}$$

since

$$\begin{aligned}
\alpha \circ \pi_1 &= (\text{because } \pi_1 \circ \Delta = 1_A) \\
&\quad F(\pi_1 \circ \Delta) \circ \alpha \circ \pi_1 \\
&\leq F(\pi_1) \circ F(\Delta) \circ \alpha \circ \pi_2 \\
&= F(\pi_1) \circ \beta,
\end{aligned}$$

and

$$\begin{aligned}
\alpha \circ \pi_2 &= F(\pi_2 \circ \Delta) \circ \alpha \circ \pi_2 \\
&= F(\pi_2) \circ \beta.
\end{aligned}$$

Conversely, consider an ordered F-bisimulation (R, β) on (A, α):

$$\begin{tikzcd}
R \arrow[r, "\pi_1"] \arrow[d, "\beta"'] \arrow[dr, phantom, "\geq"] & A \arrow[d, "\alpha"] \arrow[dr, phantom, "*"] & R \arrow[l, "\pi_2"'] \arrow[d, "\beta"] \\
F(R) \arrow[r, "F(\pi_1)"'] & F(A) & F(R) \arrow[l, "F(\pi_2)"]
\end{tikzcd}$$

We prove $R \subseteq \sqsubseteq_A$ or rather, equivalently, $\pi_1 \leq \pi_2$. We use fixed-point induction on 1_A (which by (4) is equal to $\mu h.\ \alpha^{-1} \circ F(h) \circ \alpha$) to show $1_A \circ \pi_1 \leq \pi_2$:

- $(\lambda a \in A.\ \bot_A) \circ \pi_1 \leq \pi_2$.
- Suppose $h \circ \pi_1 \leq \pi_2$. Then

$$\begin{aligned}
\alpha^{-1} \circ F(h) \circ \alpha \circ \pi_1 &\leq \alpha^{-1} \circ F(h) \circ F(\pi_1) \circ \beta \\
&= \alpha^{-1} \circ F(h \circ \pi_1) \circ \beta \\
&\leq (\text{because } h \circ \pi_1 \leq \pi_2 \text{ and } F \text{ is locally monotonic}) \\
&\quad \alpha^{-1} \circ F(\pi_2) \circ \beta \\
&= \alpha^{-1} \circ \alpha \circ \pi_2 \\
&= \pi_2
\end{aligned}$$

(3) ⇒ (2′) : We prove this implication, from which the equivalence of (1) – (5) follows, under the assumption that F weakly preserves ordered kernel pairs.

By assumption α is dense-epi. Consider a homomorphism of coalgebras $e : (A,\alpha) \to (B,\beta)$ and suppose e is dense-epi. We shall prove that e is an isomorphism. Define

$$R_e \equiv \{(a,a') \in A \times A \mid e(a) \sqsubseteq e(a')\}.$$

The continuity and the strictness of e imply that R_e is a cpo. Below it is shown that it can be extended to an F-coalgebra (R_e,γ), such that (R_e,γ) is an ordered F-bisimulation on (A,α). Then from the order strong-extensionality of (A,α) it follows that $R_e \subseteq \sqsubseteq_A$. Hence e is a strict order-monic and since e is also dense-epi, it is an isomorphism (see the Appendix).

For the existence of an arrow $\gamma : R_e \to F(R_e)$ the assumption that F weakly preserves ordered kernel pairs will be used.

$$\begin{tikzcd}
R_e \arrow[rr, "\pi_2"] \arrow[dr, "\pi_1"'] \arrow[dd, "\exists\gamma"'] & & A \arrow[dr, "e"] \arrow[dd, "\alpha"'] & \\
& A \arrow[rr, "e"] \arrow[dd, "\alpha"'] & & B \arrow[dd, "\beta"] \\
F(R_e) \arrow[rr, "F(\pi_2)"] \arrow[dr, "F(\pi_1)"'] & & F(A) \arrow[dr, "F(e)"'] & \\
& F(A) \arrow[rr, "F(e)"'] & & F(B)
\end{tikzcd}$$

Since (R_e,π_1,π_2) is an ordered kernel pair for e, $(F(R_e),F(\pi_1),F(\pi_2))$ is by assumption a weak ordered kernel pair for $F(e)$. Now

$$\begin{aligned}
F(e)\circ\alpha\circ\pi_1 &= \beta\circ e\circ\pi_1 \\
&\leq \beta\circ e\circ\pi_2 \\
&= F(e)\circ\alpha\circ\pi_2,
\end{aligned}$$

from which the existence of an arrow $\gamma : R_e \to F(R_e)$, with $\alpha\circ\pi_1 \leq F(\pi_1)\circ\gamma$ and $\alpha\circ\pi_2 = F(\pi_2)\circ\gamma$ follows. Thus R_e is an ordered F-bisimulation. □

The fact that the final F-coalgebra (D,i) from Theorem 4 is order strongly-extensional was already proved in [RT93]. (The proof given there makes explicit

use of the way D is constructed (as the projective limit of its defining ω-chain).) The equivalence of finality and maximal-finality ((1) and (5)) is due to [Plo91].

The main contribution of the above theorem is the proof of (3) $\Rightarrow$ (2), showing—for functors that weakly preserve ordered kernel pairs—that coalgebras are final if they are strongly extensional. Most functors (lifting, sum and so on) weakly preserve ordered kernel pairs.

Note that for locally continuous functors on $CPO_\perp$ there always exists an arrow from any F-coalgebra to an F-coalgebra (A, α) for which α is an isomorphism. For such functors, therefore, a final coalgebra is completely determined by the uniqueness part in the definition of finality. This explains why order strong-extensionality can be shown to be equivalent to finality.

Clearly, the clauses (1), (2) and (4) are fairly straightforward dualizations of the corresponding clauses in Plotkin's induction theorem (repeated here as Theorem 10 in the appendix). The proofs of the equivalence of (1) and (2), and of the implications (1) $\Rightarrow$ (4) and (4) $\Rightarrow$ (2) are immediate from the corresponding parts in the proof of the induction theorem. Clause (3) above cannot be seen as a dualisation of any of the clauses of Theorem 10. For a further remark on this poin see Section 8.

6 Ordered final semantics

Final coalgebras are furthermore characterized by the following theorem, which shows that they present a natural way of modelling bisimulation.

Theorem 9. *Let $F : CPO_\perp \to CPO_\perp$ be a locally continuous functor, and suppose that F weakly preserves ordered kernel pairs. Let (A, α) be a final F-coalgebra and let $f : (B, \beta) \to (A, \alpha)$ be a coalgebra homomorphism (which is unique by finality of (A, α)). For all $b, b' \in B$,*

$$b \sqsubseteq^F b' \Leftrightarrow f(b) \sqsubseteq_A f(b').$$

Proof:
From left to right: consider $b, b' \in B$ with $b \sqsubseteq^F b'$. Let (R, γ) be an ordered F-bisimulation on (B, β) with bRb'. From

$$\begin{tikzcd}
R \arrow[r, "\pi_1"] \arrow[d, "\gamma"'] \arrow[dr, phantom, "\geq"] & B \arrow[r, "f"] \arrow[d, "\beta"] \arrow[dr, phantom, "*"] & A \arrow[d, "\alpha"] \arrow[dr, phantom, "*"] & B \arrow[l, "f"'] \arrow[d, "\beta"] \arrow[dr, phantom, "*"] & R \arrow[l, "\pi_2"'] \arrow[d, "\gamma"] \\
F(R) \arrow[r, "F(\pi_1)"'] & F(B) \arrow[r, "F(f)"'] & F(A) & F(B) \arrow[l, "F(f)"] & F(R) \arrow[l, "F(\pi_2)"]
\end{tikzcd}$$

it follows that $f \circ \pi_1$ is a lax-homomorphism from (R, γ) to (A, α) and that $f \circ \pi_2$ is the (by finality of (A, α)) unique coalgebra homomorphism from (R, γ)

to (A,α). It follows from Theorem 8 (clause (5)) that $f \circ \pi_1 \leq f \circ \pi_2$. Thus $f(b) \sqsubseteq_A f(b')$.

As in the proof of (3) $\Rightarrow$ (2′) in Theorem 8, it can be shown that the ordered kernel pair

$$R_f \equiv \{(b,b') \in B \times B \mid f(b) \sqsubseteq_A f(b')\}$$

of f can be extended to an ordered F-bisimulation (R_f,γ) on (B,β) (using the fact that F weakly preserves ordered kernel pairs), from which the implication from right to left follows. □

The unique arrow $f : (B,\beta) \to (A,\alpha)$ could be called (having in mind, e.g., a transition system represented by (B,β)) the *ordered final semantics* for (B,β). Cf. the final semantics of [Acz88, RT93], where symmetric F-bisimulations are used.

The above theorem can be seen as yet another characterization of final coalgebras, since its reverse also holds: if (A,α) is an F-coalgebra such that for all coalgebra homomorphisms $f : (B,\beta) \to (A,\alpha)$ and, for all $b,b' \in B$,

$$b \sqsubseteq^F b' \Leftrightarrow f(b) \sqsubseteq_A f(b'),$$

then (A,α) is a final F-coalgebra. Take (A,α) for (B,β) and 1_A for f to see that (A,α) is order strongly-extensional (using in addition the fact that $\sqsubseteq_A$ is itself an ordered F-bisimulation); by Theorem 8, (A,α) is final.

Example 1, continued. Let N be the set of natural numbers with the usual ordering and extended with a top element ω, and let $\phi : N \to (N)_\bot$ be the obvious isomorphism. Then (N,ϕ) is a final coalgebra of the functor $(\cdot)_\bot : CPO_\bot \to CPO_\bot$. For a deterministic partial transition system $(S,\to)$, represented as a $(\cdot)_\bot$-coalgebra (S,α), the final semantics $f : (S,\alpha) \to (N,\phi)$ maps a state $s \in S$ to the natural number (possibly ω) corresponding to the number of transition steps that can be taken starting in s. □

Example 2, continued. The functor $\mathcal{P} : CPO_\bot \to CPO_\bot$, which takes a cpo D to the Plotkin powerdomain (with empty set) of $(D)_\bot$ is locally continuous (see [Plo81]) and has by Theorem 5 a final coalgebra (P,ψ). By Theorem 8, we know that (P,ψ) is order strongly-extensional, thus finding back (an "unlabelled" version of) Proposition 3.10 from [Abr91]. Since $\mathcal{P}$ can be shown to preserve weakly ordered kernel pairs, Theorem 9 applies. Thus for the final semantics $f : (S,\alpha) \to (P,\psi)$ of a nondeterministic transition system $(S,\to,\uparrow)$, represented as the $\mathcal{P}$-coalgebra (S,α), we have for all $s,t \in S$,

$$s \sqsubseteq^{\mathcal{P}} t \Leftrightarrow f(s) \sqsubseteq_P f(t),$$

sometimes called the full abstractness of f. (Similar results are obtained in [Abr91] by means of Stone duality.) □

7 Metric spaces

In [AM89], bisimulations are defined as coalgebras (R, β) (in a category of classes) for which both projections π_1 and π_2 are coalgebra homomorphisms (not only π_2). For such symmetric bisimulations, the category of complete metric spaces offers a suitable framework as well. It has been studied in great detail in [RT93]. In this section, we shall point out that the preceding co-induction theorem also applies to metric spaces, and next use the resulting theorem to prove some properties of a metric hyperuniverse.

Let *CMS* be the category with (1-bounded) complete metric spaces (D, d_D) as objects and non-expansive (non-distance-increasing) functions as arrows. (For basic facts on metric spaces see, e.g., [Eng89].) Hom sets in *CMS* are themselves complete metric spaces, using as a metric on arrows the usual pointwise extension. A functor F on *CMS* is *locally contracting* if there exists ϵ with $0 \leq \epsilon < 1$ such that, for all D, E, the mapping $F_{D,E}$ is a contraction with factor ϵ. In [RT93], it is shown (extending earlier results of [AR89]) that every locally contracting functor F has a unique fixed point which is both an initial F-algebra and a final F-coalgebra.

A 'metric version' of Theorem 8 is obtained by dropping—both in the formulation of the theorem and in its proof—the word 'order(ed)' everywhere; considering in clause (3) only symmetric bisimulations; replacing in clause (4) the least fixed-point characterization of 1_A by the statement that it is the *unique* fixed point; and by dropping clause (5) (the notion of lax-homomorphism does not make sense in a metric setting). Note that the definitions of 'weakly preserving kernel pairs' and 'dense-epi' can be adapted straightforwardly. The proof can be almost literally copied: the proof of (4) $\Rightarrow$ (2) becomes somewhat simpler because of the uniqueness of 1_A; and in the proof of (3) $\Rightarrow$ (2$'$), the kernel pair of f should be taken rather than the ordered kernel pair.

Example 3. Let $\mathcal{P}_c : CMS \longrightarrow CMS$ be defined by, for all $(D, d_D) \in CMS$,

$$\mathcal{P}_c(D) \equiv \{X \subseteq D \mid X \text{ is compact (w.r.t. } d_D) \}.$$

(The metric on $\mathcal{P}_c(D)$ is the so-called Hausdorff metric.) For every ϵ with $0 \leq \epsilon < 1$, the 'shrinking' functor id_ϵ is given by, for any (D, d_D),

$$id_\epsilon((D, d_D)) \equiv (D, \epsilon \cdot d_D).$$

Clearly id_ϵ is locally contracting. Taking the composition $\mathcal{P}_c \circ id_\epsilon$ (which we shall by abuse of notation again denote by $\mathcal{P}_c$) yields again a locally contractive functor. Thus there exists a fixed point

$$\gamma : H \cong \mathcal{P}_c(H),$$

and (H, γ) is a final $\mathcal{P}_c$-coalgebra. □

Because the metric space H is isomorphic to the collection of its compact subsets (note the presence of the 'metric shrinker' id_ϵ, though), it is an instance of a *hyperuniverse.* (See [FH92] for a general construction of hyperuniverses, and [FH83] and [Acz88] for a hyperuniverse based on a non-standard collection of axioms. Cf. [Abr88, MMO89, Rut91].) By putting, for $p, p' \in H$,

$$p' \in_H p \equiv p' \in \gamma(p),$$

H can be easily seen to contain all so-called hereditarily finite sets and their limits (with respect to the metric on H). Note that these limits need not be hereditarily finite themselves.

As pointed out in [Abr88], the standard axioms of set theory hold in H, with topological versions of separation, replacement and choice. By (the metric version of) Theorem 8, strong extensionality can be added to these axioms: two sets in H are equal if and only if they are $\mathcal{P}_c$-bisimilar. E.g., for $p, q \in H$ with (omitting the isomorphism γ)

$$p = \{p\}, \quad q = \{q\},$$

$p = q$ follows from the fact that $\{(p, q)\}$ is a $\mathcal{P}_c$-bisimulation on H.

8 Conclusion

As was observed above, the characterization of final coalgebras in terms of strong extensionality (clause (3) of Theorem 8) does not have a dual counterpart among the clauses of the structural induction theorem (Theorem 10 in the Appendix). However, the latter theorem can be extended with a fifth, equivalent clause that comes close to being the dual of clause (3) of Theorem 8, as follows. An *F-congruence* on an *F-algebra* (A, α) is an F-algebra (R, β) with R a relation on A such that the projections $\pi_1, \pi_2 : (R, \beta) \to (A, \alpha)$ are homomorphisms of F-algebras. This definition generalizes the standard notion of a congruence on Σ-algebras. Note that it is dual to the definition of symmetric bisimulation. Clauses (1) through (4) of Theorem 10 can be shown to be equivalent to the following statement: there exists $\beta : F(\Delta) \to \Delta$ (with $\Delta \equiv \{(a, a') \in A \times A \mid a = a'\}$) such that (Δ, β) is the smallest F-congruence on (A, α).

Acknowledgements

As always, we have benefitted from presentations of this work for the Amsterdam Concurrency Group, headed by Jaco de Bakker. We are in particular indebted to Daniele Turi: this paper builds directly on our previous joint work, and we have had some lively discussions on the contents of this paper. One of the anonymous referees is thanked for suggesting a useful reference.

References

[Abr88] S. Abramsky. A Cook's tour of the finitary non-well-founded sets. Department of Computing, Imperial College, London, 1988.

[Abr91] S. Abramsky. A domain equation for bisimulation. *Information and Computation*, 92:161–218, 1991.

[Acz88] P. Aczel. *Non-well-founded sets*. Number 14 in CSLI Lecture Notes. Stanford University, 1988.

[AM89] P. Aczel and N. Mendler. A final coalgebra theorem. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *Proceedings Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 357–365, 1989.

[AR89] P. America and J.J.M.M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *Journal of Computer and System Sciences*, 39(3):343–375, 1989.

[Eng89] R. Engelking. *General Topology*, volume 6 of *Sigma Series in Pure Mathematics*. Heldermann Verlag, Berlin, revised and completed edition, 1989.

[FH83] M. Forti and F. Honsell. Set theory with free construction principles. *Annali Scuola Normale Superiore, Pisa*, X(3):493–522, 1983.

[FH92] M. Forti and F. Honsell. A general construction of hyperuniverses. Technical Report 1992/9, Istituto di Matematiche Applicate 'U. Dini', Facoltà di Ingegneria, Università di Pisa, 1992.

[Fio93] M. Fiore. A coinduction principle for recursive data types based on bisimulation. In *Proceedings of the Eighth IEEE Symposium on Logic In Computer Science*, 1993.

[Lan71] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[LP82] D. Lehmann and A. Pasztor. Epis need not to be dense. *Theoretical Computer Science*, 17:151–161, 1982.

[LS81] D. Lehmann and M.B. Smyth. Algebraic specifications of data types: a synthetic approach. *Mathematical Systems Theory*, 14:97–139, 1981.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MMO89] M.W. Mislove, L.S. Moss, and F.J. Oles. Non-well-founded sets obtained from ideal fixed points. In *Proc. of the Fourth IEEE Symposium on Logic in Computer Science*, pages 263–272, 1989. To appear in Information and Computation.

[MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[Par81] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[Pit92] A.M. Pitts. A co-induction principle for recursively defined domains. Technical Report 252, Computer Laboratory, University of Cambridge, 1992.

[Pit93] A.M. Pitts. Relational properties of recursively defined domains. In *Proceedings 8th Annual Symposium on Logic in Computer Science*, pages 86–97. IEEE Computer Society Press, 1993.

[Plo81] G.D. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the "Pisa Notes"). Department of Computer Science, University of Edinburgh, 1981.

[Plo91] G.D. Plotkin. Some notes on recursive domain equations. Handwritten notes for the Domain Theory PG Course. University of Edinburgh, 1991.

[RT93] J.J.M.M. Rutten and D. Turi. On the foundations of final semantics: Non-standard sets, metric spaces, partial orders. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX workshop on*

Semantics: Foundations and Applications, volume 666 of *Lecture Notes in Computer Science*, pages 477–530. Springer-Verlag, 1993.

[Rut91] J.J.M.M. Rutten. Hereditarily-finite sets and complete metric spaces. Technical Report CS-R9148, Centre for Mathematics and Computer Science, Amsterdam, 1991.

[Smy92] M.B. Smyth. I-categories and duality. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of categories in computer science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 270–287. Cambridge University Press, 1992.

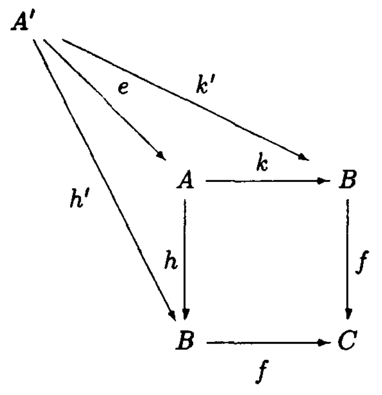
[SP32] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11:761–783, 1982.

9 Appendix

Some categorical notions

Let C be a category. An arrow $m : A \to B$ is called *monic* if for any two arrows $f, g : D \to A$ the equality $m \circ f = m \circ g$ implies $f = g$. An arrow $e : A \to B$ is called *epi* if for any two arrows $f, g : B \to D$ the equality $f \circ e = g \circ e$ implies $f = g$.

A kernel pair (see [Lan71]) for an arrow $f : B \to C$ in C consists of an object A and a pair of arrows $h : A \to B$ and $k : A \to B$ such that $f \circ h = f \circ k$, and such that for any other object A' and arrows $h' : A' \to B$ and $k' : A' \to B$ with $f \circ h' = f \circ k'$, there exists a unique arrow $e : A' \to A$ satisfying $h' = h \circ e$ and $k' = k \circ e$:


$$\begin{tikzcd}
A' \arrow[dr, "e"] \arrow[drr, "k'"] \arrow[ddr, "h'"'] & & \\
& A \arrow[r, "k"] \arrow[d, "h"] & B \arrow[d, "f"] \\
& B \arrow[r, "f"'] & C
\end{tikzcd}$$

Ordered kernel pairs

In $C = CPO_\perp$, the above definition can be generalized as follows. An *ordered* kernel pair for a function $f : B \to C$ in $CPO_\perp$ consists of a cpo A and a pair of functions $h : A \to B$ and $k : A \to B$ such that $f \circ h \leq f \circ k$, and such that for any other cpo A' and functions $h' : A' \to B$ and $k' : A' \to B$ with $f \circ h' \leq f \circ k'$, there exists a unique arrow $e : A' \to A$ satisfying $h' = h \circ e$ and $k' = k \circ e$.

The cpo A with functions h and k is called a *weak ordered kernel pair* for f if for any other cpo A' and functions $h' : A' \to B$ and $k' : A' \to B$ with

$f \circ h' \leq f \circ k'$, there exists an arrow $e : A' \to A$ (not necessarily unique) satisfying $h' \leq h \circ e$ (rather than $h' = h \circ e$) and $k' = k \circ e$.

A functor $F : CPO_\perp \to CPO_\perp$ *weakly preserves ordered kernel pairs* if it transforms ordered kernel pairs for functions f into weak ordered kernel pairs for $F(f)$.

Some further order-theoretic notions

Let D be a cpo and consider a continuous function $f : D \to D$. (That is, f preserves least upperbounds of ω-chains.) Then f has a least fixed point, which is denoted by $\mu x.\ f(x)$.

A subset $P \subseteq D$ is called ω-inductive if every chain $\langle x_n \rangle_n$ in P has its least upperbound in P.

The following is called the principle of fixed-point induction. Let $f : D \to D$ be continuous and let $P \subseteq D$ be ω-inductive. Then

$$(\perp \in P \ \wedge \ (\forall x \in D[x \in P \Rightarrow f(x) \in P]) \ \Rightarrow \ (\mu x.\ f(x)) \in P$$

A *strict order-monic* (see [Plo81]) is a strict continuous function (in $CPO_\perp$) $m : A \to B$ such that for any two arrows $f, g : D \to A$ the inequality $m \circ f \leq m \circ g$ implies $f \leq g$. It is easy to see that m is a strict order-monic if and only if, for all $a, a' \in A$,

$$a \sqsubseteq a' \Leftrightarrow m(a) \sqsubseteq m(a').$$

A strict continuous function $e : A \to B$ is *dense-epi* if it is epi and moreover satisfies $cl(e(A)) = B$, where $cl(e(A))$ is the least subset of B that contains $e(A)$ and that is closed under least upperbounds of ω-chains. (In fact the condition $cl(e(A)) = B$ can be shown, by transfinite induction, to imply the fact that e is epi. See [LP82] for an explanation why "Epis need not to be dense".)

If $m : A \to B$ is both a strict order-monic and dense-epi, then m is an isomorphism: $m(A) = cl(m(A))$ since e is a strict order-monic, and $cl(m(A)) = B$, since e is dense-epi. Thus e is a bijective order-embedding.

The structural induction theorem

In [Plo81] (Theorem 4 of Chapter 5), the following theorem is proved. (See also [LS81] for a similar result.)

Theorem 10. *Let $F : CPO_\perp \to CPO_\perp$ be a locally continuous functor which preserves inclusions. (That is, if $\iota : A \subseteq B$ then $F(\iota) : F(A) \subseteq F(B)$.) Let $\alpha : F(A) \to A$ be an F-algebra. Then the following four statements are equivalent:*

1. *(A, α) is an initial F-algebra.*
2. *α is a strict order-monic, and for every strict order-monic $m : B \to A$: if there exists $\beta : F(B) \to B$ such that $m : (B, \beta) \to (A, \alpha)$ is a homomorphism of algebras (i.e., $m \circ \beta = \alpha \circ F(m)$), then m is an isomorphism.*

3. *α is a strict order-monic, and for every ω-inductive $P \subseteq A$ the following principle of structural induction holds:*

$$(\bot \in P \ \wedge\ (\forall x \in F(A)[x \in F(P) \Rightarrow \alpha(x) \in P])) \ \Rightarrow\ P = A$$

4. *α is an isomorphism and $1_A = \mu h.\, \alpha \circ F(h) \circ \alpha^{-1}$.*

The assumption that F preserves inclusions is only used to prove the equivalence of (2) and (3). This property is satisfied by most covariant functors.

Three Metric Domains of Processes for Bisimulation

Franck van Breugel[1,2,*]

[1] Department of Software Technology
CWI, Kruislaan 413, 1098 SJ Amsterdam
[2] Department of Mathematics and Computer Science
Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam

Abstract. A new metric domain of processes is presented. This domain is located in between two metric process domains introduced by De Bakker and Zucker. The new process domain characterizes the collection of image finite processes. This domain has as advantages over the other process domains that no complications arise in the definitions of operators like sequential composition and parallel composition, and that image finite language constructions like random assignment can be modelled in an elementary way. As in the other domains, bisimilarity and equality coincide in this domain.
The three domains are obtained as unique (up to isometry) solutions of equations in a category of 1-bounded complete metric spaces. In the case the action set is finite, the three domains are shown to be equal (up to isometry). For infinite action sets, e.g., equipollent to the set of natural or real numbers, the process domains are proved not to be isometric.

Introduction

In semantics, a *process* is usually understood as a behaviour of a system. *Labelled transition systems* have proved to be suitable for describing the behaviour (or operational semantics) of a system (cf. [Plo81]). A labelled transition system can be viewed as a rooted directed graph of which the edges are labelled by actions (cf. [BK87]), or as a tree of which the edges are labelled by actions, which is obtained by unfolding the graph. The semantic notion of a process is usually defined by means of a suitable behavioural equivalence over the labelled transition systems. *Bisimilarity* (cf. [Par81]) is commonly accepted as the finest behavioural equivalence over labelled transition systems (cf. [Gla90, Gla93]).

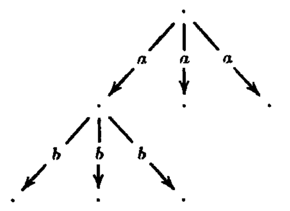
In this paper, processes are studied from the point of view of denotational semantics. In the literature, domains of processes are found for several mathematical structures. For complete partial orders, process domains are presented by Milne and Milner in [MM79], and Abramsky in [Abr91]. Aczel introduces in [Acz88] a process domain for non-well-founded sets. For complete metric spaces,

* This work was partially supported by the Netherlands *Nationale Faciliteit Informatica* programme, project Research and Education in Concurrent Systems (REX).

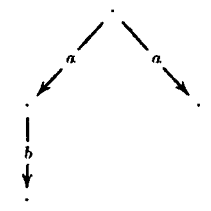
process domains are presented by De Bakker and Zucker in [BZ82, BZ83], and Golson and Rounds in [GR83, Gol84].

Aczel shows in [Acz88] that processes can be viewed as labelled transition systems. Bisimulation relations on these labelled transition systems induce bisimulation relations on the processes. A process domain is called *strongly extensional* (or *internally fully abstract*) if bisimilarity - being the largest bisimulation relation - coincides with equality, i.e. processes are bisimilar if and only if they are equal. Abramsky and Aczel prove that their process domains are strongly extensional. The process domains introduced by De Bakker and Zucker in [BZ82] and [BZ83] are shown to be strongly extensional by Van Glabbeek and Rutten in [GR89] and [Rut92].

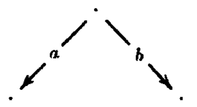
The metric process domains introduced by De Bakker and Zucker in [BZ82] and [BZ83], which will be denoted by P_1 and P_2 in the sequel, and a third r process domain, which will be denoted by P_3, are studied in detail in this pa Processes can be viewed as trees (both finite and infinite in depth) of which t edges are labelled by actions, and which are absorptive, i.e. for all nodes of a tree the collection of subtrees of that node is a set instead of a multiset, and commutative. For example, the tree



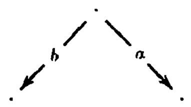
is not a process, and



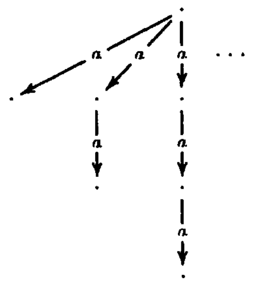
is the process obtained by absorption. Furthermore, the processes



and

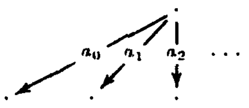


are identified by commutativity. The processes are endowed with a metric such that the distance between processes decreases if the maximal depth at which the truncations of the processes coincide increases. All processes considered in this paper are closed with respect to this metric. For example, the process

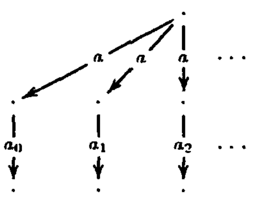


including the infinite branch is closed in contrast with the process not containing this infinite branch.

A process is called *finitely branching* if each node has only finitely many outgoing edges. A process is called *image finite* if, for each action, each node has only finitely many outgoing edges labelled with that action. A finitely branching process is image finite, but an image finite process is in general not finitely branching. For example, the process



is image finite but not finitely branching.



is an example of a general (or unrestricted) process being not finitely branching nor image finite. The process domains P_1, P_2, and P_3 can be shown to correspond to the collections of (finite in depth and)

- general processes,
- finitely branching processes, and
- image finite processes.

For example, the correspondence between the process domain P_3 and the collection of image finite processes of finite depth will be accomplished as follows. First, the space of image finite processes of finite depth is completed. In this way, a complete metric space of (finite and infinite in depth) processes is obtained. Second, the completed space is shown to be isometric to the process domain P_3.

The three process domains can be related in the following way. The process domain P_2 can be isometrically embedded in the process domain P_3 and the process domain P_3 can be isometrically embedded in the process domain P_1. If the action set is finite, then the three process domains can be shown to be isometric. If the action set is infinite, e.g., equipollent to the set of natural or real numbers, then it can be demonstrated that the three process domains are not isometric.

For P_1-processes, complications arise in the definitions of the following operators:

- sequential composition (cf. [BZ82, BM88]),
- parallel composition (cf. [BZ82, BM88, ABKR89, AR92]),
- trace set as defined by De Bakker et al. in [BBKM84], and

- fairification as defined by Rutten and Zucker in [RZ92].

For example, it is not possible to give a (denotational) definition of the sequential composition of P_1-processes, which coincides with the operational definition of the sequential composition. (Note that processes can be viewed as labelled transition systems.) In [BM88], the sequential composition of P_1-processes is not well-defined. The definition of the sequential composition in [BZ82] is well-defined, but does not coincide to the operational one. It can be shown that these complications do not arise in the definitions of the operators mentioned above on P_2- and P_3-processes.

Unlike the process domain P_2, the process domain P_3 makes an elementary semantic modelling of image finite language constructions like random assignment possible (cf. [Bre94]). (For a detailed overview of metric semantic models the reader is referred to [BR92].)

Novel in the present paper are

- the process domain P_3, which can be shown to correspond to the class of image finite processes and to be strongly extensional,
- the detailed comparison of the process domains P_1, P_2, and P_3 showing that the three process domains are isometric if the action set is finite and that they are not isometric for infinite action sets, and
- the relation of the process domains P_1, P_2, and P_3 with the classes of general, finitely branching, and image finite processes, extending results concerning the process domains P_1 and P_2 of [BZ82] and [BZ83].

In the first section of this paper, some preliminaries concerning metric spaces can be found. In the second section, the three process domains are introduced. In the third section, the correspondence between P_1-, P_2-, and P_3-processes and general, finitely branching, and image finite processes is studied. The process domains are related as described above in the fourth section. In the fifth section, the process domains are shown to be strongly extensional. In the sixth section, some complications arising in the definition of the sequential composition of P_1-processes are pinpointed. Furthermore, it is shown that these complications do not arise in the definition of this operator on P_3-processes. The other three operators, viz parallel composition, trace set, and fairification, are considered in [Bre94].

In this paper, several definitions from other papers have been modified slightly to stress the correspondence with the other definitions.

1 Metric spaces

Some preliminaries concerning metric spaces are presented. Only some nonstandard notions, i.e. notions which are not found in the main text of [Eng89], are introduced.

Contractive functions, which are called contractions, are introduced in

Definition 1. Let (X, d_X) and $(X', d_{X'})$ be metric spaces. A function $f : X \rightarrow X'$ is called *contractive* if there exists an ε, with $0 \leq \varepsilon < 1$, such that, for all x and x',

$$d_{X'}(f(x), f(x')) \leq \varepsilon \cdot d_X(x, x').$$

These contractions play a central rôle in

Theorem 2 (Banach's theorem). *Let (X, d_X) be a complete metric space. If $f : X \rightarrow X$ is a contraction then f has a unique fixed point $\mathit{fix}(f)$. For all x,*

$$\lim_n f^n(x) = \mathit{fix}(f)$$

where

$$f^0(x) = x \text{ and } f^{n+1}(x) = f(f^n(x)).$$

Proof. See Theorem II.6 of [Ban22]. □

In this paper, several recursive definitions are presented (cf. Definition 12, 14, 15, 22, and 24). Banach's theorem can be used to prove the well-definedness of these definitions (cf. [KR90]).

The embeddings to be introduced in Section 4 will be defined by means of nonexpansive functions.

Definition 3. Let (X, d_X) and $(X', d_{X'})$ be metric spaces. A function $f : X \rightarrow X'$ is called *nonexpansive* if, for all x and x',

$$d_{X'}(f(x), f(x')) \leq d_X(x, x').$$

2 Three process domains

Three process domains are presented. These process domains are defined by means of recursive domain equations.

In [AR89], America and Rutten present a category theoretic technique to solve recursive domain equations. The objects of the category are 1-bounded complete metric spaces. With a domain equation a functor is associated. If this functor satisfies certain conditions, then it has a unique fixed point (up to isometry) which is the intended solution of the domain equation.

The recursive domain equations, by which the process domains are defined, are built from an action set A, which is endowed with the discrete metric, and the constructions described in

Definition 4. Let (X, d_X) and $(X', d_{X'})$ be 1-bounded complete metric spaces. A metric on the Cartesian product of X and X', $X \times X'$, is defined by

$$d_{X \times X'}((x, x'), (\bar{x}, \bar{x}')) = \max\{ d_X(x, \bar{x}), d_{X'}(x', \bar{x}') \}.$$

A metric on the collection of functions from X to X', $X \rightarrow X'$, is defined by

$$d_{X \to X'}(f, f') = \sup \{ d_{X'}(f(x), f'(x)) \mid x \in X \}.$$

A new metric on X is defined by

$$d_{id_{\frac{1}{2}}(X)}(x, x') = \tfrac{1}{2} \cdot d_X(x, x').$$

The Hausdorff metric on the set of closed subsets of X, $\mathcal{P}_{cl}(X)$, and on the set of compact subsets of X, $\mathcal{P}_{co}(X)$, is defined by

$$\begin{aligned} d_{\mathcal{P}(X)}(A, B) = \max \{ & \sup \{ \inf \{ d_X(x, x') \mid x' \in B \} \mid x \in A \}, \\ & \sup \{ \inf \{ d_X(x, x') \mid x' \in A \} \mid x \in B \} \} \end{aligned}$$

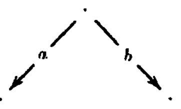
where $\sup \emptyset = 0$ and $\inf \emptyset = 1$.

The three process domains are introduced in

Definition 5. The process domains P_1, P_2, and P_3 are defined by the recursive domain equations

$$\begin{aligned} P_1 &\cong \mathcal{P}_{cl}(A \times id_{\frac{1}{2}}(P_1)) \\ P_2 &\cong \mathcal{P}_{co}(A \times id_{\frac{1}{2}}(P_2)) \\ P_3 &\cong A \to \mathcal{P}_{co}(id_{\frac{1}{2}}(P_3)) \end{aligned}$$

Processes as described in the introduction can be represented by elements of these process domains. For example, the process



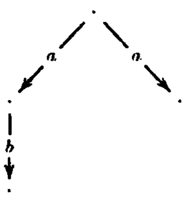
is represented by the P_1- and P_2-process

$$\{(a, \emptyset), (b, \emptyset)\}$$

and by the P_3-process

$$\lambda a' . \begin{cases} \{\lambda a'' . \emptyset\} & \text{if } a' = a \text{ or } a' = b \\ \emptyset & \text{otherwise} \end{cases}$$

The process



is represented by the P_1- and P_2-process

$$\{(a, \{(b, \emptyset)\}), (a, \emptyset)\}$$

and by the P_3-process

$$\lambda a' \,.\, \begin{cases} \{p_0, p_1\} & \text{if } a' = a \\ \emptyset & \text{otherwise} \end{cases}$$

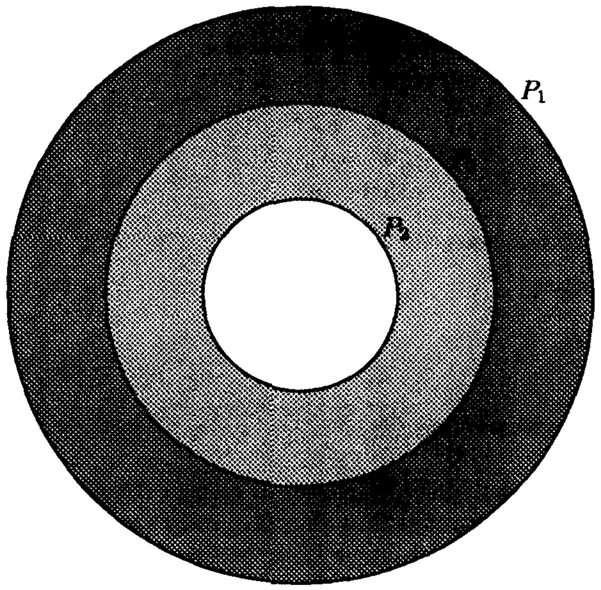
where

$$p_0 = \lambda a'' \,.\, \begin{cases} \{\lambda a''' \,.\, \emptyset\} & \text{if } a'' = b \\ \emptyset & \text{otherwise} \end{cases}$$

and

$$p_1 = \lambda a'' \,.\, \emptyset.$$

Not every process can be represented in all three process domains. In Section 4, we will show that the process domain P_3 is located in between P_1 and P_2, i.e. P_2 can be isometrically embedded in P_3 and P_3 can be isometrically embedded in P_1.



Next, processes in the shaded regions of the above picture are presented. The process

$$\begin{tikzcd}
 & & & \cdot \arrow[dlll, "a_0"'] \arrow[dl, "a_1"'] \arrow[d, "a_2"] & \cdots \\
\cdot & & \cdot & \cdot &
\end{tikzcd}$$

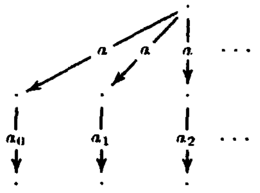
is represented by the P_1-process

$$\{\, (a_n, \emptyset) \mid n \in \mathbb{N} \,\}.$$

However, this is not a P_2-process, because the above set is closed but not compact. The process is also represented by the P_3-process

$$\lambda a' \,.\, \begin{cases} \{\lambda a'' \,.\, \emptyset\} & \text{if } a' = a_n \text{ for some } n \\ \emptyset & \text{otherwise} \end{cases}$$

The process



is represented by the P_1-process

$$\{\, (a, \{(a_n, \emptyset)\}) \mid n \in \mathbb{N} \,\}.$$

Again, this is not a P_2-process, because the above set is not compact. The process can also not be represented by a P_3-process. The obvious candidate

$$\lambda a' \,.\, \begin{cases} \{\, p_n \mid n \in \mathbb{N} \,\} & \text{if } a' = a \\ \emptyset & \text{otherwise} \end{cases}$$

where

$$p_n = \lambda a'' \,.\, \begin{cases} \{\lambda a''' \,.\, \emptyset\} & \text{if } a'' = a_n \\ \emptyset & \text{otherwise} \end{cases}$$

is not a P_3-process, since the set

$$\{\, p_n \mid n \in \mathbb{N} \,\}$$

is not compact.

3 Finite processes

The three process domains are related to certain collections of *finite* (in depth) processes. It is demonstrated that P_1-, P_2-, and P_3-processes correspond to general, finitely branching, and image finite processes, respectively.

The set of processes of finite depth is introduced in

Definition 6. The set P_1^* of *processes of finite depth* is defined by

$$P_1^* = \bigcup \{\, P_1^n \mid n \in \mathbb{N} \,\}$$

where

$$P_1^n = \begin{cases} \{\emptyset\} & \text{if } n = 0 \\ \mathcal{P}(A \times P_1^{n-1}) & \text{otherwise} \end{cases}$$

Obviously, each P_1^*-process is a P_1-process. The P_1^*-processes are endowed with the restriction of the metric on the P_1-processes. The obtained metric space is not complete. For example, the sequence $(p_n)_n$ of P_1^*-processes defined by

$$p_n = \begin{cases} \emptyset & \text{if } n = 0 \\ \{(a, p_{n-1})\} & \text{otherwise} \end{cases}$$

is a Cauchy sequence but does not have a limit in P_1^* (the sequence converges to a process of infinite depth). The metric completion of the metric space of P_1^*-processes, which is denoted by $\widetilde{P_1^*}$, is shown to be isometric to the process domain P_1 in

Theorem 7. $\widetilde{P_1^*} \cong P_1$.

Proof. See Theorem 2.11 of [BZ82]. □

The set of finitely branching processes of finite depth is introduced in the following definition, in which $\mathcal{P}_{fi}$ denotes the set of all finite subsets.

Definition 8. The set P_2^* of *finitely branching processes of finite depth* is defined by

$$P_2^* = \bigcup \{ P_2^n \mid n \in \mathbb{N} \}$$

where

$$P_2^n = \begin{cases} \{\emptyset\} & \text{if } n = 0 \\ \mathcal{P}_{fi}(A \times P_2^{n-1}) & \text{otherwise} \end{cases}$$

Similarly, the metric completion of the metric space of P_2^*-processes is proved to be isometric to the complete metric space of P_2-processes in

Theorem 9. $\widetilde{P_2^*} \cong P_2$.

Proof. See Theorem 3.2 of [BZ83]. □

The set of image finite processes of finite depth is introduced in

Definition 10. The set P_3^* of *image finite processes of finite depth* is defined by

$$P_3^* = \bigcup \{ P_3^n \mid n \in \mathbb{N} \}$$

where

$$P_3^n = \begin{cases} \{\lambda a \cdot \emptyset\} & \text{if } n = 0 \\ A \to \mathcal{P}_{fi}(P_3^{n-1}) & \text{otherwise} \end{cases}$$

The process domain P_3 can be shown to be isometric to the metric completion of the metric space of P_3^*-processes.

Theorem 11. $\widetilde{P_3^*} \cong P_3$.

Proof. Similar to the proofs of the Theorems 7 and 9. □

4 Comparison of the process domains

The three process domains are related. It is shown that the process domain P_2 can be isometrically embedded in the process domain P_3 and that the process domain P_3 can be isometrically embedded in the process domain P_1. Furthermore, if the action set A is finite, then the process domain P_1 can be isometrically embedded in the process domain P_2 such that the diagram

$$\begin{tikzcd}
 & P_3 \arrow[loop above, "id"] \arrow[dr, "i_2"] & \\
P_2 \arrow[ur, "i_1"] \arrow[loop below, "id"] & & P_1 \arrow[ll, "i_3"] \arrow[loop below, "id"]
\end{tikzcd}$$

commutes. Consequently, if the action set A is finite, then the process domains P_1, P_2, and P_3 are isometric. If the action set A is infinite, then it can be proved that the process domains P_1, P_2, and P_3 are not isometric.

The embedding i_1 from the process domain P_2 to the process domain P_3 is introduced in

Definition 12. The embedding $i_1 : P_2 \to P_3$ is defined by

$$i_1(p) = \lambda a \cdot \{ i_1(p') \mid (a, p') \in p \}.$$

In order to prove the well-definedness of the above recursive definition of the embedding i_1, a so-called higher-order transformation Ψ_{i_1} is introduced in

Definition 13. The higher-order transformation

$$\Psi_{i_1} : (P_2 \to^1 P_3) \to (P_2 \to^1 P_3)$$

is defined by

$$\Psi_{i_1}(\psi)(p) = \lambda a \cdot \{ \psi(p') \mid (a, p') \in p \}.$$

In order to be well-defined, the higher-order transformation Ψ_{i_1} is restricted to nonexpansive functions, i.e.

$$\Psi_{i_1} \in (P_2 \to^1 P_3) \to (P_2 \to^1 P_3).$$

(The collection of nonexpansive functions from P_2 to P_3, $P_2 \to^1 P_3$, endowed with the restriction of the metric on functions from P_2 to P_3 is a complete metric space.) Although only continuity, which is implied by nonexpansiveness, is needed in the well-definedness proof of the higher-order transformation Ψ_{i_1}, the restriction induces half of the proof that the embedding i_1 is isometric (see below). This higher-order transformation Ψ_{i_1} can be shown to be contractive (here the $id_{\frac{1}{2}}$ in the domain equation of process domain P_3 is crucial). According to Banach's theorem (cf. Theorem 2), the higher-order transformation Ψ_{i_1} has a unique fixed point which is the intended embedding i_1, i.e.

$$i_1 = \mathit{fix}(\Psi_{i_1}).$$

Consequently, $i_1 \in P_2 \to^1 P_3$. To show that the embedding i_1 is isometric it is left to prove that, for all p and p',

$$d(i_1(p), i_1(p')) \geq d(p, p').$$

This can be demonstrated by fixed point induction using Banach's theorem.

The embedding i_2 from the process domain P_3 to the process domain P_1 is introduced in

Definition 14. The embedding $i_2 : P_3 \to P_1$ is defined by

$$i_2(p) = \{ (a, i_2(p')) \mid p' \in p(a) \}.$$

As the embedding i_1, also the embedding i_2 can be shown to be well-defined and isometric.

Assume the action set A is finite. Then the process domain P_1 can be isometrically embedded in the process domain P_2. The embedding i_3 from the process domain P_1 to the process domain P_2 is introduced in

Definition 15. The embedding $i_3 : P_1 \to P_2$ is defined by

$$i_3(p) = \{ (a, i_3(p')) \mid (a, p') \in p \}.$$

Also this embedding can be shown to be well-defined by means of a higher-order transformation. In the well-definedness proof of the higher-order transformation the compactness of the process domain P_1 is exploited. The process domain P_1 is compact, since the solution of a recursive domain equation built from 1-bounded compact metric spaces (e.g., the finite action set A endowed with the discrete metric), $\mathcal{P}_{cl}$, $\times$, and $id_{\frac{1}{2}}$ is a 1-bounded compact metric space as is proved in [BW93].

The embedding i_3 can also be shown to be isometric. Furthermore, it can be demonstrated that the above diagram commutes. For example, it can be proved that

$$d(i_3 \circ i_2 \circ i_1, id) \leq \tfrac{1}{2} \cdot d(i_3 \circ i_2 \circ i_1, id)$$

and hence $i_3 \circ i_2 \circ i_1 = id$. As a consequence, the process domains P_1, P_2, and P_3 are isometric.

Theorem 16. *If A is finite, then $P_1 \cong P_2$, $P_2 \cong P_3$, and $P_1 \cong P_3$.*

Assume the action set is infinite. More precisely, assume A is equipollent to $2 \uparrow n$, for some n, where $2 \uparrow n$ is defined in

Definition 17. The sets $2 \uparrow n$ are defined by

$$2 \uparrow n = \begin{cases} \mathbb{N} & \text{if } n = 0 \\ 2^{2 \uparrow (n-1)} & \text{otherwise} \end{cases}$$

$p \xrightarrow{a} p'$ if and only if $p' \in p(a)$.

Also the process domain P_3 can be shown to be strongly extensional.

Theorem 21. *P_3 is strongly extensional.*

Proof. Similar to the proofs of the Theorems 19 and 20. □

6 Sequential composition

Some complications arising in the definition of the sequential composition of P_1-processes are pinpointed. Furthermore, it is shown that these complications do not arise in the definition of the sequential composition of P_3-processes.

In Definition 4.4 of [BM88], the sequential composition of P_1-processes is defined by

Definition 22. The operator $;$: $P_1 \times P_1 \to P_1$ is defined by

$$p \,;\, p' = \begin{cases} p' & \text{if } p = \emptyset \\ \{\, (a, p'' \,;\, p') \mid (a, p'') \in p \,\} & \text{otherwise} \end{cases}$$

This definition coincides with the operational definition of the sequential composition. (Note that processes can be seen as labelled transition systems.) However, the above definition is not well-defined, as Warmerdam ([War90]) showed (cf. Appendix A).

Also in Definition 2.14 of [BZ82], the sequential composition of P_1-processes is defined.

Definition 23. For a finite process p, $p \,;\, p'$ is defined as in Definition 22, and for an infinite process p,

$$p \,;\, p' = \lim_n (p\,[n] \,;\, p')$$

where $p\,[n]$ denotes the truncation of process p at depth n.

This definition is well-defined. However, the above definition does not coincide with the operational definition of the sequential composition (cf. Appendix A).

For P_3-processes, the sequential composition is defined in

Definition 24. The operator $;$: $P_3 \times P_3 \to P_3$ is defined by

$$p \,;\, p' = \begin{cases} p' & \text{if } p = \lambda a \cdot \emptyset \\ \lambda a \cdot \{\, p'' \,;\, p' \mid p'' \in p(a) \,\} & \text{otherwise} \end{cases}$$

The well-definedness of the above definition of the sequential composition can be proved along the lines of the well-definedness proof of the embedding i_1 in the fourth section of this paper.

Also in the definitions of the operators parallel composition, trace set, and fairification on P_1-processes similar complications arise (cf. [BK87, BBKM84, Bre94]). These complications do not arise in the definitions of the operators on P_3-processes (cf. [Bre94]). Also process domain P_2 does not give rise to these complications (cf. [KR90]). However, unlike process domain P_3, process domain P_2 does not allow an elementary modelling of image finite language constructions like random assignment (cf. [Bre94]).

Concluding remarks

In this concluding section, some related work is discussed and some points for further research are mentioned.

A fourth process domain P_4 defined by the recursive domain equation $P_4 \cong A \rightarrow \mathcal{P}_{cl}(id_{\frac{1}{2}}(P_4))$ is considered in [Bre94]. The process domain P_4 can be shown to be isometric to the process domain P_1 (independent of the size of the action set A).

An alternative metric process domain is introduced by Golson and Rounds in [GR83, Gol84]. The processes are Milner's rigid synchronization trees endowed with a pseudometric. The pseudometric is induced by the (strong) behavioural equivalence relation introduced in [Mil80]. This behavioural equivalence relation and the bisimilarity equivalence relation considered in Section 5 do not coincide (cf. [Mil90]). Golson and Rounds show that their process domain is isometric to the process domain P_1 in case the action set is finite or countably infinite (for the countably infinite case, the power set construction used in the domain equation defining P_1 should be restricted to the collection of countable subsets).

In [Ole87], Oles defines a denotational semantics for a nonuniform language with the so-called angelic choice operator. The mathematical domain of this denotational semantics is defined as the solution of a recursive domain equation over bounded complete directed sets. For a uniform language with the conventional choice operator, the mathematical domain defined by the recursive domain equation $P \cong A \rightarrow \mathcal{P}_{fi}(P)$ has been suggested ([Ole92]). This domain equation shows some resemblance with the domain equation for process domain P_3.

Some topics for further research are the study of the process domains P_1, P_2, and P_3 with the action set endowed with an arbitrary complete metric instead of the discrete metric, and process domains corresponding to general, finitely branching, and image finite processes for complete partial orders and non-well-founded sets.

Acknowledgements

The author would like to thank Jaco de Bakker, Jan Rutten, and Fer-Jan de Vries for several discussions and their comments on a preliminary version of this paper. Furthermore, the author is grateful to Marcello Bonsangue, Frank Oles, Daniele Turi, Erik de Vink, and Jeroen Warmerdam for discussion.

References

[ABKR89] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Denotational Semantics of a Parallel Object-Oriented Language. *Information and Computation*, 83(2):152–205, November 1989.

[Abr91] S. Abramsky. A Domain Equation for Bisimulation. *Information and Computation*, 92(2):161–218, June 1991.

[Acz88] P. Aczel. *Non-Well-Founded Sets*. Number 14 in CSLI Lecture Notes. Centre for the Study of Languages and Information, Stanford, 1988.

[AR89] P. America and J.J.M.M. Rutten. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *Journal of Computer and System Sciences*, 39(3):343–375, December 1989.

[AR92] P. America and J.J.M.M. Rutten. A Layered Semantics for a Parallel Object-Oriented Language. *Formal Aspects of Computing*, 4(4):376–408, 1992.

[Ban22] S. Banach. Sur les Opérations dans les Ensembles Abstraits et leurs Applications aux Equations Intégrales. *Fundamenta Mathematicae*, 3:133–181, 1922.

[BBKM84] J.W. de Bakker, J.A. Bergstra, J.W. Klop, and J.-J.Ch. Meyer. Linear Time and Branching Time Semantics for Recursion with Merge. *Theoretical Computer Science*, 34(1/2):135–156, 1984.

[BK87] J.A. Bergstra and J.W. Klop. A Convergence Theorem in Process Algebra. Report CS-R8733, CWI, Amsterdam, July 1987. Appeared in [BR92], pages 164–195.

[BM88] J.W. de Bakker and J.-J.Ch. Meyer. Metric Semantics for Concurrency. *BIT*, 28:504–529, 1988.

[BR92] J.W. de Bakker and J.J.M.M. Rutten, editors. *Ten Years of Concurrency Semantics, selected papers of the Amsterdam Concurrency Group*. World Scientific, Singapore, September 1992.

[Bre94] F. van Breugel. *Topological Models in Comparative Semantics*. PhD thesis, Vrije Universiteit, Amsterdam, 1994. In preparation.

[BW93] F. van Breugel and J.H.A. Warmerdam. Solving Recursive Domain Equations in a Category of Compact Metric Spaces. CWI, Amsterdam. Preprint, to appear.

[BZ82] J.W. de Bakker and J.I. Zucker. Processes and the Denotational Semantics of Concurrency. *Information and Control*, 54(1/2):70–120, July/August 1982.

[BZ83] J.W. de Bakker and J.I. Zucker. Compactness in Semantics for Merge and Fair Merge. In E. Clarke and D. Kozen, editors, *Proceedings of 4th Workshop on Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 18–33, Pittsburgh, June 1983. Springer-Verlag.

[Eng89] R. Engelking. *General Topology*, volume 6 of *Sigma Series in Pure Mathematics*. Heldermann Verlag, Berlin, revised and comp[illegible]d edition, 1989.

[Gla90] R.J. van Glabbeek. The Linear Time - Branching Time Spectrum. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings of CONCUR'90*, volume

458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, August 1990. Springer-Verlag.

[Gla93] R.J. van Glabbeek. The Linear Time - Branching Time Spectrum II. To appear in *Proceedings of CONCUR'93*, Hildesheim, August 1993.

[Gol84] W.G. Golson. *Denotational Models based on Synchronous Communicating Processes.* PhD thesis, University of Michigan, Ann Arbor, 1984.

[GR83] W.G. Golson and W.C. Rounds. Connections between Two Theories of Concurrency: Metric Spaces and Synchronization Trees. *Information and Control*, 57(2/3):102–124, May/June 1983.

[GR89] R.J. van Glabbeek and J.J.M.M. Rutten. The Processes of De Bakker and Zucker represent Bisimulation Equivalence Classes. In *J.W. de Bakker, 25 jaar semantiek*, pages 243–246. CWI, Amsterdam, April 1989.

[KR90] J.N. Kok and J.J.M.M. Rutten. Contractions in Comparing Concurrency Semantics. *Theoretical Computer Science*, 76(2/3):179–222, 1990.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[Mil90] R. Milner. Operational and Algebraic Semantics of Concurrent Processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. The MIT Press/Elsevier, Cambridge/Amsterdam, 1990.

[MM79] G. Milne and R. Milner. Concurrent Processes and Their Syntax. *Journal of the ACM*, 26(2):302–321, April 1979.

[Ole87] F.J. Oles. Semantics for Concurrency without Powerdomains. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 211–222, Munich, January 1987.

[Ole92] F.J. Oles, August 1992. Personal communication.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *Proceedings of 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Karlsruhe, March 1981. Springer-Verlag.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Report DAIMI FN-19, Aarhus University, Aarhus, September 1981.

[Rut92] J.J.M.M. Rutten. Processes as Terms: Non-Well-Founded Models for Bisimulation. *Mathematical Structures in Computer Science*, 2(3):257–275, September 1992.

[RZ92] J.J.M.M. Rutten and J.I. Zucker. A Semantic Approach to Fairness. *Fundamenta Informaticae*, 16(1):1–38, January 1992.

[War90] J.H.A. Warmerdam, November 1990. Personal communication.

A Warmerdam's counterexample

Warmerdam ([War90]) showed that the sequential composition of P_1-processes as defined in Definition 4.4 of [BM88] (cf. Definition 22) is not well-defined by proving that the set

$$\{ (a, p'' \,;\, p') \mid (a, p'') \in p \}$$

is in general not closed. Here, Warmerdam's counterexample is presented. Furthermore, this counterexample is used to illustrate that the sequential composition as defined in Definition 2.14 of [BZ82] (cf. Definition 23) does not correspond to the operational definition of the sequential composition.

Let P_1-process p be defined by

$$p = \{\,(a, p_n) \mid n \in \mathbb{N}\,\}$$

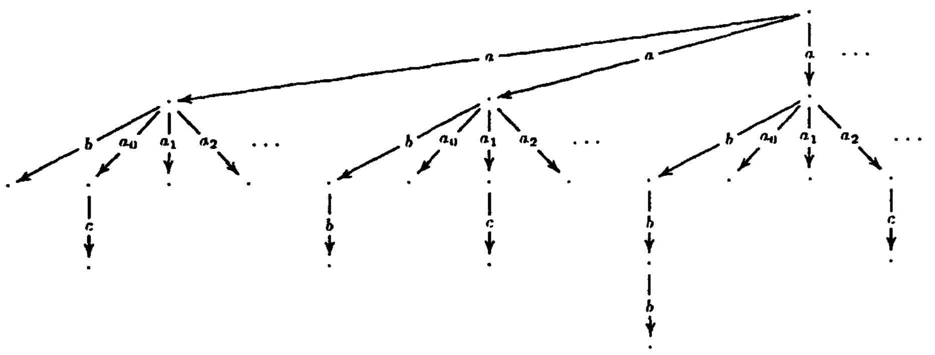
where

$$p_n = \{b^n, (a_0, \emptyset), \ldots, (a_{n-1}, \emptyset), (a_n, \{(c, \emptyset)\}), (a_{n+1}, \emptyset), \ldots\}$$

and

$$b^n = \begin{cases} (b, \emptyset) & \text{if } n = 0 \\ (b, \{b^{n-1}\}) & \text{otherwise} \end{cases}$$

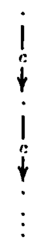
This P_1-process p is depicted by



Let P_1-process p' be defined by

$$p' = \{\lim_n c^n\}.$$

This P_1-process p' is depicted by



According to Definition 4.4 of [BM88] (cf. Definition 22), the sequential composition of the P_1-processes p and p' is defined by

$$p\,;p' = \{\,(a, p''_n) \mid n \in \mathbb{N}\,\}$$

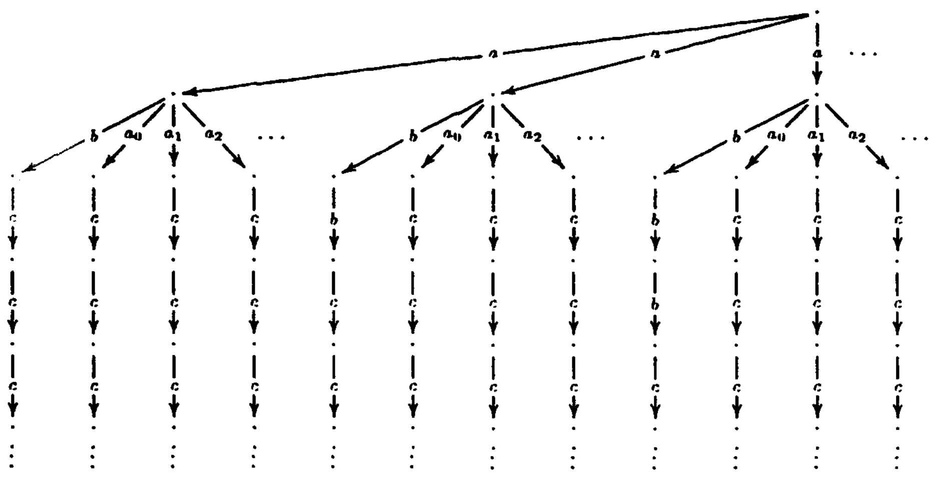
where

$$p''_n = \{b^n ; p', (a_0, p'), (a_1, p'), \ldots\}$$

and

$$b^n ; p' = \begin{cases} (b, p') & \text{if } n = 0 \\ (b, \{b^{n-1} ; p'\}) & \text{otherwise} \end{cases}$$

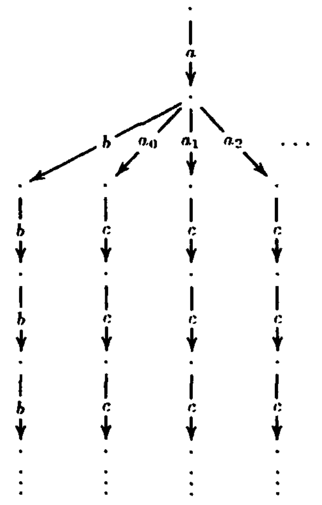
This process $p ; p'$ is depicted by



However, $p ; p'$ is not a P_1-process, since the set $p ; p'$ is not closed. The set $p ; p'$ contains the Cauchy sequence $((a, p''_n))_n$ but not its limit (a, p'') where

$$p'' = \{\lim_n b^n, (a_0, p'), (a_1, p'), \ldots\}$$

which is depicted by



The above counterexample also shows that the limit construction in the definition of the sequential composition presented in Definition 2.14 of [BZ82] (cf. Definition 23) adds unexpected subprocesses; the limit construction $\lim_n (p[n]\,; p')$ adds subprocess (a, p'').

Topological Models for Higher Order Control Flow

J.W. de Bakker[1,2] and F. van Breugel[1,2,*]

[1] Department of Software Technology
CWI, Kruislaan 413, 1098 SJ Amsterdam
[2] Department of Mathematics and Computer Science
Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam

Abstract. Semantic models are presented for two simple imperative languages with higher order constructs. In the first language the interesting notion is that of second order assignment $x := s$, for x a procedure variable and s a statement. The second language extends this idea by a form of higher order communication, with statements $c\,!\,s$ and $c\,?\,x$, for c a channel. We develop operational and denotational models for both languages, and study their relationships. Both in the definitions and the comparisons of the semantic models, convenient use is made of some tools from (metric) topology. The operational models are based on (SOS-style) transition systems; the denotational definitions use domains specified as solutions of domain equations in a category of 1-bounded complete ultrametric spaces. In establishing the connection between the two kinds of models, fruitful use is made of Rutten's *processes as terms* technique. Another new tool consists in the use of *metric* transition systems, with a metric defined on the configurations of the system. In addition to higher order programming notions, we use higher order definitional techniques, e.g., in defining the semantic mappings as fixed points of (contractive) higher order operators. By Banach's theorem, such fixed points are unique, yielding another important proof principle for our paper.

Introduction

In recent years, the study of higher order programming notions has become a central topic in the field of semantics. Seminal in this development have been two schools of research, viz. that of (typed) λ-calculus in the area of functional programming (see, e.g., [Bar92] for a survey of the current situation), and that of higher order processes in the theory of concurrency (see, e.g., [AR87, Tho90, MPW92]). ([LTLG92] can be used for a quick overview of much of the relevant literature.) The aim of the present paper is to provide another perspective on this problem area by studying higher order notions embedded in the traditional setting of imperative languages. First, we study *second order assignment*: the

[*] This work was partially supported by the Netherlands *Nationale Faciliteit Informatica* programme, project Research and Education in Concurrent Systems (REX).

statement $x := s$, for x a procedure variable and s a statement, assigns s to x. In the operational semantics, this is modelled by storing (the syntactic entity) s in the current 'syntactic' state. Denotationally, the (function which is the) meaning of s is stored in the 'semantic' state. The second notion we study is *second order communication*. Recall that in a CSP- or occam-like language value-passing communication is expressed by the two actions $c\,!\,e$ and $c\,?\,v$ occurring in two parallel components (c a channel, e some expression, and v an individual variable), and synchronised execution of these actions results in the transmission of the current value of e to v. A second order variant of this is the pair of communication constructs $c\,!\,s$ and $c\,?\,x$ (c, s, and x as above). Now a higher order value is passed at the moment of synchronised execution: in the operational semantics, we pass s (again a syntactic object); denotationally, the meaning of s is transmitted.

Though these notions are, we hope, conceptually quite simple, a not so simple arsenal of semantic tools is necessary to make the ideas just sketched precise, and to obtain a full picture of the relationships between the operational ($\mathcal{O}$) and denotational ($\mathcal{D}$) models. In both kinds of models, topological techniques play an essential rôle. More specifically, we work in a category of 1-bounded complete ultrametric spaces, and a variety of functors on this category is used to specify the domains we work with. (This type of domain equations originated with [BZ82]; the general theory is due to [AR89]. See also [BR92] for many further applications.)

For reasons of presentation, in addition to the languages with higher order assignment ($\mathcal{L}_{as_2}$) and communication ($\mathcal{L}_{co_2}$) we also discuss two simpler languages with only first order assignment ($\mathcal{L}_{as}$) and communication ($\mathcal{L}_{co}$), respectively. This allows a more leisurely development of the machinery: in particular, we are able to demonstrate in a simple setting another higher order phenomenon which is pervasive in this paper, viz. the use of (contractive) higher order mappings in both the definition and the comparison of semantic meaning functions. Each of the $\mathcal{O}$'s or $\mathcal{D}$'s to be defined is obtained as (unique) fixed point of some higher order mapping $\Phi_{\mathcal{O}}$ or $\Phi_{\mathcal{D}}$. By the uniqueness property, in order to establish $\mathcal{O} = \mathcal{D}$, it suffices to show, e.g., that $\Phi_{\mathcal{O}}(\mathcal{D}) = \mathcal{D}$.

The definition of each of the $\mathcal{O}$'s follows the customary pattern in that it is derived from some (SOS-style) transition system ([Plo81]). Mostly, these systems are *finitely branching*, a property on which the *compactness* of the resulting sets of meanings is based. However, in the comparative study of $\mathcal{L}_{co_2}$ we need a generalisation to *compactly branching* transition systems. This is, in turn, based on an extension of the metric framework consisting in the introduction of a metric on the configurations of the transition system (rather than only having a metric based on the standard distance between sequences of actions generated by successive transitions).

The key idea in the semantic analysis of $\mathcal{L}_{as_2}$ is the introduction of both syntactic and semantic states, and of a suitable mapping linking the two. Whereas the syntactic states are an immediate extension of those used for $\mathcal{L}_{as}$, the set of semantic states requires a system of (reflexive) domain equations for its specification. Once the appropriate definitions have become available, a concise (statement and) proof of the relationship between $\mathcal{O}$ and $\mathcal{D}$ is possible, thanks to the rather powerful general methodology.

The first order language $\mathcal{L}_{co}$ is a fairly typical language with imperative concurrency. Our design of $\mathcal{O}$ for $\mathcal{L}_{co}$ exhibits only some mild variations compared with the traditional approach. The denotational $\mathcal{D}$ is based on a 'branching time' process domain P of the 'nonuniform' variety (processes have a functional dependence on the state). It is not difficult to show (and implicit in [BZ82]) that P is *strongly extensional*: with a slight adaptation of the usual definition of bisimilarity, we have that bisimilarity on P coincides with identity. The various semantic operators on P may as well be defined by higher order techniques. The relationship between $\mathcal{O}$ and $\mathcal{D}$ for $\mathcal{L}_{co}$ involves a *trace* mapping from the denotational 'branching time' to the operational 'linear time' domain: among others, the branching structure is collapsed, and failing attempts at communication are deleted (and deadlock is delivered if no 'proper' action remains).

The paper culminates in the semantic study of $\mathcal{L}_{co_2}$, bringing a synthesis of many of the earlier techniques. The denotational domain, albeit rather complex due to the use of three domain equations, allows an appealingly simple denotational definition. This domain can also be shown to be strongly extensional (with some higher order generalisation of the bisimilarity definition, cf., e.g., [AGR92, MS92]). More work is needed to link $\mathcal{O}$ and $\mathcal{D}$. First, an idea already used for $\mathcal{L}_{co}$, viz. to design a variant of $\mathcal{O}$ delivering results in the denotational domain, is applied again. However, for $\mathcal{L}_{co_2}$ a complication arises, inducing the appearance of 'processes as terms' ([Rut92]). Also, this is the point where, as signalled earlier, a compactly branching transition system appears, a notion which presupposes a metric on the configurations ([Bre94]). In the final stage of the proof relating $\mathcal{O}$ and $\mathcal{D}$, a lemma relating the transitions of both the original system (on which $\mathcal{O}$ for $\mathcal{L}_{co_2}$ is based) and of the extended system (in which the configurations may involve semantic processes) provides the key technical step.

In the final section, the paper summarises the relationships between $\mathcal{O}$ and $\mathcal{D}$ for the four languages considered. We see as one of the achievements of our paper the transparency of the successive refinements, going from the simple $\mathcal{O} = \mathcal{D}$ result for $\mathcal{L}_{as}$ to the more elaborate theorem for $\mathcal{L}_{co_2}$.

We conclude this introduction with some remarks on related work. The idea to handle second order assignment $x := s$ through the storing of a pair (x, s) in the (syntactic) state is close to the explicit substitution (in the framework of the λ-calculus) of [Cur88, ACCL90], albeit that some stack-like nesting of states - omitted in this paper not to overload the presentation - would be needed to allow a full correspondence. The language $\mathcal{L}_{co_2}$ should, after some massaging of the specific operator for parallelism, be able to at least model a key part of Thomsen's CHOCS ([Tho89, Tho90]), viz. that sublanguage which he uses to encode the lazy λ-calculus. However, a precise statement and, especially, a full proof of this claim demands a lot of further work. Other connections to explore include the relationships with the π-calculus ([MPW92, Mil92]), the higher order π-calculus ([San92, San93]), and the γ-calculus ([Bou89, BB92], cf. also [JP90]). In the π-

calculus, channel names are transmitted rather than processes, so an immediate correspondence is not to be expected. For another reason, the same holds for the γ-calculus: the notion of sequential composition used there is essentially different from ours.

1 A sequential language with assignment

The first language we discuss, viz. $\mathcal{L}_{as}$, is quite simple, and chosen especially to illustrate the use of higher order techniques in defining and relating semantic models. Also, it prepares the way for the more interesting language with second order assignment considered in the next section. For $\mathcal{L}_{as}$, we shall define both $\mathcal{O}$ (operational) and $\mathcal{D}$ (denotational) semantics as (unique) fixed point of a suitable contractive mapping [1]. Banach's theorem[2] applies, since all spaces involved are complete. The semantics $\mathcal{O}$ and $\mathcal{D}$ shall be related by showing that both are fixed points of the same contractive mapping.

Let $(v \in) \mathit{IVar}$, $(x \in) \mathit{PVar}$ be alphabets of individual and procedure variables. Let $(e \in) \mathit{Exp}$ be a class of simple expressions (syntax left unspecified).

Definition 1. The language $\mathcal{L}_{as}$ is defined by

$$s ::= v := e \mid s \,;\, s \mid s + s \mid x \mid \mu x\,[s].$$

The prefix μx binds occurrences of procedure variable x. Our semantic definitions will throughout be given for closed constructs (no free procedure variables) only. To define the operational semantics we shall use transition systems. The configurations of the transition system are pairs of resumptions and states.

Definition 2. The class Res_1 of resumptions is defined by

$$r ::= \mathrm{E} \mid s : r.$$

The set State_1 of states is defined by

$$(\sigma \in) \mathit{State}_1 = \mathit{IVar} \to \mathit{Val},$$

for $(\alpha \in) \mathit{Val}$ some set of values.

The (empty) resumption E will be used to denote termination. The state $\sigma\{\alpha/v\}$ has value α in v and equals σ elsewhere. Let $\mathcal{V}(e)(\sigma)$ denote the value of expression e in state σ. Let $s\{s'/x\}$ denote syntactic substitution of statement s' for the free occurrences of procedure variable x in statement s. The transition system $\mathcal{T}_1$ is introduced in

[1] Let (X, d_X) and $(X', d_{X'})$ be metric spaces. A function $f : X \to X'$ is called contractive if there exists an ϵ, with $0 \leq \epsilon < 1$, such that, for all x and x',

$$d_{X'}(f(x), f(x')) \leq \epsilon \cdot d_X(x, x').$$

[2] Let (X, d_X) be a complete metric space. If $f : X \to X$ is contractive then f has a unique fixed point $\mathit{fix}(f)$ (cf. [Ban22]).

Definition 3. The transition relation $\rightarrow$ of $\mathcal{T}_1$ is the smallest subset of $(\mathit{Res}_1 \times \mathit{State}_1) \times (\mathit{Res}_1 \times \mathit{State}_1)$ satisfying the rules given below. A rule of the form

$$\text{if } [r_1, \sigma_1] \rightarrow [r, \sigma] \text{ then } [r_2, \sigma_2] \rightarrow [r, \sigma]$$

will be abbreviated to

$$[r_2, \sigma_2] \rightarrow_0 [r_1, \sigma_1];$$

the 0-subscript indicates that we have here a *zero-step* transition.

$$\begin{array}{lll}
(1) & [v := e : r, \sigma] & \rightarrow \; [r, \sigma\{\alpha/v\}], \text{ where } \alpha = \mathcal{V}(e)(\sigma) \\
(2) & [(s_1 ; s_2) : r, \sigma] & \rightarrow_0 [s_1 : (s_2 : r), \sigma] \\
(3) & [(s_1 + s_2) : r, \sigma] & \rightarrow_0 [s_1 : r, \sigma] \\
(4) & [(s_1 + s_2) : r, \sigma] & \rightarrow_0 [s_2 : r, \sigma] \\
(5) & [\mu x [s] : r, \sigma] & \rightarrow \; [s\{\mu x [s]/x\} : r, \sigma]
\end{array}$$

In the operational semantics we collect successive transitions. Each resumption is mapped to an element of the semantic domain P_1 presented in

Definition 4. The domain P_1 is defined by

$$(p \in) P_1 = \mathit{State}_1 \rightarrow \mathcal{P}_{nc}(\mathit{State}_1^{\infty}).$$

The set $(\varsigma \in) \mathit{State}_1^{\infty} = \mathit{State}_1^{*} \cup \mathit{State}_1^{\omega}$ of finite and infinite sequences of states is endowed with the 1-bounded complete ultrametric d specified by

$$d(\varsigma, \varsigma') = \begin{cases} 0 & \text{if } \varsigma = \varsigma' \\ 2^{-n} & \text{otherwise} \end{cases}$$

where n is the length of the longest common prefix of ς and ς'. According to Kuratowski's theorem[3], the set $\mathcal{P}_{nc}(\mathit{State}_1^{\infty})$ of nonempty compact subsets of $\mathit{State}_1^{\infty}$ endowed with the Hausdorff metric is a 1-bounded complete ultrametric space.

Definition 5. The higher order mapping $\Phi_{\mathcal{O}^*} : (\mathit{Res}_1 \rightarrow P_1) \rightarrow (\mathit{Res}_1 \rightarrow P_1)$ is defined by

$$\begin{aligned}
\Phi_{\mathcal{O}^*}(\phi)(\mathrm{E}) &= \lambda\sigma . \{\varepsilon\} \\
\Phi_{\mathcal{O}^*}(\phi)(s : r) &= \lambda\sigma . \bigcup \{ \sigma' \cdot \phi(r')(\sigma') \mid [s : r, \sigma] \rightarrow [r', \sigma'] \}
\end{aligned}$$

The operational semantics $\mathcal{O}^* : \mathit{Res}_1 \rightarrow P_1$ is defined by

$$\mathcal{O}^* = \mathit{fix}(\Phi_{\mathcal{O}^*}).$$

[3] If (X, d_X) is a 1-bounded complete ultrametric space then the set of nonempty and compact subsets of X, $\mathcal{P}_{nc}(X)$, endowed with the Hausdorff metric based on d_X is a 1-bounded complete ultrametric space (cf. [Kur56]).

In the above definition of $\Phi_{\mathcal{O}^*}$, $\sigma' \cdot \phi(r')(\sigma')$ is the result of prefixing the set of state sequences $\phi(r')(\sigma')$ by the state σ'. The well-definedness proof of $\Phi_{\mathcal{O}^*}$ exploits the fact that $\mathcal{T}_1$ is finitely branching. Obviously, $\Phi_{\mathcal{O}^*}$ is contractive. According to Banach's theorem, $\Phi_{\mathcal{O}^*}$ has a unique fixed point.

Definition 6. The operational semantics $\mathcal{O} : \mathcal{L}_{as} \to P_1$ is defined by

$$\mathcal{O}(s) = \mathcal{O}^*(s : \mathrm{E}).$$

In the denotational semantics, we restrict ourselves to nonexpansive mappings[4] (notation $\to^1$).

Definition 7. The higher order mapping

$$\Phi_{\mathcal{D}} : (\mathcal{L}_{as} \to P_1 \to^1 P_1) \to (\mathcal{L}_{as} \to P_1 \to^1 P_1)$$

is defined by

$$\begin{aligned}
\Phi_{\mathcal{D}}(\phi)(v := e)(p) &= \lambda\sigma . (\sigma\{\alpha/v\} \cdot p(\sigma\{\alpha/v\})), \text{ where } \alpha = \mathcal{V}(e)(\sigma) \\
\Phi_{\mathcal{D}}(\phi)(s_1 ; s_2)(p) &= \Phi_{\mathcal{D}}(\phi)(s_1)(\Phi_{\mathcal{D}}(\phi)(s_2)(p)) \\
\Phi_{\mathcal{D}}(\phi)(s_1 + s_2)(p) &= \lambda\sigma . (\Phi_{\mathcal{D}}(\phi)(s_1)(p)(\sigma) \cup \Phi_{\mathcal{D}}(\phi)(s_2)(p)(\sigma)) \\
\Phi_{\mathcal{D}}(\phi)(\mu x[s])(p) &= \lambda\sigma . (\sigma \cdot \phi(s\{\mu x[s]/x\})(p)(\sigma))
\end{aligned}$$

The denotational semantics $\mathcal{D} : \mathcal{L}_{as} \to P_1 \to^1 P_1$ is defined by

$$\mathcal{D} = \mathit{fix}(\Phi_{\mathcal{D}}).$$

The nonexpansiveness of $\Phi_{\mathcal{D}}(\phi)(s)$ and the contractiveness of $\Phi_{\mathcal{D}}$ can be proved by structural induction. Note that this definition of $\mathcal{D}$ implies, e.g., that $\mathcal{D}(\mu x[s])(p) = \lambda\sigma . (\sigma \cdot \mathcal{D}(s\{\mu x[s]/x\})(p)(\sigma))$. Well-definedness of $\mathcal{D}$ is a consequence of the contractiveness of $\Phi_{\mathcal{D}}$ (here ensured by the σ-step) rather than of a direct argument by structural induction on s.

Definition 8. The denotational semantics $\mathcal{D}^* : \mathit{Res}_1 \to P_1$ is defined by

$$\begin{aligned}
\mathcal{D}^*(\mathrm{E}) &= \lambda\sigma . \{\varepsilon\} \\
\mathcal{D}^*(s : r) &= \mathcal{D}(s)(\mathcal{D}^*(r))
\end{aligned}$$

The operational and denotational semantics are related in

Theorem 9. $\mathcal{O}^* = \mathcal{D}^*$.

Proof. For this theorem, we will sketch two alternative proofs.

[4] Let (X, d_X) and $(X', d_{X'})$ be metric spaces. A function $f : X \to X'$ is called nonexpansive if, for all x and x',

$$d_{X'}(f(x), f(x')) \le d_X(x, x').$$

1. We can prove that, for all r,

   $$\Phi_{\mathcal{O}^*}(\mathcal{D}^*)(r) = \mathcal{D}^*(r)$$

   by induction on the complexity of r. For example, for the resumption $(s_1 ; s_2) : r$ we have that

   $$\begin{aligned} &\Phi_{\mathcal{O}^*}(\mathcal{D}^*)((s_1 ; s_2) : r) \\ &= \Phi_{\mathcal{O}^*}(\mathcal{D}^*)(s_1 : (s_2 : r)) && \text{[the definition of the complexity is such} \\ &= \mathcal{D}^*(s_1 : (s_2 : r)) && \text{that the induction hypothesis applies here]} \\ &= \mathcal{D}(s_1)(\mathcal{D}(s_2)(\mathcal{D}^*(r))) \\ &= \mathcal{D}^*((s_1 ; s_2) : r). \end{aligned}$$

   Since $\mathcal{O}^*$ and $\mathcal{D}^*$ are both fixed point of $\Phi_{\mathcal{O}^*}$ and $\Phi_{\mathcal{O}^*}$ has a unique fixed point, $\mathcal{O}^*$ and $\mathcal{D}^*$ must be equal.
2. We can also prove that, for all r,

   $$d(\mathcal{O}^*(r), \mathcal{D}^*(r)) \leq \tfrac{1}{2} \cdot \sup\{ d(\mathcal{O}^*(r'), \mathcal{D}^*(r')) \mid r' \in Res_1 \}$$

   by induction on the complexity of r. For example, for the resumption $v{:=}e{:}r$ we have that

   $$\begin{aligned} &d(\mathcal{O}^*(v := e : r), \mathcal{D}^*(v := e : r)) \\ &= d(\lambda\sigma . (\sigma\{\alpha/v\} \cdot \mathcal{O}^*(r)(\sigma\{\alpha/v\})), \lambda\sigma . (\sigma\{\alpha/v\} \cdot \mathcal{D}^*(r)(\sigma\{\alpha/v\}))) \\ &= \tfrac{1}{2} \cdot d(\mathcal{O}^*(r), \mathcal{D}^*(r)) \\ &\leq \tfrac{1}{2} \cdot \sup\{ d(\mathcal{O}^*(r'), \mathcal{D}^*(r')) \mid r' \in Res_1 \}. \end{aligned}$$

   Consequently, for all r, $d(\mathcal{O}^*(r), \mathcal{D}^*(r)) = 0$. Hence $\mathcal{O}^* = \mathcal{D}^*$.

□

The first proof follows [KR90] (cf. [BM88]), but with a substantial simplification thanks to our avoiding procedure environments.

Corollary 10. *For all s, $\mathcal{O}(s) = \mathcal{D}(s)(\lambda\sigma . \{\varepsilon\})$.*

2 A sequential lânguage with second order assignment

The central notion of this section is second order assignment, in the form of the statement $x := s$, for s itself a statement. In the operational semantics, the routine (program text) s is stored in the syntactic state σ as value for x; in the denotational semantics, the meaning $\mathcal{D}(s)$ is stored as value for x in the semantic state ρ. The definition of $\mathcal{O}$ and $\mathcal{D}$ for $\mathcal{L}_{a,s_2}$ allows a particularly succinct (statement and) proof of the relationship between $\mathcal{O}$ and $\mathcal{D}$.

Definition 11. The language $\mathcal{L}_{a,s_2}$ is defined by

$$s ::= v := e \mid s ; s \mid s + s \mid x \mid x := s.$$

The configurations of the transition system defining the operational semantics are pairs of resumptions (defined as in the previous section, but now named Res_2) and ***syntactic states***, which are introduced in

Definition 12. The set $SynState_2$ of syntactic states is defined by

$$(\sigma \in) \, SynState_2 = (IVar \rightarrow Val) \times (PVar \rightarrow \mathcal{L}_{as_2}).$$

Let, for the state $\sigma = (\sigma_1, \sigma_2)$, the states $\sigma\{\alpha/v\}$ and $\sigma\{s/x\}$ be short for $(\sigma_1\{\alpha/v\}, \sigma_2)$ and $(\sigma_1, \sigma_2\{s/x\})$, respectively. The transition system $\mathcal{T}_2$ is introduced in

Definition 13. The transition relation $\rightarrow$ of $\mathcal{T}_2$ is the smallest subset of $(Res_2 \times SynState_2) \times (Res_2 \times SynState_2)$ satisfying (1), (2), (3), (4) from Definition 3, and

$$\begin{array}{lll} (6) & [x : r, \sigma] & \rightarrow [\sigma(x) : r, \sigma] \\ (7) & [x := s : r, \sigma] & \rightarrow [r, \sigma\{s/x\}] \end{array}$$

The definitions of $\mathcal{O}^*$ and $\mathcal{O}$ follow those of $\mathcal{O}^*$ and $\mathcal{O}$ of the previous section, but now using transition system $\mathcal{T}_2$ and semantic domain P_2, which is obtained from P_1 by replacing $State_1$ by $SynState_2$. We next present the (system of) domain equations[5] for the collection of *semantic states* $SemState_2$ and P_3, the denotational domain for $\mathcal{L}_{as_2}$.

Definition 14. The domains $SemState_2$ and P_3 are defined by

$$\begin{array}{lll} (\rho \in) \, SemState_2 & \cong & (IVar \rightarrow Val) \times (PVar \rightarrow id_{\frac{1}{2}}(P_3 \rightarrow^1 P_3)) \\ (p \in) \, P_3 & \cong & SemState_2 \rightarrow^1 \mathcal{P}_{nc}(SemState_2^\infty) \end{array}$$

Definition 15. The denotational semantics $\mathcal{D} : \mathcal{L}_{as_2} \rightarrow P_3 \rightarrow^1 P_3$ is defined by

$$\begin{array}{ll} \mathcal{D}(v := e)(p) & = \lambda\rho \, . \, (\rho\{\alpha/v\} \cdot p(\rho\{\alpha/v\})), \text{ where } \alpha = \mathcal{V}(e)(\rho) \\ \mathcal{D}(s_1 \, ; s_2)(p) & = \mathcal{D}(s_1)(\mathcal{D}(s_2)(p)) \\ \mathcal{D}(s_1 + s_2)(p) & = \lambda\rho \, . \, (\mathcal{D}(s_1)(p)(\rho) \cup \mathcal{D}(s_2)(p)(\rho)) \\ \mathcal{D}(x)(p) & = \lambda\rho \, . \, (\rho \cdot \rho(x)(p)(\rho)) \\ \mathcal{D}(x := s)(p) & = \lambda\rho \, . \, (\rho\{\psi/x\} \cdot p(\rho\{\psi/x\})), \text{ where } \psi = \mathcal{D}(s) \end{array}$$

[5] To solve these domain equations, we work in a category of 1-bounded complete ultrametric spaces and apply the methodology of solving domain equations in this category as developed in [AR89]. Functors F appearing in domain equations $X \cong F(X)$ - or rather $(X, d_X) \cong F(X, d_X)$ - with $\cong$ denoting isometry, may be built from the familiar operations on 1-bounded complete ultrametric spaces such as Cartesian product, disjoint union, (nonexpansive) function space, and (nonempty) compact power set, and the operation $id_{1/2}$ $(id_{1/2}(X, d_X) = (X, \frac{1}{2} \cdot d_X))$, starting from given 1-bounded complete ultrametric spaces (A, d_A) and the unknown space (X, d_X). The operation $id_{1/2}$ is used in particular to ensure contractiveness of the functor F, which induces uniqueness of the solution up to isometry.

The denotational semantics $\mathcal{D}$ closely follows the structure of the rules in transition system $\mathcal{T}_2$. Consider, for example, the case that a rule $[r, \sigma] \rightarrow [r', \sigma']$ (or $[r, \sigma] \rightarrow_0 [r', \sigma']$) is the sole rule for configuration $[r, \sigma]$ in $\mathcal{T}_2$. Let p and p' denote the denotational meanings of r and r', and let ρ and ρ' be the semantic states corresponding to σ and σ' (cf. Definition 16). Then the formula $p(\rho) = \rho' \cdot p'(\rho')$ (or $p(\rho) = p'(\rho')$) expresses the denotational counterpart of this rule. In this way the clause for $\mathcal{D}(x)(p)(\rho)$ may be understood from clause (6) of Definition 13.

The definition of $\mathcal{D}^*$ follows that of $\mathcal{D}^*$ of the previous section. To each syntactic state a corresponding semantic state is assigned by the mapping *sem* introduced in

Definition 16. The mapping $sem : SynState_2 \rightarrow SemState_2$ is defined by

$$sem(\sigma) = (\sigma_1, \lambda x \,.\, \lambda p \,.\, \mathcal{D}(\sigma_2(x))(p)).$$

The mapping *sem* is extended in the natural way to a mapping from $\mathcal{P}_{nc}(SynState_2^\infty)$ to $\mathcal{P}_{nc}(SemState_2^\infty)$. By means of this mapping the operational and denotational semantics are related in

Theorem 17. *For all r and σ, $sem(\mathcal{O}^*(r)(\sigma)) = \mathcal{D}^*(r)(sem(\sigma))$.*

Proof. This proof follows the second proof of Theorem 9. For example, for resumption $x := s : r$ we have that

$$\begin{aligned}
& d(sem(\mathcal{O}^*(x := s : r)(\sigma)), \mathcal{D}^*(x := s : r)(sem(\sigma))) \\
&= d(sem(\sigma\{s/x\} \cdot \mathcal{O}^*(r)(\sigma\{s/x\})), \mathcal{D}(x := s)(\mathcal{D}^*(r))(sem(\sigma))) \\
&= d(sem(\sigma\{s/x\}) \cdot sem(\mathcal{O}^*(r)(\sigma\{s/x\})), \\
&\qquad sem(\sigma)\{\mathcal{D}(s)/x\} \cdot \mathcal{D}^*(r)(sem(\sigma)\{\mathcal{D}(s)/x\})) \\
&= \tfrac{1}{2} \cdot d(sem(\mathcal{O}^*(r)(\sigma\{s/x\})), \mathcal{D}^*(r)(sem(\sigma)\{\mathcal{D}(s)/x\})) \\
&\leq \tfrac{1}{2} \cdot \sup\{d(sem(\mathcal{O}^*(r')(\sigma')), \mathcal{D}^*(r')(sem(\sigma')) \mid r' \in Res_2, \sigma' \in State_2\},
\end{aligned}$$

since $sem(\sigma\{s/x\}) = sem(\sigma)\{\mathcal{D}(s)/x\}$.

□

Corollary 18. *For all s and σ, $sem(\mathcal{O}(s)(\sigma)) = \mathcal{D}(s)(\lambda\rho \,.\, \{\varepsilon\})(sem(\sigma))$.*

3 A parallel language with communication

The language $\mathcal{L}_{co}$ studied here has first order communication (synchronised transmission of simple values) as its main concept. $\mathcal{L}_{co}$ is close to a language such as CSP ([Hoa85]); again, its main motivation in the present context is to pave the way for the second order variant. A further simplification with respect to the usual languages of this kind is that we assume one global state, rather than a distribution of local states over the various parallel components. The design of a mechanism for local states is well-understood (see, e.g., [ABKR89]), and we have kept it separate from the present development in order not to burden the presentation.

Let ($c \in$) *Chan* be an alphabet of channel names.

Definition 19. The language $\mathcal{L}_{co}$ is defined by

$$s ::= v := e \mid c\,!\,e \mid c\,?\,v \mid s\,;s \mid s+s \mid s \parallel s \mid x \mid \mu x\,[s].$$

The configurations of the transition system are pairs of resumptions and extended states.

Definition 20. The class Res_3 of resumptions is defined by

$$r ::= \mathrm{E} \mid s.$$

The set $State_3$ of states is defined by

$$(\sigma \in) State_3 = State_1.$$

The set $State_3^{ext}$ of extended states is defined by

$$(\eta \in) State_3^{ext} = State_3 \cup (Chan \times Val) \cup (Chan \times IVar).$$

In the transition system, we will use the extended state (c, α) to denote that the value α is sent on channel c, and we will use (c, v) to denote that the value received on channel c should be assigned to the individual variable v. The transition system $\mathcal{T}_3$ is introduced in

Definition 21. The transition relation $\rightarrow$ of $\mathcal{T}_3$ is the smallest subset of $(Res_3 \times State_3^{ext}) \times (Res_3 \times State_3^{ext})$ satisfying

(1) $[v := e,\ \sigma] \rightarrow [\mathrm{E},\ \sigma\{\alpha/v\}]$, where $\alpha = \mathcal{V}(e)(\sigma)$

(2) $[c\,!\,e,\ \sigma] \rightarrow [\mathrm{E},\ (c, \alpha)]$, where $\alpha = \mathcal{V}(e)(\sigma)$

(3) $[c\,?\,v,\ \sigma] \rightarrow [\mathrm{E},\ (c, v)]$

(4) $[s_1 + s_2,\ \sigma] \rightarrow_0 [s_1,\ \sigma]$

(5) $[s_1 + s_2,\ \sigma] \rightarrow_0 [s_2,\ \sigma]$

(6) $[\mu x\,[s],\ \sigma] \rightarrow [s\{\mu x\,[s]/x\},\ \sigma]$

(7) if $[s_1,\ \sigma] \rightarrow [r_1,\ \eta]$ then $[s_1\,;s_2,\ \sigma] \rightarrow [r_1\,;s_2,\ \eta]$

(8) if $[s_1,\ \sigma] \rightarrow [r_1,\ \eta]$ then $[s_1 \parallel s_2,\ \sigma] \rightarrow [r_1 \parallel s_2,\ \eta]$

(9) if $[s_2,\ \sigma] \rightarrow [r_2,\ \eta]$ then $[s_1 \parallel s_2,\ \sigma] \rightarrow [s_1 \parallel r_2,\ \eta]$

(10) if $[s_1,\ \sigma] \rightarrow [r_1,\ (c, \alpha)]$ and $[s_2,\ \sigma] \rightarrow [r_2,\ (c, v)]$
then $[s_1 \parallel s_2,\ \sigma] \rightarrow [r_1 \parallel r_2,\ \sigma\{\alpha/v\}]$

(11) if $[s_1,\ \sigma] \rightarrow [r_1,\ (c, v)]$ and $[s_2,\ \sigma] \rightarrow [r_2,\ (c, \alpha)]$
then $[s_1 \parallel s_2,\ \sigma] \rightarrow [r_1 \parallel r_2,\ \sigma\{\alpha/v\}]$

In the above, we adopt the convention that $\text{E} \,;\, s = \text{E} \parallel s = s \parallel \text{E} = s$, and $\text{E} \parallel \text{E} = \text{E}$. We say that $[s, \sigma]$ *blocks* if there do not exist a resumption r and a state (not an extended state) σ' such that $[s, \sigma] \to [r, \sigma']$. The semantic domain for the operational semantics is introduced in

Definition 22. The domain P_4 is defined by

$$(p \in) P_4 = State_3 \to \mathcal{P}_{nc}((State_3)^{\infty}_{\delta}).$$

The set $(\varsigma \in)(State_3)^{\infty}_{\delta} = State_3^* \cup State_3^{\omega} \cup State_3^* \cdot \{\delta\}$ of finite and infinite sequences of states possibly ending with δ is endowed with the ultrametric described after Definition 4.

Definition 23. The operational semantics $\mathcal{O}^* : Res_3 \to P_4$ is the unique mapping satisfying

$$\mathcal{O}^*(\text{E}) = \lambda\sigma \,.\, \{\varepsilon\}$$

$$\mathcal{O}^*(s) = \lambda\sigma \,.\, \begin{cases} \{\delta\} & \text{if } [s, \sigma] \text{ blocks} \\ \bigcup \{ \sigma' \cdot \mathcal{O}^*(r)(\sigma') \mid [s, \sigma] \to [r, \sigma'] \} & \text{otherwise} \end{cases}$$

The operational semantics $\mathcal{O}$ is defined as the restriction of $\mathcal{O}^*$ to $\mathcal{L}_{co}$. It is important to observe that $\mathcal{O}^*$, and hence $\mathcal{O}$, is not compositional, i.e. there is no semantic operator $\parallel$ satisfying $\mathcal{O}^*(s_1 \parallel s_2) = \mathcal{O}^*(s_1) \parallel \mathcal{O}^*(s_2)$.

The semantic domain for the denotational semantics is presented in

Definition 24. The domain P_5 is defined by

$$(p \in) P_5 \cong \{\text{E}\} \,\bar{\cup}\, (State_3 \to \mathcal{P}_{co}(State_3^{ext} \times id_{\frac{1}{2}}(P_5))).$$

In the above definition, $\bar{\cup}$ denotes the disjoint union and $\mathcal{P}_{co}$ the compact power set operator. The domain P_5 is a branching domain. Its core structure is as that of a P_5' solving $P_5' \cong \mathcal{P}_{co}(State_3^{ext} \times id_{\frac{1}{2}}(P_5'))$; additional structure is provided by the nil process E and by P_5's functional dependence on arguments in $State_3$. It is not difficult to define (a natural extension of) bisimilarity (notation $\sim$) on P_5, and to show that P_5 is strongly extensional, viz. $p_1 \sim p_2$ if and only if $p_1 = p_2$ (cf. [RT92, Bre93]).

Definition 25. The operator $;$: $P_5 \times P_5 \to^1 P_5$ is the unique mapping satisfying

$$p_1 \,;\, p_2 = \begin{cases} p_2 & \text{if } p_1 = \text{E} \\ \lambda\sigma \,.\, \{ (\eta, p_1' \,;\, p_2) \mid (\eta, p_1') \in p_1(\sigma) \} & \text{otherwise} \end{cases}$$

The operator $+ : P_5 \times P_5 \to^1 P_5$ is defined by

$$p_1 + p_2 = \begin{cases} p_2 & \text{if } p_1 = \text{E} \\ p_1 & \text{if } p_2 = \text{E} \\ \lambda\sigma \,.\, (p_1(\sigma) \cup p_2(\sigma)) & \text{otherwise} \end{cases}$$

The operator $\parallel : P_5 \times P_5 \to^1 P_5$ is the unique mapping satisfying

$$p_1 \parallel p_2 = (p_1 \mathbin{\lfloor\!\lfloor} p_2) + (p_2 \mathbin{\lfloor\!\lfloor} p_1) + (p_1 \mathbin{\lfloor} p_2) + (p_2 \mathbin{\lfloor} p_1)$$

where

$$p_1 \mathbin{\lfloor\!\lfloor} p_2 = \begin{cases} p_2 & \text{if } p_1 = \mathrm{E} \\ \lambda\sigma \,.\, \{\, (\eta, p_1' \parallel p_2) \mid (\eta, p_1') \in p_1(\sigma) \,\} & \text{otherwise} \end{cases}$$

and, for $p_1 = \mathrm{E}$ or $p_2 = \mathrm{E}$,

$$p_1 \mathbin{\lfloor} p_2 = \mathrm{E},$$

otherwise

$$p_1 \mathbin{\lfloor} p_2 = \lambda\sigma \,.\, \{\, (\sigma\{\alpha/v\}, p_1' \parallel p_2') \mid ((c,\alpha), p_1') \in p_1(\sigma),\ ((c,v), p_2') \in p_2(\sigma) \,\}.$$

The above definition can be made rigorous by another appeal to higher order techniques. For example, for the operator ; we should introduce a higher order mapping $\Phi_; : (P_5 \times P_5 \to^1 P_5) \to (P_5 \times P_5 \to^1 P_5)$ defined by

$$\Phi_;(\phi)(p_1, p_2) = \begin{cases} p_2 & \text{if } p_1 = \mathrm{E} \\ \lambda\sigma \,.\, \{\, (\eta, \phi(p_1', p_2)) \mid (\eta, p_1') \in p_1(\sigma) \,\} & \text{otherwise} \end{cases}$$

Definition 26. The denotational semantics $\mathcal{D} : \mathcal{L}_{co} \to P_5$ is the unique mapping satisfying

$$\begin{aligned}
\mathcal{D}(v := e) &= \lambda\sigma \,.\, \{(\sigma\{\alpha/v\}, \mathrm{E})\}, \text{ where } \alpha = \mathcal{V}(e)(\sigma) \\
\mathcal{D}(c\,!\,e) &= \lambda\sigma \,.\, \{((c,\alpha), \mathrm{E})\}, \text{ where } \alpha = \mathcal{V}(e)(\sigma) \\
\mathcal{D}(c\,?\,v) &= \lambda\sigma \,.\, \{((c,v), \mathrm{E})\} \\
\mathcal{D}(s_1 \,;\, s_2) &= \mathcal{D}(s_1) \,;\, \mathcal{D}(s_2) \\
\mathcal{D}(s_1 + s_2) &= \mathcal{D}(s_1) + \mathcal{D}(s_2) \\
\mathcal{D}(s_1 \parallel s_2) &= \mathcal{D}(s_1) \parallel \mathcal{D}(s_2) \\
\mathcal{D}(\mu x[s]) &= \lambda\sigma \,.\, \{(\sigma, \mathcal{D}(s\{\mu x[s]/x\}))\}
\end{aligned}$$

We now prepare the way for the statement relating $\mathcal{O}$ and $\mathcal{D}$. We first define a 'hybrid' operational semantics, based on $\mathcal{T}_3$ but yielding elements in the denotational domain P_5.

Definition 27. The operational semantics $\mathcal{O}^{\#} : Res_3 \to P_5$ is the unique mapping satisfying

$$\begin{aligned}
\mathcal{O}^{\#}(\mathrm{E}) &= \mathrm{E} \\
\mathcal{O}^{\#}(s) &= \lambda\sigma \,.\, \{\, (\eta, \mathcal{O}^{\#}(r)) \mid [s, \sigma] \to [r, \eta] \,\}
\end{aligned}$$

Second, we extend the denotational semantics $\mathcal{D}$ to a denotational semantics $\mathcal{D}^{\#}$ from Res_3 to P_5 by defining $\mathcal{D}^{\#}(\mathrm{E}) = \mathrm{E}$.

Lemma 28. $\mathcal{O}^{\#} = \mathcal{D}^{\#}$.

Proof. Following the first proof of Theorem 9, it suffices to show that the higher order mapping $\Phi_{\mathcal{O}^{\#}}$ underlying Definition 27 has $\mathcal{D}^{\#}$ as fixed point.

□

Finally, we show how the operational semantics $\mathcal{O}^{\#}$ and $\mathcal{O}^{*}$ are connected. Semantic domain $(p_4 \in) P_4$ is simpler than $(p_5 \in) P_5$ in three ways;

- for all σ, the branching structure of $p_5(\sigma)$ is collapsed, leaving in $p_4(\sigma)$ only a set of paths of $p_5(\sigma)$,
- failing attempts at communication (c, α) or (c, v) appear in $p_5(\sigma)$ but not in $p_4(\sigma)$, and
- $p_5(\sigma)$ contains, in general, pairs (σ', p_5'). Here p_5' models the continuation of the execution after σ' has been delivered. This allows that an interleaving action of some $\bar{p}_5$ might change σ' before p_5' is applied. However, this does not hold for $p_4(\sigma)$ which contains sets of the form $\sigma' \cdot p_4'(\sigma')$.

The combined effect of these simplifications is yielded by *trace* defined in

Definition 29. The mapping $\mathit{trace} : P_5 \rightarrow^1 P_4$ is the unique mapping satisfying

$$\mathit{trace}(\mathrm{E}) = \lambda\sigma \,.\, \{\varepsilon\}$$

$$\mathit{trace}(p) = \lambda\sigma \,.\, \begin{cases} \{\delta\} & \text{if } p(\sigma) \text{ blocks} \\ \bigcup \{\, \sigma' \cdot \mathit{trace}(p')(\sigma') \mid (\sigma', p') \in p(\sigma) \,\} & \text{otherwise} \end{cases}$$

where $p(\sigma)$ blocks if there does not exist a pair (σ', p') in $p(\sigma)$.

The well-definedness proof of the higher order mapping Φ_{trace} underlying the above definition relies on Michael's theorem[6].

Lemma 30. $\mathcal{O}^{*} = \mathit{trace} \circ \mathcal{O}^{\#}$.

Proof. Again we can follow the first proof of Theorem 9 by showing that the higher order mapping $\Phi_{\mathcal{O}^{*}}$ underlying Definition 23 has $\mathit{trace} \circ \mathcal{O}^{\#}$ as fixed point. □

Theorem 31. $\mathcal{O} = \mathit{trace} \circ \mathcal{D}$.

4 A parallel language with second order communication

This is the culminating section of our paper, providing a synthesis of ideas from the Sections 2 and 3. In addition, we need some novel techniques to establish the relationship between $\mathcal{O}$ and $\mathcal{D}$ for $\mathcal{L}_{co_2}$. In particular, we use

- the 'processes as terms' approach of [Rut92], and
- a metric on configurations of a transition system ([Bre94]).

As in Section 2, a more realistic language could be based on local states. In such a setting it would be meaningful to transmit a *closure*, a pair consisting of a statement and a local state, rather than just a statement (as we do in the operational model for $\mathcal{L}_{co_2}$).

[6] Let (X, d_X) be a metric space. If $\mathcal{X} \in \mathcal{P}_{co}(\mathcal{P}_{co}(X))$ then $\bigcup \mathcal{X} \in \mathcal{P}_{co}(X)$ (cf. [Mic51]).

Definition 32. The language $\mathcal{L}_{co_2}$ is defined by

$$s ::= v := e \mid s \,;\, s \mid s + s \mid s \,\|\, s \mid x \mid c\,!\,s \mid c\,?\,x.$$

The configurations of the transition system are pairs of resumptions (defined as in the previous section, but now named Res_4) and extended syntactic states.

Definition 33. The set $SynState_4$ of syntactic states is defined by

$$(\sigma \in)\, SynState_4 = (IVar \to Val) \times (PVar \to \mathcal{L}_{co_2}).$$

The class $SynState_4^{ext}$ of extended syntactic states is defined by

$$(\eta \in)\, SynState_4^{ext} = SynState_4 \cup (Chan \times \mathcal{L}_{co_2}) \cup (\overline{Chan} \times PVar),$$

where $\overline{Chan} = \{\, \bar{c} \mid c \in Chan \,\}$.

We introduce $\overline{Chan}$ to avoid a possible ambiguity: we distinguish between the extended state denoting that statement x is sent on channel c - denoted by (c, x) - and the extended state denoting that the statement received on channel c should be assigned to procedure variable x - denoted by $(\bar{c}, x)$. The transition system $\mathcal{T}_4$ is presented in

Definition 34. The transition relation $\to$ of $\mathcal{T}_4$ is the smallest subset of $(Res_4 \times SynState_4^{ext}) \times (Res_4 \times SynState_4^{ext})$ satisfying (1), (4), (5), (7), (8), (9) from Definition 21, and

(12) $[x, \sigma] \to [\sigma(x), \sigma]$

(13) $[c\,!\,s, \sigma] \to [\mathrm{E}, (c, s)]$

(14) $[c\,?\,x, \sigma] \to [\mathrm{E}, (\bar{c}, x)]$

(15) if $[s_1, \sigma] \to [r_1, (c, s)]$ and $[s_2, \sigma] \to [r_2, (\bar{c}, x)]$
then $[s_1 \,\|\, s_2, \sigma] \to [r_1 \,\|\, r_2, \sigma\{s/x\}]$

(16) if $[s_1, \sigma] \to [r_1, (\bar{c}, x)]$ and $[s_2, \sigma] \to [r_2, (c, s)]$
then $[s_1 \,\|\, s_2, \sigma] \to [r_1 \,\|\, r_2, \sigma\{s/x\}]$

The definitions of $\mathcal{O}^*$ and $\mathcal{O}$ follow those of $\mathcal{O}^*$ and $\mathcal{O}$ of the previous section, but now using transition system $\mathcal{T}_4$ and semantic domain P_6 introduced in

Definition 35. The domain P_6 is defined by

$$(p \in)\, P_6 = SynState_4 \to \mathcal{P}_{nc}((SynState_4)_\delta^\infty).$$

Next, we define the collection of (extended) semantic states $SemState_4$ ($SemState_4^{ext}$), and the domain P_7 of denotational meanings for $\mathcal{L}_{co_2}$.

Definition 36. The domains $SemState_4$, $SemState_4^{ext}$, and P_7 are defined by

$$\begin{aligned}
(\rho \in)\, SemState_4 &\cong (IVar \to Val) \times (PVar \to id_{\frac{1}{2}}(P_7)) \\
(\xi \in)\, SemState_4^{ext} &\cong SemState_4 \,\bar{\cup}\, (Chan \times id_{\frac{1}{2}}(P_7)) \,\bar{\cup}\, (\overline{Chan} \times PVar) \\
(p \in)\, P_7 &\cong \{\mathrm{E}\} \,\bar{\cup}\, (SemState_4 \to^1 \mathcal{P}_{co}(SemState_4^{ext} \times id_{\frac{1}{2}}(P_7)))
\end{aligned}$$

Note the correspondence of the definitions of the domains $SemState_4$, $SemState_4^{ext}$, and P_7 with those of $SynState_4$, $SynState_4^{ext}$, and P_6, respectively. On domain P_7 we can define (higher order) bisimilarity in several ways. Based on these definitions, the domain can be shown to be strongly extensional. Whether one of the bisimilarity notions gives us the 'right' equivalence needs further study.

Definition 37. The denotational semantics $\mathcal{D} : \mathcal{L}_{co_2} \to P_7$ is defined by

$$\begin{aligned}
\mathcal{D}(v := e) &= \lambda\rho\,.\,\{(\rho\{\alpha/v\}, \mathrm{E})\}, \text{where } \alpha = \mathcal{V}(e)(\rho)\\
\mathcal{D}(s_1\,;\,s_2) &= \mathcal{D}(s_1)\,;\,\mathcal{D}(s_2)\\
\mathcal{D}(s_1 + s_2) &= \mathcal{D}(s_1) + \mathcal{D}(s_2)\\
\mathcal{D}(s_1 \parallel s_2) &= \mathcal{D}(s_1) \parallel \mathcal{D}(s_2)\\
\mathcal{D}(x) &= \lambda\rho\,.\,\{(\rho, \rho(x))\}\\
\mathcal{D}(c\,!\,s) &= \lambda\rho\,.\,\{((c,p), \mathrm{E})\}, \text{ where } p = \mathcal{D}(s)\\
\mathcal{D}(c\,?\,x) &= \lambda\rho\,.\,\{((\bar{c},x), \mathrm{E})\}
\end{aligned}$$

The semantic operators used here are defined quite similarly to those of Definition 25. For example, for the operator $\lfloor$ we have, for $p_1 \neq \mathrm{E}$ and $p_2 \neq \mathrm{E}$,

$$p_1 \lfloor p_2 = \lambda\rho\,.\,\{\,(\rho\{p/x\}, p_1' \parallel p_2') \mid ((c,p),p_1') \in p_1(\rho), ((\bar{c},x),p_2') \in p_2(\rho)\,\}.$$

In order to relate $\mathcal{O}$ and $\mathcal{D}$, we need various preparations. First, we want to mimic the introduction of $\mathcal{O}^{\#}$ (cf. Definition 27), delivering denotational meanings. This requires using ρ's rather than σ's. Clause (12) of Definition 34 then obtains the form $[x, \rho] \to [\rho(x), \rho]$. As a consequence, semantic entities $p \in P_7$ appear in the new $\mathcal{T}_4'$, with respect to the extended class of resumptions introduced in Definition 38. In Definition 39, we introduce the induced transition system. Note that $\mathcal{T}_4'$ is no more finitely branching, and the higher order definition of $\mathcal{O}^{\#}$ based on $\mathcal{T}_4'$ requires separate justification.

Definition 38. The class Res_4' is defined by

$$u ::= \mathrm{E} \mid t$$

where

$$t ::= v := e \mid t\,;t \mid t + t \mid t \parallel t \mid x \mid c\,!\,t \mid c\,?\,x \mid p.$$

Definition 39. The transition relation $\to$ of $\mathcal{T}_4'$ is the smallest subset of $(Res_4' \times SemState_4^{ext}) \times (Res_4' \times SemState_4^{ext})$ satisfying

(1) $[v := e,\ \rho] \ \rightarrow\ [\mathrm{E},\ \rho\{\alpha/v\}]$, where $\alpha = \mathcal{V}(e)(\rho)$

(2) $[t_1 + t_2,\ \rho] \rightarrow_0 [t_1,\ \rho]$

(3) $[t_1 + t_2,\ \rho] \rightarrow_0 [t_2,\ \rho]$

(4) $[x,\ \rho] \ \rightarrow\ [\rho(x),\ \rho]$

(5) $[c\,!\,t,\ \rho] \ \rightarrow\ [\mathrm{E},\ (c, p)]$, where $p = \mathcal{D}^{\#}(t)$ (cf. Definition 43)

(6) $[c\,?\,x,\ \rho] \ \rightarrow\ [\mathrm{E},\ (\bar{c}, x)]$

(7) if $[t_1,\ \rho] \rightarrow [u_1,\ \xi]$ then $[t_1\,;\,t_2,\ \rho] \ \rightarrow [u_1\,;\,t_2,\ \xi]$

(8) if $[t_1,\ \rho] \rightarrow [u_1,\ \xi]$ then $[t_1 \parallel t_2,\ \rho] \rightarrow [u_1 \parallel t_2,\ \xi]$

(9) if $[t_2,\ \rho] \rightarrow [u_2,\ \xi]$ then $[t_1 \parallel t_2,\ \rho] \rightarrow [t_1 \parallel u_2,\ \xi]$

(10) if $[t_1,\ \rho] \rightarrow [u_1,\ (c,p)]$ and $[t_2,\ \rho] \rightarrow [u_2,\ (\bar{c},x)]$
then $[t_1 \parallel t_2,\ \rho] \rightarrow [u_1 \parallel u_2,\ \rho\{p/x\}]$

(11) if $[t_1,\ \rho] \rightarrow [u_1,\ (\bar{c},x)]$ and $[t_2,\ \rho] \rightarrow [u_2,\ (c,p)]$
then $[t_1 \parallel t_2,\ \rho] \rightarrow [u_1 \parallel u_2,\ \rho\{p/x\}]$

(12) if $(\xi, p') \in p(\rho)$ then $[p,\ \rho] \rightarrow [p',\ \xi]$

Definition 40. The operational semantics $\mathcal{O}^{\#} : Res_4' \rightarrow^1 P_7$ is the unique mapping satisfying

$$\mathcal{O}^{\#}(\mathrm{E}) = \mathrm{E}$$
$$\mathcal{O}^{\#}(t) \ = \lambda\rho\,.\,\{\,(\xi, \mathcal{O}^{\#}(u)) \mid [t,\ \rho] \rightarrow [u,\ \xi]\,\}$$

Note that the $\rightarrow^1$ in the above definition assumes a metric on Res_4'. This is presented in

Definition 41. The metric $d : Res_4' \times Res_4' \rightarrow [0,1]$ is defined by

$$d(u,u') = 0$$

if $u \equiv u'$, otherwise

$$d(u,u') = \begin{cases} d_{P_7}(u,u') & \text{if } u \in P_7 \text{ and } u' \in P_7 \\ \max\{d(t_1,t_1'), d(t_2,t_2')\} & \text{if } u \equiv t_1\,;\,t_2 \text{ and } u' \equiv t_1'\,;\,t_2' \\ \max\{d(t_1,t_1'), d(t_2,t_2')\} & \text{if } u \equiv t_1 + t_2 \text{ and } u' \equiv t_1' + t_2' \\ \max\{d(t_1,t_1'), d(t_2,t_2')\} & \text{if } u \equiv t_1 \parallel t_2 \text{ and } u' \equiv t_1' \parallel t_2' \\ d(t,t') & \text{if } u \equiv c\,!\,t \text{ and } u' \equiv c\,!\,t' \\ 1 & \text{otherwise} \end{cases}$$

We shall also need the mapping $\mathcal{S}$ defined in

Definition 42. The mapping

$$\mathcal{S} : (Res_4' \times SemState_4) \rightarrow^1 \mathcal{P}_{co}(Res_4' \times SemState_4^{ext})$$

is defined by

$$\mathcal{S}(u,\rho) = \{\,[u',\ \xi] \mid [u,\ \rho] \rightarrow [u',\ \xi]\,\}.$$

Let $\Phi_{\mathcal{O}^\#}$ be the higher order mapping associated in the natural way with the definition of $\mathcal{O}^\#$. Well-definedness of $\Phi_{\mathcal{O}^\#}$ follows by noting that

- $\mathcal{S}$ is well-defined, i.e., for all u and ρ, $\mathcal{S}(u, \rho)$ is compact and $\mathcal{S}$ is nonexpansive,
- for all t and ρ, the set $\{ (\xi, \phi(u)) \mid [t, \rho] \rightarrow [u, \xi] \}$ is compact, since $\mathcal{S}$ delivers compact sets and ϕ is nonexpansive,
- for all t, the mapping $\lambda\rho . \{ (\xi, \phi(u)) \mid [t, \rho] \rightarrow [u, \xi] \}$ is nonexpansive, since $\mathcal{S}$ and ϕ are nonexpansive.

Second, we extend the denotational semantics $\mathcal{D}$.

Definition 43. The denotational semantics $\mathcal{D}^\# : Res'_4 \rightarrow^1 P_7$ is defined by

$$\begin{aligned}
\mathcal{D}^\#(\mathrm{E}) &= \mathrm{E}\\
\mathcal{D}^\#(v := e) &= \lambda\rho . \{(\rho\{\alpha/v\}, \mathrm{E})\}, \text{ where } \alpha = \mathcal{V}(e)(\rho)\\
\mathcal{D}^\#(t_1 ; t_2) &= \mathcal{D}^\#(t_1) ; \mathcal{D}^\#(t_2)\\
\mathcal{D}^\#(t_1 + t_2) &= \mathcal{D}^\#(t_1) + \mathcal{D}^\#(t_2)\\
\mathcal{D}^\#(t_1 \parallel t_2) &= \mathcal{D}^\#(t_1) \parallel \mathcal{D}^\#(t_2)\\
\mathcal{D}^\#(x) &= \lambda\rho . \{(\rho, \rho(x))\}\\
\mathcal{D}^\#(c\,!\,t) &= \lambda\rho . \{((c, p), \mathrm{E})\}, \text{ where } p = \mathcal{D}^\#(t)\\
\mathcal{D}^\#(c\,?\,x) &= \lambda\rho . \{((\bar{c}, x), \mathrm{E})\}\\
\mathcal{D}^\#(p) &= p
\end{aligned}$$

Lemma 44. $\mathcal{O}^\# = \mathcal{D}^\#$.

Proof. This proof follows the first proof of Theorem 9. For example, for resumption x we have that

$$\begin{aligned}
&\Phi_{\mathcal{O}^\#}(\mathcal{D}^\#)(x)\\
&= \lambda\rho . \{(\rho, \mathcal{D}^\#(\rho(x)))\}\\
&= \lambda\rho . \{(\rho, \rho(x))\}\\
&= \mathcal{D}^\#(x).
\end{aligned}$$

□

To each extended syntactic state an extended semantic state is assigned by the mapping *sem*.

Definition 45. The mapping $sem : SynState_4^{ext} \rightarrow SemState_4^{ext}$ is defined by

$$\begin{aligned}
sem(\sigma) &= (\sigma_1, \lambda x . \mathcal{D}^\#(\sigma_2(x)))\\
sem((\bar{c}, x)) &= (\bar{c}, x)\\
sem((c, s)) &= (c, \mathcal{D}^\#(s))
\end{aligned}$$

The mapping *sem* is, again, extended in the natural way to a mapping from $\mathcal{P}_{nc}((SynState_4)_\delta^\infty)$ to $\mathcal{P}_{nc}((SemState_4)_\delta^\infty)$. The next lemma is the key technical result on which the relationship between $\mathcal{O}$ and $\mathcal{D}$ is based. The lemma expresses a canonical correspondence between transitions of $\mathcal{T}_4$ and $\mathcal{T}'_4$.

Lemma 46. *For all s, r, u, σ, σ', and ξ,*

> *if* $[s, \sigma] \to [r, \sigma']$ *then* $[s, sem(\sigma)] \to [u', sem(\sigma')]$
> *and* $\mathcal{O}^{\#}(u') = \mathcal{O}^{\#}(r)$ *for some* u'

and

> *if* $[s, sem(\sigma)] \to [u, \xi]$ *then* $[s, \sigma] \to [r', \sigma'']$
> *and* $\mathcal{O}^{\#}(r') = \mathcal{O}^{\#}(u)$ *and* $sem(\sigma'') = \xi$ *for some* r' *and* σ''.

Proof. This lemma can be proved by structural induction on s. We will only consider the first part for statement $s_1 ; s_2$. We distinguish two cases.

1. Assume $[s_1 ; s_2, \sigma] \to [s_2, \sigma']$. Then $[s_1, \sigma] \to [\text{E}, \sigma']$. By induction, $[s_1, sem(\sigma)] \to [u', sem(\sigma')]$ and $\mathcal{O}^{\#}(u') = \mathcal{O}^{\#}(\text{E})$. Consequently, $u' \equiv \text{E}$. So, $[s_1 ; s_2, sem(\sigma)] \to [s_2, sem(\sigma')]$.
2. Assume $[s_1 ; s_2, \sigma] \to [s_1' ; s_2, \sigma']$. Then $[s_1, \sigma] \to [s_1', \sigma']$. By induction, $[s_1, sem(\sigma)] \to [u', sem(\sigma')]$ and $\mathcal{O}^{\#}(u') = \mathcal{O}^{\#}(s_1')$. Consequently, $u' \not\equiv \text{E}$. So, $[s_1 ; s_2, sem(\sigma)] \to [u' ; s_2, sem(\sigma')]$ and

$$\begin{aligned}
&\mathcal{O}^{\#}(u' ; s_2) \\
&= \mathcal{D}^{\#}(u' ; s_2) \\
&= \mathcal{O}^{\#}(u') ; \mathcal{D}^{\#}(s_2) \\
&= \mathcal{O}^{\#}(s_1') ; \mathcal{D}^{\#}(s_2) \\
&= \mathcal{O}^{\#}(s_1' ; s_2).
\end{aligned}$$

□

The mapping *trace* used for $\mathcal{L}_{co_2}$ is obtained from Definition 29 by replacing σ's by ρ's:

Definition 47. The mapping $trace : P_7 \to^1 SemState_4 \to^1 \mathcal{P}_{nc}((SemState_4)^{\infty}_{\delta})$ is defined by

$$\begin{aligned}
trace(\text{E}) &= \lambda\rho . \{\varepsilon\} \\
trace(p) &= \lambda\rho . \begin{cases} \{\delta\} & \text{if } p(\rho) \text{ blocks} \\ \bigcup \{\, \rho' \cdot trace(p')(\rho') \mid (\rho', p') \in p(\rho) \,\} & \text{otherwise} \end{cases}
\end{aligned}$$

The operational semantics $\mathcal{O}^*$ and $\mathcal{O}^{\#}$ are related by means of the mappings *sem* and *trace*.

Lemma 48. *For all r and σ,* $sem(\mathcal{O}^*(r)(\sigma)) = trace(\mathcal{O}^{\#}(r))(sem(\sigma))$.

Proof. We can prove this lemma by means of the proof principle exploited in the second proof of Theorem 9 using Lemma 46.

□

Theorem 49. *For all s and σ,* $sem(\mathcal{O}(s)(\sigma)) = trace(\mathcal{D}(s))(sem(\sigma))$.

Summary

The results from the Sections 1 to 4 relating $\mathcal{O}$ and $\mathcal{D}$ for the four languages considered are summarised in the following table (putting $\mathcal{O}[\![s]\!] = \mathcal{O}(s)$ for each of the four languages, $\mathcal{D}[\![s]\!] = \mathcal{D}(s)(\lambda\sigma \cdot \{\varepsilon\})$ for $\mathcal{L}_{as}$, $\mathcal{D}[\![s]\!] = \mathcal{D}(s)(\lambda\rho \cdot \{\varepsilon\})$ for $\mathcal{L}_{as_2}$, and $\mathcal{D}[\![s]\!] = \mathcal{D}(s)$ for $\mathcal{L}_{co}$ and $\mathcal{L}_{co_2}$):

$$\begin{array}{ll}
\mathcal{L}_{as}: & \mathcal{O}[\![s]\!] = \mathcal{D}[\![s]\!] \\
\mathcal{L}_{as_2}: & sem \circ \mathcal{O}[\![s]\!] = \mathcal{D}[\![s]\!] \circ sem \\
\mathcal{L}_{co}: & \mathcal{O}[\![s]\!] = (trace \circ \mathcal{D})[\![s]\!] \\
\mathcal{L}_{co_2}: & sem \circ \mathcal{O}[\![s]\!] = (trace \circ \mathcal{D})[\![s]\!] \circ sem
\end{array}$$

References

[ABKR89] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Denotational Semantics of a Parallel Object-Oriented Language. *Information and Computation*, 83(2):152–205, November 1989.

[ACCL90] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, January 1990.

[AGR92] E. Astesiano, A. Giovini, and G. Reggio. Observational Structures and their Logics. *Theoretical Computer Science*, 96(1):249–283, April 1992.

[AR87] E. Astesiano and G. Reggio. SMoLCS-driven Concurrent Calculi. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 249 of *Lecture Notes in Computer Science*, pages 169–201, Pisa, March 1987. Springer-Verlag.

[AR89] P. America and J.J.M.M. Rutten. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *Journal of Computer and System Sciences*, 39(3):343–375, December 1989.

[Ban22] S. Banach. Sur les Opérations dans les Ensembles Abstraits et leurs Applications aux Equations Intégrales. *Fundamenta Mathematicae*, 3:133–181, 1922.

[Bar92] H.P. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, Background: Computational Structures, chapter 2, pages 117–309. Clarendon Press, Oxford, 1992.

[BB92] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[BM88] J.W. de Bakker and J.-J.Ch. Meyer. Metric Semantics for Concurrency. *BIT*, 28:504–529, 1988.

[Bou89] G. Boudol. Towards a Lambda-Calculus for Concurrent and Communicating Systems. In J. Diaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 351 of *Lecture Notes in Computer Science*, pages 149–162, Barcelona, March 1989. Springer-Verlag.

[BR92] J.W. de Bakker and J.J.M.M. Rutten, editors. *Ten Years of Concurrency Semantics, selected papers of the Amsterdam Concurrency Group.* World Scientific, Singapore, September 1992.

[Bre93] F. van Breugel. Three Metric Domains of Processes for Bisimulation. This volume.

[Bre94] F. van Breugel. *Topological Models in Comparative Semantics*. PhD thesis, Vrije Universiteit, Amsterdam, 1994. In preparation.

[BZ82] J.W. de Bakker and J.I. Zucker. Processes and the Denotational Semantics of Concurrency. *Information and Control*, 54(1/2):70–120, July/August 1982.

[Cur88] P.-L. Curien. The $\lambda\rho$-calculus: An Abstract Framework for Environment Machines. Report, LIENS, Paris, October 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice/Hall International, London, 1985.

[JP90] R. Jagadeesan and P. Panangaden. A Domain-theoretic Model for a Higher-order Process Calculus. In M.S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 181–194, Coventry, July 1990. Springer-Verlag.

[KR90] J.N. Kok and J.J.M.M. Rutten. Contractions in Comparing Concurrency Semantics. *Theoretical Computer Science*, 76(2/3):179–222, 1990.

[Kur56] K. Kuratowski. Sur une Méthode de Métrisation Complète des Certains Espaces d'Ensembles Compacts. *Fundamenta Mathematicae*, 43:114–138, 1956.

[LTLG92] J.-J. Lévy, B. Thomsen, L. Leth, and A. Giacalone. CONcurrency and Functions: Evaluation and Reduction. *Bulletin of the European Association for Theoretical Computer Science*, 48:88–106, October 1992.

[Mic51] E. Michael. Topologies on Spaces of Subsets. *Transactions of the American Mathematical Society*, 71:152–182, 1951.

[Mil92] R. Milner. Functions as Processes. *Mathematical Structures in Computer Science*, 2(2):119–141, June 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 1(100):1–40 and 41–77, September 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed Bisimulation. In W. Kuich, editor, *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695, Vienna, July 1992. Springer-Verlag.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Report DAIMI FN-19, Aarhus University, Aarhus, September 1981.

[RT92] J.J.M.M. Rutten and D. Turi. On the Foundations of Final Semantics: non-standard sets, metric spaces, partial orders. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 477–530, Beekbergen, June 1992. Springer-Verlag.

[Rut92] J.J.M.M. Rutten. Processes as Terms: Non-Well-Founded Models for Bisimulation. *Mathematical Structures in Computer Science*, 2(3):257–275, September 1992.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburg, Edinburg, 1992.

[San93] D. Sangiorgi. An Investigation into Functions as Processes. This volume.

[Tho89] B. Thomsen. A Calculus of Higher Order Communicating Systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming*

Languages, pages 143–154, Austin, January 1989.

[Tho90] B. Thomsen. *Calculi for Higher Order Communicating Systems.* PhD thesis, Imperial College, London, September 1990.

An Investigation into Functions as Processes

Davide Sangiorgi[1]

Abstract. In [Mil90] Milner examines the encoding of the λ-calculus into the π-calculus [MPW92]. The former is the universally accepted basis for computations with *functions*, the latter aims at being its counterpart for computations with *processes.* The primary goal of this paper is to continue the study of Milner's encodings. We focus mainly on the lazy λ-calculus [Abr87]. We show that its encoding gives rise to a λ-model, in which a weak form of extensionality holds. However the model is not fully abstract: To obtain full abstraction, we examine both the *restrictive* approach, in which the semantic domain of processes is cut down, and the *expansive* approach, in which λ-calculus is enriched with *constants* to obtain a *direct* characterisation of the equivalence on λ-terms induced, via the encoding, by the behavioural equivalence adopted on the processes. Our results are derived exploiting an intermediate representation of Milner's encodings into the *Higher-Order π-calculus,* an ω-order extension of π-calculus where also agents may be transmitted. For this, essential use is made of the fully abstract compilation from the Higher-Order π-calculus to the π-calculus studied in [San92a].

1 Introduction

In [Mil90] Milner examines the encoding of the λ-calculus into the π-calculus [MPW92]; the former is the universally accepted basis for computations with *functions*, the latter aims at being its counterpart for computations with *processes.* More precisely, Milner shows how the evaluation strategies of *lazy λ-calculus* and *call-by-value λ-calculus* [Abr87, Plo75] can be faithfully mimicked. The characterisation of the equivalence induced on λ-terms by the encodings is left as an open problem; it also remains to be studied which kind of λ-model — if any — can be constructed from the process terms.

The primary goal of this paper is to continue the study of Milner's encodings. A deep comparison between a process calculus and λ-calculus is interesting for several reasons; indeed, virtually all proposals for process calculi with the capability of treating — directly or indirectly — processes as first class objects have incorporated attempts at embedding the λ-calculus [Bou89, Tho90]. From the process calculus point of view, it is a significant test of expressiveness, and helps in getting deeper insight into its theory. From the λ-calculus point of view, it provides the means to study λ-terms in contexts other than purely sequential ones, and with the instruments developed in the process calculus. For example, an important behavioural equivalence upon process terms gives rise to an interesting equivalence upon λ-terms. Moreover, the relevance of those λ-calculus evaluation strategies which can be efficiently encoded is strengthened. More practical motivations for describing

1 Address: Department of Computer Science, University of Edinburgh, JCMB, Mayfield road, Edinburgh EH9 3JZ, U.K. Email: sad@dcs.ed.ac.uk

functions as processes are to provide a semantic foundation for languages which combine concurrent and functional programming and to develop parallel implementations of functional languages.

Our other major goal is more centered on process calculi. The paradigm on which π-calculus is constructed is first order: Reductions cause instantiations of *names*. This contrasts with what happens in λ-calculus, where reductions cause instantiations of *terms*. Higher-order communications are avoided in π-calculus because of their complexity and because they can be represented at first order. The latter is showed in [San92a] by comparing π-calculus with the *Higher-Order π-calculus* (HOπ), an ω-order extension of π-calculus where not only names but also processes and parametrised processes of arbitrary high order can be communicated: A compilation from HOπ to π-calculus is defined and proved fully abstract with respect to the semantics of the calculi. Thus, the second goal of this paper is to illustrate the use of HOπ and of the representability result of HOπ into π-calculus.

Using the abstraction power of HOπ, for both the lazy and the call-by-value λ-calculus we give encodings which are easier to understand and to deal with than those available in the π-calculus. By applying the compilation from HOπ to π-calculus, we can turn them into π-calculus encodings which can then be compared with Milner's; this is a significant test for the canonicity of the encodings involved. In the lazy λ-calculus the correspondence is exact. That is, if $\mathcal{P}$ and $\mathcal{H}$ are, respectively, the π-calculus and HOπ encodings, and $\mathcal{C}$ is the compilation from HOπ to π-calculus, then the following diagram commutes:

$$\begin{tikzcd}
\lambda \arrow[r, "\mathcal{H}"] \arrow[dr, "\mathcal{P}"'] & \mathrm{HO}\pi \arrow[d, "\mathcal{C}"] \\
 & \pi
\end{tikzcd}$$

In consequence, since $\mathcal{C}$ is fully abstract, any result proved for one of the encodings can be transferred to the other. By working with HOπ, we show in this paper that the encodings do give rise to a λ-model, where conditional extensionality holds. It is not fully abstract, though. To obtain full abstraction we follow two directions: In the *restrictive* approach, based on the use of *barbed bisimulation* [MS92], the semantic domain of processes is cut down; in the *expansive* approach λ-calculus is enriched with constants to obtain a *direct* characterisation of the equivalence on λ-terms induced, via the encoding, by the behavioural equivalence adopted on the processes.

For call-by-value the situation is less sharp. In [Mil90] Milner presents two candidates for the encoding, and it is not obvious which one should be preferred: The first allows easier reasoning, but the second is more efficient. Moreover, when applied to the HOπ encoding, compilation $\mathcal{C}$ dces not return either of them. Apparently, to obtain them some code transformation has to be carried out. The study of these transformations leads to interesting outcomes. Firstly, it suggests a correction of the order in which some actions appear in Milner's encodings. This rearrangement does not affect the operational correspondence between λ- and π-terms. However, it affects the behavioural equivalence on the encoding π-terms, in a way which makes the encoding more faithful to the encoded call-by-value discipline. Secondly, the study of the transformations reveals that β-reduction is not valid in Milner's second encoding, which severely reduces its importance.[1] The counter-example is fairly sophisticated and we doubt we could have obtained it without going through HOπ.

This paper is an extract from (mainly chapter 6 of) the author's Ph.D. thesis [San92a]; we refer to it for details and proofs of the results reported.

Acknowledgements. I wish to thank Robin Milner for discussions and suggestions, and Benjamin Pierce, Peter Sewell and the anonymous referees for comments on an earlier draft. The paper was written during my stay at INRIA-Rocquencourt; I am grateful to Jean-Jacques Levy for having invited me.

2 The π-calculus and the Higher-Order π-calculus

In this section we review the syntax and semantics of π-calculus and HOπ, before moving on to the study of the representation of functions as processes, the core of this paper.

2.1 Syntax

We shall look explicitly only at the syntax and the semantics of the *Higher-Order π-calculus (HOπ)*, since the π-calculus is a subcalculus of it. Actually, we only present a *fragment* of these languages, but one which is sufficient for the encoding of the λ-calculus. A more detailed description of the operators involved and their meaning can be found in [San92a].

A HOπ *agent* (or *term*) can be a *process* or an *abstraction*, i.e. a parametrised process. In the following, P and Q stand for processes, F and G for abstractions, A for agents. We use X, Y to range over the set of variables; as in λ-calculus, a variable is supposed to be instantiated with a term. The letters $a, b, \ldots, x, y, \ldots$ stand for names. Moreover, K stands for an agent or a name and U for a variable or a name. We use a tilde to denote a finite (possibly empty) tuple. In the fragment of HOπ we consider, a process is built from names using the operators of parallel composition, restriction, replication, variable application, input and output prefixing, and nil.

$$P :: P_1 \mid P_2 \;\mid\; \nu x\, P \;\mid\; !P \;\mid\; X\langle \tilde{K} \rangle \;\mid\; x(\tilde{U}).P \;\mid\; \overline{x}\langle \tilde{K} \rangle.P \;\mid\; 0$$

An agent is an abstraction over a process or over a partial application:

$$A :: (\tilde{U})P \;\mid\; (\tilde{U})X\langle \tilde{K} \rangle$$

Variable application $X\langle \tilde{K} \rangle$ is needed to provide an abstraction received as an input with the appropriate arguments. The other process operators resemble those of the (polyadic) π-calculus and CCS (see [Mil91, Mil89]); we only remind the reader that

1 The version of [Mil90] which appeared in the *Jour. of Math. Structures* was written when the results in this chapter were already known and thus presents only the first encoding.

the replication $!P$ represents an unbounded number of copies of P in parallel, and allows us to describe processes with infinite behaviour. Application has the highest precedence; abstraction the lowest. Sometimes we abbreviate $\alpha.0$ as α. In the above expressions, when a tuple is empty the surrounding brackets $()$ or $\langle\rangle$ are omitted. Note that also a variable X is an agent, corresponding to the case in which $\tilde{U}$ and $\tilde{K}$ are empty.

An abstraction is an agent which takes some arguments before becoming a process. The typical form of an abstraction is $(\tilde{U})P$; it is like a procedure, in which $(\tilde{U})$ represents the parameters. For instance, $F \overset{def}{=} (Y)(P \mid Y)$ abstracts on a process variable; F takes a process and runs it in parallel with P. We can also abstract on abstraction variables, as in $G \overset{def}{=} (X)(P \mid X\langle Q\rangle)$; then F applied to G yields $P \mid P \mid Q$. The machinery can be iterated, progressively increasing the order of the resulting abstraction. In this sense HOπ is an ω-order calculus: There is no bound on the order of the agents which can be written and communicated. In contrast, in π-calculus abstractions and communications can only be first order, i.e. abstractions and communications of names. Thus, in π-calculus syntax, variable application does not appear and tuples $\tilde{K}$ and $\tilde{U}$ are replaced by simple tuples of names.

W.r.t. the language in [San92a], we have omitted the operators of matching and summation, and recursive definitions of agents have been replaced by replication. The latter is a limitation because while replication can be encoded using recursive definitions, the other way round only holds if the number of recursive definitions is finite (see [Mil91, section 3.1]). Further, in the presented sublanguage an agent may only have a finite number of free names. By contrast, in the full syntax in [San92a], since infinite recursive definitions and infinite summations are allowed, it is possible to write agents which have an infinite number of free names, like a counter which at each step emits a signal on a different channel.

The restriction $\nu b\, P$, the input prefix $a(\tilde{U}).P$ and the abstraction $(\tilde{U})P$ are formal binders for names and variables in $\tilde{U}$ and b; they give rise in the expected way to the definitions of α-conversion, *free names* and *free variables* of a term. An *open* agent is an agent possibly containing free variables.

It is crucial in practice to avoid disagreements in what is carried by a given name or expected in applications; for instance we reject expressions like $\overline{a}\langle b_1, b_2\rangle.P|a(x).Q$ or $X\langle b\rangle \mid X\langle b_1, b_2\rangle$, due to the mismatching in the use of the names. In HOπ this need is very compelling: It is not only a question of arities, but we also have to avoid any confusion between instantiation of names and of agents as well as instantiation of agents of different order. To this end, Milner proposed the use of *sorts* [Mil91]. Sorts have, very roughly, the flavour of types in λ-calculus; however in the process calculi not only terms are assigned a sort (or a type), but also names, the latter depending upon the (sorts of the) objects which that name can carry. The sorting system is also useful to understand the passage from π-calculus to HOπ: The ω-order sorts of HOπ can be derived by removing certain constraints on the first-order sort language of π-calculus. We will not present the sorting system because it is not essential to understand the contents of this paper. The reader should take for granted that all agents described obey a sorting.

It is worth pointing out that we do not lose expressiveness in our language by having application only with variables. In fact, every "well-sorted expression" $A\langle\widetilde{K}\rangle$ can be put into this form by "executing" the applications it contains; for instance from $((X)Y\langle X\rangle)\langle P\rangle$, we get $Y\langle P\rangle$. This makes the definition of substitution more elaborate, but facilitates the proofs in the calculus. However, we shall sometimes use $A\langle F\rangle$ as metanotation; for instance, if $G \overset{def}{=} (X)P$, then $G\langle F\rangle$ is $P\{F/X\}$.

2.2 Operational semantics

Following Milner [Mil90, Mil91], we give the operational semantics of the language as a *reduction system*. We begin by defining *structural congruence*, written $\equiv$, as the smallest congruence over the class of processes which satisfies the rules below.

1. $P \equiv Q$ if P is α-convertible to Q;
2. abelian monoid laws for $|$: $P\,|\,Q \equiv Q\,|\,P$, $P\,|\,(Q\,|\,R) \equiv (P\,|\,Q)\,|\,R$, $P\,|\,0 \equiv P$;
3. $\nu x\, 0 \equiv 0$; $\nu x\, \nu y\, P \equiv \nu y\, \nu x\, P$; $(\nu x\, P)\,|\,Q \equiv \nu x\,(P\,|\,Q)$; if $x \notin fn(Q)$;
4. $!P \equiv P\,|\,!P$.

The structural congruence axioms are used to act upon the structure of terms so that processes willing to interact can be brought into contiguous positions. Then the *reduction relation* can be described with a few simple rules:

$$\text{COM: } x(\widetilde{U}).P\,|\,\overline{x}\langle\widetilde{K}\rangle.Q \longrightarrow P\{\widetilde{K}/\widetilde{U}\}\,|\,Q \qquad \text{PAR: } \frac{P \longrightarrow P'}{P\,|\,Q \longrightarrow P'\,|\,Q}$$

$$\text{RES: } \frac{P \longrightarrow P'}{\nu x\, P \longrightarrow \nu x\, P'} \qquad \text{STRUCT: } \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

2.3 Barbed bisimulation

Barbed bisimulation was first proposed in [MS92]. One of the motivations was to be able to *uniformly* define bisimulation-based equivalences in different calculi. This is a very important property for the kind of work conducted in this paper, since it allows us to have the same definition of equivalence in the calculi considered (including the λ-calculus, as we shall see in Section 6).

Barbed bisimulation focuses on the reduction relation. It goes a little further though, since the reduction relation by itself is not enough to yield the desired discriminanting power. The choice in [San92a] was to introduce, for each name a, an *observation predicate* $\downarrow_a$ which detects the possibility of performing a communication with the environment along a. We can check whether $P \downarrow_a$ holds from the syntactic form of P: There must be a prefix $a(\widetilde{U})$ or $\overline{a}\langle\widetilde{K}\rangle$ which is not underneath another prefix and not in the scope of a restriction on a. For example, if P is $(\nu c)(\overline{c}.b\,|\,a.d)$, then $P\downarrow_a$, but not $P\downarrow_c$, $P\downarrow_b$ or $P\downarrow_d$.

Definition 1. *Strong barbed bisimulation*, written $\dot{\sim}$, is the largest symmetrical relation on the class of processes of the language such that $P \dot{\sim} Q$ implies:

1. whenever $P \longrightarrow P'$ then there is a Q' such that $Q \longrightarrow Q'$ and $P' \dot{\sim} Q'$;
2. for each name a, if $P\downarrow_a$ then $Q\downarrow_a$. □

By itself, barbed bisimulation is too coarse (it is not even preserved by parallel composition). By parametrisation over contexts, we get a finer relation.

Definition 2. Two processes P and Q are *strong barbed-congruent*, written $P \sim Q$, if for each context $C[\cdot]$, it holds that $C[P] \overset{\cdot}{\sim} C[Q]$. □

It is important to stress that the proofs in [San92a] dealing with barbed congruence use the *full* languages of π-calculus and HOπ — as opposed to the fragments presented here — in the construction of the contexts with which processes are tested. Therefore, these contexts use matching, summation, infinite recursive definitions and infinite free names. A challenging problem for future research is to see whether such results can also be proved in the finitary calculus, i.e. without infinite definitions and names.

The weak version of the equivalence, in which one abstracts away from the length of the reductions in two matching actions, is obtained in the standard way: Let $\Longrightarrow$ be the reflexive and transitive closure of $\longrightarrow$ and $\Downarrow_a$ be $\Longrightarrow\downarrow_a$ (the composition of the two relations). Then *weak barbed bisimulation*, written $\overset{\cdot}{\approx}$, is defined by replacing, in definition 1, the transition $Q \longrightarrow Q'$ with $Q \Longrightarrow Q'$ and the predicate $Q \downarrow_a$ with $Q \Downarrow_a$; and *weak barbed congruence*, written $\approx$, by replacing in definition 2 $\overset{\cdot}{\sim}$ with $\overset{\cdot}{\approx}$. The definition of barbed congruence on abstractions and open agents is given in the expected way, by requiring instantiation of variables and of abstracted names with all admissible agents or names.

The discriminatory power of barbed bisimulation is tested in [San92a], by proving that in the strong and in the weak case barbed congruence coincides in CCS and π-calculus with the ordinary bisimilarity congruences.

2.4 The compilation from HOπ to π-calculus

We present the compilation from HOπ to π-calculus on agents which can transmit only *one* value — a name or an abstraction — and which only use *unary* abstractions. This is purely to make the definition of the compilation (and of the operational correspondence for it) more readable — the generalisation to the calculus with arbitrary arities does not introduce semantic complications. We use $(\overline{a}\langle m\rangle.P)\{m := F\}$ to stand for $\nu m\,(\overline{a}\langle m\rangle.P \mid\, !m(U).F\langle U\rangle)$, where U is a name or a variable, depending upon the sort of m.[1] One should think of m as a pointer to F and $\{m := F\}$ as a "local environment" for P. We call m a *name-trigger*.

The compilation $\mathcal{C}$ from HOπ to π-calculus is defined in Table 1. The idea is that the communication of the HOπ agent F should be represented at first order by the communication of a name-trigger m which gives *access* to (the encoding of) F; the name m is used by the recipient to activate the needed copies of F with the appropriate arguments. The other delicate rules are those for application and for variable. Consider the application $X\langle F\rangle$: When X is instantiated to an agent G, it becomes $G\langle F\rangle$. Translating $X\langle F\rangle$, we expect to receive just a name-trigger to G, and we are expected to use this name to activate G, providing it with the

1 When P is 0, the occurrence of $\mid$ is unnecessary and hence $(\overline{a}\langle m\rangle.0)\{m := F\}$ should be read as $\nu m\,(\overline{a}\langle m\rangle.\,!m(U).F\langle U\rangle)$

$$\mathcal{C}[\![X]\!] \overset{def}{=} \begin{cases} \mathcal{C}[\![(Y)X\langle Y\rangle]\!] & \text{if } X \text{ is a higher-order abstraction} \\ \mathcal{C}[\![(a)X\langle a\rangle]\!] & \text{otherwise} \end{cases}$$

$$\mathcal{C}[\![\alpha.P]\!] \overset{def}{=} \begin{cases} (\overline{a}\langle m\rangle.\mathcal{C}[\![P]\!])\{m := \mathcal{C}[\![F]\!]\} & \text{if } \alpha = \overline{a}\langle F\rangle \\ a(x).\mathcal{C}[\![P]\!] & \text{if } \alpha = a(X) \\ \alpha.\mathcal{C}[\![P]\!] & \text{otherwise} \end{cases}$$

$$\mathcal{C}[\![X\langle F\rangle]\!] \overset{def}{=} (\overline{x}\langle m\rangle.0)\{m := \mathcal{C}[\![F]\!]\} \qquad \mathcal{C}[\![X\langle b\rangle]\!] \overset{def}{=} \overline{x}\langle b\rangle.0$$

$$\mathcal{C}[\![P \mid Q]\!] \overset{def}{=} \mathcal{C}[\![P]\!] \mid \mathcal{C}[\![Q]\!] \quad \mathcal{C}[\![\nu\, a\, P]\!] \overset{def}{=} \nu\, a\, \mathcal{C}[\![P]\!] \quad \mathcal{C}[\![!\, P]\!] \overset{def}{=} !\mathcal{C}[\![P]\!]$$

$$\mathcal{C}[\![(X)P]\!] \overset{def}{=} (x)\mathcal{C}[\![P]\!] \qquad \mathcal{C}[\![(a)P]\!] \overset{def}{=} (a)\mathcal{C}[\![P]\!]$$

Table 1. The compilation $\mathcal{C}$

argument F. Since we cannot pass agents at first order, as in the rule for output, this is resolved by sending a name-trigger for F. In the rule for variable an η-conversion is employed. This is to make explicit all possible applications and hence to introduce all necessary name-triggers; the use of full triggered forms is needed to get the soundness of Theorem 3 below [San92a]. In this rule, the termination of $\mathcal{C}$ is guaranteed by the well-sortedness hypothesis which ensures that, in $(Y)X\langle Y\rangle$, the sort of Y is "smaller" than the sort of X. In the table, a variable X is mapped to its lower case letter x; we assume that both this name x and the name-trigger m are fresh, i.e. do not occur in the source agent.

As a simple example, suppose $F \overset{def}{=} (b)0$, Y is a first-order variable of the same sort as F, and X is a second-order variable which may take F or Y as arguments. Then

$$\mathcal{C}[\![(X)(X\langle F\rangle \mid \overline{b}\langle Y\rangle)]\!] = (x)(\overline{x}\langle m\rangle\,\{m := F\} \mid \overline{b}\langle m\rangle\,\{m := (z)\overline{y}\langle z\rangle\})$$

Theorem 3 (full abstraction for $\mathcal{C}$). *For each HOπ agent A_1 and A_2, it holds that $A_1 \approx A_2$ iff $\mathcal{C}[\![A_1]\!] \approx \mathcal{C}[\![A_2]\!]$* ☐

3 The lazy λ-calculus

We take for granted the basic concepts of the λ-calculus (see [Bar84]). We use Λ for the class of closed pure λ-terms and M, N, L to range over Λ. We denote by Ω the divergent term $(\lambda x.xx)(\lambda x.xx)$. In Abramsky's *lazy λ-calculus* [Abr87], a redex is always at the extreme left of a term: The reduction rules are those for reflexivity and transitivity plus

$$(\beta)\ (\lambda x.M)N \Longrightarrow M\{N/x\} \qquad\qquad (App)\ \frac{M \Longrightarrow M'}{MN \Longrightarrow M'N}$$

When embedding the λ-calculus into a process calculus, functional application becomes a particular parallel combination of two agents, the function and its argument, and β reduction a particular case of interaction. The encoding below of lazy

λ-calculus into HOπ makes this idea very transparent. The translation of a λ-term is an abstraction over a name; this name will be the only access to that agent and will be used to interact with the appropriate λ-term. Thus $\mathcal{H}[\lambda x.M]\langle p\rangle$ receives at p its λ-argument and the name q which will give access to M. In the translation of application, the restriction on q prevents interference from other processes. For simplicity, a variable x of the λ-calculus is mapped to its upper-case variable X in HOπ.

$$\begin{aligned}\mathcal{H}[\lambda x.M] &\overset{def}{=} (p)p(X,q).\mathcal{H}[M]\langle q\rangle\\ \mathcal{H}[x] &\overset{def}{=} X\\ \mathcal{H}[MN] &\overset{def}{=} (p)\nu q\,(\mathcal{H}[M]\langle q\rangle \mid \overline{q}\langle \mathcal{H}[N], p\rangle.0)\end{aligned}$$

The higher-order features of HOπ allow us a simpler encoding than Milner's into π-calculus [Mil90]. Indeed, there is a one-to-one correspondence between reductions in λ-terms and in their HOπ counterparts. Therefore, following Boudol's terminology [Bou89], we can claim that *lazy λ-calculus is a subcalculus of HOπ.*

***Proposition 4** (operational correspondence for $\mathcal{H}$). Let M and M' be closed λ-terms.*

1. *If $M \longrightarrow M'$ then $\mathcal{H}[M]\langle p\rangle \longrightarrow \mathcal{H}[M']\langle p\rangle$,*
2. *the converse, i.e. if $\mathcal{H}[M]\langle p\rangle \longrightarrow Q$ then there is an M' such that $M \longrightarrow M'$ and $Q \equiv \mathcal{H}[M']\langle p\rangle$.*

PROOF: Induction on the structure of M. □

If we apply compilation $\mathcal{C}$ to the encoding $\mathcal{H}$, the output is precisely Milner's encoding $\mathcal{P}$ in [Mil90]; the symbol '$\circ$' denotes function composition:

***Proposition 5.** $\mathcal{C} \circ \mathcal{H} = \mathcal{P}$.* □

Consequently, by appealing to the full abstraction for $\mathcal{C}$, we can freely switch between the two encodings. We shall exploit this in Sections 5 and 6 to study them from the point of view of the model theory of the λ-calculus.

4 The Call-by-Value λ-calculus

In call-by-value λ-calculus, reductions may only occur when the argument is a value, i.e. an abstraction. The reduction relation used by Milner in [Mil90] is described by the usual rules for reflexivity and transitivity plus the rules β_v, App_L, App_R:

$$(\beta_v)\ (\lambda x.M)\lambda y.N \Longrightarrow M\{\lambda y.N/x\} \qquad (App_L)\ \frac{M \Longrightarrow M'}{MN \Longrightarrow M'N}$$

$$(App_R)\ \frac{N \Longrightarrow N'}{MN \Longrightarrow MN'}$$

We shall try to repeat for call-by-value what we did in the previous section for lazy λ-calculus: We propose an encoding into HOπ and then we compare it with Milner's into π-calculus through compilation $\mathcal{C}$. The call-by-value encodings are slightly more involved than those for the lazy λ-calculus. They also lose a neat canonicity, which is implicitly confirmed by the fact that in his original work [Mil90], Milner presents *two* candidates for the encoding. Basically, the problems in the translation of call-by-value come from the following *dichotomy* in the behaviour of a λ-abstraction. Take the term MN: Both M and N could reduce to an abstraction; but if M does, then the abstraction is destined to perform the *input* of a value, whereas if N does, the abstraction represents an *output* value. This causes disagreement on whether the process which encodes a λ-abstraction should first perform an input or an output.

In the encoding below into HOπ, in contrast with the one for the lazy λ-calculus, the translation of an application allows the two arguments M and N to run in parallel. The HOπ process $\mathcal{H}[\![M]\!]\langle p\rangle$ uses p to communicate (with an output action) that M has reduced to a value; then the dichotomy in the behaviour of a λ-abstraction is solved by the arbiter $App\langle p,q,r\rangle$ which imposes the correct interaction between M and N.

$$\begin{aligned}
\mathcal{H}[\![\lambda x.M]\!] &\stackrel{def}{=} (p)\overline{p}\langle (w)w(X,q).\mathcal{H}[\![M]\!]\langle q\rangle\rangle.0\\
\mathcal{H}[\![x]\!] &\stackrel{def}{=} (p)\overline{p}\langle X\rangle.0\\
\mathcal{H}[\![MN]\!] &\stackrel{def}{=} (p)(\nu\, q,r)(\mathcal{H}[\![M]\!]\langle q\rangle \mid \mathcal{H}[\![N]\!]\langle r\rangle \mid App\langle p,q,r\rangle)\\
&\qquad\text{where } App \stackrel{def}{=} (p,q,r)q(X).r(Y).\nu\, v\,(X\langle v\rangle \mid \overline{v}\langle Y,p\rangle.0)
\end{aligned}$$

It is enlightening to relate this encoding to the one for the lazy λ-calculus. In the above rules for abstraction and variable, the "core" is the object part of the output at p, and it has the same format as the corresponding rule for the lazy λ-calculus. Now the rule for "call-by-value application" should become clear: The arbiter App receives along p and r the "cores" of $\mathcal{H}[\![M]\!]$ and $\mathcal{H}[\![N]\!]$ and then imposes on them the "lazy application". Therefore a reduction on the λ-terms is matched by *three* reductions on the process side. Let us apply compilation $\mathcal{C}$ to $\mathcal{H}$ and see what we get back.

$$\begin{aligned}
\mathcal{C}[\![\mathcal{H}[\![\lambda x.M]\!]]\!] &\stackrel{def}{=} (p)\,\overline{p}\langle m\rangle\,\{m := (w)w(x,q).\mathcal{C}[\![\mathcal{H}[\![M]\!]]\!]\langle q\rangle\}\\
\mathcal{C}[\![\mathcal{H}[\![x]\!]]\!] &\stackrel{def}{=} (p)\,\overline{p}\langle m\rangle\,\{m := (w)\overline{x}\langle w\rangle\}\\
\mathcal{C}[\![\mathcal{H}[\![MN]\!]]\!] &\stackrel{def}{=} (p)\,(\nu\, q,r)(\mathcal{C}[\![\mathcal{H}[\![M]\!]]\!]\langle q\rangle \mid \mathcal{C}[\![\mathcal{H}[\![N]\!]]\!]\langle r\rangle \mid App_\pi\langle p,q,r\rangle)
\end{aligned}$$

where with simple algebraic manipulations, $App_\pi\langle p,q,r\rangle$ can be written as

$$App_\pi\langle p,q,r\rangle \stackrel{def}{=} q(x).r(y).(\nu\, v)\overline{x}\langle v\rangle.(\overline{v}\langle m,p\rangle\,\{m := (w)\overline{y}\langle w\rangle\})$$

In his original work [Mil90], Milner presents two candidates for the encoding of call-by-value λ-calculus into π-calculus, which we shall call $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. They only differ in the rule for the translation of a variable: In $\mathcal{P}_2$ this rule is simpler, but $\mathcal{P}_1$ allows easier reasoning and proofs. There are two differences between the encoding $\mathcal{C} \circ \mathcal{H}$ and Milner's $\mathcal{P}_i$'s. The first difference is that the order of the actions $r(y)$ and $\nu v\, \overline{x}\langle v\rangle$ in $App_\pi\langle p,q,r\rangle$ is reversed. Let us call $\mathcal{P}'_i$, $i = 1,2$ the encoding obtained from $\mathcal{P}_i$ by commuting such actions $r(y)$ and $\nu v\, \overline{x}\langle v\rangle$. This action rearrangement causes a semantic difference between $\mathcal{P}'_i[\![MN]\!]$ and $\mathcal{P}_i[\![MN]\!]$ only when M and N are open. The encodings $\mathcal{P}'_i$'s appear closer than the $\mathcal{P}_i$'s to the call-by-value intuition. We justify this with an example. Consider the λ-term $\lambda x.x\Omega$: Since the call-by-value application $x\Omega$ has a divergent argument Ω, the term $x\Omega$ is supposed not to produce any visible behaviour. Consequently, we expect that a faithful encoding of call-by-value equates $\lambda x.x\Omega$ and $\lambda x.\Omega$. But this is true only for the encodings $\mathcal{P}'_i$'s, whereas it fails for the $\mathcal{P}_i$'s. In consequence, we consider the former an improvement of the latter.

The second difference between the encoding $\mathcal{C}\circ\mathcal{H}$ and Milner's $\mathcal{P}_i$'s (everything we shall say for the $\mathcal{P}_i$'s holds for their "rectified" $\mathcal{P}'_i$'s) is that the component $\overline{v}\langle m,p\rangle\, \{m := (w)\overline{y}\langle w\rangle\}$ of $App_\pi\langle p,q,r\rangle$ is "optimised" as $\overline{v}\langle y,p\rangle.0$ in $\mathcal{P}_1$ and $\mathcal{P}_2$ and, further, in $\mathcal{P}_2$ a similar optimisation occurs in the rule for variable, which is translated as $(p)\overline{p}\langle x\rangle.0$. We call the former *optimisation 1* and the latter *optimisation 2*. It can be shown that both of them are instances of the same potential optimisation of the compilation $\mathcal{C}$ in the rule for output of a variable, namely

$$\mathcal{C}[\![\overline{a}\langle X\rangle.Q]\!] \overset{def}{=} \overline{a}\langle x\rangle.\mathcal{C}[\![Q]\!] \qquad (*)$$

The intuitive justification for (*) would be the following. Suppose that previous interactions have instantiated the variable X of the HOπ term $\overline{a}\langle X\rangle.Q$ with F. In the encoding π-calculus terms the simulation of these interactions causes the instantiation of the name x with a trigger, say m_F, to $\mathcal{C}[\![F]\!]$. Now, with the rule (*) this same trigger m_F is then transmitted in the output along a. Instead, with the original rule of Table 1 a new name-trigger m to the term $(y)\overline{m_F}\langle y\rangle$ is transmitted: This just seems to introduce a further level of indirection to the activation of $\mathcal{C}[\![F]\!]$. Indeed, rule (*) is often sound, and we believe that optimisation 1 is. This would give us a factorisation for $\mathcal{P}_1$ (or better, for its rectified $\mathcal{P}'_1$) through the HOπ encoding and the compilation $\mathcal{C}$, up-to some code-optimisation. We defer the analysis of the soundness of optimisation 1, as well as of possible other optimisations of $\mathcal{C}$, for future research.

But rule (*) is not sound *in general*. The problem has to do with sharing. With rule (*) two outputs of the same variable become at first-order outputs of pointers to the same "environment entry"; this identity can be recognised and affects the successive behaviour. For instance, the encodings of the strongly congruent HOπ processes (here F is any abstraction)

$$P \overset{def}{=} \nu a\,(\overline{a}\langle F\rangle.0 \mid a(X).\overline{b}\langle X\rangle.\overline{b}\langle X\rangle.0)$$
$$Q \overset{def}{=} \nu a\,(\overline{a}\langle F\rangle.0 \mid a(X).\overline{b}\langle F\rangle.\overline{b}\langle F\rangle.0)$$

would not be equivalent. For the same reason, β-conversion is not valid for Milner's second encoding $\mathcal{P}_2$, as can be shown using the terms $M = (\lambda x.(\lambda y.x))(\lambda z.z)$ and $N = \lambda y.(\lambda z.z)$: In one β-step M reduces to N; however $\mathcal{P}_2[\![M]\!] \not\approx \mathcal{P}_2[\![N]\!]$. The difference between them appears after a sequence of interactions with the external environment of length at least 7 (!).

Nevertheless, $\mathcal{P}_2$ yields a precise operational correspondence between λ-terms and their process encodings and, intuitively, one "expects" $\mathcal{P}_2$ to be correct. Recently, in a collaboration with Benjamin Pierce [PS93] we have studied a stronger sorting discipline than Milner's, in which one distinguishes the ability of using channels of a given sort for performing inputs, outputs or both of them. In this system, we have indeed been able to prove the validity of β-reduction for $\mathcal{P}_2$. It would be interesting to see whether the adoption of such a refined sort systems would also validate rule $(*)$.

5 A λ-model from the process terms

Having shown the exact operational correspondence between lazy λ-terms and their process images (Proposition 4), it is legitimate to ask ourselves whether the encoding gives rise to a λ-model, and if so, what kind of λ-model it represents. We chose the lazy λ-calculus because of the simplicity of its encodings. We shall work within HOπ; therefore from now on up to the end of the paper, the word encoding and the symbol $\mathcal{H}$ refer to the HOπ encoding of lazy λ-calculus given in Section 3. All results can be transported onto Milner's encoding into π-calculus via Theorem 3 and Proposition 5.

There are simple syntax-free definitions of λ-model (i.e. they do not mention λ-terms). However, since we already have the mapping from λ to process terms, it is more convenient to use a definition where we can use such a mapping explicitly. A *valuation* is a function from the set of λ-variables to the domain D of the λ-model; $[d/x]\rho$ is the valuation which maps x to d and which behaves like ρ on the remaining elements.

Definition 6 (λ-model, from [HS86]). A *λ-model* is a triple $< D, \cdot, \mathcal{M} >$, where D is a set with at least two elements, '$\cdot$' is a mapping from $D \times D$ to D and $\mathcal{M}$ is a mapping which assigns, to each λ-term M and valuation ρ, a member $\mathcal{M}[\![M]\!]_\rho \in D$ such that:

1. $\mathcal{M}[\![x]\!]_\rho = \rho(x)$ 2. $\mathcal{M}[\![MN]\!]_\rho = \mathcal{M}[\![M]\!]_\rho \cdot \mathcal{M}[\![N]\!]_\rho$
3. $\mathcal{M}[\![\lambda x.M]\!]_\rho \cdot d = \mathcal{M}[\![M]\!]_{[d/x]\rho}$, for all $d \in D$
4. $\mathcal{M}[\![M]\!]_\rho = \mathcal{M}[\![M]\!]_\sigma$, if $\rho(x) = \sigma(x)$ for all x free in M
5. $\mathcal{M}[\![\lambda x.M]\!]_\rho = \mathcal{M}[\![\lambda y.M\{y/x\}]\!]_\rho$, for y not free in M
6. if $\mathcal{M}[\![M]\!]_{[d/x]\rho} = \mathcal{M}[\![N]\!]_{[d/x]\rho}$ for all $d \in D$, then $\mathcal{M}[\![\lambda x.M]\!]_\rho = \mathcal{M}[\![\lambda x.N]\!]_\rho$.□

Our λ-model should respect the semantic relation adopted in HOπ. So, let us denote by $[\,A\,]_\approx$ the equivalence class of the agent A, namely

$$[\,A\,]_\approx = \{A' \ :\ A' \text{ is an HO}\pi \text{ agent and } A \approx A'\}$$

The elements of the domain D of the model will be the equivalence classes of the closed HOπ agents with the same sort S as the agents encoding λ-terms.

$$D \overset{def}{=} \{[\,F\,]_\approx \ :\ F \in \text{HO}\pi \text{ and } F \text{ has sort } S\}$$

The definition of application on these elements follows the translation of λ-application in $\mathcal{H}$:

$$[G]_{\approx} \cdot [F]_{\approx} \stackrel{def}{=} [(p)\nu q\,(G\langle q\rangle \mid \overline{q}\langle F,p\rangle)]_{\approx} \quad \text{for } p,q \text{ not free in } G,F$$

Note that the definition of application is consistent: by the congruence properties of $\approx$, the result of the application does not depend upon the representatives G and F chosen from the equivalence classes. We are left with the definition of the mapping $\mathcal{M}[\![M]\!]_\rho$. The valuation ρ maps λ-variables to equivalence classes of $\approx$. Given a valuation ρ, we denote by ρ^H a substitution from HOπ variables to HOπ agents s.t.

$$\text{for each } \lambda\text{-variable } x, \quad \rho^H(\mathcal{H}[\![x]\!]) \in \rho(x)$$

Therefore, ρ^H is the "conversion" of ρ which operates on the HOπ variable $\mathcal{H}[\![x]\!]$ and which selects a representative out of the equivalence class of $\rho(x)$. Now, the mapping $\mathcal{M}$ of the λ-model is defined using $\mathcal{H}$ as follows:

$$\mathcal{M}[\![M]\!]_\rho \stackrel{def}{=} [\mathcal{H}[\![M]\!]\rho^H]_{\approx}$$

Note that since $\approx$ is a congruence, this definition is independent of the representatives of the equivalence classes selected by ρ^H. We denote by $\mathcal{D}$ be the triple $< D, \cdot, \mathcal{M} >$ so obtained.

Theorem 7. *$\mathcal{D}$ is a λ-model.*

PROOF: Use the definition of $\mathcal{D}$ plus the congruence properties of $\approx$ to show that each clause of Definition 6 is satisfied. □

We could have tried to be more selective in the definition of the domain $\mathcal{D}$, and take as domain $D^* = \{[\mathcal{H}[\![M]\!]]_{\approx} \;:\; M \in \Lambda\}$; then $\mathcal{D}^* =< D^*, \cdot, \mathcal{M} >$ represents the *interior* of $\mathcal{D}$ [HS86]. But it turns out that $\mathcal{D}^*$ is *not* a λ-model. Clause (6) in Definition 6 fails. As counterexample, take the terms L_1 and L_2 as will be defined in Section 6. Their encodings are not equivalent, i.e. $\mathcal{H}[\![L_1]\!] \not\approx \mathcal{H}[\![L_2]\!]$; however, for all closed N it holds that $\mathcal{H}[\![L_1\{N/x\}]\!] \approx \mathcal{H}[\![L_2\{N/x\}]\!]$. Therefore $\mathcal{D}$ is an example of a λ-model whose interior is not a λ-model; see [HL80] for two more examples.

Now that we know that $\mathcal{D}$ is a λ-model, we can infer all properties of λ-models for it; in particular we get that

- Every provable equation of $\lambda\beta$ is valid for the encoding, up to $\approx$ (where $\lambda\beta$ is the formal theory given by α and β conversion plus the rules of inference for equivalence and congruence).
- $< D, \cdot >$ is a combinatory algebra (and hence is combinatorially complete) where the two distinguished elements k and s can be defined as $k = [\mathcal{H}[\![\lambda xy.x]\!]]_{\approx}$, and $s = [\mathcal{H}[\![\lambda xyz.xz(yz)]\!]]_{\approx}$.

However model $\mathcal{D}$ is *not* extensional, i.e. it is not a $\lambda\eta$ model. As counterexample, take Ω and $\lambda x.\Omega x$. Then $\mathcal{H}[\![\Omega]\!]\langle p\rangle \not\approx \mathcal{H}[\![\lambda x.\Omega x]\!]\langle p\rangle$, since $\mathcal{H}[\![\Omega]\!]\langle p\rangle \approx 0$, whereas $\mathcal{H}[\![\lambda x.\Omega x]\!]\langle p\rangle$ can perform a visible action at p. This failure is not too surprising, since our encoding mimics the lazy λ-calculus, in which the η rule is not valid. However, as in the lazy λ-calculus, the η rule holds if M is convergent:

Theorem 8 conditional extensionality.
$\mathcal{H}[\![M]\!]\langle p\rangle \Downarrow_p$ *implies* $\mathcal{H}[\![\lambda x.Mx]\!] \approx \mathcal{H}[\![M]\!]$, *for* $x \notin fv(M)$.

PROOF: Use Proposition 4 and the definition of the encoding. □

6 Full abstraction

Full abstraction, first studied by Milner [Mil77] and Plotkin [Plo77], is the problem of finding a denotational interpretation for a programming language such that the resulting semantic equality coincides with a notion of operational indistinguishability.

Inspired by the work of Milner and Park in concurrency [Par81, Mil89], Abramsky [Abr87] introduces an operational equivalence on the lazy λ-calculus terms called *applicative bisimulation*, built on the idea that convergence is the only observable property.

Definition 9. *Applicative bisimulation* is the largest symmetric relation $\simeq \subseteq \Lambda \times \Lambda$ such that if $M \simeq N$ and $M \Longrightarrow \lambda x.M'$, then there is an N' such that $N \Longrightarrow \lambda x.N'$ and $M'\{L/x\} \simeq N'\{L/x\}$, for all $L \in \Lambda$. □

If we take λ as the only port of the λ-calculus and $M \Downarrow_\lambda$ as meaning "M can reduce to an abstraction", then applicative bisimulation is the λ-calculus version of weak barbed congruence. This follows from the characterisation of applicative bisimulation in terms of "convergence in all contexts" given in [AO89].

The classical setting in which the full abstraction problem has been developed is the simply typed λ-calculus. With the introduction of the operational equivalence resulting from applicative bisimulation, it can be neatly transferred to the untyped λ-calculus and it has motivated elegant works by Abramsky, Ong and Boudol ([AO89, Bou91]).

A denotational interpretation is said to be *sound* if it only equates operationally equivalent terms, *complete* if it equates all operationally equivalent terms, and *fully abstract* if it is sound and complete. Let us consider what happens with the encoding $\mathcal{H}$. It is sound, since $\mathcal{H}[\![M]\!] \approx \mathcal{H}[\![N]\!]$ implies $M \simeq N$; this can be established using (mainly) Proposition 4. However, $\mathcal{H}$ is not complete. For this, take:

$$L_1 = x(\lambda y.(x\Xi\Omega y))\Xi \qquad\qquad L_2 = x(x\Xi\Omega)\Xi.$$

where Ξ is an always-convergent term (that is, for all $\tilde{N}$, $\Xi\tilde{N} \Downarrow$), like the term $(\lambda x.\lambda y.(xx))(\lambda x.\lambda y.(xx))$. Terms L_1 and L_2 are used by Abramsky and Ong [AO89] to show that their canonical model for lazy λ-calculus is not fully abstract. They show that L_1 and L_2 are applicative bisimilar but can be distinguished using *convergence test*, an operator which is definable in the canonical model but is not in the pure lazy λ-calculus. We also have $\mathcal{H}[\![L_1]\!] \not\approx \mathcal{H}[\![L_2]\!]$; by using Theorem 3, this follows from a similar result for the encoding into π-calculus, which Milner obtained by implementing the convergence test as a π-process [Mil90]. In terms of the model $\mathcal{D}$ of the previous section, this inequality means that L_1 and L_2 have different denotations and that $\mathcal{D}$ is not fully abstract. Given a denotational interpretation which is not fully abstract, there are two natural directions to achieve full abstraction:

- to cut down the existing "over-generous" semantic domain (*restrictive approach*);
- to enrich the language (*expansive approach*).

These two approaches are exemplified by the solutions to the full abstraction problem for PCF (a typed λ-calculus extended with fixed points, boolean and arithmetic features) proposed by Milner [Mil77] and Plotkin [Plo77]; in the latter, PCF is augmented with a 'parallel or' operator. We shall see that in our case both directions lead to interesting constructions.

6.1 The restrictive approach

The first approach exploits the possibility of quantifying barbed bisimulation on a *particular* class of contexts, which allows us to specify exactly the way in which certain agents are supposed to be used. As λ-terms are *only* used in λ-calculus contexts, so we can require that their encodings be used only in encodings of λ-contexts. The encoding $\mathcal{H}$ is extended to λ-contexts by mapping the λ hole to the HOπ hole, i.e. $\mathcal{H}[\![[\cdot]]\!] \stackrel{def}{=} [\cdot]$. Thus, the class of contexts we are interested in is

$$\mathcal{L} = \{\mathcal{H}[\![C_\lambda[\cdot]]\!] \text{ such that } C_\lambda[\cdot] \text{ is a } \lambda\text{-context}\}$$

For $P, Q \in HO\pi$, we set $P \approx_{\mathcal{L}} Q$ if for every $\mathcal{L}$-context $C[\cdot]$, the processes $C[P]$ and $C[Q]$ are barbed bisimilar.

Proposition 10. *For each $M, N \in \Lambda$, it holds that $M \simeq N$ iff $\mathcal{H}[\![M]\!] \approx_{\mathcal{L}} \mathcal{H}[\![N]\!]$*

PROOF: By use of the operational correspondence (Proposition 4), the characterisation of $\simeq$ in terms of barbed congruence, the congruence properties of $\simeq$. $\Box$

This result allows us to construct a fully abstract model for the lazy λ-calculus. Let $[A]_{\approx_{\mathcal{L}}}$ be the equivalence class of A modulo $\approx_{\mathcal{L}}$, '$\cdot$' and $\mathcal{M}[\![M]\!]$ as defined in Section 5 but with $[\,]_{\approx_{\mathcal{L}}}$ in place of $[\,]_{\approx}$, and

$$D' = \{[\mathcal{H}[\![M]\!]]_{\approx_{\mathcal{L}}} \ : \ M \in \Lambda\}.$$

Theorem 11 full abstraction. *$\mathcal{D}' = < D', \cdot, \mathcal{M} >$ is a fully abstract model for the lazy λ-calculus.*

PROOF: Full abstraction follows from Proposition 10. The proof that $\mathcal{D}'$ is a λ-model is analogous to the proof that $\mathcal{D}$ is a λ-model in Theorem 7. (The proof of Theorem 7 used the congruence properties of $\approx$; in this case, we need the congruence of $\approx_{\mathcal{L}}$ on encodings of λ-contexts; that is, if $C[\cdot]$ is the encoding of a λ-context and $\mathcal{H}[\![M]\!] \approx_{\mathcal{L}} \mathcal{H}[\![N]\!]$, then $C[\mathcal{H}[\![M]\!]] \approx_{\mathcal{L}} C[\mathcal{H}[\![N]\!]]$). $\Box$

Indeed, the model $\mathcal{D}'$ is also *fully expressive*: all objects of the domain of interpretation are λ-definable. These results show that if from π-calculus and HOπ we discard everything which is extraneous to the encoding, in particular adopting $\approx_{\mathcal{L}}$ as semantic equivalence, then the structure that we get back is the "same thing" as the lazy λ-calculus. In our view, this was really the decisive test for the correctness of the HOπ and π-calculus encodings.

The domain D' weakens the domain D of Section 5 in two aspects: The behavioural equivalence is $\approx_{\mathcal{L}}$ rather than the more discriminanting $\approx$; and only the interior of D is taken into account. The first restriction is necessary to get full abstraction; the second to make the definition of application consistent. It is interesting to note the relationship between the choice of the class of HOπ agents and the choice of the behavioural equivalence in the definition of the domains D and D'. In the former, we took the class $\mathcal{A}$ of *all* admissible agents, and then in the behavioural equivalence we had to use quantification over the class *Cnt* of *all* admissible contexts (definition of $\approx$); in the latter we restricted to the *"interior"* of $\mathcal{A}$, and then in the behavioural equivalence we had to restrict to the *"interior"* of *Cnt* (definition of $\approx_{\mathcal{L}}$).

6.2 The expansive approach

We next study the equivalence induced on λ-terms by the encoding, called ***λ-observational equivalence***; it equates the λ-terms M and N if $\mathcal{H}[M] \approx \mathcal{H}[N]$. In other words, we look at the effect on λ-terms of the use of "richer" contexts, in which also concurrent features may be present. To derive a direct characterisation of λ-observational equivalence (i.e. a characterisation not mentioning the encoding) we have to enrich the λ-calculus with constants. A *constant* is a symbol which is added to the language without specifying any operational rule; in this sense they are opposed to *operators*, for which the behavioural rules are given (examples are convergence test and non-deterministic choice). Constants can be found in the well-known technique of the *top down specification and analysis*, where a system is developed through a series of refinement steps each representing a different level of abstraction; a lower level implements some details which at a higher level are left undefined. A constant c is then a high level primitive standing for some lower level procedure K_c; Now for closed terms, $c\tilde{M}$ becomes a sensible normal form. Operationally, we really can see it as the *output* of the tuple $\tilde{M}$ along the channel c and towards K_c.

Let Λ_C be the class of λ-terms enriched with constants. When generalising applicative bisimulation to terms in Λ_C, the main question is which condition should be imposed on the equality between the terms $c\tilde{M}$ and $c\tilde{N}$. According to the above interpretation of constants, it is natural to require that the *ordered* sequence of arguments represented by $\tilde{M}$ and $\tilde{N}$ be equivalent (clause (2) in the following definition).

Definition 12. *Applicative bisimulation over Λ_C*, written $\simeq_C$, is the largest symmetrical relation on $\Lambda \times \Lambda$ such that $M \simeq_C N$ implies:

1. if $M \Longrightarrow \lambda x.M'$ then there is an N' such that $N \Longrightarrow \lambda x.N'$ and $M'\{L/x\} \simeq_C N'\{L/x\}$, for all $L \in \Lambda_C$;
2. if $M \Longrightarrow cM_1...M_n$, for some $n \geq 0$ and $c \in C$, then there are $N_1, \ldots N_n$ such that $N \Longrightarrow cN_1 \ldots N_n$ and $M_i \simeq_C N_i$, $1 \leq i \leq n$.

The proof (in [San92a]) that $\simeq_C$ coincides with λ-observational equivalence (in particular the implication from right to left) is delicate. Milner's encoding is extended to Λ_C by mapping constants to a special kind of agents called *triggers*. These were introduced in [San92a] to obtain a simple characterisation of barbed congruence in HOπ; it turns out that their discriminanting power is the same as that of constants in the λ-calculus.

Theorem 13 (direct characterisation of λ-observational equivalence). *If $M, N \in \Lambda$, it holds that $M \simeq_C N$ iff $\mathcal{H}[M] \approx \mathcal{H}[N]$* □

Let $\mathcal{D}_C$ be the extension to Λ_C of the model $\mathcal{D}$ of Section 5; $\mathcal{D}_C$ is defined as $\mathcal{D}$ with Λ_C in place of Λ, and utilising the extension of $\mathcal{H}$ to Λ_C.

Corollary 14 (full abstraction for $\mathcal{D}$). $\mathcal{D}_C$ is a fully abstract model for the lazy λ-calculus enriched with constants. □

Starting from these results, the study of λ-observational equivalence has been continued in [San92b]. The outcomes suggest that it is a *robust* equivalence. First, it enjoys simple operational and denotational characterisations. Secondly it coincides with the equivalence obtained when the λ-calculus is augmented with the whole class of *well-formed operators*, a fairly large class of operators whose behaviour depends only on the semantics — not on the syntax — of their operands; that is to say, the encoding into π-calculus/HOπ induces maximal observational discrimination on λ-terms.

References

[Abr87] S. Abramsky. *Domain Theory and the Logic of Observable Properties.* PhD thesis, University of London, 1987.

[AO89] S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. Technical report, Imperial College, 1989. To appear in *Information and Computation.*

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics,* volume 103 of *Studies in Logic.* North Holland, 1984. Revised edition.

[Bou89] G. Boudol. Towards a lambda calculus for concurrent and communicating systems. In *TAPSOFT '89,* volume 351 of *Lecture Notes in Computer Science,* pages 149–161, 1989.

[Bou91] G. Boudol. A lambda calculus for (strict) parallel functions. Rapport de recherche 1387, INRIA-Sophia Antipolis, 1991. To appear in Information and Computation.

[HL80] J.R. Hindley and G. Longo. Lambda calculus models and extensionality. *Zeitschr. f. math. Logik und Grundlagen d. Math.,* 26:289–310, 1980.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[Mil77] R. Milner. Fully abstract models of typed lambda calculus. *Theoretical Computer Science,* 4:1–22, 1977.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil90] R. Milner. Functions as processes. Research Report 1154, INRIA, Sofia Antipolis, 1990. Final version in *Journal of Mathem. Structures in Computer Science* 2(2):119–141, 1992.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. To appear in the *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[Par81] D.M. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conf. on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*. Springer Verlag, 1981.

[Plo75] G.D Plotkin. Call by name, call by value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] G.D Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[PS93] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. In *8th LICS Conf.* IEEE Computer Society Press, 1993.

[San92a] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* CST–99–93, Department of Computer Science, University of Edinburgh, 1992. Also published as ECS–LFCS–93–266.

[San92b] D. Sangiorgi. The lazy lambda calculus in a concurrency scenario. In *7th LICS Conf.* IEEE Computer Society Press, 1992.

[Tho90] B. Thomsen. *Calculi for Higher Order Communicating Systems.* PhD thesis, Department of Computing, Imperial College, 1990.

Time Abstracted Bisimulation: Implicit Specifications and Decidability

Kim G. Larsen[*]and Wang Yi[†]

Abstract

In the last few years a number of real–time process calculi have emerged with the purpose of capturing important quantitative aspects of real–time systems. In addition, a number of process equivalences sensitive to time–quantities have been proposed, among these the notion of timed (bisimulation) equivalence in [RR86, DS89, HR91, BB89, NRSV90, MT90, Wan91b].

In this paper, we introduce a *time–abstracting* (bisimulation) equivalence, and investigate its properties with respect to the real–time process calculus of [Wan90]. Seemingly, such an equivalence would yield very little information (if any) about the timing properties of a process. However, time–abstracted reasoning about a composite process may yield important information about the relative timing–properties of the components of the system. In fact, we show as a main theorem that such implicit reasoning will reveal *all* timing aspects of a process. More precisely, we prove that two processes are interchangeable in any context up to time–abstracted equivalence precisely if the two processes are themselves timed equivalent.

As our second main theorem, we prove that time–abstracted equivalence is decidable for the calculus of [Wan90] using classical methods based on a finite–state symbolic, structured operational semantics.

1 Introduction

During the last few years various process calculi have been extended to include real–time in order to handle quantitative aspects of real–time systems, for instance that some critical event must not or should happen within a certain time period. The extensions often include timed versions of classical process equivalences, e.g. timed bisimulation equivalence, timed failure equivalence and timed trace equivalence [RR86, DS89, HR91, NRSV90, MT90, Wan91b]. Loosely speaking, for two processes to be equivalent they should not only agree on *what* actions they can perform, they must also agree on *when* these actions are performable. Alternatively, one can say that an observer is assumed to be sensitive to passage of time including the quantity by which time is passing.

A fundamental problem induced by any new process calculus is that of axiomatization and decidability of the associated process equivalence. Normally, these problems are solved in two stages: the problems are first solved for the class of regular processes, i.e. processes with no parallel composition, after which it is shown how to remove parallel composition through the use of a so–called expansion theorem. However, for real–time calculi where time is represented

[*]Department of Mathematics and Computer Science, Institute for Electronic Systems, Aalborg University, DK9220 Aalborg, Denmark. E-mail: kgl@iesd.auc.dk. The work of this author was supported by the Danish Natural Science Research Council project DART and partially by the ESPRIT Basic Research Action 7166, CONCUR2.

[†]Department of Computer Systems, Box 325, Uppsala University, S751 05, Uppsala, Sweden. E-mail: yi@docs.uu.se. The work of this author was done during a visit to the deparment of mathematics and computer science at Aalborg University, and supported by the Danish Natural Science Research Council project DART and partially by the ESPRIT Basic Research Action, REACT.

by some dense time domain (such as the non-negative reals) processes will have infinitely many states, and it has been shown in [GL92] that no expansion theorem exists for timed bisimulation equivalence — i.e. parallel composition can not in general be removed. This explains why axiomatization and decidability of various equivalences between real-time processes based on dense time domains have proven notoriously hard problems. Recent work by Čerāns [Č92], Chen [Che91b] and Fokkink and Klusener [FK91] offers the first examples of decidability and axiomatization for real-time calculi based on dense time.

In this paper we introduce a *time-abstracting* (bisimulation) equivalence between real-time processes, i.e. in comparing real-time processes we shall abstract away from passage of time [1]. Seemingly, such an equivalence would yield very little information (if any at all) about the timing behaviour of a real-time system. However, if the real-time system is a combination of real-time systems, $O(P_1, \dots, P_n)$ say, time-abstracted reasoning will at least yield some information about the relationship between the concrete timing properties of the components $P_1, \dots, P_n$. In fact, as we shall prove as a main theorem of this paper, in a certain formal sense *all* timing aspects of a real-time system may be revealed in this manner.

As the second main contribution of this paper, we demonstrate that the time-abstracted equivalence is decidable using essentially classical methods based on a finite-state symbolic, structured operational semantics. The symbolic semantics is based on a discrete version of the standard (continuous) operational semantics. In order to obtain completeness it is essential that the symbolic semantics is based on a sufficiently fine "granularity". In fact, we show that the "granularity" required is linearly dependent on the number of parallel components.

To further motivate the usefulness of time-abstracted equivalence consider the combined system in Figure 1 consisting of two (disposable) media A and B.

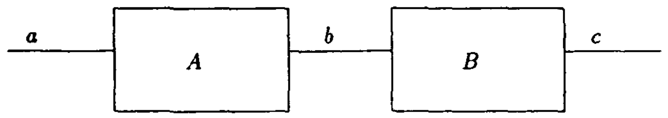


Figure 1: A Combined Medium

Functionally, the two media are nearly identical: they accept messages on the left port passing them on to the right port. However, taking time into account, there are important differences between the media: after having accepted a message on port a, A is immediately able to deliver the massage on port b. However, if the message has not been taken after a delay of t_a a timeout will occur and the massage is lost. In contrast, the medium B will never lose a message once it has been accepted. However, a message can only be accepted on port b after some initial delay t_b. Using the timed calculus of Wang [Wan90, Wan91b, Wan91a] the two media A and B may be specificed as follows:

$$\begin{aligned} A &\stackrel{def}{=} a.(\bar{b}.\mathrm{nil} + \epsilon(t_a).\tau.\mathrm{nil}) \\ B &\stackrel{def}{=} \epsilon(t_b).b.\bar{c}.\mathrm{nil} \end{aligned}$$

It should be obvious that even from a time-abstracted point of view, the behaviour of the combined system $(A \,|\, B)\backslash b$ is highly dependent on the timing parameters t_a and t_b. Essentially, if $t_a > t_b$ the combined system will function as a proper (disposable) medium, i.e.:

$$(A \,|\, B)\backslash b \stackrel{\bullet}{\approx} a.\bar{c}.\mathrm{nil} \qquad (1)$$

[1]This abstraction is very similar to the abstraction from internal computation in classical process algebras.

where $\overset{\bullet}{\approx}$ denotes our (weak) time-abstracting equivalence [2]. In contrast, if $t_b > t_a$, the combined medium may not be able to successfully deliver messages; in fact the following will hold [3]:

$$(A \mid B)\backslash b \overset{\bullet}{\approx} a.(\tau.\text{nil} + \tau.\bar{c}.\text{nil}) + \tau.a.\bar{c}.\text{nil} \tag{2}$$

Even though, we gain information about the *relationship* of the timing behaviours of A and B in both (1) and (2), we have no information about the timing behaviour of the combined system. Obviously, in the case (1) a message can be delivered on port c after a delay of less than t_b from the acceptance of the message. Using the (weak) timed bisimulation equivalence from [Wan91a] such properties can be specified:

$$(A \mid B)\backslash b \approx a.\epsilon(t_b).\bar{c}.\text{nil}^4$$

Alternatively, one can express such *explicit* timing properties using Timed Modal Logics, e.g. [ACD90, HLW91, HNJ92, RH92]. However, we can also formulate explicit timing properties using time-abstracted equivalence by resorting to *implicit specifications*: i.e. instead of specifying properties of $S = (A \mid B)\backslash b$ directly we specify properties of the system S in certain *contexts*. Concretely, specifying that S must be able to deliver on port c after a delay of no more than d after acceptance on port c can be expressed as follows:

$$(\bar{a}.(c.w.\text{nil} + \epsilon(d).\tau.\text{nil}) \mid S)\backslash\{a, c\} \quad \overset{\bullet}{\approx} \quad w.\text{nil} \tag{3}$$

where w is a distinguished (success) action. Here, we are exploiting the *maximal progress* property of the calculus in [Wan91a] [5].

The previously announced main theorem, that all explicit timing properties can be captured using time-abstracted equivalence, can now be made more precise: we show that implicit time-abstracting specifications of the form (3) precisely characterizes *timed* bisimulation equivalence. That is, two timed processes are timed bisimulation equivalent just in case they satisfy the same implicit time-abstracted specifications. Thus, without any loss of discriminating power, one may use time-abstracting bisimulation equivalence instead of timed bisimulation equivalence.

The outline of the paper is as follows: in section 2 we review the timed calculus of [Wan90, Wan91b, Wan91a] together with the notion of timed bisimulation; in section 3 strong and weak notions of time-abstracted bisimulations are introduced; in section 4 we prove as our first main theorem that implicit time-abstracting specifications are as discriminating as timed bisimulation; section 5 contains our second main contribution: decidability of strong and weak time-abstracted bisimulation equivalence. Finally, in section 6 we give some concluding remarks. To achieve readability while maintaining credibility we enclose full proofs in the appendices.

2 Timed Processes

2.1 Syntax and Semantics

The language we use to describe timed processes is essentially, Milner's CCS extended with a delay construct $\epsilon(d).P$. Informally, $\epsilon(d).P$ means "wait for d units of time and then behave like P", where $d \in \mathcal{R}_+$ is a nonnegative real.

[2] Weak indicating that $\overset{\bullet}{\approx}$ also abstracts from internal computation.

[3] The summand $\cdots \tau.a.\bar{c}.\text{nil}$ reflects that messages *may* successfully be delivered in case A delays sufficiently long before accepting a messages as this will reduce the remaining delay for B.

[4] The displayed equivalence does in fact not hold as the delay required before the delivery depends on the delay before the acceptance. Using time-variables as in [Wan91b] a valid equation would be: $(A \mid B)\backslash b \approx a@t.\epsilon(t_b - t).\bar{c}.\text{nil}$

[5] Maximal progress means that time is not allowed to pass if a system can perform internal computation.

As in CCS, we assume a set $\Lambda = \Delta \cup \bar{\Delta}$ with $\bar{\bar{\alpha}} = \alpha$ for all $\alpha \in \Lambda$, ranged over by α, β representing external actions, and a distinct symbol τ representing internal actions. We use $\mathcal{A}ct$ to denote the set $\Lambda \cup \{\tau\}$ ranged over by a, b representing both internal and external actions.

Further, assume a set of process variables ranged over by X.

We adopt a two-phase syntax to describe networks of regular timed processes. First, regular timed process expressions are generated by the following grammar:

$$E ::= \mathrm{nil} \mid X \mid \epsilon(d).E \mid a.E \mid E + E \mid X \overset{def}{=} E$$

We shall restrict process expressions to be *well-guarded* in the following sense:

Definition 1 *X is well-guarded in E if and only if every free occurrence of X in E is within a subexpression (a guard) of the form $a.F$ in E.*

E is well-guarded if and only if every free variable in E is well-guarded in E, and for every subexpression of the form $X \overset{def}{=} F$ in E, X is well-guarded in F. □

Closed and well-guarded expressions generated by the grammar above are called *regular timed processes*. Networks of regular timed processes are described by CCS parallel composition:

$$P_1 | \ldots | P_n$$

where P_i are regular timed processes. For simplicity, we have ignored the other CCS operators. However, the results of this paper can be easily extended to more general types of networks modelled by the combination of parallel composition, restriction and relabelling:

$$(P_1[S_1] | \ldots | P_n[S_n]) \backslash A$$

$$\begin{array}{ccc}
\dfrac{}{a.P \xrightarrow{a} P} & \dfrac{P \xrightarrow{a} P'}{\epsilon(0).P \xrightarrow{a} P'} & \\[2ex]
\dfrac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} & \dfrac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'} & \dfrac{P \xrightarrow{a} P'}{X \xrightarrow{a} P'} \; [X \overset{def}{=} P] \\[2ex]
\dfrac{Q \xrightarrow{a} Q'}{P|Q \xrightarrow{a} P|Q'} & \dfrac{P \xrightarrow{a} P'}{Q \xrightarrow{a} P'|Q} & \dfrac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}
\end{array}$$

Table 1: Action Rules for Timed Semantics.

$$\begin{array}{ccc}
\dfrac{}{\mathrm{nil} \xrightarrow{\epsilon(d)} \mathrm{nil}} & \dfrac{}{\epsilon(c+d).P \xrightarrow{\epsilon(d)} \epsilon(c).P} & \dfrac{P \xrightarrow{\epsilon(d)} P'}{\epsilon(c).P \xrightarrow{\epsilon(c+d)} P'} \\[2ex]
\dfrac{}{\alpha.P \xrightarrow{\epsilon(d)} \alpha.P} & \dfrac{P \xrightarrow{\epsilon(d)} P' \quad Q \xrightarrow{\epsilon(d)} Q'}{P + Q \xrightarrow{\epsilon(d)} P' + Q'} & \dfrac{P \xrightarrow{\epsilon(d)} P'}{X \xrightarrow{\epsilon(d)} P'} \; [X \overset{def}{=} P] \\[2ex]
\dfrac{P \xrightarrow{\epsilon(d)} P' \quad Q \xrightarrow{\epsilon(d)} Q'}{P|Q \xrightarrow{\epsilon(d)} P'|Q'} \; [Sort_d(P) \cap \overline{Sort_d(Q)} = \emptyset] & &
\end{array}$$

Table 2: Delay Rules for Timed Semantics .

We will use P, Q to range over timed processes.

A timed operational semantics for the language has been developed in [Wan90]. We present the transition rules in two groups: rules for actions in table 1 [6] and rules for delays in table 2 [7].

Note that the side condition for the delay rule of parallel composition is to guarantee that the parallel processes satisfy the maximal progress assumption, that is, a *timed process will never wait if it can perform an internal action τ*. The condition is formalized by means of $Sort_d(P)$ defined inductively on the structure of processes P, in table 3. Intuitively, $Sort_d(P)$ includes all external actions that P is able to perform within d time units; whereas $Sort_d(P) \cap \overline{Sort_d(Q)} = \emptyset$ [8] means that P and Q cannot communicate with each other within d time units.

Definition 2 *Given a process P, we define $Sort_0(P) = \emptyset$ and $Sort_c(P)$ for $c \neq 0$ to be the least set satisfying the equations [9] given in table 3.* □

$Sort_c(\text{nil})$	$=$	$\emptyset$
$Sort_c(\alpha.P)$	$=$	$\{\alpha\}$
$Sort_c(\tau.P)$	$=$	$\emptyset$
$Sort_c(\epsilon(d).P)$	$=$	$Sort_{c \dot{-} d}(P)$
$Sort_c(P+Q)$	$=$	$Sort_c(P) \cup Sort_c(Q)$
$Sort_c(X)$	$=$	$Sort_c(P) \quad [X \overset{def}{=} P]$
$Sort_c(P\|Q)$	$=$	$Sort_c(P) \cup Sort_c(Q)$

Table 3: Equations for $Sort_c(P)$.

The following properties of timed processes will be often referred in the later sections.

Proposition 1

1. *(maximal progress) If $P \xrightarrow{\tau} P'$ for some P', then $P \xrightarrow{\epsilon(d)} P''$ for no d and P''.*
2. *(time determinism) Whenever $P \xrightarrow{\epsilon(d)} P'$ and $P \xrightarrow{\epsilon(d)} P''$ then $P' = P''$.*
3. *(persistency) If $P \xrightarrow{\epsilon(d)} P'$ and $P \xrightarrow{\alpha} Q$ for some P' and Q, then $P' \xrightarrow{\alpha} Q'$ for some Q'.*
4. *(time continuity) For all c, d and P'', $P \xrightarrow{\epsilon(c+d)} P''$ iff $P \xrightarrow{\epsilon(c)} P' \xrightarrow{\epsilon(d)} P''$ for some P'.* □

We end this section with notation:

- $\overline{P}$ stands for a network $P_1|...|P_n$ where P_i are regular timed processes.
- Whenever $P \xrightarrow{\epsilon(d)} P'$, P^d stands for P' [10] : note that P^d is well-defined due to time-determinism property stated above.
- $\overline{P}^{\overline{x}}$ stands for $P_1^{x_1}|...|P_n^{x_n}$ for $\overline{x} = (x_1, ..., x_n)$.

[6]Note that apart from the rule for $\epsilon(0).P$, the action rules are exactly the same as in CCS.

[7]In table 2, we use d to stand for a non-zero real; this implies that a $\epsilon(0)$-transition can never be inferred by the inference rules. However, we shall apply the convention that $P \xrightarrow{\epsilon(0)} P$ for all P.

[8]Here, $\overline{Sort_d(Q)}$ is defined to be the set $\{\bar{\alpha} \mid \alpha \in Sort_d(Q)\}$.

[9]In table 3, $c \dot{-} d$ is defined to be $c - d$ if $c > d$, 0 otherwise.

[10]Note that P^0 stands for P following the convention that $P \xrightarrow{\epsilon(0)} P$ for all P.

Conceptually, one can imagine each component P_i of a network $\overline{P}$ to be equipped with a private clock. All clocks proceed at the same speed and a clock-value will be reset to 0 when the corresponding component perform a real action; $\overline{P}^{\overline{x}}$ denotes the state of $\overline{P}$ in which the clock-values are $x_1, ..., x_n$.

2.2 Timed Bisimulation

We have developed a labelled transition system: $\langle \mathcal{P}_R, \longrightarrow, \mathcal{L} \rangle$ where $\mathcal{P}_R$ is the set of timed processes generated by the two-phase syntax; $\longrightarrow$ is the least relation satisfying the inference rules given in table 1 and table 2; $\mathcal{L}$ is the set of labels, $\mathcal{A}ct \cup \{\epsilon(d) \mid d \in \mathcal{R}_+\}$. To compare timed processes, strong and weak notions of timed bisimulation have been defined based on this transition system in [Wan90].

Definition 3 *(strong timed bisimulation) A binary relation $\mathcal{S}$ on $\mathcal{P}_R$ is a strong timed simulation if $(P, Q) \in \mathcal{S}$ implies that for all $a \in \mathcal{A}ct$ and $d \in \mathcal{R}_+$,*

1. *Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{S}$*
2. *Whenever $P \xrightarrow{\epsilon(d)} P'$ then, for some Q', $Q \xrightarrow{\epsilon(d)} Q'$ and $(P', Q') \in \mathcal{S}$*

We call such a simulation $\mathcal{S}$ a strong timed bisimulation if it is symmetrical. The largest strong timed bisimulation is called strong timed equivalence, denoted $\sim$. □

Weak timed equivalence is defined by abstracting away from internal actions.

Definition 4

1. $P \overset{\tau}{\Longrightarrow} Q$ *if* $P(\xrightarrow{\tau})^*Q$
2. $P \overset{a}{\Longrightarrow} Q$ *if* $P(\xrightarrow{\tau})^* \xrightarrow{a} (\xrightarrow{\tau})^*Q$
3. $P \overset{\epsilon(d)}{\Longrightarrow} Q$ *if* $P(\xrightarrow{\tau})^* \xrightarrow{\epsilon(d_1)} (\xrightarrow{\tau})^* ... (\xrightarrow{\tau})^* \xrightarrow{\epsilon(d_n)} (\xrightarrow{\tau})^*Q$ *where* $d = \sum_{i \leq n} d_i$. □

Definition 5 *(weak timed bisimulation) A binary relation $\mathcal{S}$ on $\mathcal{P}_R$ is a weak timed simulation if $(P, Q) \in \mathcal{S}$ implies that for all $a \in \mathcal{A}ct$ and $d \in \mathcal{R}_+$,*

1. *Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q \overset{a}{\Longrightarrow} Q'$ and $(P', Q') \in \mathcal{S}$*
2. *Whenever $P \xrightarrow{\epsilon(d)} P'$ then, for some Q', $Q \overset{\epsilon(d)}{\Longrightarrow} Q'$ and $(P', Q') \in \mathcal{S}$*

We call such a simulation $\mathcal{S}$ a weak timed bisimulation if it is symmetrical. The largest weak timed bisimulation is called weak timed equivalence, denoted $\approx$. □

In [Wan91a], it has been shown that $\sim$ is a congruence w.r.t all CCS operators and $\approx$ is a congruence w.r.t. all the other operators except summation and recursion.

3 Time Abstracted Equivalences

In analyzing a large system, we often need to make proper abstractions according to what properties of the system we are interested. One such example is weak timed equivalence, which abstracts away from internal actions. In this section, we develop notions of bisimulation abstracting away from both time delays and internal actions.

Definition 6 *(abstracting away from time)*

1. $P \xrightarrow{\epsilon}\!\!\bullet\, Q$ *if* $P(\xrightarrow{\epsilon(d)})^{*}Q$
2. $P \xrightarrow{a}\!\!\bullet\, Q$ *if* $P \xrightarrow{\epsilon}\!\!\bullet \xrightarrow{a} \xrightarrow{\epsilon}\!\!\bullet\, Q$ □

For example, $\epsilon(2).\alpha.P \xrightarrow{\epsilon}\!\!\bullet\, \epsilon(0.3).\alpha.P, \ldots, \epsilon(2).\alpha.P \xrightarrow{\epsilon}\!\!\bullet\, \alpha.P$. Here, we simply consider a timed transition like $P \xrightarrow{\epsilon(d)} Q$ as an empty transition $P \xrightarrow{\epsilon}\!\!\bullet\, Q$ where the quantitative part i.e. d of the transition is ignored. This assumes that the observer (or environment) who makes the observation is *insensitive to time–quantities.* Naturally, we may identify two processes if they can not be distinguished by any time insensitive environment.

Definition 7 *(strong time abstracted equivalence) A binary relation $\mathcal{S}$ on $\mathcal{P}_R$ is a strong time abstracted simulation if $(P,Q) \in \mathcal{S}$ implies that for all $a \in \mathcal{A}ct$ and $d \in \mathcal{R}_{+}$,*

1. *Whenever $P \xrightarrow{a} P'$ then, for some Q', $Q \xrightarrow{a}\!\!\bullet\, Q'$ and $(P',Q') \in \mathcal{S}$*
2. *Whenever $P \xrightarrow{\epsilon(d)} P'$ then, for some Q', $Q \xrightarrow{\epsilon}\!\!\bullet\, Q'$ and $(P',Q') \in \mathcal{S}$*

We call such a simulation $\mathcal{S}$ a strong time abstracted bisimulation if it is symmetrical. The largest strong time abstracted bisimulation is called strong time abstracted equivalence, denoted $\overset{\bullet}{\sim}$. □

For example, $\epsilon(2).\tau.\text{nil}|\epsilon(1).\beta.\text{nil} \overset{\bullet}{\sim} \tau.\text{nil}|\beta.\text{nil} \overset{\bullet}{\sim} \tau.\beta.\text{nil} + \beta.\tau.\text{nil}$. Note that in terms of timed bisimulation equivalence $\sim$, there is no regular process equivalent to the parallel process.

We make a further abstraction to abstract away from internal actions.

Definition 8 *(abstracting away from time and τ)*

1. $P \overset{\epsilon}{\Longrightarrow}\!\!\bullet\, Q$ *if* $P(\xrightarrow{\epsilon(d)} \cup \xrightarrow{\tau})^{*}Q$
2. $P \overset{\alpha}{\Longrightarrow}\!\!\bullet\, Q$ *if* $P \overset{\epsilon}{\Longrightarrow}\!\!\bullet \xrightarrow{\alpha} \overset{\epsilon}{\Longrightarrow}\!\!\bullet\, Q$ □

Definition 9 *(weak time abstracted equivalence) A binary relation $\mathcal{S}$ on $\mathcal{P}_R$ is a weak time abstracted simulation if $(P,Q) \in \mathcal{S}$ implies that for all $\alpha \in \mathcal{A}ct - \{\tau\}$ and $\theta \in \{\tau\} \cup \{\epsilon(d) \mid d \in \mathcal{R}_{+}\}$,*

1. *Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \overset{\alpha}{\Longrightarrow}\!\!\bullet\, Q'$ and $(P',Q') \in \mathcal{S}$*
2. *Whenever $P \xrightarrow{\theta} P'$ then, for some Q', $Q \overset{\epsilon}{\Longrightarrow}\!\!\bullet\, Q'$ and $(P',Q') \in \mathcal{S}$*

We call such a simulation $\mathcal{S}$ a weak time abstracted bisimulation if it is symmetrical. The largest weak time abstracted bisimulation is called weak time abstracted equivalence, denoted $\overset{\bullet}{\approx}$. □

Now, we can further simplify our example process $\epsilon(2).\tau.\text{nil}|\epsilon(1).\beta.\text{nil}$ to $\beta.\text{nil}$ by the equation: $\epsilon(2).\tau.\text{nil}|\epsilon(1).\beta.\text{nil} \overset{\bullet}{\approx} \beta.\text{nil}$.

It seems that every timed process would be time–abstracted equivalent to an untimed process which contains no delay–construct. This is not true for $\overset{\bullet}{\sim}$. For instance,

$$(\epsilon(1).\alpha.\text{nil}|\beta.(\tau.\text{nil} + \bar{\alpha}.\omega.\text{nil}))\backslash\{\alpha\} \not\overset{\bullet}{\sim} P_{cc},$$

$$\begin{tikzcd} & \stackrel{\bullet}{\approx} & \\ \stackrel{\bullet}{\sim} \arrow[ur] & & \approx \arrow[ul] \\ & \sim \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Figure 2: Ordering Timed and Time–Abstracted Equivalences with Strength.

for all untimed processes P_{ccs}. However, it is true for weak time abstracted equivalence that for each timed process P, there will be an untimed process P_{ccs} such that $P \stackrel{\bullet}{\approx} P_{ccs}$. For instance, it is easy to prove $(\epsilon(1).\alpha.\mathrm{nil}|\beta.(\tau.\mathrm{nil} + \bar{\alpha}.\omega.\mathrm{nil}))\backslash\{\alpha\} \stackrel{\bullet}{\approx} (\tau.\alpha.\mathrm{nil}|\beta.(\tau.\mathrm{nil} + \bar{\alpha}.\omega.\mathrm{nil}))\backslash\{\alpha\}$.

We conclude this section with the commuting diagram shown in figure 2, which illustrates the relationship between *timed* and *time–abstracted* equivalences. The arrow in the diagram should be understood as set inclusion, that is: $\sim\subseteq\stackrel{\bullet}{\sim}\subseteq\stackrel{\bullet}{\approx}$ and $\sim\subseteq\approx\subseteq\stackrel{\bullet}{\approx}$. The proofs of these inclusions, that they are strict and also the only inclusions among the four equivalences are straightforward.

4 Implicit Time Abstraction

In this section we present our first main theorem: *two timed processes are strong (weak) timed equivalent if and only if they satisfy the same strong (weak) implicit time–abstracted specifications.* Here, a strong implicit time–abstracted specification of a process P is an equation of the form:

$$A\,|\,P \;\stackrel{\bullet}{\sim}\; B \qquad (4)$$

where A and B are real–time processes. That is $P \sim Q$ if and only if P and Q satisfy the same equations of the form (4). Alternatively, the results in this section say that $\sim$ ($\approx$) is the coarsest equivalence contained in $\stackrel{\bullet}{\sim}$ ($\stackrel{\bullet}{\approx}$) which is preserved by the parallel composition of our calculus [11].

Theorem 1 *$P \sim Q$ if and only if $P\,|\,N \stackrel{\bullet}{\sim} Q\,|\,N$ for all N.*

Proof: *Only If:* As $\sim$ is preserved by all operators of the calculus and since $\sim$ is contained in $\stackrel{\bullet}{\sim}$, it is obvious that this direction holds.

If: We show that the relation:

$$\mathcal{R} = \{(P,Q) \mid \text{for all } N,\ P|N \stackrel{\bullet}{\sim} Q|N\}$$

is a strong timed bisimulation. Thus consider $(P,Q) \in \mathcal{R}$.

First consider an action–transition $P \xrightarrow{a} P'$ and let $\{Q_1,\ldots,Q_m\}$ be the set of all a–derivatives for Q [12].

In case $m = 0$ (i.e. Q has no a–transitions), $P\,|\,N \not\stackrel{\bullet}{\sim} Q\,|\,N$ for $N = \bar{a}.w.\mathrm{nil} + \tau.\mathrm{nil}$, where w is a distinguished action not occurring in neither P nor Q. However, this contradicts the assumption that $(P,Q) \in \mathcal{R}$.

[11] As $\sim$ is preserved by *all* operators of the calculus, $\sim$ is in fact the congruence induced by $\stackrel{\bullet}{\sim}$. This fact does not extend to the weak case, as $\approx$ is not — as usual — preserved by $+$.

[12] We use the easily established fact, that all processes definable in our calculus are *image–finite* in the sense that the set of derivatives under any action is finite.

Thus, $m > 0$. Now, assume that $(P', Q_i) \notin \mathcal{R}$ for all i. We shall show that this leads to a contradiction. However, under this assumption it follows from the definition of $\mathcal{R}$ that for each i there exists a process N_i such that $P' \mid N_i \not\stackrel{\bullet}{\sim} Q_i \mid N_i$. Now let:

$$N \stackrel{def}{=} \bar{a}.N' \qquad N' \stackrel{def}{=} \sum_{i=1}^{m} w_i.N_i + \tau.N'$$

where w_i are distinct actions not occurring in neither P nor Q. Note, that N' is a time-stopped process (and $P \mid N'$ is time-stopped for any P) in the sense that no delay-transitions can take place. Now we claim that $P \mid N \not\stackrel{\bullet}{\sim} Q \mid N$ contradicting that $(P, Q) \in \mathcal{R}$. To argue for this consider the transition:

$$P \mid N \xrightarrow{\tau} P' \mid N' \tag{5}$$

A possible match for $Q \mid N$ must be of the form $Q \mid N \xrightarrow{\tau}\!\!\bullet\ R$. Due to the maximal progress property of our calculus, and as N' (and hence $Q_i \mid N'$) is time-stopped, the only possible such transitions are either of the form (a) $Q \mid N \xrightarrow{\tau}\!\!\bullet\ Q_i \mid N'$ or of the form (b) $Q \mid N \xrightarrow{\tau}\!\!\bullet\ Q'' \mid N$ with $Q \xrightarrow{\tau}\xrightarrow{\epsilon}\!\!\bullet\ Q''$. Clearly, transitions of the form (b) can not match (5) as $P' \mid N' \xrightarrow{w_i}$ whereas $Q'' \mid N \not\xrightarrow{w_i}\!\!\bullet$. Let us thus compare behaviours of $P' \mid N'$ and $Q_i \mid N'$: first note that with respect to w_i both possess the following unique transitions: $P' \mid N' \xrightarrow{w_i} P' \mid N_i$ and $Q_i \mid N' \xrightarrow{w_i} Q_i \mid N_i$. Thus, if $P' \mid N' \stackrel{\bullet}{\sim} Q_i \mid N'$ it follows that $w_i.(P' \mid N_i) \stackrel{\bullet}{\sim} w_i.(Q_i \mid N_i)$. However, this contradicts the assumption that $P' \mid N_i \not\stackrel{\bullet}{\sim} Q_i \mid N_i$ and the easily established fact that whenever $a.U \stackrel{\bullet}{\sim} a.V$ then also $U \stackrel{\bullet}{\sim} V$. Thus, $Q \mid N$ has no match for the transition (5) of $P \mid N$ and hence $P \mid N \not\stackrel{\bullet}{\sim} Q \mid N$ contradicting the assumption that $(P, Q) \in \mathcal{R}$.

Now consider a delay transition $P \xrightarrow{\epsilon(d)} P'$. If $Q \not\xrightarrow{\epsilon(d)}$ then clearly $P \mid N \not\stackrel{\bullet}{\sim} Q \mid N$ for $N = \epsilon(d).w.\text{nil}$ contradicting $(P, Q) \in \mathcal{R}$. Otherwise assume that $Q \xrightarrow{\epsilon(d)} Q'$ (due to time-determinism Q' is unique). Assume $(P', Q') \notin \mathcal{R}$, that is $P' \mid N' \not\stackrel{\bullet}{\sim} Q' \mid N'$ for some N'. In this case $P \mid N \not\stackrel{\bullet}{\sim} Q \mid N$ for $N = \epsilon(d).N'$ again violating the basic assumption that $(P, Q) \in \mathcal{R}$. □

Example. Consider the two processes [13]:

$$P = \epsilon(1).a \mid b \qquad Q = b.\epsilon(1).a + \epsilon(1).(a \mid b)$$

It is obvious that these two processes are not strong timed equivalent, i.e. $P \not\sim Q$. To see this, note that P possesses the following transition-sequence:

$$P \xrightarrow{\epsilon(.5)} \epsilon(.5).a \mid b \xrightarrow{b} \epsilon(.5).a \mid \text{nil} \xrightarrow{\epsilon(.5)} a \mid \text{nil}$$

The only possible match Q for is the following:

$$Q \xrightarrow{\epsilon(.5)} b.\epsilon(1).a + \epsilon(.5).(a \mid b) \xrightarrow{b} \epsilon(1).a \xrightarrow{\epsilon(.5)} \epsilon(.5).a$$

However, it is clear that this is not a proper match as $a \mid \text{nil} \xrightarrow{a}$ whereas $\epsilon(.5).a \not\xrightarrow{a}$. Now using the construction of the above theorem 1 we obtain the following process:

$$N = \epsilon(.5).\bar{b}.w_1.\epsilon(.5).(\bar{a}.w + \tau.\text{nil})$$

which distinguishes P and Q, i.e. $P \mid N \not\stackrel{\bullet}{\sim} Q \mid N$. □

We have a similar result for weak *timed* and *time abstracted* equivalences.

Theorem 2 *$P \approx Q$ if and only if $P \mid N \stackrel{\bullet}{\approx} Q \mid N$ for all N.*

[13]We are using the convention of dropping trailing nil's. That is, we write simply a for a.nil.

Proof: *Only if:* As $\approx$ is preserved by parallel composition and since $\approx$ is contained in $\overset{\bullet}{\approx}$, it is obvious that this direction holds.

If: We show that the relation: $\mathcal{R} = \{(P,Q) \mid$ for all $N.\ P|N \overset{\bullet}{\approx} Q|N\}$ is a weak timed bisimulation. A complete proof is given in the full version of the paper [LW93]. □

5 Decidability

From the delay rules in table 2, we can easily see that the timed processes are infinite-state w.r.t. $\longrightarrow$ and also $\longrightarrow\!\!\!\bullet$. For example, $\epsilon(1).P|Q \xrightarrow{\epsilon}\!\!\bullet\ \epsilon(0.3).P|Q \xrightarrow{\epsilon}\!\!\bullet \ldots \xrightarrow{\epsilon}\!\!\bullet\ \epsilon(0.0005).P|Q \xrightarrow{\epsilon}\!\!\bullet \ldots \xrightarrow{\epsilon}\!\!\bullet\ P|Q$. The infinite-stateness makes the decidability problem of $\sim$ and $\overset{\bullet}{\sim}$ notoriously hard.

To achieve decidability, we shall study a particular class of processes $\mathcal{P}_N$ ranged over by $\overline{P}, \overline{Q}$, called integer processes in which, only naturals are allowed to occur in a delay operator $\epsilon(d)$. However, we should point out that the decidability result is easily extended to processes using rational numbers in delay operators: before comparing two such processes simply multiply all delays with a common constant, sufficiently large to make all delays integers.

In this section, we prove that strong (weak) time abstracted equivalence over integer processes is decidable. The proof is constructed in two steps. First, we show that the state-space of a timed process can be partitioned into equivalence classes according to the notion of time region due to Alur and Dill, [AD90]. Secondly, we develop a time-step semantics called k-semantics which is parameterized with a granularity $1/k$. Intuitively, the k-semantics describes how a process shall behave in every $1/k$ time units. The idea is to use each state of such a time-step semantics to represent an equivalence class of states of the timed semantics. Based on the parameterized k-semantics we define a family of symbolic time abstracted equivalences $\overset{\circ}{\sim}_k$ which is also relativized to the granularity $1/k$. It turns out that $\overset{\circ}{\sim}_{n+2}$ coincides with $\overset{\bullet}{\sim}$, that is:

$$\overline{P} \overset{\bullet}{\sim} \overline{Q} \textit{ if and only if } \overline{P} \overset{\circ}{\sim}_{n+2} \overline{Q}$$

where n is the maximal number of components in the networks $\overline{P}$ and $\overline{Q}$.

Since the integer processes in the $(n+2)$-semantics are finite-state, $\overset{\circ}{\sim}_{n+2}$ can be checked using the existing techniques and algorithms for bisimulation-checking, such as [KS90, SV89, PT87, JGZ89, CPS89] and hence so can $\overset{\bullet}{\sim}$. Finally, we extend the results to weak time abstracted equivalence.

5.1 Partitioning State-Space into Equivalence Classes

To illustrate the idea, we consider a simple regular process:

$$P \overset{def}{=} \alpha.Q + \epsilon(1).\tau.R$$

The process may offer α before 1 and will time out at 1. Indeed it is infinite-state since by performing an empty transition (delay) it may reach a continuum of states, $\{P^x | x < 1\}$. However, $P^x \overset{\bullet}{\sim} P^y$ for all $x, y < 1$, that is, $\{P^x | x < 1\}$ is an equivalence class.

Naturally, we may say that all time points such as $x = 0, 0.1, \ldots, 0.9$ in the region $x < 1$ are equivalent in the sense that they give rise to an equivalence class of states. This motivates a notion of equivalence over time points in a multi-dimensional time vector.

Let $\overline{x}$ and $\overline{y}$ range over $\mathcal{R}_+^n$, understood as time points in the n-dimensional time vector. For $\overline{x} \in \mathcal{R}_+^n$ and $d \in \mathcal{R}_+$, we shall write $\overline{x} + d$ for $(x_1 + d, \ldots, x_n + d)$.

Definition 10 *$\overline{x}$ and $\overline{y}$ are equivalent, denoted by $\overline{x} \doteq \overline{y}$ if*

1. $\forall i : (\lfloor x_i \rfloor = \lfloor y_i \rfloor)$,
2. $\forall i,j : (\{x_i\} \leq \{x_j\} \Longleftrightarrow \{y_i\} \leq \{y_j\})$ *and*
3. $\forall i : (\{x_i\} = 0 \Longleftrightarrow \{y_i\} = 0)$.

where $\lfloor d \rfloor$ is the lower integer part of d and $\{d\}$ is the fractional part of d. The equivalence classes of $\mathcal{R}_+^n$ are called time regions. □

The definition above is the standard one for time region, taken from [AD90]. The first clause requires that the lower integer parts of $\overline{x}$ and $\overline{y}$ must be equal; the second clause requires that the fractional parts of $\overline{x}$ and $\overline{y}$ must be ordered in the same way; the third requires that some fractional parts of $\overline{x}$ are 0 if and only if the corresponding fractional parts of $\overline{y}$ are 0.

The following is an important property of $\doteq$, saying that equivalent points —which must be in the same region—can always reach the same regions by delays.

Lemma 1 *Whenever $\overline{x} \doteq \overline{y}$, then for all $d \in \mathcal{R}_+$, $\overline{x} + d \doteq \overline{y} + e$ for some $e \in \mathcal{R}_+$.*

Proof: It is given in the full version of the paper [LW93]. □

We intend to establish that for any integer parallel process $\overline{P}$, a time region denotes an equivalence class of states $\overline{P}^{|\overline{x}|}$ [14] in terms of $\dot{\sim}$. Thus, two states in a time region should agree on what actions they can perform and then reach the same regions; they should also be able to reach the same regions by delays.

Lemma 2 *For all $\overline{P} \in \mathcal{P}_N$, $d \in \mathcal{R}_+$ and $a \in \mathcal{A}ct$, whenever $\overline{x} \doteq \overline{y}$, then*

1. $\overline{P}^{\overline{x}} \xrightarrow{a} \overline{P'}^{\overline{x}'}$ *for some $\overline{P'}$ and $\overline{x}'$, implies $\overline{P}^{\overline{y}} \xrightarrow{a} \overline{P'}^{\overline{y}'}$ for some $\overline{y}' \doteq \overline{x}'$ and*
2. $\overline{P}^{\overline{x}} \xrightarrow{\epsilon(d)} \overline{P}^{\overline{x}+d}$ *implies $\overline{P}^{\overline{y}} \xrightarrow{\epsilon(e)} \overline{P}^{\overline{y}+e}$ and $\overline{x} + d \doteq \overline{y} + e$ for some $e \in \mathcal{R}_+$.*

Proof: It is given in the full version of the paper [LW93]. □

Now, we are ready to state the partition theorem, which asserts that the infinite state-space of integer processes can be divided into equivalence classes according to time regions. In fact, many of such classes belong to a large equivalence class and the number of such classes is finite.

Theorem 3 *(partition) Whenever $\overline{x} \doteq \overline{y}$, then $\overline{P}^{\overline{x}} \dot{\sim} \overline{P}^{\overline{y}}$ for all $\overline{P} \in \mathcal{P}_N$.*

Proof: By lemma 2, it should be obvious that the relation: $\mathcal{S} = \{(\overline{P}^{\overline{x}}, \overline{P}^{\overline{y}}) \mid \overline{x} \doteq \overline{y}, \overline{P} \in \mathcal{P}_N\}$ is a strong time abstracted bisimulation. □

In the next section, we want to find a representative state for each equivalence class and then construct a symbolic transition system in terms of the representative states. In order to do so, we need first find a representative point for each time region of $\mathcal{R}_+^n$ for a given n.

Let $\mathcal{N}$ denote the naturals. We define the set of grids with granularity $1/k$: $\mathcal{N}_k = \{m/k \mid m \in \mathcal{N}\}$ ranged over by g, h and the set of grid points with granularity $1/k$: $\mathcal{N}_k^n = \{\overline{r} \mid 1 \leq i \leq n, r_i \in \mathcal{N}_k\}$ ranged over by $\overline{r}, \overline{s}$. An obvious choice is to use the the grid points $\mathcal{N}_m^n$ as representative points for $\mathcal{R}_+^n$, for some fixed granularity $1/m$.

[14] $\overline{P}^{|\overline{x}|} = \{\overline{P}^{\overline{y}} \mid \overline{y} \doteq \overline{x}\}$.

We claim that the grid points with granularity $1/(n+1)$ are enough to represent the n-dimensional time points $\mathcal{R}^n_+$, that is.

Lemma 3 *For all $\overline{x} \in \mathcal{R}^n_+$, there exists $\overline{r} \in \mathcal{N}^n_{n+1}$ such that $\overline{x} \doteq \overline{r}$.*

Proof: It is given in the full version of the paper [LW93]. □

Clearly, the lemma above will hold for any granularity finer than $1/(n+1)$ such as $1/(n+2), 1/(n+3)$ etc. However, it doesn't hold for a granularity coarser than $1/(n+1)$. To see this, consider the case of $n = 2$: with the granularity $1/2$ one can not find a grid point representing $(1/3, 2/3)$.

Thus $1/(n+1)$ is the coarsest granularity allowing any time region in the n-dimensional time space to be represented up to $\doteq$. However, we need a slightly finer granularity (which is in fact $1/(n+2)$ as shown in the following lemma) in order for a region to reach all regions by grid-valued delays, which are reachable by real-valued delays. The following lemma will be heavily used in proving the decidability results.

Lemma 4 *For all $\overline{r} \in \mathcal{N}^n_{n+2}$ and all $d \in \mathcal{R}_+$, there exist $\overline{r'} \in \mathcal{N}^n_{n+2}$ and $g \in \mathcal{N}_{n+2}$ such that $\overline{r} \doteq \overline{r'}$ and $\overline{r} + d \doteq \overline{r'} + g$.*

Proof: It is given in the full version of the paper [LW93]. □

Note that $\overline{r'} + g \in \mathcal{N}^n_{n+2}$, which will prove an essential property for the applicability of our finitary, symbolic semantics to follow. Also, note that it is not always possible to choose $\overline{r'} = \overline{r}$. To see this, consider the case of $n = 2$, $\overline{r} = (3/4, 0)$ and $d = 1/8$. The only possible choices for g is 0 and 1/4. However in both cases we see that $\overline{r} + g \not\doteq \overline{r} + d$. However, taking $\overline{r'} = (1/2, 0)$ and $g = 1/4$ we obtain as desired $\overline{r'} \doteq \overline{r}$ and $\overline{r'} + g \doteq \overline{r} + d$.

$$\frac{}{a.P \xrightarrow{a}_k P} \qquad \frac{P \xrightarrow{a}_k P'}{\epsilon(0).P \xrightarrow{a}_k P'}$$

$$\frac{P \xrightarrow{a}_k P'}{P + Q \xrightarrow{a}_k P'} \qquad \frac{Q \xrightarrow{a}_k Q'}{P + Q \xrightarrow{a}_k Q'} \qquad \frac{P \xrightarrow{a}_k P'}{X \xrightarrow{a}_k P'} \; [X \overset{def}{=} P]$$

$$\frac{P \xrightarrow{a}_k P'}{P|Q \xrightarrow{a}_k P'|Q} \qquad \frac{Q \xrightarrow{a}_k Q'}{P|Q \xrightarrow{a}_k P|Q'} \qquad \frac{P \xrightarrow{a}_k P' \quad Q \xrightarrow{\bar{a}}_k Q'}{P|Q \xrightarrow{\tau}_k P'|Q'}$$

Table 4: Action Rules for k-Semantics.

$$\frac{}{\text{nil} \xrightarrow{\chi}_k \text{nil}} \qquad \frac{}{\epsilon(r + \frac{1}{k}).P \xrightarrow{\chi}_k \epsilon(r).P} \qquad \frac{P \xrightarrow{\chi}_k P'}{\epsilon(0).P \xrightarrow{\chi}_k P'}$$

$$\frac{}{\alpha.P \xrightarrow{\chi}_k \alpha.P} \qquad \frac{P \xrightarrow{\chi}_k P' \quad Q \xrightarrow{\chi}_k Q'}{P + Q \xrightarrow{\chi}_k P' + Q'} \qquad \frac{P \xrightarrow{\chi}_k P'}{X \xrightarrow{\chi}_k P'} \; [X \overset{def}{=} P]$$

$$\frac{P \xrightarrow{\chi}_k P' \quad Q \xrightarrow{\chi}_k Q'}{P|Q \xrightarrow{\chi}_k P'|Q'} \; [P|Q \overset{\tau}{\not\longrightarrow}_k]$$

Table 5: Delay Rules for k-Semantics.

5.2 Time–Step Semantics: Sampling

The timed semantics describes how a process will behave at every real-valued time point with arbitrarily fine precision. This introduces the infinite-stateness of timed processes.

In practice, the "sampling" technique is often used to analyze a system. Instead of doing experiment on the system under consideration at *every* time point, only certain *typical* time points are chosen to capture or approximate the full system behaviour. Based on this idea, we develop a time-step semantics called k-semantics relativized by the granularity $1/k$, which describes how a process shall behave in every $1/k$ units of time. To achieve finer precision, we can choose a finer granularity. However, the timed processes will be finite-state for any fixed granularity $1/k$. As we shall see latter it is possible to completely capture time abstracted equivalences by sampling with a sufficiently fine granularity. In fact, the granularity required turns out to be $1/(n+2)$ where n is the number of parallel components.

We present the inference rules for the k-semantics in two steps: rules for real actions in table 4 and rules for delays in table 5. Note that apart from the index k associated with the arrow, the action rules are the same as in table 1 and the delay rules are parameterized with k.

We claim that the processes $\mathcal{P}_N$ are finite-state w.r.t. the transition relation $\longrightarrow_k$ for any non-zero natural k. This can be established based on the following facts on processes:

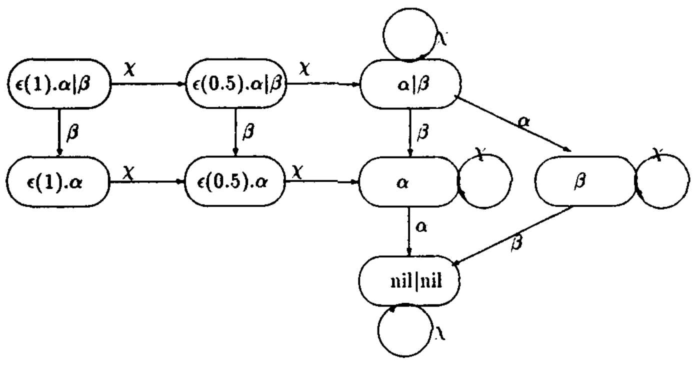


Figure 3: Transition Graph for $\epsilon(1).\alpha.\text{nil}|\beta.\text{nil}$ with Granularity 1/2.

- There is no infinite summation allowed;
- All recursive definitions are well-guarded;
- No parallel composition occurs within a recursion;
- Every process $\overline{P}$ must be *time-stable* after some maximal delay d_m, in the sense that $\overline{P}^{d_m} \xrightarrow{\epsilon(d)} \overline{P}^{d_m}$ [15] for all d or $\overline{P}^{d_m} \xrightarrow{\tau}$.

Example. In figure 3, we have a transition graph for $\epsilon(1).\alpha.\text{nil}|\beta.\text{nil}$ with granularity 1/2. For clarity, we have omitted nil in the graph. □

[15] Here, $\overline{P}^{d_m}$ stands for $P_1^{d_m}|\ldots|P_n^{d_m}$.

5.3 Symbolic Time Abstracted Bisimulation

We shall use a grid state $\overline{P}^{\overline{r}}$ to stand for an equivalence class of (real-valued) states. More precisely, we define:

$$\overline{P}^{[\overline{r}]} = \{\overline{P}^{\overline{x}} \mid \overline{x} \doteq \overline{r}\}$$

A class like $\overline{P}^{[\overline{r}]}$ shall be called as a *symbolic state* (or a symbolic process). We shall use R, S to denote symbolic states. Now, we define a symbolic transition relation $\longmapsto_k$ over symbolic states —(*classes* of real-valued states) as follows:

Definition 11 *For $\overline{r}, \overline{s} \in \mathcal{N}_k^n$ and $\overline{P}, \overline{Q} \in \mathcal{P}_N$.*

1. $\overline{P}^{[\overline{r}]} \overset{\chi}{\longmapsto}_k \overline{P}^{[\overline{s}]}$ *if* $\overline{P}^{\overline{r}} \xrightarrow{\chi}_k \overline{P}^{\overline{s}}$
2. $\overline{P}^{[\overline{r}]} \overset{a}{\longmapsto}_k \overline{Q}^{[\overline{s}]}$ *if* $\overline{P}^{\overline{r}} \xrightarrow{a}_k \overline{Q}^{\overline{s}}$ □

Intuitively, if there is a real transition in the k–semantics between two grid states, then there is a symbolic transition between the two equivalence classes they represent. Note that the definition above contains much more information than it looks. In fact, according to the definition, we can infer a symbolic transition like $\overline{P}^{[\overline{r}]} \overset{\chi}{\longmapsto}_k \overline{P}^{[\overline{s}]}$ whenever $\overline{P}^{\overline{r}'} \overset{\chi}{\longmapsto}_k \overline{P}^{\overline{s}'}$ for some grid states $\overline{r}' \doteq \overline{r}$ and $\overline{s}' \doteq \overline{s}$. However, the numbers of grid states in $\overline{P}^{[\overline{r}]}$ and $\overline{P}^{[\overline{s}]}$ are finite and hence, the symbolic processes are finite–state w.r.t the symbolic transition relation $\longmapsto_k$.

Like in defining time abstracted equivalence, we now abstract away from the symbolic time steps between symbolic states.

Definition 12

1. $R \overset{\epsilon}{\longrightarrow\!\!\circ}_k S$ *if* $R(\overset{\chi}{\longmapsto}_k)^* S$
2. $R \overset{a}{\longrightarrow\!\!\circ}_k S$ *if* $R \overset{\epsilon}{\longrightarrow\!\!\circ}_k \overset{a}{\longmapsto}_k \overset{\epsilon}{\longrightarrow\!\!\circ}_k S$ □

Definition 13 *(strong symbolic k–equivalence) A binary relation $\mathcal{S}$ over symbolic states is a strong k–simulation if $(R, S) \in \mathcal{S}$ implies that for all $a \in \mathcal{A}ct$ and χ,*

1. *Whenever $R \overset{a}{\longmapsto}_k R'$ then, for some S', $S \overset{a}{\longrightarrow\!\!\circ}_k S'$ and $(R', S') \in \mathcal{S}$*
2. *Whenever $R \overset{\chi}{\longmapsto}_k R'$ then, for some S', $S \overset{\epsilon}{\longrightarrow\!\!\circ}_k S'$ and $(R', S') \in \mathcal{S}$*

We call such a simulation $\mathcal{S}$ a strong k–bisimulation if it is symmetrical. The largest strong k–bisimulation is called strong symbolic k–equivalence. denoted $\overset{\circ}{\sim}_k$.

We define $\overline{P}^{\overline{r}} \overset{\circ}{\sim}_k \overline{Q}^{\overline{s}}$ whenever $\overline{P}^{[\overline{r}]} \overset{\circ}{\sim}_k \overline{Q}^{[\overline{s}]}$. □

Note that $\overset{\circ}{\sim}_k$ is decidable for any fixed k because of the finite–stateness of symbolic processes. The following is the main result of this section.

Theorem 4 *For all $\overline{P}, \overline{Q} \in \mathcal{P}_N$ and $\overline{r}, \overline{s} \in \mathcal{N}_{n+2}^n$, $\overline{P}^{\overline{r}} \overset{\circ}{\sim} \overline{Q}^{\overline{s}}$ if and only if $\overline{P}^{\overline{r}} \overset{\circ}{\sim}_{n+2} \overline{Q}^{\overline{s}}$, where n is the maximal number of components of $\overline{P}$ and $\overline{Q}$* [16].

Proof: For the direction:*Only If*, we show that the relation: $\mathcal{R} = \{(\overline{P}^{[\overline{r}]}, \overline{Q}^{[\overline{s}]}) \mid \overline{r}, \overline{s} \in \mathcal{N}_{n+2}^n, \overline{P}, \overline{Q} \in \mathcal{P}_N$ and $\overline{P}^{\overline{r}} \overset{\circ}{\sim} \overline{Q}^{\overline{s}} \mid \}$ is a strong symbolic $(n+2)$–bisimulation; for the other direction, we show

[16]Note that we can always extend $\overline{P}$ or $\overline{Q}$ with nil–processes as auxiliary components so that they own the same number of components.

that the relation: $\mathcal{S} = \{(\overline{P}^{\overline{r}}, \overline{Q}^{\overline{s}}) \mid \overline{r}, \overline{s} \in \mathcal{N}^n_{n+2},\ \overline{P}, \overline{Q} \in \mathcal{P}_N$ and $\overline{P}^{|\overline{r}|} \overset{\circ}{\sim}_{n+2} \overline{Q}^{|\overline{s}|} \mid \}$ is a strong time abstracted bisimulation up to $\overset{\bullet}{\sim}$. A complete proof is given in the full version of the paper [LW93]. □

We extend the results to weak time abstracted equivalence.

Definition 14

1. $R \overset{\epsilon}{\Longrightarrow}\!\circ_k S$ *if* $R(\overset{\chi}{\longmapsto}_k \cup \overset{\tau}{\longmapsto}_k)^* S$
2. $R \overset{\alpha}{\Longrightarrow}\!\circ_k S$ *if* $R \overset{\epsilon}{\Longrightarrow}\!\circ_k \overset{\alpha}{\longmapsto}_k \overset{\epsilon}{\Longrightarrow}\!\circ_k S$ □

Definition 15 *(weak symbolic k–equivalence) A binary relation $\mathcal{S}$ over symbolic states is a weak k–simulation if $(R, S) \in \mathcal{S}$ implies that for all $\alpha \in \mathcal{A}ct - \{\tau\}$ and $\theta \in \{\chi, \tau\}$,*

1. *Whenever $R \overset{\alpha}{\longmapsto}_k R'$ then, for some S', $S \overset{\alpha}{\Longrightarrow}\!\circ_k S'$ and $(R', S') \in \mathcal{S}$*
2. *Whenever $R \overset{\theta}{\longmapsto}_k R'$ then, for some S', $S \overset{\epsilon}{\Longrightarrow}\!\circ_k S'$ and $(R', S') \in \mathcal{S}$*

We call such a simulation $\mathcal{S}$ a weak k–bisimulation if it is symmetrical. The largest weak k–bisimulation is called weak symbolic k–equivalence, denoted $\overset{\circ}{\approx}_k$.

We define $\overline{P}^{\overline{r}} \overset{\circ}{\approx}_k \overline{Q}^{\overline{s}}$ whenever $\overline{P}^{|\overline{r}|} \overset{\circ}{\approx}_k \overline{Q}^{|\overline{s}|}$. □

Finally, we achieve the decidability result for weak time abstracted equivalence.

Theorem 5 *For all $\overline{P}, \overline{Q} \in \mathcal{P}_N$ and $\overline{r}, \overline{s} \in \mathcal{N}^n_{n+2}$, $\overline{P}^{\overline{r}} \overset{\bullet}{\approx} \overline{Q}^{\overline{s}}$ if and only if $\overline{P}^{\overline{r}} \overset{\circ}{\approx}_{n+2} \overline{Q}^{\overline{s}}$, where n is the maximal number of components of $\overline{P}$ and $\overline{Q}$.*

Proof: It is similar to the proof for theorem 4. A complete proof is given in the full version of the paper [LW93]. □

6 Conclusion

In this paper we have introduced a notion of *time-abstracting* bisimulation equivalence.

As the first main result of this paper, we have demonstrated that two processes are interchangeable in any context up to time–abstracted equivalence precisely when they are timed equivalent. Thus, by resorting to *implicit* specifications — i.e. specifications of a system in contexts — we may reveal *all* timing properties of a system.

As our second main result we have established the decidab of the time–abstracted equivalence by providing a finite–state and symbolic yet structured, operational semantics of processes. The symbolic semantics can be seen as sampling a process with a given frequency; we prove that sufficiently frequent sampling — $1/(n+2)$ where n is the number of parallel components — yields a symbolic equivalence completely capturing the time–abstracted equivalence.

The minimization algorithm presented in [ACH92] can be seen to minimize timed graphs [AD90] with respect to time–abstract bisimulation equivalence even though no notion of time–abstracted bisimulation is given in the paper. Despite the purpose of the minimization effort being to obtain more efficient model–checking algorithms with respect to a real–time temporal logic, we believe that the results of [ACH92] can provide an alternative method for deciding time–abstracted equivalences. However, we are of the opinion that our approach is simpler (certainly from a process algebraic point of view) as it is based directly on a traditional structured, operational semantics.

Recently, we have completed a prototype implementation of a tool-set for timed and time-abstracted bisimulation equivalences based on the methods described in this paper and in [Č92]. In addition the tool-set applies the efficient, local checking technique described in [La92], thus avoiding to explore the state-space more than necessary. We hope to report upon this work in a forthcoming paper [CGL92].

References

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In Proceedings of the Fifth IEEE Symposium on Logic in Computer Science, 1990.

[ACH92] R. Alur, C Courcoubetis, N. Halbwachs, D. Dill, H. Wong-Toi. Minimization of Timed Transition Systems. CONCUR92, LNCS 630, 1992.

[AD90] Rajeev Alur and David Dill. Automata for modelling real-time systems. In Automata, Languages and Programming: Proceedings of the 17th ICALP, LNCS 443. Springer-Verlag, 1990.

[BB89] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. Technical Report P8916, University of Amsterdam, 1989.

[CGL92] K. Cerans, J.C. Godskesen, K.G. Larsen. JANUS: a tool for analyzing real-time processes, Aalborg University, (in preparation). 1992.

[Che91b] Liang Chen. An interleaving model for real-time systems. Technical report, LFCS, University of Edinburgh, Scotland, 1991. Preliminary version.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. Technical report, LFCS, University of Edinburgh, Scotland, 1989.

[DS89] Jim Davis and Steve Schneider. An introduction to timed CSP. Technical Report PRG-75, Oxford University Computing Laboratory, 1989.

[FK91] W.J. Fokkink and S. Klusener. Real time algebra with prefixed integration. Technical report, CWI, Amsterdam, 1991.

[GL92] Jens Chr. Godskesen and Kim G. Larsen. Real-time calculi and expansion theorems. In Twelfth Conference on the FST and TCS, Lecture Notes in Computer Science. Springer-Verlag, December 1992. To appear.

[HLW91] Uno Holmer, Kim Larsen, and Yi Wang. Deciding properties of regular timed processes. In the proceedings of CAV91, volume 575 of Lecture Notes in Computer Science. Springer-Verlag, 1991.

[HNJ92] T. Henzinger, X. Nicollin, J. Sifakis. and J. Voiron. Symbolic Model Checking for Real-Time Systems. Proceedings of the 7th IEEE Symposium on Logic in Computer Science, 1992.

[Hoa85] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.

[HR91] Matthew Hennessy and Tim Regan. A process algebra for timed systems. Technical Report 5/91, University of Sussex, 1991.

[Jef91b] Allan Jeffrey. A linear time process algebra. In the proceedings of CAV91, volume 575 of Lecture Notes in Computer Science. Springer-Verlag. July 1991.

[JGZ89] K.G. Larsen J.C. Godskesen and M. Zeeberg. Tav — tools for automatic verification — users manual. Technical Report R 89-19. Department of Mathematics and Computer Science, Aalborg University, 1989. Presented at workshop on Automatic Methods for Finite State Systems, Grenoble, France, Juni 1989.

[KS90] P.C. Kanellakis and S.A. Smolka, CCS Expressions, finite state processes, and three problems of equivalence. Information and Control Vol 86, 1990.

[La92] Kim G. Larsen. Efficient Local Correctness Checking In Proceedings of CAV92, Montreal, Canada.

[LW93] Kim G. Larsen and Wang Yi. Time Abstracted Bisimulation: Implicit Specifications and Decidability. Tech Report, Department of Computer Systems, Uppsala University, Sweden, 1993.

[Mil89] Robin Milner. Communication and Concurrency. Series in Computer Science. Prentice-Hall International, 1989.

[MT90] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In CONCUR'90, volume 458 of Lecture Notes in Computer Science. Springer-Verlag, 1990.

[NRSV90] Thomas A. Henzinger, X. Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems IEEE Proc. 7th Sym. Logic in Computer Science, California, June, 1992.

[NRSV90] X. Nicollin, J.-L. Richier, Joseph Sifakis, and J. Voiron. ATP: an algebra for timed processes. In Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Gallilee, Israel. April 1990.

[NSY91] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. From ATP to timed graphs and hybrid systems. In Real-Time: Theory in Practice, volume 600 of Lecture Notes in Computer Science. Springer-Verlag, 1991.

[RH92] Rajeev Alur and T. Henzinger. Logics and Models of Real Time: a Survey. REX Workshop on Real Time: Theory and Practice, 1991.

[RR86] G.M. Reed and A.W. Roscoe, A Timed Model for Communicating Sequential Processes. LNCS No. 226, 1986.

[PT87] Paige and Tarjan. Three partition refinement algorithms. SIAM Journal of Computing, 16(6), 1987.

[Sch91] Steve Schneider. An operational semantics for timed CSP. April 1991.

[Č92] Kārlis Čerāns. Decidability of bisimulation equivalences for processes with parallel timers. To appear in Proceedings of CAV'92, 1992.

[SV89] R. De Simone and D. Vergamini. Aboard AUTO. Technical Report 111, INRIA, Sofia-Antipolis, 1989.

[Wan90] Yi Wang. Real-time behaviour of asynchronous agents. In CONCUR '90, volume 458 of Lecture Notes in Computer Science. Springer-Verlag, 1990.

[Wan91a] Yi Wang. A Calculus of Real Time Systems. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1991.

[Wan91b] Yi Wang. CCS + time = an interleaving model for real time systems. In ICALP91, LNCS 510. Springer-Verlag, 1991.

Timewise Refinement for Communicating Processes

Steve Schneider

Oxford University Computing Laboratory
11 Keble Road, Oxford OX1 3QD, UK
sas@comlab.ox.ac.uk

Abstract

A theory of timewise refinement is presented. This allows the translation of specifications and proofs of correctness between semantic models, permitting each stage in the verification of a system to take place at the appropriate level of abstraction. The theory is presented within the context of CSP. A denotational characterisation is given in terms of relations between behaviours at different levels of abstraction, and various properties for the preservation of refinement through parallel composition are discussed. An operational characterisation is also given in terms of timed and untimed tests, and observed to coincide with the denotational characterisation.

1 Introduction and general theory

Verification of time-critical systems requires the application of necessarily complicated and detailed techniques, reflecting the complex nature of such systems and the detailed and precise requirements upon them. Yet it is often the case that a significant proportion of specifications on timed systems will be concerned with logical behaviour rather than timing behaviour, and proposed implementations will often be correct with respect to these parts of the specification by virtue of their functional properties, independently of their timing properties.

This paper proposes a way of avoiding the need to carry out the entire analysis of a system at the most complicated level. We investigate refinement relations between processes in different models of the CSP hierarchy [Ree88]. It is important to identify which properties (such as deadlock-freedom or determinism) can be translated between models, since only for such properties can verifications be mapped up the hierarchy. The more mature and powerful techniques available in the more abstract models, such as model-checking, algebraic techniques, theories for deadlock-freedom, and simply more abstract reasoning, may then be used in conjunction with the more cumbersome and difficult methods required for the more detailed aspects of the verification.

This paper investigates two refinement relations in detail, both from an untimed to a timed model of CSP. The first untimed model is concerned only with safety specification. The second is also able to address fairness and (untimed) liveness requirements. The relationships between these two models and the timed infinite model for timed CSP [Sch92, MRS92] will be presented.

General framework

We model a process in terms of the observations that may be made of it, which may also be considered as the behaviours it may exhibit. If we have a set $\mathcal{O}$ of all possible observations, then a process is identified with a subset of $\mathcal{O}$. The corresponding semantic model $\mathcal{M}$ consists of those subsets of $\mathcal{O}$ that may be considered to represent some process. A set of healthiness conditions, or axioms, for $\mathcal{M}$ are used to characterise these subsets of $\mathcal{O}$.

A programming language $\mathcal{L}$ is used for describing processes. Each program in $\mathcal{L}$ is associated with an element of $\mathcal{M}$, called its semantics or meaning, by means of a semantic function $\mathcal{F} : \mathcal{L} \rightarrow \mathcal{M}$. This function is compositional, in the sense that the process associated with any particular program depends only on the processes associated with its components, and how these components are composed.

Specifications are given in terms of predicates upon observations. A process P meets a specification S if all of its observations meet the corresponding predicate. In this case, we write P **sat** S.

$$P \textbf{ sat } S \quad \Leftrightarrow \quad \forall o : \mathcal{O} \bullet (o \in \mathcal{F}[\![P]\!]) \Rightarrow S$$

A program meets a specification when its semantics meets it.

A process P_1 is refined by another process P_2 when every possible behaviour of P_2 is also a possible behaviour of P_1. In this case we write $P_1 \sqsubseteq P_2$, and consider P_2 to be more deterministic than P_1, since P_1 can do everything P_2 can, and possibly more. If $P_1 \sqsubseteq P_2$, and P_1 **sat** S, then it follows that P_2 **sat** S; refining a process maintains correctness with respect to specifications. This approach also allows processes P to act as specifications: P_2 meets specification P if it is a refinement of P.

The nature of the semantic model is dependent upon the nature of the observation set $\mathcal{O}$. Observations describe executions of systems at a particular level of abstraction. For example, the use of traces as observations provides only the sequences of events that a system may perform; refusals provide information about contexts in which a system may deadlock; and timed traces also provide information about the times at which events may occur. The use of a particular kind of observation depends on the kind of specification we wish to consider, and the level of abstraction at which we need to consider the system in order to establish correctness.

If we have two different semantic models $\mathcal{M}_A$ and $\mathcal{M}_C$, based upon different sets of observations $\mathcal{O}_A$ and $\mathcal{O}_C$ respectively, then we are able to analyse systems at two different levels of abstraction; and we may ask when a description at the level of $\mathcal{M}_C$ refines a description at the level of $\mathcal{M}_A$.

We firstly employ a relation ${}_A\mathcal{R}_C \subseteq \mathcal{O}_A \times \mathcal{O}_C$ to relate observations at the different levels of abstraction. The intention is that if $b_A \; {}_A\mathcal{R}_C \; b_C$ then b_A and b_C are both descriptions, at different levels of abstraction, of the same execution; or alternatively, that b_A is an abstract description of b_C. There is of course no guarantee that the relation ${}_A\mathcal{R}_C$ captures a useful relationship between behaviours; this depends upon the intended application of the theory. The refinement relation between processes from $\mathcal{M}_A$ and processes from $\mathcal{M}_C$ with respect to the relation ${}_A\mathcal{R}_C$ is then given by the following definition.

Definition 1.1

$$P_A \sqsubseteq_{{}_A\mathcal{R}_C} P_C \quad \Leftrightarrow \quad {}_A\mathcal{R}_C{}^{-1}\,(P_C) \subseteq P_A$$

□

We consider P_C to refine P_A if P_A admits every abstract view of every behaviour of P_C.

Refinement may be promoted to programs:

Definition 1.2

$$Q_A \sqsubseteq_{{}_A\mathcal{R}_C} Q_C \quad \Leftrightarrow \quad \mathcal{F}_A[\![Q_A]\!] \sqsubseteq_{{}_A\mathcal{R}_C} \mathcal{F}_C[\![Q_C]\!]$$

□

A verification of Q_A may be translated into a verification of Q_C by use of the following inference rule, whose soundness follows from the definitions above:

$$\frac{\begin{array}{l} Q_A \ \mathbf{sat}_A \ S_A \\ Q_A \sqsubseteq_{{}_A\mathcal{R}_C} Q_C \end{array}}{Q_C \ \mathbf{sat}_C \ \forall\, b_A \bullet (b_A \ {}_A\mathcal{R}_C \ b_C \Rightarrow S_A)}$$

We may thus consider the specification $\forall\, b_A \bullet (b_A \ {}_A\mathcal{R}_C \ b_C \Rightarrow S_A)$ to be the translation of S_A. (Here we use b_A as the free variable in S_A ranging over behaviours in $\mathcal{O}_A$ in the **sat** relation; and b_C similarly.)

Observe that if $\mathcal{M}_A = \mathcal{M}_C$, and the relation ${}_A\mathcal{R}_C$ is the identity relation, then the refinement relation $\sqsubseteq_{{}_A\mathcal{R}_C}$ is simply refinement under the non-deterministic ordering; and the rule states that if a program meets a specification, then so too does any refinement of it.

Definition 1.3 A refinement relation ${}_A\mathcal{R}_C$ is said to be *complete* if whenever the conclusion of the above rule holds, then there is some Q_A for which the two antecedents hold. □

Lemma 1.4 The relation ${}_A\mathcal{R}_C$ is complete if and only if ${}_A\mathcal{R}_C{}^{-1}\,(Q)$ is an element of $\mathcal{M}_A$ whenever Q is an element of $\mathcal{M}_C$ □

Proof Assume that ${}_A\mathcal{R}_C{}^{-1}\,(Q)$ is an element of $\mathcal{M}_A$ whenever Q is an element of $\mathcal{M}_C$. Consider the conclusion $Q_C \ \mathbf{sat}_C \ \forall\, b_A \bullet (b_A \ {}_A\mathcal{R}_C \ b_C \Rightarrow S_A)$. Then it follows that ${}_A\mathcal{R}_C{}^{-1}\,(Q_C)$ **sat** S_A, and also ${}_A\mathcal{R}_C{}^{-1}\,(Q_C) \sqsubseteq_{{}_A\mathcal{R}_C} Q_C$; thus the rule is complete.

If on the other hand there is some Q for which ${}_A\mathcal{R}_C{}^{-1}\ (Q)$ is not a process, then given any $P \sqsubseteq_{{}_A\mathcal{R}_C} Q$, P will not meet the specification $b_A \in {}_A\mathcal{R}_C{}^{-1}\ (Q)$. This is because ${}_A\mathcal{R}_C{}^{-1}\ (Q)$ is a subset of P, so P will contain some behaviour b_A which breaks the specification. Yet the process Q meets the timed version of that specification. $\square$

Completeness of the refinement relation allows translation in the following direction:

$$\frac{Q_C \ \mathbf{sat}\ \forall\, b_A \bullet (b_A \ {}_A\mathcal{R}_C\ b_C \Rightarrow S_A)}{{}_A\mathcal{R}_C{}^{-1}\ (Q_C)\ \mathbf{sat}\ S_A}$$

The following weakening of the conclusion to this rule is often useful:

$$\exists\, Q_A \bullet Q_A \sqsubseteq_{{}_A\mathcal{R}_C} Q_C \wedge Q_A \ \mathbf{sat}\ S_A$$

The discussion so far has all been on the semantic level. In order to prove that one program refines another using the above theory, it is necessary to calculate the semantics of each program, and then check that the refinement relation holds between them. Compositionality often plays a critical role in breaking down verification obligations on large systems to manageable components. We aim to exploit the compositional nature of program semantics, and so we investigate when refinements established between components of abstract and concrete systems mean that the entire abstract system is refined by the entire concrete system.

A context $C_A(X)$ is said to be refined by another context $C_C(Y)$ if there is a relationship $C_A(Q_A) \sqsubseteq_{{}_A\mathcal{R}_C} C_C(Q_C)$ whenever $Q_A \sqsubseteq_{{}_A\mathcal{R}_C} Q_C$. Our aim is to find relationships concerning the operators of the language $\mathcal{L}$ so that refinement between contexts and programs may be established without resorting to explicit calculation of their semantics, by reasoning at the syntactic level.

A syntactic operator $\oplus'$ of $\mathcal{L}$ is a refinement of operator $\oplus$ if combinations of refinements of processes refine combinations of the processes:

Definition 1.5 An operator $\oplus'$ of the language $\mathcal{L}$ with arity α refines operator $\oplus$ with the same arity, if

$$(\forall\, i < \alpha \bullet P_i \sqsubseteq_{{}_A\mathcal{R}_C} Q_i) \quad \Rightarrow \quad \oplus\langle P_i \mid i < \alpha\rangle \sqsubseteq_{{}_A\mathcal{R}_C} \oplus'\langle Q_i \mid i < \alpha\rangle$$

$\square$

The framework presented above is very well-known. But to go further, we must focus on particular models, languages, and refinement relations. We are interested in conditions for refinement relations to exist between programs (which will vary from relation to relation), and how specifications translate between models.

In this paper we are concerned with mapping results up the hierarchy of untimed and timed models for CSP. We will concentrate on two relations in detail, both from an untimed to a timed model: one from the untimed traces model, which is used for analysis of safety properties; and one from the untimed infinite traces model [Ros88] (which also contains failures and divergences), a more sophisticated untimed model supporting consideration of liveness issues.

2 Communicating Sequential Processes

Syntax

The language of Communicating Sequential is given by the following Backus-Naur form:

$$\begin{aligned} P \; ::= \; & \mathit{Chaos} \mid \mathit{Stop} \mid \mathit{Skip} \mid P \,;P \mid P \overset{t}{\rhd} P \mid P \,\Box\, P \mid a : A \longrightarrow P_a \mid \sqcap_{i \in I} P_i \\ & \mid P \;{}_A\|_A\; P \mid P \interleave P \mid P \setminus A \mid f(P) \mid f^{-1}(P) \mid X \mid \mu X \circ P \end{aligned}$$

Here the set A is a subset of the universal set of events Σ; I is a subset of the set of indexes $\mathcal{I}$; f is a function $\Sigma \rightarrow \Sigma$; X is drawn from the set of process variables VAR; and t is drawn from the set of times, the non-negative real numbers. The *programs* of the language are those terms with no free process variables.

The constructors given by the BNF above represent respectively: the most non-deterministic process; deadlock; successful termination; sequential composition; timeout; external choice; prefix choice; non-deterministic choice; synchronised parallel; interleaving parallel; interface abstraction or hiding; two forms of alphabet renaming; process variable; and recursion. For a more detailed discussion of the language, the reader is referred to [DaS92b].

The following abbreviations often prove useful:

$$\begin{aligned} \mathit{Wait}\; t &= \mathit{Stop} \overset{t}{\rhd} \mathit{Skip} \\ b \longrightarrow P &= a : \{b\} \longrightarrow P(a) \;\text{ where } P(b) = P \\ b \xrightarrow{t} P &= b \longrightarrow \mathit{Wait}\; t \,; P \\ P \,\|\, Q &= P \;{}_\Sigma\|_\Sigma\; Q \\ P \sqcap Q &= \sqcap_{i \in \{1,2\}} P_i \;\text{ where } P_1 = P \text{ and } P_2 = Q \end{aligned}$$

When modelling timed processes, we must take care to ensure that recursive calls are time guarded, so that a minimum delay must elapse between successive recursive calls. This is achieved by ensuring that every instance of the process variable of a recursive term should appear in the right-hand argument of a non-zero timeout. A set of rules for determining when a term is time guarded is detailed in [DaS92a].

Notation

The set Σ is the set of visible events. Variables a, b, c are taken to range over Σ. The variables t and u range over $\mathbf{R}^+$, the set of non-negative real numbers. Variable tr ranges over Σ^*, finite sequences of events from Σ; u ranges over Σ^ω, infinite sequences of events from Σ; $X \subseteq \Sigma$ denotes a set of events; s ranges over $(\mathbf{R}^+ \times \Sigma)^\omega$, the (finite and infinite) sequences of timed visible events; we use $\aleph \subseteq \mathbf{R}^+ \times \Sigma$ to represent a timed refusal, a set of timed visible events.

We use the following operations on (untimed and timed) sequences of events: $\#w$ is the length of the sequence w; $w_1 \frown w_2$ denotes the concatenation of w_1 and w_2. The notation $w_1 \preceq w_2$ means that w_1 is a subsequence of w_2.

The following projections are defined on untimed sequences by list comprehension:

$$\begin{aligned}
tr \restriction A &= \langle a \mid a \leftarrow tr, a \in A\rangle \\
tr \setminus A &= \langle a \mid a \leftarrow tr, a \notin A\rangle \\
tr \downarrow c &= \langle x \mid a \leftarrow tr, a = c.x\rangle \\
\sigma(tr) &= \{a \mid tr \restriction \{a\} \neq \langle\rangle\}
\end{aligned}$$

For timed sequences, we define the beginning and end of a sequence in the following way: $begin(\langle(t,\mu)\rangle \frown s) = t$, $end(s \frown \langle(t,\mu)\rangle = t$, and for convenience $begin(\langle\rangle) = \infty$ and $end(\langle\rangle) = 0$. The following projections on timed sequences are defined by list comprehension:

$$\begin{aligned}
s \lhd t &= \langle(u,a) \mid (u,a) \leftarrow s, u \leq t\rangle \\
s \blacktriangleleft t &= \langle(u,a) \mid (u,a) \leftarrow s, u < t\rangle \\
s \uparrow t &= \langle(u,a) \mid (u,a) \leftarrow s, u = t\rangle \\
s \restriction A &= \langle(u,a) \mid (u,a) \leftarrow s, a \in A\rangle \\
s \setminus A &= \langle(u,a) \mid (u,a) \leftarrow s, a \notin A\rangle \\
s - t &= \langle(u-t,a) \mid (u,a) \leftarrow s, u \geq t\rangle \\
strip(s) &= \langle a \mid (u,a) \leftarrow s\rangle \\
\sigma(s) &= \{a \mid s \restriction \{a\} \neq \langle\rangle\}
\end{aligned}$$

We also define a number of projections on timed refusal sets:

$$\begin{aligned}
\aleph \blacktriangleleft t &= \{(u,a) \mid (u,a) \in \aleph, u < t\} \\
\aleph \rhd t &= \{(u,a) \mid (u,a) \in \aleph, u \geq t\} \\
\aleph \restriction A &= \{(u,a) \mid (u,a) \in \aleph, a \in A\} \\
\aleph - t &= \{(u-t,a) \mid (u,a) \in \aleph, u \geq t\} \\
\sigma(\aleph) &= \{a \mid (u,a) \in \aleph\} \\
end(\aleph) &= sup\{u \mid (u,a) \in \aleph\}
\end{aligned}$$

We will use $(s,\aleph) - t$ as an abbreviation for $(s - t, \aleph - t)$, and $end(s,\aleph)$ for $max\{end(s), end(\aleph)\}$.

Semantic models

The hierarchy of models presented in [Ree88] supports reasoning at a number of levels of abstraction, allowing aspects of behaviour dependent upon refusal information, stability information, or timing information to be included as required. In addition to Reed's hierarchy of models, we have the infinite timed model $\mathcal{M}_{TI}$, presented in [Sch92] and [MRS92]; and the untimed infinite traces model $\mathcal{M}_{UI}$ of [Ros88], which is an extension of the failures-divergences model of [BrR85]. In this paper we will focus on the three models which yield the most general results concerning refinement: the untimed traces model $\mathcal{M}_{UT}$, and the two infinite models. These three models are presented in full, together with their corresponding semantic functions, in Appendix A.

$$\begin{tikzcd}
\mathcal{M}_{TI} \arrow[ddd, dashed] \arrow[dr] & & & \mathcal{M}_{TFS} \arrow[dddlll] \arrow[dd] \arrow[ddr] & & \\
& \mathcal{M}_{UI} \arrow[dr] & & & & \\
& & \mathcal{M}_{UFD} \arrow[r, dashed, leftrightarrow] \arrow[d, dashed] & \mathcal{M}_{UFS} \arrow[dl] \arrow[dr] & \mathcal{M}_{UFTS} \arrow[l] \arrow[dr] & \\
\mathcal{M}_{TF} \arrow[rr] \arrow[ddrrr] & & \mathcal{M}_{UF} \arrow[dr] & & \mathcal{M}_{US} \arrow[dl] & \mathcal{M}_{TS} \arrow[l] \arrow[ddll] \\
& & & \mathcal{M}_{UT} & & \\
& & & \mathcal{M}_{TT} \arrow[u] & &
\end{tikzcd}$$

Figure 1: Reed's hierarchy and additional models

The untimed traces model

Observations in the model $\mathcal{M}_{UT}$ are simply finite sequences of events, or *traces*. A trace of a system is a record of the events performed during some (partial) execution of the system. Thus the observation set $\mathcal{O}_{UT}$ is defined to be Σ^*, where Σ is the universal set of events.

The model $\mathcal{M}_{UT}$ is the set of nonempty prefix closed subsets S of $\mathcal{O}_{UT}$.

The untimed infinite traces model

This model is first described in [Ros88]. In other presentations, processes consist of three components, modelling the three kinds of observation that may be made: a failure set $F \subseteq \Sigma^* \times \mathbf{P}(\Sigma)$; a divergence set $D \subseteq \Sigma^*$; and an infinite traces set $I \subseteq \Sigma^\omega$. A divergence tr is a sequence of events such that after some prefix of tr the system may perform an infinite sequence of internal actions. A failure (tr, X) is an observation of a system if either the sequence of external events tr may be observed during an execution, after which no further internal progress may be made and the process refuses to engage in any event from the set X; or else tr is a divergent trace. An infinite trace u is an infinite sequence of actions such that either the system may perform the whole trace during a single execution, or else some prefix of it is a divergent trace.

For the sake of uniformity within this presentation, we consider a process to consist of a single set S of pairs, where the first component is a label from the set $\{f, d, i\}$, and the second component is a behaviour from the corresponding behaviour set. Thus S is a subset of

$$\{f\} \times (\Sigma^* \times \mathbf{P}(\Sigma)) \cup \{d\} \times \Sigma^* \cup \{i\} \times \Sigma^\omega$$

The timed infinite traces model

In this model, the times at which events are performed and refused are recorded. This model assumes that systems are finitely variable: an infinite sequence of internal and external actions may not be performed in a finite time. Thus the only infinite traces that may be observed must take infinitely long to occur. Furthermore, since a change in the set of events made available to the environment is considered to correspond to an internal action, this model needs to consider only those refusal sets which contain finitely many changes in any finite interval.

The set of traces $T\Sigma^\omega_\leq$ and refusal sets $IRSET$ are adequate for capturing all possible observations of finitely variable systems:

$$\begin{aligned}
T\Sigma^\omega_\leq &= \{s \in (\mathbf{R}^+ \times \Sigma)^\omega \mid \langle (t_1, a_1), (t_2, a_2)\rangle \preceq s \Rightarrow t_1 \leq t_2 \\
&\qquad\qquad\qquad\qquad \wedge \#s = \infty \Rightarrow end(s) = \infty\} \\
RTOK &= \{[b, e) \times A \mid 0 \leq b < e < \infty \wedge A \subseteq \Sigma\} \\
RSET &= \{\bigcup R \mid R \subseteq RTOK \wedge R \text{ is finite}\} \\
IRSET &= \{\bigcup R \mid R \subseteq RTOK \wedge \forall t \bullet (\bigcup R) \lhd t \in RSET\}
\end{aligned}$$

Behaviours consist of (trace,refusal) pairs. In contrast to the untimed case, the refusal is observed during the occurrence of the trace, rather than simply afterwards.

For example, the behaviour $(\langle (3, a), (3, d), (8, b) \rangle, [0, 20) \times \{c\})$ is a record indicating that the process was observed to refuse event c beginning at time 0, that while it was continuing to do so, it performed event a and then d at time 3, and then event b at time 8. Finally, the observer stopped watching c being refused at time 20.

As usual, a process consists of the set of possible behaviours that may be observed of it. The model is presented in full in [Sch92, MRS92].

Example

Define the program AB as follows:

$$AB \;=\; \mu X \circ (a \xrightarrow{3} X \overset{5}{\rhd} b \longrightarrow Stop)$$

Then $\mathcal{F}_{UT}[\![AB]\!]$ contains both the traces $\langle a \rangle$ and $\langle a, a, b \rangle$, but not trace $\langle b, a \rangle$. The untimed infinite semantics $\mathcal{F}_{UI}[\![AB]\!]$ contains failures $(f, (\langle a, a \rangle, \{a\}))$ and $(f, (\langle a, b \rangle, \{b, c\}))$, but not $(f, \langle a \rangle, \{b\})$ or $(f, \langle b, a \rangle, \{\})$; it contains the infinite trace $(i, \langle a, a, a, \ldots \rangle)$; and it contains no divergences.

The timed behaviours $\mathcal{F}_{TI}[\![AB]\!]$ include $(\langle (2, a), (9, a) \rangle, [0, 6) \times \{b\})$: the process may perform event a at time 2, and again at time 9, while refusing to perform b between times 0 and 6. The behaviour $(\langle\rangle, [0, 5) \times \{b\} \cup [5, \infty) \times \{a\})$ is also possible: if no external events are performed, then b will be refused for the first 5 units of time, after which the timeout will occur, and a will be refused thereafter. Neither $(\langle (2, a) \rangle, [0, 1) \times \{a\})$ nor $(\langle\rangle, [0, 10) \times \{b\})$ are possible timed behaviours of $\mathcal{F}_{UI}[\![AB]\!]$.

3 Timed refinement

3.1 Trace refinement

We consider an untimed trace to be an abstract description of a timed failure if the trace corresponds to the sequence of events in the timed trace. We thus define the refinement relation between untimed traces $\mathcal{O}_{UT}$ and timed failures $\mathcal{O}_{TI}$ as follows:

$$tr \;{}_{UT}\mathcal{R}_{TI}\; (u, \aleph) \;\;\Leftrightarrow\;\; tr = strip(u)$$

For a timed trace s, the sequence $strip(s)$ is the trace s with the times removed from the events.

Theorem 3.1 This refinement relation is complete □

Proof By Lemma 1.4 it is enough to show that $P =_{UT}\mathcal{R}_{TI}{}^{-1}\ (Q)$ is a well-defined process for any timed process Q. But ${}_{UT}\mathcal{R}_{TI}{}^{-1}\ (Q) = \{strip(s) \mid (s,\aleph) \in Q\}$, and this set is clearly non-empty (since Q is) and prefix closed (since Q is), and hence it is a well-defined process, meeting the definition in Appendix A. □

It turns out that all of the CSP operators preserve this refinement relation:

Theorem 3.2 Given any CSP operator $\oplus$, and two vectors of processes of length $arity(\oplus)$, $\underline{P} \sqsubseteq_{UT\mathcal{R}_{TI}} \underline{Q}$,

$$\oplus(\underline{P}) \sqsubseteq_{UT\mathcal{R}_{TI}} \oplus(\underline{Q})$$

□

Proof By an analysis of the timed and untimed semantics (given in Appendix A) of each CSP operator in turn. □

Corollary 3.3 For any program P, $P \sqsubseteq_{UT\mathcal{R}_{TI}} P$ □

$$\begin{tikzcd}
P \arrow[d] \arrow[dr] & \\
\mathcal{F}_{UT}[\![P]\!] \arrow[r, leftrightarrow, "\sqsubseteq_{UT\mathcal{R}_{TI}}"] & \mathcal{F}_{TI}[\![P]\!]
\end{tikzcd}$$

The payoff from this result is that any trace specification may be verified of a CSP program in the untimed traces model, and it follows immediately that its translation into the timed model will hold for the same program on its more complicated semantics. Also $Skip \sqsubseteq_{UT\mathcal{R}_{TI}} \sqcap_{t \in I} Wait\ t$ for any set of times I, so arbitrary delays can be introduced into programs while still preserving refinement, since if $P \sqsubseteq_{UT\mathcal{R}_{TI}} Q$, then it follows that $Skip\ ; P = P \sqsubseteq_{UT\mathcal{R}_{TI}} \sqcap_{t \in I} Wait\ t\ ; Q$. Thus an untimed verification can be carried out and delays inserted subsequently.

The translation of a specification S on traces, (with free variable tr) will be

$$\forall\, tr \in \Sigma^* \bullet (tr\ {}_{UT}\mathcal{R}_{TI}\ (s,\aleph) \Rightarrow S)$$

If S is admissible (i.e. $(\forall\, tr < u \bullet S) \Rightarrow S[u/tr]$) then this is equivalent on processes to the specification $S[strip(s)/tr]$. Thus admissible specifications may be translated to timed specifications by a simple substitution of the free variable. Since most safety specifications are admissible, this does not amount to a practical limitation.

As an example, consider the safety requirement that b should always be the last event performed. This is given by

$$S \;=\; \forall\, tr_0, tr_1 \bullet (tr = tr_0 {}^\frown \langle b \rangle {}^\frown tr_1) \Rightarrow tr_1 = \langle\rangle$$

A verification in the traces model that program AB satisfies this specification would be quite straightforward. We may translate this verification to the timed model, and conclude that AB **sat** $S[strip(s)/tr]$ in that model. This may then be used in a timed verification. For example, consider the timed specification that event a should never be performed within 8 time units of any b:

$$(t, b) \text{ in } s \;\Rightarrow\; a \notin \sigma(s \uparrow (t-8, t+8))$$

This specification reads as follows: if (t, b) is recorded in the trace s, then a does not appear in the set of events recorded in s during the interval $(t-8, t+8)$. Then the untimed specification tells us that a cannot occur after b, i.e. in the interval $(t, t+8)$, so the only cases to consider in the timed model are a occurring before b, or at the same time, i.e. the interval $(t-8, t]$. For this case, a timed analysis on AB is required.

In general, S is translated to $(\#s < \infty \Rightarrow S[strip(s)/tr])$.

3.2 Failures Refinement

We may think of a process refusing a particular set, in the untimed sense, if it eventually reaches a state after which no event from that set is possible. In the timed world, this corresponds to the information that there is some time after which the set may be continuously refused. Thus for a timed behaviour $(u, \aleph)$ with finite timed trace u, an abstract view of this behaviour would be an untimed version $strip(u)$ of the trace, and for any set X, if there is some t for which $[t, \infty) \times X$ is contained in $\aleph$, then $\aleph$ is evidence that X may eventually be refused forever.

Relating timed infinite traces to untimed ones, we obtain the following refinement relation between $\mathcal{O}_{UI}$ and $\mathcal{O}_{TI}$:

$$\begin{aligned}
(f, (tr, X)) \;{}_{UI}\mathcal{R}_{TI}\; (u, \aleph) \;&\Leftrightarrow\; tr = strip(u) \land \exists\, t \bullet [t, \infty) \times X \subseteq \aleph \\
(i, tr_0) \;{}_{UI}\mathcal{R}_{TI}\; (u, \aleph) \;&\Leftrightarrow\; tr_0 = strip(u)
\end{aligned}$$

Observe that there is no timed version of divergence in this model.

Theorem 3.4 This refinement relation is complete. □

Proof We need to show that $P =_{UI}\mathcal{R}_{TI}{}^{-1}\,(Q)$ is a well-defined process for any timed process Q (i.e. meets axioms 1–8 given in Appendix A).

It follows from axiom 1 for $\mathcal{M}_{TI}$ that ${}_{UI}\mathcal{R}_{TI}{}^{-1}\,(Q) = P$ meets axiom 1 for $\mathcal{M}_{UI}$. Axiom 2 for $\mathcal{M}_{TI}$ yields axioms 2 and 6 for P; Axiom 3 yields axiom 3;

and axioms 4,5 and 7 are trivial for P since P has no behaviours of the form (d, tr).

We have only to establish axiom 8. Consider a behaviour $(f, (s, \{\})) \in P$. Then there must be some timed trace s_0 such that $(s_0, \{\}) \in Q$ and $strip(s_0) = s$. Now by axiom 4 for $\mathcal{M}_{TI}$ there must be some process $R \in \mathcal{CL}$ such that $(s_0, \{\}) \in R$ and $Q \sqsubseteq R$.

Let $c : \mathbf{P}(\mathbf{R}) \to \mathbf{R}$ be a choice function. Then define:

$$\begin{array}{rcll} T_0 & = & \{s_0\} & \\ T_{n+1} & = & \{s^\frown\langle(t_a, a)\rangle \mid & s \in T_n \wedge a \in \Sigma \\ & & & \{t \mid (s^\frown\langle(t, a)\rangle, \{\}) \in R\} \neq \{\} \\ & & & t_a = c(\{t \mid (s^\frown\langle(t, a)\rangle, \{\}) \in R\})\} \\ & & & \\ T & = & \overline{\bigcup_{i \in \mathbf{N}} T_i} & \end{array}$$

Since R is finitely variable and closed, any infinite trace in T comes from a legitimate infinite trace in R. And since R meets axiom 3 for $\mathcal{M}_{TI}$, the set of all events that do not extend a given finite trace in T must be refusible for all time after the corresponding timed trace in R, since there is no time after that timed trace at which any of those events is possible. It follows that T is a set of traces which establishes that axiom 8 holds for P. $\square$

A study of the CSP operators reveals the following:

Theorem 3.5 Every CSP operator except parallel composition preserves ${}_{UI}\mathcal{R}_{TI}$ refinement $\square$

We again obtain that $Skip \sqsubseteq_{{}_{UI}\mathcal{R}_{TI}} \bigsqcap_{t \in I} Wait\ t$ for any set of times I, so arbitrary delays can be introduced into programs while still preserving timewise failures refinement.

Unfortunately, parallel composition does not preserve refinement in general. One example where it fails is in the case of two processes Q_1 and Q_2, illustrated in Figure 3.2. They are always willing to perform an event at some time in the future, by offering it periodically (so neither will eventually always refuse it), but they are unable to find any time on which they can synchronise, so their combination is able to refuse the offer forever.

$$\begin{array}{rcl} Q_1 & = & \mu X \circ (a \longrightarrow Stop) \overset{1}{\rhd} Wait\ 3\ ; X \\ Q_2 & = & Wait\ 2\ ; Q_1 \end{array}$$

Each of Q_1 and Q_2 are refinements of $P = a \longrightarrow Stop$, but $Q_1 \parallel Q_2$ is not a refinement of $P \parallel P$, since it may refuse a forever, as Q_1 and Q_2 can never synchronise on a; yet $P \parallel P$ is unable initially to refuse a.

However, we do obtain the following theorem.

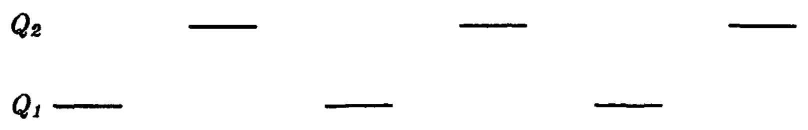


Figure 2: alternating offers

Theorem 3.6 Every CSP program P has that $P \ {}_{UI}\mathcal{R}_{TI} \ P$ □

This follows from Theorem 4.2 and Theorem 4.3, presented later.

Parallel composition and refinement

The importance of the parallel operator moves us to investigate conditions under which it does preserve refinement.

Non-Retraction

The example above illustrates one of the ways in which refinement may be lost by parallel composition: the periodic withdrawal of offers. It seems that one way to ensure synchronisation is to maintain offers until they are accepted.

A process which does not withdraw offers (though it may make new ones) until it next performs a visible event is termed *non-retracting*. This is similar to (though slightly weaker than) the notion of *nonpre-emptive* given in [ClZ92], although that definition is given in operational terms.

Definition 3.7 A process $S \in \mathcal{M}_{TI}$ is *non-retracting* if

$$(s, \aleph) \in S \quad \Rightarrow \quad (s, \aleph \cup \{(t, a) \mid \exists\, u \bullet (u, a) \in \aleph \wedge end(s) \leq t < u\}) \in S$$

□

If an event may be refused at a time u, then it must be possible that it was continuously refused since the occurrence of the last visible event, at time $end(s)$. Thus once an event is guaranteed to have been offered, it must be continually offered thereafter.

As expected, we obtain that parallel composition preserves refinement for non-retracting processes:

Lemma 3.8 If $P_1 \sqsubseteq_{UI}\mathcal{R}_{TI}\ Q_1$ and $P_2 \sqsubseteq_{UI}\mathcal{R}_{TI}\ Q_2$, and Q_1 and Q_2 are both non-retracting, then $(P_1 \parallel P_2) \sqsubseteq_{UI}\mathcal{R}_{TI}\ (Q_1 \parallel Q_2)$ □

Proof This is a special case of Lemma 3.10 below, with $A_1 = A_2 = \Sigma$, and the fact that non-retraction is stronger than eventual non-retraction on Σ. □

This idea of non-retraction may be generalised, so that it is concerned only with particular events rather than all events, and with the fact that a time after which offers should be maintained is reached eventually rather than immediately.

Definition 3.9 A process $S \in \mathcal{M}_{TI}$ is *eventually non-retracting on* A if for any trace s there is some time $t(s, S)$ such that

$$(s, \aleph) \in S \quad \Rightarrow \quad (s, \aleph \cup \{(t, a) \mid a \in A \wedge \exists u \bullet (u, a) \in \aleph \wedge t(s, S) \leq t < u\}) \in S$$

□

Observe that if S is eventually non-retracting on A, and $B \subseteq A$, then it is also eventually non-retracting on B.

This form of eventual non-retraction allows a period of unstable behaviour before settling down. This permits some timeout behaviour disallowed by a non-retraction requirement. For example, $a \longrightarrow P \overset{3}{\rhd} b \longrightarrow Q$ is eventually non-retracting (if P and Q are) although the offer of a will be retracted at time 3.

Lemma 3.8 may be generalised to interface parallel, by considering processes that are non-retracting on their common interface.

Lemma 3.10 If $P_1 \sqsubseteq_{UI\mathcal{R}_{TI}} Q_1$ and $P_2 \sqsubseteq_{UI\mathcal{R}_{TI}} Q_2$, and Q_1 and Q_2 are both eventually non-retracting on $A_1 \cap A_2$, then

$$(P_1 \ {}_{A_1}\|_{A_2} \ P_2) \sqsubseteq_{UI\mathcal{R}_{TI}} (Q_1 \ {}_{A_1}\|_{A_2} \ Q_2)$$

□

Proof If $(i, u) \ {}_{UI}\mathcal{R}_{TI} \ (s, \aleph)$ and $(s, \aleph) \in \mathcal{F}_{TI}[\![Q_1 \ {}_{A_1}\|_{A_2} \ Q_2]\!]$, then it follows immediately that $(i, u \restriction A_1) \in \mathcal{F}_{UT}[\![P_1]\!] \wedge (i, u \restriction A_2) \in \mathcal{F}_{UT}[\![P_2]\!]$, and hence that $(i, u) \in \mathcal{F}_{UI}[\![P_1 \ {}_{A_1}\|_{A_2} \ P_2]\!]$.

Consider a behaviour $(s, \aleph) \in \mathcal{F}_{TI}[\![Q_1 \ {}_{A_1}\|_{A_2} \ Q_2]\!]$, with $(f, (tr, X)) \ {}_{UI}\mathcal{R}_{TI}$ $(s, \aleph)$. Then $tr = strip(s)$, and $\exists t \bullet [t, \infty) \times X \subseteq \aleph$. Now by the semantics of the parallel operator, there are $\aleph_1$ and $\aleph_2$ such that $(s, \aleph_1) \in \mathcal{F}_{TI}[\![Q_1]\!]$, $(s, \aleph_2) \in \mathcal{F}_{TI}[\![Q_2]\!]$, and $\aleph \restriction (A_1 \cup A_2) = (\aleph_1 \restriction A_1) \cup (\aleph_2 \restriction A_2)$. Define

$$\begin{aligned} X_1 &= \{a \in A_1 \mid \forall t \bullet a \in \sigma(\aleph_1 \rhd t)\} \\ X_2 &= \{a \in A_2 \mid \forall t \bullet a \in \sigma(\aleph_2 \rhd t)\} \end{aligned}$$

Then $X_1 \cup X_2 = X \restriction (A_1 \cup A_2)$. Furthermore, by the eventual non-retraction of Q_1 and Q_2, it follows that there are t_1 and t_2 such that

$$\begin{aligned} &(s \restriction A_1, \aleph_1 \cup [t_1, \infty) \times X_1)) \in \mathcal{F}_{TI}[\![Q_1]\!] \\ &(s \restriction A_2, \aleph_2 \cup [t_2, \infty) \times X_2)) \in \mathcal{F}_{TI}[\![Q_2]\!] \end{aligned}$$

Since each P_i is refined by the corresponding Q_i, it follows that

$$\begin{aligned} &(f, (strip(s) \restriction A_1, X_1)) \in \mathcal{F}_{UI}[\![P_1]\!] \\ &(f, (strip(s) \restriction A_2, X_2)) \in \mathcal{F}_{UI}[\![P_2]\!] \end{aligned}$$

and so $(f, (strip(s), X_1 \cup X_2)) \in \mathcal{F}_{UI}[\![P_1 \ {}_{A_1}\|_{A_2}\ P_2]\!]$ by the semantics of the parallel operator. Hence the parallel operator preserves refinement for eventually non-retracting processes. □

However, if only one of the processes is non-retracting, then the refinement need not be preserved through a parallel combination. For example, consider the following processes, illustrated in Figure 3.2.

$$\begin{aligned} Q_1 &= \mathit{Wait}\ 2\ ; \mu X \circ (0 \longrightarrow \mathit{Stop} \ \Box\ \mathit{Wait}\ 1\ ; \mathit{succ}(X)) \\ Q_2 &= \mu X \circ (n : \mathbf{N} \longrightarrow \mathit{Stop}) \overset{1}{\rhd} \mathit{succ}(X) \end{aligned}$$

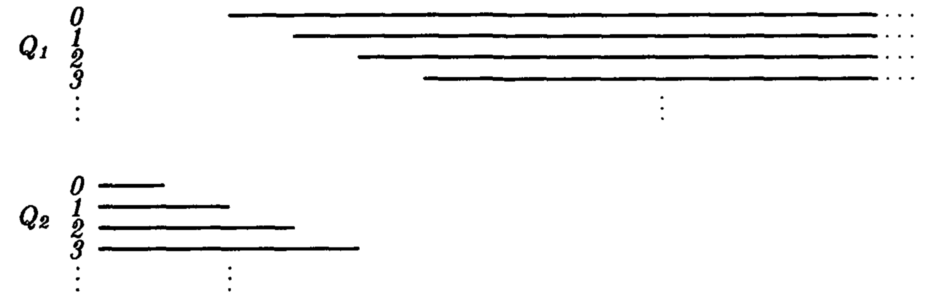


Figure 3: non-synchronising offers

The process Q_1 makes natural numbers available, one at a time. It is non-retracting, and it is also a refinement of $P_1 = n : \mathbf{N} \longrightarrow \mathit{Stop}$; on the empty trace, nothing may be refused forever. The process Q_2 begins with all natural numbers available, and retracts them one at a time. It is a refinement of

$$P_2 = \sqcap_{F \subseteq^{fin} \mathbf{N}}\ n : (\mathbf{N} \setminus \mathbf{F}) \longrightarrow \mathit{Stop}$$

for which all events are possible, and any finite set of events may eventually be refused forever. The parallel combination $Q_1 \parallel Q_2$ is equivalent to *Stop*, since there is no event that Q_1 and Q_2 may cooperate on: Q_1 is prepared to perform event m from time $m + 2$ onwards, but Q_2 is not prepared to perform it beyond time $m + 1$. On the other hand, $P_1 \parallel P_2$ is equivalent to P_2, which is unable to deadlock before any events have been performed. Hence $Q_1 \parallel Q_2$ is not a refinement of $P_1 \parallel P_2$, even though Q_1 is non-retracting.

Promptness

The above example highlights other ways in which parallel combination can fail to preserve refinement. If Q_1 had made all of its offers by some time t, then the counterexample would not work, since the non-retraction of Q_1 ensure that all of the offers, made by time t, must remain on offer until acceptance occurs.

We define a process to be t-prompt if it must make its offers by time t: if a set may be refused up to time t, then it may be refused forever.

Definition 3.11 A process $S \in \mathcal{M}_{TI}$ is *t-prompt* if

$$(s, \aleph) \in S \wedge [u, u+t) \times A \subseteq \aleph \wedge u \geq end(s) \quad \Rightarrow \quad (s, \aleph \cup [u, \infty) \times A) \in S$$

□

We then obtain the following alternative result, which places no constraints upon Q_2:

Lemma 3.12 If $P_1 \sqsubseteq_{UI\mathcal{R}_{TI}} Q_1$ and $P_2 \sqsubseteq_{UI\mathcal{R}_{TI}} Q_2$, Q_1 is non-retracting and t-prompt for some t, then $(P_1 \parallel P_2) \sqsubseteq_{UI\mathcal{R}_{TI}} (Q_1 \parallel Q_2)$ □

Proof This is a special case of Lemma 3.14 below. □

Promptness may be generalised to apply only to a particular set of events A.

Definition 3.13 A process $S \in \mathcal{M}_{TI}$ is *t-prompt on* A if

$$\begin{aligned}(s, \aleph) \in S \wedge B \subseteq A \wedge [u, u+t) \times B \subseteq \aleph \wedge u \geq end(s) \\ \Rightarrow \quad (s, \aleph \cup [u, \infty) \times B) \in S\end{aligned}$$

□

Then the condition for parallelism to preserve refinement may be correspondingly generalised: if the interface between two processes may be split into two parts, and each process is prompt and non-retracting on a different part, then refinement will be preserved by parallel composition.

Lemma 3.14 If $P_1 \sqsubseteq_{UI\mathcal{R}_{TI}} Q_1$, $P_2 \sqsubseteq_{UI\mathcal{R}_{TI}} Q_2$, $B_1 \cup B_2 = A_1 \cap A_2$, Q_1 is eventually non-retracting on B_1 and prompt on B_1, and Q_2 is eventually non-retracting on B_2 and prompt on B_2, then

$$(P_1 \;{}_{A_1}\|_{A_2}\; P_2) \quad \sqsubseteq_{UI\mathcal{R}_{TI}} \quad (Q_1 \;{}_{A_1}\|_{A_2}\; Q_2)$$

□

Proof It is clear that infinite traces of the timed process will appear in untimed form in the untimed process. So we need only show that

$$\begin{aligned}(f, (tr, X)) \;{}_{UI}\mathcal{R}_{TI}\; (s, \aleph) \wedge (s, \aleph) \in \mathcal{F}_{TI}[\![Q_1 \;{}_{A_1}\|_{A_2}\; Q_2]\!] \\ \Rightarrow \quad (f, (tr, X)) \in \mathcal{F}_{UI}[\![P_1 \;{}_{A_1}\|_{A_2}\; P_2]\!]\end{aligned}$$

Assume the antecedent, and consider $(s, \aleph) \in \mathcal{F}_{TI}[\![Q_1 \;{}_{A_1}\|_{A_2}\; Q_2]\!]$, built from behaviours $(s \upharpoonright A_1, \aleph_1) \in \mathcal{F}_{TI}[\![Q_1]\!]$ and $(s \upharpoonright A_2, \aleph_2) \in \mathcal{F}_{TI}[\![Q_2]\!]$, where we have $(\aleph_1 \upharpoonright A_1) \cup (\aleph_2 \upharpoonright A_2) = \aleph \upharpoonright (A_1 \cup A_2)$. We have by the relation ${}_{UI}\mathcal{R}_{TI}$ that $tr = strip(s)$, and also $t \geq end(s) + \max\{t(s \upharpoonright A_1, Q_1), t(s \upharpoonright A_2, Q_2)\}$ for some t, such that $[t, \infty) \times X \subseteq \aleph$. Now Q_1 is t_1-prompt for some t_1, and Q_2 is t_2-prompt for some t_2. Define

$$\begin{aligned}
X_1 &= \sigma((\aleph_1 \upharpoonright (X \cap A_1 \cap B_1)) \rhd (t_1 + t)) \\
Y_1 &= \{a \in X \cap A_1 \cap B_2 \mid [t_2 + t, \infty) \times \{a\} \subseteq \aleph_1\} \\
X_2 &= \sigma((\aleph_2 \upharpoonright (X \cap A_2 \cap B_2)) \rhd (t_2 + t)) \\
Y_2 &= \{a \in X \cap A_2 \cap B_1 \mid [t_1 + t, \infty) \times \{a\} \subseteq \aleph_2\}
\end{aligned}$$

Then $X_1 \cup Y_1 \cup X_2 \cup Y_2 = X \cap (A_1 \cup A_2)$. Now since Q_1 is eventually non-retracting on B_1, it follows (also using subset closure of refusals for Q_1) that

$$(s \upharpoonright A_1, \aleph_1 \cup [t, t + t_1) \times X_1) \in \mathcal{F}_{TI}[\![Q_1]\!]$$

And hence it follows by promptness that

$$(s \upharpoonright A_1, \aleph_1 \cup [t, \infty) \times X_1) \in \mathcal{F}_{TI}[\![Q_1]\!]$$

Also, observe that $[t_2 + t, \infty) \times Y_1 \subseteq \aleph_1$. Since Q_1 is a refinement of P_1 it follows that $(f, (tr \upharpoonright A_1, X_1 \cup Y_1)) \in \mathcal{F}_{UI}[\![P_1]\!]$. Similarly we obtain that $(f, (tr \upharpoonright A_2, X_2 \cup Y_2)) \in \mathcal{F}_{UI}[\![P_2]\!]$. Thus from the semantics of parallel, we obtain that $(f, (tr, X)) \in \mathcal{F}_{UI}[\![P_1 \; {}_{A_1}\|_{A_2} \; P_2]\!]$, yielding the result. $\square$

This result is particularly useful, as it applies immediately to systems such as those described in terms of input/output automata [LyV92] where input is always possible, and so components are always non-retracting and prompt on input. In these systems, parallel composition connects outputs from one process to corresponding inputs of the other. Thus the interface in a parallel composition may be partitioned into those events input by one component, and those input by the other. The two processes will be prompt and non-retracting on these two sets respectively.

This result may also be applied to CSP descriptions of occam programs, since in such programs output guards are not permitted. Once a process is prepared to perform an output, it remains ready to perform it until it occurs. Consequently, processes are always non-retracting on output, so parallel composition will preserve refinement for prompt components.

Compactness

The final condition we will present here concerns the nature of the untimed processes P_1 and P_2. A process is compact if its refusals are determined (in a particular way) by the finite refusal sets. If the untimed processes are compact, then it turns out that only one of the timed processes need be non-retracting for refinement to be preserved by the parallel operator. In the example above, P_2 fails this condition; any finite subset of $\mathbf{N}$ may be refused, but infinite subsets may not.

Definition 3.15 A process P in $\mathcal{M}_{UI}$ is *compact* if for any $tr \in \Sigma^*$, $Y \subseteq \Sigma$ we have

$$(\forall X \subseteq^{fin} Y \bullet (f,(tr,X)) \in P) \quad \Rightarrow \quad (f,(tr,Y)) \in P$$

$\square$

Lemma 3.16 If $P_1 \sqsubseteq_{UI}\mathcal{R}_{TI}\ Q_1$, $P_2 \sqsubseteq_{UI}\mathcal{R}_{TI}\ Q_2$, Q_1 is eventually non-retracting on $A_1 \cap A_2$, and P_1,P_2 are compact, then

$$(P_1\ {}_{A_1}\|_{A_2}\ P_2) \sqsubseteq_{UI}\mathcal{R}_{TI}\ (Q_1\ {}_{A_1}\|_{A_2}\ Q_2)$$

$\square$

Proof It is clear that infinite traces of the timed process will appear in the untimed process. So we need only show that

$$\begin{aligned}(f,(tr,X))\ {}_{UI}\mathcal{R}_{TI}\ (s,\aleph) \wedge (s,\aleph) \in \mathcal{F}_{TI}[\![Q_1\ {}_{A_1}\|_{A_2}\ Q_2]\!]\\ \Rightarrow \quad (f,(tr,X)) \in \mathcal{F}_{UI}[\![P_1\ {}_{A_1}\|_{A_2}\ P_2]\!]\end{aligned}$$

Consider $(s,\aleph) \in \mathcal{F}_{TI}[\![Q_1\ {}_{A_1}\|_{A_2}\ Q_2]\!]$, built from behaviours $(s,\aleph_1) \in \mathcal{F}_{TI}[\![Q_1]\!]$ and $(s,\aleph_2) \in \mathcal{F}_{TI}[\![Q_2]\!]$, where $\aleph_1 \upharpoonright A_1 \cup \aleph_2 \upharpoonright A_2 = \aleph \upharpoonright (A_1 \cup A_2)$. We have by the relation ${}_{UI}\mathcal{R}_{TI}$ that $tr = strip(s)$, and also that there is some $t \geq end(s) + t(s \upharpoonright A_1, Q_1)$ such that $[t,\infty) \times X \subseteq \aleph$. Define

$$\begin{aligned}X_1 &= \{a \in X \cap A_1 \mid \forall u \bullet a \in \sigma(\aleph_1 \rhd u)\}\\ X_2 &= \{a \in X \cap A_2 \mid \exists u \bullet [u,\infty) \times \{a\} \subseteq \aleph_2\}\end{aligned}$$

Since $\aleph_1 \upharpoonright A_1 \cup \aleph_2 \upharpoonright A_2 = \aleph \upharpoonright (A_1 \cup A_2)$, we have that $X_1 \cup X_2 = X$. Since Q_1 is eventually non-retracting, we obtain that $(s, \aleph_1 \cup [t,\infty) \times X_1) \in \mathcal{F}_{TI}[\![Q_1]\!]$. Since P_1 is refined by Q_1, we have that $(tr, X_1) \in \mathcal{F}_{UI}[\![P_1]\!]$.

Now consider a finite set $\{a_1, a_2 \ldots a_n\} = Y \subseteq X_2$. For each a_i there is a corresponding time t_i such that $[t_i,\infty) \times \{a_i\} \subseteq \aleph_2$. Thus there is a time $t_0 = max\{t_i\}$ such that $[t_0,\infty) \times Y \subseteq \aleph_2$. Since P_2 is refined by Q_2, it follows that $(f,(tr,Y)) \in \mathcal{F}_{UI}[\![P_2]\!]$. This is true for all finite subsets Y of X_2, so by compactness of P_2 we have that $(f,(tr,X_2)) \in \mathcal{F}_{UI}[\![P_2]\!]$. Hence $(f,(tr,X)) \in \mathcal{F}_{UI}[\![P_1\ {}_{A_1}\|_{A_2}\ P_2]\!]$ as required. $\square$

Compactness is often easy to check, since it will be present in any process not containing infinite non-determinism. Thus any program not containing any infinite choice will automatically be compact.

Specification

In the untimed infinite traces model specifications may be considered as consisting of three components, dealing with the failures, divergences and infinite traces. In other words, for any given $S(l,b)$ there are S_f, S_d and S_i such that

$$\begin{aligned} S(l,b) \quad \Leftrightarrow \quad & (l = \text{`}f\text{'} \wedge b = (tr, X)) \Rightarrow S_f(tr, X) \\ & \wedge (l = \text{`}d\text{'} \wedge b = tr) \Rightarrow S_d(tr) \\ & \wedge (l = \text{`}i\text{'} \wedge b = u) \Rightarrow S_i(u) \end{aligned}$$

Then a specification (S_f, S_d, S_i) translates to the timed specification

$$\begin{aligned} & \#u < \infty \wedge [t, \infty) \times X \subseteq \aleph \Rightarrow S_f(strip(u), X) \\ \wedge \quad & \#u = \infty \Rightarrow S_i(strip(u)) \end{aligned}$$

For example, the specification 'deadlock-free' constrains only the possible failure set, with $S_f(tr, X) \Leftrightarrow X \neq \Sigma$. The translation is equivalent to

$$\#u < \infty \Rightarrow \neg \exists\, t \bullet [t, \infty) \times \Sigma \subseteq \aleph$$

which is the timed version of deadlock-freedom. Thus an untimed verification of deadlock-freedom for a system remains valid under timewise refinement.

The untimed specification of a buffer may be given simply as a predicate S_f:

$$\begin{aligned} S_f(tr, X) \quad \Leftrightarrow \quad & \quad tr \downarrow out \leq tr \downarrow in \\ & \wedge \quad tr \downarrow out = tr \downarrow in \Rightarrow X \cap in = \{\} \\ & \wedge \quad tr \downarrow out < tr \downarrow in \Rightarrow out \not\subseteq X \end{aligned}$$

where $tr \downarrow c$ is the sequence of messages recorded in tr on channel c.

The translation is equivalent to the following timed specification:

$$\begin{aligned} & strip(u) \downarrow out \leq strip(u) \downarrow in \\ \wedge \quad & strip(u) \downarrow out = strip(u) \downarrow in \Rightarrow \neg \exists\, t, m \bullet [t, \infty) \times \{in.m\} \subseteq \aleph \\ \wedge \quad & strip(u) \downarrow out < strip(u) \downarrow in \Rightarrow \neg \exists\, t \bullet [t, \infty) \times out \subseteq \aleph \end{aligned}$$

which is the specification of a timed buffer.

As an example of an application of the theory, consider Roscoe's first (untimed) buffer law presented in [Hoa85], which tells us that the chaining together of two buffers is again a buffer. The chaining operator is defined in terms of parallel, hiding, and renaming (where $swap_{a,b}$ renames channel a to b and vice versa):

$$P_1 \gg P_2 \quad \widehat{=} \quad (swap_{out,c}(P_1)\ {}_{\{in,c\}}\|_{\{out,c\}}\ swap_{in,c}(P_2)) \setminus c$$

However, this law does not hold in general in the timed model. As we have seen, B_1 and B_2 might fail to agree on a time to synchronise on their common internal channel, resulting in their combination refusing ever to output.

In order to establish conditions under which the law does hold, we will make use of the fact that untimed buffers are compact (since if some input can be refused, then so too can all possible inputs), and also of the fact that the refinement relations in this paper are complete, which yields that every timed buffer is a refinement of some untimed buffer. We may then obtain conditions under which a chain of buffers again yields a buffer.

For example, if every buffer B_i is eventually non-retracting on input, then the chain $B_1 \gg B_2 \gg \ldots \gg B_n$ is again a buffer, eventually non-retracting on input. This follows from the fact that each B_i is a refinement of some buffer A_i; that we have a condition which may be applied at every step of building up the chain to ensure that the timed chain refined the untimed chain (in the general parallel case, we require only non-retraction on the interface); and that the chain $A_1 \gg A_2 \gg \ldots \gg A_n$ is an untimed buffer (from Roscoe's law), from which it follows that any refinement of it is a timed buffer. A similar result holds if each B_i is non-retracting on output; or if odd (or even) numbered buffers are non-retracting on both input and output. It follows that the combination $B_1 \gg COPY \gg B_2 \ldots COPY \gg B_n$ is a buffer, for any timed buffers B_i.

4 An operational view

An alternative semantic approach that is often employed in the theory of process algebra is operational: processes are defined in terms of transitions that they may perform and subsequent states that may be reached. Within this framework, equivalence between processes may be characterised in terms of bisimulation relations [Mil89], or by means of equivalence under some notion of testing [Hen88].

In the testing approach, a test is defined to be a process T which also has the capacity to perform a special success event ω, which is considered to be distinct from the set of synchronisation events Σ. An execution of a process P is a maximal (finite or infinite) sequence of transitions starting from P. Then we say that P $\underline{\text{may}}$ T if there is some execution of $(P \parallel T) \setminus \Sigma$ which passes through a state from which ω is a possible transition; and P $\underline{\text{must}}$ T if every execution of $(P \parallel T) \setminus \Sigma$ passes through such a state. Then P is equivalent to Q under $\underline{\text{may}}$ testing if for any test T, P $\underline{\text{may}}$ $T \Leftrightarrow Q$ $\underline{\text{may}}$ T; and P and Q are equivalent under $\underline{\text{must}}$ testing if P $\underline{\text{must}}$ $T \Leftrightarrow Q$ $\underline{\text{must}}$ T for any test T.

An operational semantics has been given for CSP in [BRW9x] and [Ros88]. Transitions are given as $P \xrightarrow{\mu} P'$, indicating that a process P may perform a μ event (i.e. an internal or visible event) and then behave as P'. In this section we will subscript the transition with a u to indicate that this is an untimed transition. Equivalence in the untimed traces model $\mathcal{M}_{UT}$ is exactly the same as equivalence under $\underline{\text{may}}$ testing using the transitions given in [Ros88]; and equivalence in the untimed infinite traces model $\mathcal{M}_{UI}$ is exactly the same as equivalence under $\underline{\text{must}}$ testing using those transitions. More details may be found in

[Hen88, BRW9x, Ros88]. The important properties from our point of view is that each trace of P predicted by the traces model corresponds to an execution of P in which that sequence of visible events is performed (as well as possibly some internal events); that any divergence corresponds to an execution in which some prefix of the divergent trace is performed, followed by an infinite sequence of internal τ steps; any failure (tr, X) corresponds either to a divergence (i.e. an infinite sequence of τ steps after some prefix of the trace) or to an execution in which the entire sequence tr of events is performed, and a state is reached from which no internal progress can be made, and from which no event in the refusal set X is possible; and for every infinite trace u there is an execution which either diverges after some prefix of u or performs the entire sequence of events u. And converseiy, any execution given by the operational semantics is recorded appropriately in the denotational semantics.

An operational semantics has also been provided for timed CSP in [Sch93], where processes may undergo timed transitions: $P \xrightarrow{(t,\mu)} P'$ indicates that the process P may perform event μ at time t, and subsequently behave as P'. We will subscript timed transitions with t to distinguish them from untimed transitions. Evolutions, or time passing transitions, were also provided in the operational semantics. Equivalence in the infinite timed failures model $\mathcal{M}_{TI}$ is the same as equivalence under $\underline{\text{must}}$ testing using the transitions given in [Sch93]. Again, timed failures $(s, \aleph)$ are present in the denotational semantics of a process P precisely when there is some execution of P in which events are performed at the times recorded in s, passing through states in which the events recorded in the refusal set $\aleph$ were not possible.

Every CSP operator except for timeout has an untimed operational semantics. We may also describe untimed transitions for the timeout operator. The timeout may always be resolved by its left-hand argument performing a visible action, but any internal progress made by that argument does not resolve the timeout.

$$\frac{P \xrightarrow{a}_u P'}{P \overset{t}{\rhd} Q \xrightarrow{a}_u P'} \qquad\qquad \frac{P \xrightarrow{\tau}_u P'}{P \overset{t}{\rhd} Q \xrightarrow{\tau}_u P' \overset{t}{\rhd} Q}$$

Furthermore, the timeout may occur:

$$\frac{}{P \overset{t}{\rhd} Q \xrightarrow{\tau}_u Q}$$

Thus every timed CSP process has both an untimed and a timed operational semantics. We may then consider a test T both at the timed and at the untimed level. Then we will say $P \ \underline{\text{may}}_u\ T$ if some execution of $(P \parallel T) \setminus \Sigma$ in terms of untimed $\rightarrow_u$ transitions passes through a state in which ω is possible; and we will use $P \ \underline{\text{must}}_u\ T$, $P \ \underline{\text{may}}_t\ T$ and $P \ \underline{\text{must}}_t\ T$ in a similar fashion.

A useful relationship between a timed process and an untimed one is that of untimed/timed similarity, which essentially says that executions of the timed process can be matched by executions of the untimed.

Definition 4.1 Relation R is an *untimed/timed similarity* if whenever $R(P, Q)$, then

1. If $Q \xrightarrow{(t,\mu)}_t Q'$ then there is some P' such that $P \xrightarrow{\mu}_u P'$ and $R(P', Q')$
2. If $P \xrightarrow{\mu}_u$ then there is some process P', Q', and time t such that $P \xrightarrow{\mu}_u P'$ and $Q \xrightarrow{(t,\mu)}_t Q'$ and $R(P', Q')$.

□

We say that P and Q are untimed/timed similar if there is some untimed/timed similarity that holds between them.

Theorem 4.2 If P and Q are untimed/timed similar, then $P \sqsubseteq_{UT} \mathcal{R}_{TI}\ Q$ and $P \sqsubseteq_{UI} \mathcal{R}_{TI}\ Q$. □

Proof (sketch) This follows from the above-mentioned equivalence of the denotational and operational semantics in both the untimed cases and the timed case. In the first case, if there is a timed trace s of Q, then there is some execution of Q which gives rise to this trace. But then by untimed/timed similarity, every step of this execution can be matched by an untimed step, so there is an equivalent untimed execution of P, which corresponds to the trace $strip(s)$. Since the untimed operational and denotational semantics are equivalent, the trace $strip(s)$ appears in the trace set of P.

In the case of failures refinement, similar reasoning shows that infinite timed traces will be matched by infinite untimed ones; and a timed failure $(s, [t, \infty) \times X)$ with finite trace s will correspond to some execution of Q. After the trace s has been performed there are two possibilities. The execution may contain an infinite sequence of internal events; these can be matched by P, leading to a divergence and the inclusion of $(f, (strip(s), X))$ as a failure of P. The other possibility is that a final state is reached from which no event in X, or any further internal progress, is possible (since X is refused from that point onwards); in this case a corresponding untimed state in which X may be refused is reachable from P by means of a corresponding execution, and the failure $(f, (strip(s), X))$ again appears as a failure of P, □

Theorem 4.3 Every CSP process P is untimed/timed similar to itself □

Proof Let the relation R hold between two processes if they are syntactically identical up to the values of timeouts. A straightforward structural induction on the structure of the untimed process shows that R is an untimed/timed simulation. It follows that any process P is untimed/timed similar to itself.

□

In the traces model, $P \sqsubseteq Q$ is true exactly when $\forall\, T \bullet (Q\ \underline{\text{may}}\ T \Rightarrow P\ \underline{\text{may}}\ T)$. By analogy, we may characterise an operational version of timed refinement, where if Q may pass a timed test T, then P may pass the same T considered as an untimed test.

Definition 4.4 $P \sqsubseteq_t Q$ is defined by

$$P \sqsubseteq_t Q \quad \Leftrightarrow \quad \forall T : Q \ \underline{\text{may}}_t \ T \Rightarrow P \ \underline{\text{may}}_u \ T$$

□

It turns out that this notion of refinement is the same as the denotational version of traces refinement.

Theorem 4.5 $P \sqsubseteq_t Q \Leftrightarrow P \sqsubseteq_{UT\mathcal{R}_{TI}} Q$ □

Proof "$\Rightarrow$" Assume $P \not\sqsubseteq_{UT\mathcal{R}_{TI}} Q$. Then there is some trace s of Q such that $strip(s)$ is not a trace of P. Let $s = \langle (t_1, a_1), \ldots, (t_n, a_n) \rangle$. Then define the test T by

$$T = a_1 \longrightarrow \ldots \longrightarrow a_n \longrightarrow \omega \longrightarrow Stop$$

Since the timed operational semantics are equivalent to the denotational semantics, there is some execution of Q giving rise to trace s, so there is some execution of $(Q \parallel T) \setminus \Sigma$ which reaches a state in which T can perform ω. However, there is no such execution of $(P \parallel T) \setminus \Sigma$, since if there were then this would correspond to P performing the events in $strip(s)$, which would mean that $strip(s)$ is a trace of P, yielding a contradiction. Thus $Q \ \underline{\text{may}}_t \ T$ but $\neg(P \ \underline{\text{may}}_u \ T)$, and so $\neg(P \sqsubseteq_t Q)$

"$\Leftarrow$" Assume $P \sqsubseteq_{UT\mathcal{R}_{TI}} Q$, and consider a test T for which $Q \ \underline{\text{may}}_t \ T$. Then there is some execution of $(Q \parallel T) \setminus \Sigma$ which leads to a state in which $\overset{\omega}{\longrightarrow}_t$ is possible. The contribution of Q to this execution corresponds to some timed trace s. Then $strip(s)$ is a trace of P, so P has some execution giving rise to $strip(s)$. Now since T is untimed/timed similar to itself, $(P \parallel T) \setminus \Sigma$ has an execution which takes T through untimed states that are untimed/timed similar to the timed states T passed through in the successful execution of $(Q \parallel T) \setminus \Sigma$, so it reaches a state in which an $\overset{\omega}{\longrightarrow}_u$ transition is possible. Thus $P \ \underline{\text{may}}_u \ T$.

□

In the failures/divergences model, $P \sqsubseteq Q$ is equivalent to $P \ \underline{\text{must}} \ T \Rightarrow Q \ \underline{\text{must}} \ T$ for any T. Again by analogy, we characterise an operational version of timed refinement:

Definition 4.6 $P \sqsubseteq_f Q$ is defined by

$$P \sqsubseteq_f Q \quad \Leftrightarrow \quad \forall T : P \ \underline{\text{must}}_u \ T \Rightarrow Q \ \underline{\text{must}}_t \ T$$

□

This formulation of refinement is equivalent to the denotational version of failures refinement.

Theorem 4.7 $P \sqsubseteq_f Q \Leftrightarrow P \sqsubseteq_{UI}\mathcal{R}_{TI}\ Q$ □

Proof "$\Rightarrow$" If $P \not\sqsubseteq_{UI}\mathcal{R}_{TI}\ Q$ then either (1) there is some $(s, [u, \infty) \times X) \in \mathcal{F}_{TI}[\![Q]\!]$ with $(strip(s), X) \notin \mathcal{F}_{UF}[\![P]\!]$, or (2) there is some infinite trace s such that $(s, \{\}) \in \mathcal{F}_{TI}[\![Q]\!]$ and $strip(s) \notin \mathcal{F}_{UI}[\![P]\!]$.

1 Let $s = \langle (t_1, a_1), \ldots, (t_n, a_n) \rangle$, and let $t_0 = 0$. Define

$$\begin{array}{rcll} T_i & = & Wait\,(t_i - t_{i-1})\,;((a_i \longrightarrow T_{i+1}) \overset{0}{\rhd} \omega \longrightarrow Stop)) & 0 < i \leq n \\ T_{n+1} & = & Wait\,(u - t_n)\,; x : X \longrightarrow \omega \longrightarrow Stop & \end{array}$$

If $(P \parallel T_1) \setminus \Sigma$ has an execution that is not successful, then the contribution from P must correspond to the failure $(strip(s), X)$, yielding a contradiction. Thus $P\ \underline{\text{must}}_u\ T_1$. On the other hand, Q has an execution corresponding to $(s, [u, \infty) \times X)$, and so $(Q \parallel T) \setminus \Sigma$ does have an unsuccessful execution, thus $\neg(Q\ \underline{\text{must}}_t\ T)$.

2 Let $s = \langle (t_1, a_1), \ldots, (t_i, a_i), \ldots \rangle$. Then let the trace during an interval $[n, n+1)$ be given by $\langle (t_{n1}, a_{n1}), \ldots, (t_{nm}, a_{nm}) \rangle$. This must be finite for any interval, since the trace s is finitely variable, i.e. its restriction to any finite interval is finite. Define

$$\begin{array}{rcll} T_{n,i} & = & Wait\,(t_{n,i} - t_{n,i-1})\,; & \\ & & \quad ((a_{n,i} \longrightarrow T_{n,i+1}) \overset{0}{\rhd} \omega \longrightarrow Stop) & 0 < i \leq m \\ T_{n,m+1} & = & Stop & \\ T_n & = & T_{n,1} \parallel Wait\ 1\ ; T_{n+1} & \end{array}$$

This formulation is required to ensure that each of the equations for the T_i is 1-guarded.) Then if $(P \parallel T_0) \setminus \Sigma$ has an unsuccessful execution, the contribution of P must correspond to $strip(s)$, yielding a contradiction; thus $P\ \underline{\text{must}}_u\ T_0$. However, $(Q \parallel T_0) \setminus \Sigma$ does have an unsuccessful execution, driven by an execution of Q corresponding to s. Thus $\neg(Q\ \underline{\text{must}}_t\ T_0)$.

"$\Leftarrow$" Assume that $P \sqsubseteq_{UI}\mathcal{R}_{TI}\ Q$, and that $\neg(Q\ \underline{\text{must}}_t\ T)$. It will be enough to prove that $\neg(P\ \underline{\text{must}}_u\ T)$. Consider an unsuccessful execution of $(Q \parallel T) \setminus \Sigma$. There are a number of possibilities; we consider the events that were internalised by the $\setminus \Sigma$ abstraction:

* $Q \parallel T$ performs infinitely many events from Σ. Then there is a corresponding infinite trace s of both Q and T. Since $P \sqsubseteq_{UI}\mathcal{R}_{TI}\ P$, the trace $strip(s)$ is an infinite trace of P. If P diverges at some point along $strip(s)$, then this will give rise to an unsuccessful execution of $(P \parallel T) \setminus \Sigma$. Otherwise, since T is untimed/timed similar to itself, there is an infinite untimed execution of T performing the same events, and passing through untimed/timed

similar states to those reached in the timed execution. Hence there is an infinite execution of $(P \parallel T) \setminus \Sigma$ where ω is not possible in any state (since the possibility of ω depends purely on the state reached by T), and so $\neg(P\ \underline{\mathrm{must}}_u\ T)$.

* $Q \parallel T$ performs finitely many events from Σ:
  - If T performs infinitely many timed τ transitions, then it may perform infinitely many untimed ones, passing through similar states, so if P does not diverge (leading to an unsuccessful execution) then this will yield an unsuccessful execution of $(P \parallel T) \setminus \Sigma$.
  - If T performs finitely many τ actions, then it will [illegible] to arrive in a final state T'. Any events that T' is able to perf[illegible]e blocked by Q for all time, and so P (if it does not diverge) [illegible]each a stable state P'' in which none of those events are possible. Since T may by untimed transitions reach a state T'' untimed/timed similar to T', T'' is also unable to perform those events that T' was unable to perform, and so $P'' \parallel T''$ will be unable to progress. Thus the execution from $P \parallel T$ to $P'' \parallel T''$ is maximal, and furthermore is unsuccessful.

$\square$

5 A simple example

The well-known alternating bit protocol is a useful common example, since it has been treated by so many different formalisms that it provides a means of comparing and contrasting them. We will use it here simply to illustrate some of the techniques presented earlier.

The untimed alternating bit protocol consists of a sender and receiver communicating over two lossy channels. The nature of a generic lossy channel may be specified at the untimed level using the infinite traces model. The specification *SM* on a medium $M_{in,out}$ with input *in* and output *out* consists of three parts:

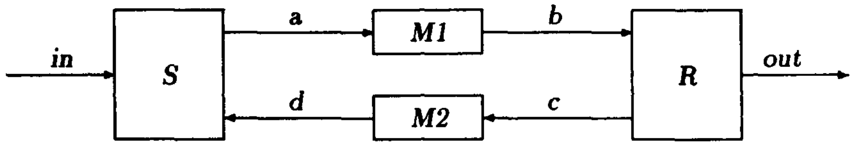
M1	$s \downarrow out \preceq s \downarrow in$
M2	$out.M \not\subseteq X \vee in.M \cap X = \{\}$
M3	$\#(u \upharpoonright in) = \infty \Rightarrow \#(u \upharpoonright out) = \infty$

M1 simply states that the sequence of messages passed on channel *out* should be a (not necessarily contiguous) subsequence of those passed on channel *in*, so messages may be lost but not corrupted; *M2* states that at least one of input and output should not be refused (where X is the refusal set) ; and *M3* is a fairness condition that requires that output should not be lost infinitely often.

The requirement we have of the entire system is that it should behave as a (one-place) buffer. Our specification is

$$\begin{array}{rrl} SPEC & = & s \downarrow out \leq_1 s \downarrow in \\ & \wedge & s \downarrow out = s \downarrow in \Rightarrow in.M \cap X = \{\} \\ & \wedge & s \downarrow out < s \downarrow in \Rightarrow out.M \not\subseteq X \end{array}$$

The network used is pictured as follows:



The basic idea of the protocol is to add an extra bit to each of the messages sent along the lossy channels which alternates between *0* and *1*. The sending process sends multiple copies of each message until it receives an acknowledgement. As soon as the receiving process gets a new message it sends acknowledgements of it until the next message arrives. The two ends can always spot a new message or acknowledgement because of the alternating bit.

The two media are described as $M1 = M_{a,b}$ and $M2 = M_{c,d}$, passing messages from a to b, and from c to d

This strategy may be captured by the following CSP descriptions of the sender S and the receiver R. We set $R = R(0)$ and $S = S(0)$, where for $s \in \{0, 1\}$ and x in the set of messages M we define

$$\begin{array}{rcl} S(s) & = & in?x \longrightarrow S'(s,x) \\ S'(s,x) & = & a!(s,x) \longrightarrow S'(s,x) \\ & & \Box\ d?s \longrightarrow S(\overline{s}) \\ & & \Box\ d?\overline{s} \longrightarrow a!(s,x) \longrightarrow S'(s,x) \\ R(s) & = & b?(s,x) \longrightarrow out!x \longrightarrow c!s \longrightarrow R(\overline{s}) \\ & & \Box\ b?(\overline{s},x) \longrightarrow c!\overline{s} \longrightarrow R(s) \end{array}$$

The entire network consists of the parallel combination of the sender and receiver together with the two media; and the channels a,b,c, and d are all made internal.

$$NETWORK = ((S \parallel R)\ {}_{\Sigma}\|_{\{a,b,c,d\}}\ (M1 \parallel M2)) \setminus \{a, b, c, d\}$$

Since the sender and receiver operate asynchronously, and the media also operate asynchronously, their combinations may be modelled using the interleaving operator $\parallel$, and the network considered as the parallel combination of the protocol and the media.

An analysis at the untimed level establishes that the system is livelock-free, essentially because of the fairness of the media which cannot lose an infinite sequence of messages. It is also deadlock-free: if S cannot make progress, then it must be waiting for both media, which must therefore both be ready to interact with R, and so R is able to make progress. Finally, it is straightforward to show that it is functionally equivalent to a one-place buffer.

Timed descriptions of the alternating bit protocol commonly employ a timeout in the description of the sender process, since the intention is that the sender should wait for an acknowledgement and then retransmit a message if this does not arrive within a certain interval. But in fact there is no need to withdraw the capability of receiving a message on the acknowledgement channel simply because a retransmission has been enabled, and so at the untimed level this behaviour may be modelled as a choice.

To provide a timed refinement of the protocol, we wish to preserve correctness of the system. The most general form of correctness that could be preserved by a timewise refinement would be for timed versions of the media to meet simply the translations of the untimed specifications with no further constraints. Thus we prefer not to impose the restriction on the media that they are non-retracting.

A timed version TS of the sender process may be obtained simply by including a delay t before retransmission of a message. The length of this delay will be influenced by such factors as the length of time before an acknowledgement would be expected to arrive, and the reluctance to send unnecessary messages. The timed receiver process TR still behaves sequentially, and has no time-critical behaviour.

Some small delays ϵ are introduced to ensure that the recursive loops are time-guarded. (These play the role of the original δ delay enforced by event prefix in earlier versions of timed CSP [ReR86]).

$$\begin{array}{rcl}
TS(s) & = & in?x \longrightarrow TS'(s,x) \\[1em]
TS'(s,x) & = & Wait\ t\ ; a!(s,x) \longrightarrow TS'(s,x) \\
& & \Box\ d?s \xrightarrow{\epsilon} S(\overline{s}) \\
& & \Box\ d?\overline{s} \xrightarrow{\epsilon} a!(s,x) \longrightarrow S'(s,x) \\[1em]
R(s) & = & b?(s,x) \xrightarrow{\epsilon} out!x \longrightarrow c!s \longrightarrow R(\overline{s}) \\
& & \Box\ b?(\overline{s},x) \xrightarrow{\epsilon} c!\overline{s} \longrightarrow R(s)
\end{array}$$

Given two timed media $TM1$ and $TM2$ that meet the timed translation of SM, by completeness there are two untimed media $M1$ and $M2$ which meet SM and which are refined by $TM1$ and $TM2$. Then $M1 \mathbin{|||} M2 \sqsubseteq_{UI}\mathcal{R}_{TI}\ TM1 \mathbin{|||} TM2$. Also, by Theorem 3.5, and since delays may be introduced into an untimed description to produce a timed refinement, and no use has been made of the synchronous parallel operator, we have that $S \mathbin{|||} R \sqsubseteq_{UI}\mathcal{R}_{TI}\ TS \mathbin{|||} TR$. Furthermore, both the sender and the receiver are non-retracting and prompt, and so $TS \mathbin{|||} TR$ is also non-retracting and prompt. Thus the timed network

$$TNETWORK \;=\; ((TS \mathbin{|||} TR)\ {}_{\Sigma}\|_{\{a,b,c,d\}}\ (TM1 \mathbin{|||} TM2)) \setminus \{a,b,c,d\}$$

is a timewise refinement of the untimed network, and so it must be a one-place buffer. Thus the functional correctness of the timed network may be deduced from an untimed analysis.

Of course, to do an analysis of the timing behaviour of the network it would be necessary to use the full power of the timed model. To consider the maximum time between input and output it is necessary to know for how long it is necessary to input messages into the media before output can be guaranteed; and to optimise the value of the timeout t it is necessary to know the expected delay in the media of a successfully transmitted message. The technique of timewise refinement cannot contribute to these concerns; its role is rather to complement them by allowing the appropriate use of more abstract methods for some analysis of aspects of a system's behaviour, even when other aspects require the use of the more complicated timed models.

6 Discussion

We have seen how verifications of specifications can be mapped up the CSP hierarchy of models, and also an example of how general laws might be translated. Other properties (such as deterministic or compact) do not translate in general. For example, the deterministic untimed process $a \longrightarrow Stop$ is refined by the non-deterministic timed process $a \longrightarrow Stop \sqcap Wait\ 5 ; a \longrightarrow Stop$, which can perform or refuse to perform a at time 2.

There has also been some work in this area in the contexts of timed CCS and of timed ACP. Larsen and Yi [LaY93] have proposed a notion of *time-abstracting* bisimulation, which specifies when timed processes are equivalent modulo timing behaviour. Thus one process may be used to specify simply the functional behaviour of a system by requiring that any proposed implementation should be time-abstracting bisimilar to it. They prove that time-abstracting equivalence is decidable for a timed CCS calculus [Wan90], in contrast to the refinement relation presented in this paper, which is not decidable. Interestingly, they also establish that time-abstracting *congruence* (i.e. equivalence in all contexts) is standard timed bisimulation. The corresponding result for this paper is that untimed traces congruence for timed processes is the same as (finite) timed failures equivalence.

Baeten and Bergstra [BaB92] have considered the embedding of untimed ACP into real time ACP. They propose a translation of untimed ACP into the timed setting, for example translating a to $\int_{t \geq 0} a(t)$: an untimed a process specifies nothing about the time the a should occur, so it translates to the timed process that can perform an a at any time. This is also the philosophy of this paper. They also consider the translation of certain identities of ACP into the timed framework; this supports reasoning at a higher (untimed) level of abstraction to be incorporated when detailed reasoning about timing issues is also required.

Earlier work [Sch89] investigated the relationship between the untimed models and the standard timed failures model of [Ree88]. The difficulties encountered in using that model to treat infinite behaviour led to the development of the infinite failures model, which supports a more natural treatment of timewise refinement from the untimed models.

We are also investigating other refinement relations. In particular, a relation between the failures/divergences model and the timed failures stabilities model that treats instability as divergence has that all CSP operators preserve refinement; and this refinement relation is complete for stable processes. When stability considerations are important then this relation would be the natural one to use. Of particular interest is the relationship between the timed models and the timed probabilistic models for CSP developed by Lowe [Low91]. Work has already been initiated in this direction (see e.g. [Low92]), which it seems should fit into the framework presented in this paper.

The underlying theory presented here is of course more general than simply CSP, and should be applicable wherever processes are modelled in terms of the behaviours they may exhibit. It may for example be applicable to Gerth and Kuiper's interface refinement [GKS92]. I feel that the theory will be useful only if refinement relations can be established at the syntactic level, since if refinement can be shown only by examining the semantics directly, then verifying abstract specifications of processes via refinement is unlikely to be much easier than performing the verification directly.

Acknowledgements

Thanks are due to Mike Mislove, Bill Roscoe, Mike Reed, Jim Davies, Dave Jackson, and members of the ESPRIT SPEC, REACT and CONCUR projects for useful discussions on this material. The author is funded on SERC research fellowship B91/RFH/312. Additional funding was received from EC Basic Research Action 7166 CONCUR 2, and the U.S. Office of Naval Research.

References

[BaB92] J.C.M. Baeten and J.A. Bergstra, *Discrete time process algebra*, University of Amsterdam, Report P9208b, 1992.

[BrR85] S.D. Brookes and A.W. Roscoe, *An improved failures model for communicating sequential processes*, Proceedings, Seminar on Concurrency, LNCS 197, 1985.

[BRW9x] S.D. Brookes, A.W. Roscoe, and D.J. Walker, *An operational semantics for CSP*, submitted for publication, 199x.

[ClZ92] R. Cleaveland and A.E. Zwarico, *A theory of testing for real-time*, North Carolina SU and Johns Hopkins, 1992.

[DaS92a] J.W. Davies and S.A. Schneider, *Recursion induction for real-time processes*, Formal Aspects of Computing, to appear.

[DaS92b] J.W. Davies and S.A. Schneider, *A brief history of timed CSP*, Oxford University Computing Laboratory technical monograph PRG-96 1992.

[GKS92] R. Gerth, R. Kuiper, and J Segers, *Interface refinement in reactive systems*, Proceedings of CONCUR '92, LNCS 630, 1992.

[Hen88] M. Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988.

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[LaY93] K.G. Larsen and Wang Yi, *Time abstracted bisimulation: implicit specifications and decidability*, Proceedings of MFPS '93, LNCS, 1993.

[Low91] G. Lowe, *Prioritized and probabilistic models of timed CSP*, Oxford University Computing Laboratory Technical Report PRG-TR-24-91, 1991.

[Low92] G. Lowe, *Relating the prioritized model of timed CSP to the timed failures model*, Oxford University Computing Laboratory, 1992.

[LyV92] N. Lynch and F. Vaandrager, *Forward and backward simulations for timing-based systems*, Proceedings, Real-time: theory in practise, LNCS 600, 1992.

[Mil89] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[MRS92] M.W. Mislove, A.W. Roscoe, and S.A. Schneider, *Fixed points without completeness*, submitted for publication, 1992.

[Ree88] G.M. Reed, *A Uniform Mathematical Theory for Real-Time Distributed Computing*, Oxford University DPhil thesis, 1988.

[ReR86] G.M. Reed and A.W. Roscoe, *A timed model for communicating sequential processes*, Proceedings, 13th ICALP, LNCS 226, 1986.

[Ros88] A.W. Roscoe, *Unbounded Nondeterminism in CSP*, Oxford University Computing Laboratory technical monograph PRG-67, 1988.

[Sch89] S.A. Schneider, *Correctness and Communication in Real-Time Systems*, Oxford University DPhil. thesis 1989.

[Sch92] S.A. Schneider, *Unbounded* nondeterminism for real-time processes, Oxford University Technical Report 13–92, 1992.

[Sch93] S.A. Schneider, *An operational semantics for timed CSP*, Information and Computation, to appear, 1993.

[Wan90] Wang Yi, *Real-time behaviour of asynchronous agents*, Proceedings of CONCUR '90, LNCS 458, 1990.

A Semantic models and functions

Traces

The traces model $\mathcal{M}_{UT}$ is defined to be those sets of traces that are non-empty, and closed under prefixing.

The semantic function $\mathcal{F}_{UT}$

The semantic function

$$\mathcal{F}_{UT} : CSP \longrightarrow \mathcal{M}_{UT}$$

is defined by the following set of equations:

$$\begin{array}{rcl}
\mathcal{F}_{UT}[\![Chaos]\!] & \hat{=} & \{tr \in \Sigma^*\} \\
\mathcal{F}_{UT}[\![Stop]\!] & \hat{=} & \{\langle\rangle\} \\
\mathcal{F}_{UT}[\![Skip]\!] & \hat{=} & \{\langle\rangle, \langle\surd\rangle\} \\
\mathcal{F}_{UT}[\![P \,;\, Q]\!] & \hat{=} & \{tr \mid tr \in \mathcal{F}_{UT}[\![P]\!] \wedge tr \upharpoonright \{\surd\} = \langle\rangle\} \\
 & & \cup \\
 & & \{tr_P \frown tr_Q \mid \; tr_P \frown \langle\surd\rangle \in \mathcal{F}_{UT}[\![P]\!] \wedge \\
 & & \qquad tr \upharpoonright \{\surd\} = \langle\rangle \wedge tr_Q \in \mathcal{F}_{UT}[\![Q]\!]\} \\
\mathcal{F}_{UT}[\![P \overset{t_0}{\rhd} Q]\!] & \hat{=} & \mathcal{F}_{UT}[\![P]\!] \cup \mathcal{F}_{UT}[\![Q]\!] \\
\mathcal{F}_{UT}[\![P \,\Box\, Q]\!] & \hat{=} & \mathcal{F}_{UT}[\![P]\!] \cup \mathcal{F}_{UT}[\![Q]\!] \\
\mathcal{F}_{UT}[\![a : A \longrightarrow P_a]\!] & \hat{=} & \bigcup_{a \in A}\{\langle a\rangle \frown tr \mid tr \in \mathcal{F}_{UT}[\![P_a]\!]\} \\
\mathcal{F}_{UT}[\![\sqcap_{i \in I} P_i]\!] & \hat{=} & \bigcup_{i \in I} \mathcal{F}_{UT}[\![P_i]\!] \\
\mathcal{F}_{UT}[\![P \;{}_A\|_B\; Q]\!] & \hat{=} & \{tr \in (A \cup B)^* \mid \; tr \upharpoonright A \in \mathcal{F}_{UT}[\![P]\!] \wedge \\
 & & \qquad tr \upharpoonright B \in \mathcal{F}_{UT}[\![Q]\!]\} \\
\mathcal{F}_{UT}[\![P \,|\!|\!|\, Q]\!] & \hat{=} & \{tr \mid \; \exists\, tr_P \in \mathcal{F}_{UT}[\![P]\!], tr_Q \in \mathcal{F}_{UT}[\![Q]\!] \bullet \\
 & & \qquad tr \; interleaves \; (tr_P, tr_Q)\} \\
\mathcal{F}_{UT}[\![P \setminus A]\!] & \hat{=} & \{tr \setminus A \mid tr \in \mathcal{F}_{UT}[\![P]\!]\} \\
\mathcal{F}_{UT}[\![f(P)]\!] & \hat{=} & \{f(tr) \mid tr \in \mathcal{F}_{UT}[\![P]\!]\} \\
\mathcal{F}_{UT}[\![f^{-1}(P)]\!] & \hat{=} & \{tr \mid f(tr) \in \mathcal{F}_{UT}[\![P]\!]\} \\
\mathcal{F}_{UT}[\![\mu X \circ F(X)]\!] & \hat{=} & \bigcup_{i \in \mathbf{N}} \mathcal{F}_{UT}[\![F^n(Stop)]\!]
\end{array}$$

A.1 Untimed infinite traces, failures, and divergences

The process axioms given in [Ros88] correspond to the following properties required of a set S for it to correspond to the set of observations of some process. Thus the semantic model $\mathcal{M}_{UI}$ is the collection of sets

$$S \subseteq \{f\} \times (\Sigma^* \times \mathbf{P}(\Sigma)) \cup \{d\} \times \Sigma^* \cup \{i\} \times \Sigma^\omega$$

(ordered under reverse inclusion) that meet these eight axioms.

(*1*) $(f,(s^\frown t,\{\})) \in S \Rightarrow (f,(s,\{\})) \in S$

(*2*) $(f,(t,X)) \in S \wedge Y \subseteq X \Rightarrow (f,(t,Y)) \in S$

(*3*) $(f,(t,X)) \in S \wedge \forall a \in Y \bullet (f,(t^\frown\langle a\rangle,\{\})) \notin S \Rightarrow (f,(t,X \cup Y)) \in S$

(*4*) $(d,s) \in S \Rightarrow (d,s^\frown t) \in S$

(*5*) $(d,s) \in S \Rightarrow (f,(s^\frown t,X)) \in S$

(*6*) $(i,s^\frown u) \in S \Rightarrow (f,(s,\{\})) \in S$

(*7*) $(d,s) \in S \Rightarrow (i,s^\frown u) \in S$

(*8*) $(f,(s,\{\})) \in S \Rightarrow \exists\, T \bullet (\forall\, t \in T \bullet$
$(f,(s^\frown t,\{a \mid t\langle a\rangle \notin T\})) \in S \wedge \{(i,s^\frown u) \mid u \in \overline{T}\} \subseteq S)$

Here $\overline{T} = \{u \in \Sigma^\omega \mid \forall\, t < u.t \in T\}$, where T ranges over finite prefix closed sets of finite traces.

The semantic function $\mathcal{F}_{UI}$

The function $\mathcal{F}_{UI}$ is defined in terms of three functions $\mathcal{F}_{UD}$, $\mathcal{F}_{UF}$, and $\mathcal{F}_I$, yielding divergences, failures, and infinite traces respectively. It is then given by

$$\begin{aligned}\mathcal{F}_{UI}[\![P]\!] &= \{(d,tr) \mid tr \in \mathcal{F}_{UD}[\![P]\!]\}\\ &\quad \cup\{(f,(tr,X)) \mid (tr,X) \in \mathcal{F}_{UF}[\![P]\!]\}\\ &\quad \cup\{(i,u) \mid u \in \mathcal{F}_I[\![P]\!]\}\end{aligned}$$

The semantic function $\mathcal{F}_{UD}$

The semantic function

$$\mathcal{F}_{UD} : CSP \longrightarrow \mathcal{M}_{UD}$$

is defined by the following set of equations:

$$\begin{aligned}\mathcal{F}_{UD}[\![Chaos]\!] &\mathrel{\widehat{=}} \{tr \mid tr \in \Sigma^*\}\\ \mathcal{F}_{UD}[\![Stop]\!] &\mathrel{\widehat{=}} \{\}\\ \mathcal{F}_{UD}[\![Skip]\!] &\mathrel{\widehat{=}} \{\}\\ \mathcal{F}_{UD}[\![P\,;Q]\!] &\mathrel{\widehat{=}} \{tr^\frown tr' \mid tr \in \mathcal{F}_{UD}[\![P]\!] \wedge tick \notin \sigma(tr) \wedge tr' \in \Sigma^*\}\\ &\quad \cup\\ &\quad \{tr^\frown tr' \mid (tr^\frown\langle\surd\rangle,\{\}) \in \mathcal{F}_{UF}[\![P]\!] \wedge \surd \notin \sigma(tr)\\ &\qquad\qquad \wedge\, tr' \in \mathcal{F}_{UD}[\![Q]\!]\}\\ \mathcal{F}_{UD}[\![P \overset{t_0}{\triangleright} Q]\!] &\mathrel{\widehat{=}} \mathcal{F}_{UD}[\![P]\!] \cup \mathcal{F}_{UD}[\![Q]\!]\\ \mathcal{F}_{UD}[\![P \,\Box\, Q]\!] &\mathrel{\widehat{=}} \mathcal{F}_{UD}[\![P]\!] \cup \mathcal{F}_{UD}[\![Q]\!]\end{aligned}$$

$$\mathcal{F}_{UD}[\![a : A \longrightarrow P_a]\!] \mathrel{\widehat{=}} \bigcup_{a\in A}\{\langle a\rangle^\frown tr \mid tr \in \mathcal{F}_{UD}[\![P_a]\!]\}$$

$$\mathcal{F}_{UD}[\![\textstyle\bigsqcap_{i\in I} P_i]\!] \mathrel{\widehat{=}} \bigcup_{i\in I} \mathcal{F}_{UD}[\![P_i]\!]$$

$$\begin{aligned}\mathcal{F}_{UD}[\![P\ {}_A\|_B\ Q]\!] \mathrel{\widehat{=}}\ & \{tr^\frown tr' \mid (tr \upharpoonright A, \{\}) \in \mathcal{F}_{UF}[\![P]\!] \wedge tr \upharpoonright B \in \mathcal{F}_{UD}[\![Q]\!] \\ & \quad\vee \\ & \quad tr \upharpoonright A \in \mathcal{F}_{UD}[\![P]\!] \wedge (tr \upharpoonright B, \{\}) \in \mathcal{F}_{UF}[\![Q]\!]\}\end{aligned}$$

$$\begin{aligned}\mathcal{F}_{UD}[\![P \mathbin{|\!|\!|} Q]\!] \mathrel{\widehat{=}}\ & \{tr \mid \exists\, tr_P \in \mathcal{F}_{UD}[\![P]\!], (tr_Q, \{\}) \in \mathcal{F}_{UF}[\![Q]\!] \bullet \\ & \qquad tr \textit{ interleaves } (tr_P, tr_Q)\} \\ & \cup \\ & \{tr \mid \exists (tr_P, \{\}) \in \mathcal{F}_{UF}[\![P]\!], tr_Q \in \mathcal{F}_{UD}[\![Q]\!] \bullet \\ & \qquad tr \textit{ interleaves } (tr_P, tr_Q)\}\end{aligned}$$

$$\begin{aligned}\mathcal{F}_{UD}[\![P \setminus A]\!] \mathrel{\widehat{=}}\ & \{tr \setminus A^\frown tr' \mid tr \in \mathcal{F}_{UD}[\![P]\!]\} \\ & \cup \\ & \{u \setminus A^\frown tr' \mid u \in \mathcal{F}_I[\![P]\!] \wedge \#(u \setminus A) < \infty\}\end{aligned}$$

$$\mathcal{F}_{UD}[\![f(P)]\!] \mathrel{\widehat{=}} \{f(tr)^\frown tr' \mid tr \in \mathcal{F}_{UD}[\![P]\!]\}$$

$$\mathcal{F}_{UD}[\![f^{-1}(P)]\!] \mathrel{\widehat{=}} \{tr \mid f(tr) \in \mathcal{F}_{UD}[\![P]\!]\}$$

The semantic function $\mathcal{F}_I$

The semantic function

$$\mathcal{F}_I : CSP \longrightarrow \mathcal{M}_I$$

is defined by the following set of equations:

$$\mathcal{F}_I[\![Chaos]\!] \mathrel{\widehat{=}} \{u \mid u \in \Sigma^\omega\}$$

$$\mathcal{F}_I[\![Stop]\!] \mathrel{\widehat{=}} \{\}$$

$$\mathcal{F}_I[\![Skip]\!] \mathrel{\widehat{=}} \{\}$$

$$\begin{aligned}\mathcal{F}_I[\![P \,;\, Q]\!] \mathrel{\widehat{=}}\ & \{u \mid u \in \mathcal{F}_I[\![P]\!] \wedge \surd \notin \sigma(u)\} \\ & \cup \\ & \{tr^\frown u' \mid (tr^\frown\langle\surd\rangle, \{\}) \in \mathcal{F}_{UF}[\![P]\!] \ \wedge \surd \notin \sigma(tr) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, u' \in \mathcal{F}_I[\![Q]\!]\} \\ & \cup \\ & \{tr^\frown u \mid tr \in \mathcal{F}_{UD}[\![P \,;\, Q]\!]\}\end{aligned}$$

$$\mathcal{F}_I[\![P \overset{t_0}{\rhd} Q]\!] \mathrel{\widehat{=}} \mathcal{F}_I[\![P]\!] \cup \mathcal{F}_I[\![Q]\!]$$

$$\mathcal{F}_I[\![P \mathbin{\Box} Q]\!] \mathrel{\widehat{=}} \mathcal{F}_I[\![P]\!] \cup \mathcal{F}_I[\![Q]\!]$$

$$\mathcal{F}_I[\![a : A \longrightarrow P_a]\!] \mathrel{\widehat{=}} \bigcup_{a\in A}\{\langle a\rangle^\frown u \mid u \in \mathcal{F}_I[\![P_a]\!]\}$$

$$\mathcal{F}_I[\![\sqcap_{i\in I} P_i]\!] \mathrel{\widehat{=}} \bigcup_{i\in I} \mathcal{F}_I[\![P_i]\!]$$

$$\begin{aligned}
\mathcal{F}_I[\![P\ {}_A\|_B\ Q]\!] \mathrel{\widehat{=}}\ & \{u \mid u \upharpoonright A \in \mathcal{F}_I[\![P]\!] \wedge u \upharpoonright [Q] \in \mathcal{F}_I[\![Q]\!]\} \\
& \cup \\
& \{tr^\frown u \mid tr \in \mathcal{F}_{UD}[\![P\ {}_A\|_B\ Q]\!]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_I[\![P \parallel Q]\!] \mathrel{\widehat{=}}\ & \{u \mid \exists\, u_P, u_Q \bullet (\#u_P = \infty \vee \#u_Q = \infty) \\
& \quad \wedge\ u\ \mathit{interleaves}(u_P, u_Q) \\
& \quad \wedge\ (u_P, \{\}) \in \mathcal{F}_{UF}[\![P]\!] \vee u_P \in \mathcal{F}_I[\![P]\!] \\
& \quad \wedge\ (u_Q, \{\}) \in \mathcal{F}_{UF}[\![Q]\!] \vee u_Q \in \mathcal{F}_I[\![Q]\!]\} \\
& \cup \\
& \{tr^\frown u \mid tr \in \mathcal{F}_{UD}[\![P \parallel Q]\!]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_I[\![P \setminus A]\!] \mathrel{\widehat{=}}\ & \{u \setminus A \mid u \in \mathcal{F}_I[\![P]\!] \wedge \#(u \setminus A) = \infty\} \\
& \cup \\
& \{tr^\frown u \mid tr \in \mathcal{F}_{UD}[\![P \setminus A]\!]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_I[\![f(P)]\!] \mathrel{\widehat{=}}\ & \{f(u)^\frown \mid u \in \mathcal{F}_I[\![P]\!]\} \\
& \cup \\
& \{tr^\frown u \mid tr \in \mathcal{F}_{UD}[\![f(P)]\!]\}
\end{aligned}$$

$$\mathcal{F}_I[\![f^{-1}(P)]\!] \mathrel{\widehat{=}} \{u \mid f(u) \in \mathcal{F}_I[\![P]\!]\}$$

The semantic function $\mathcal{F}_{UF}$

The semantic function

$$\mathcal{F}_{UF} : CSP \longrightarrow \mathcal{M}_{UF}$$

is defined by the following set of equations:

$$\mathcal{F}_{UF}[\![Chaos]\!] \mathrel{\widehat{=}} \{(tr, X) \mid tr \in \Sigma^* \wedge X \subseteq \Sigma\}$$

$$\mathcal{F}_{UF}[\![Stop]\!] \mathrel{\widehat{=}} \{(\langle\rangle, X) \mid X \subseteq \Sigma\}$$

$$\begin{aligned}
\mathcal{F}_{UF}[\![Skip]\!] \mathrel{\widehat{=}}\ & \{(\langle\rangle, X) \mid \surd \notin X\} \\
& \cup \\
& \{(\langle\surd\rangle, X) \mid X \subseteq \Sigma\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_{UF}[\![P\,;Q]\!] \mathrel{\widehat{=}}\ & \{(tr, X) \mid \surd \notin \sigma(tr) \wedge \\
& \qquad (tr, X \cup \{\surd\}) \in \mathcal{F}_{UF}[\![P]\!]\} \\
& \cup \\
& \{(tr^\frown tr', X) \mid \surd \notin \sigma(tr) \\
& \qquad \wedge\ (tr^\frown\langle\surd\rangle, \{\}) \in \mathcal{F}_{UF}[\![P]\!] \\
& \qquad \wedge\ (tr', X) \in \mathcal{F}_{UF}[\![Q]\!]\} \\
& \cup \\
& \{(tr, X) \mid tr \in \mathcal{F}_{UD}[\![P\,;Q]\!]\}
\end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{F}_{UF}[\![P \overset{t_0}{\triangleright} Q]\!] & \mathrel{\widehat{=}} & \mathcal{F}_{UF}[\![Q]\!] \cup \{(tr, X) \mid (tr, X) \in \mathcal{F}_{UF}[\![P]\!] \wedge tr \neq \langle\rangle\} \\
\mathcal{F}_{UF}[\![P \,\Box\, Q]\!] & \mathrel{\widehat{=}} & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathcal{F}_{UF}[\![P]\!] \cap \mathcal{F}_{UF}[\![Q]\!]\} \\
 & & \cup \\
 & & \{(tr, X) \mid tr \neq \langle\rangle \wedge (tr, X) \in \mathcal{F}_{UF}[\![P]\!] \cup \mathcal{F}_{UF}[\![Q]\!]\} \\
\mathcal{F}_{UF}[\![a : A \longrightarrow P_a]\!] & \mathrel{\widehat{=}} & \{(\langle\rangle, X) \mid X \cap A = \{\}\} \\
 & & \cup \\
 & & \bigcup_{a \in A} \{(\langle a\rangle^\frown tr, X) \mid (tr, X) \in \mathcal{F}_{UF}[\![P_a]\!]\} \\
\mathcal{F}_{UF}[\![\sqcap_{i \in I} P_i]\!] & \mathrel{\widehat{=}} & \bigcup_{i \in I} \mathcal{F}_{UF}[\![P_i]\!] \\
\mathcal{F}_{UF}[\![P \;{}_A\|_B\; Q]\!] & \mathrel{\widehat{=}} & \{(tr, Z) \mid \begin{array}[t]{l} (tr \upharpoonright A, X \upharpoonright A) \in \mathcal{F}_{UF}[\![P]\!] \wedge \\ (tr \upharpoonright B, Y \upharpoonright B) \in \mathcal{F}_{UF}[\![Q]\!] \wedge \\ (X \upharpoonright A) \cup (Y \upharpoonright B) = Z \upharpoonright A \cup B \wedge \\ tr = tr \upharpoonright (A \cup B)\} \end{array} \\
 & & \cup \\
 & & \{(tr, X) \mid tr \in \mathcal{F}_{UD}[\![P \;{}_A\|_B\; Q]\!]\} \\
\mathcal{F}_{UF}[\![P \,|||\, Q]\!] & \mathrel{\widehat{=}} & \{(tr, X) \mid \exists\, tr_P, tr_Q \bullet tr \; interleaves(tr_P, tr_Q) \\
 & & \qquad\qquad \wedge\, (tr_P, X) \in \mathcal{F}_{UF}[\![P]\!] \\
 & & \qquad\qquad \wedge\, (tr_Q, X) \in \mathcal{F}_{UF}[\![Q]\!]\} \\
 & & \cup \\
 & & \{(tr, X) \mid tr \in \mathcal{F}_{UD}[\![P \,|||\, Q]\!]\} \\
\mathcal{F}_{UF}[\![P \setminus A]\!] & \mathrel{\widehat{=}} & \{(tr \setminus A, X) \mid (tr, X \cup A) \in \mathcal{F}_{UF}[\![P]\!]\} \\
 & & \cup \\
 & & \{(tr, X) \mid tr \in \mathcal{F}_{UD}[\![P \setminus A]\!]\} \\
\mathcal{F}_{UF}[\![f(P)]\!] & \mathrel{\widehat{=}} & \{(f(tr), X) \mid (tr, f^{-1}(X)) \in \mathcal{F}_{UF}[\![P]\!]\} \\
 & & \cup \\
 & & \{(tr, X) \mid tr \in \mathcal{F}_{UD}[\![f(P)]\!]\} \\
\mathcal{F}_{UF}[\![f^{-1}(P)]\!] & \mathrel{\widehat{=}} & \{(tr, X) \mid (f(tr), f(X)) \in \mathcal{F}_{UF}[\![P]\!]\}
\end{array}
$$

The least fixed point is given by

$$\mathcal{F}_{UI}[\![\mu X \circ F(X)]\!] \;=\; \bigcap_\alpha \mathcal{F}_{UI}[\![F^\alpha(Chaos)]\!]$$

where α ranges over all ordinals; and for limit ordinals γ, we define the semantics $\mathcal{F}_{UI}[\![F^\gamma(Chaos)]\!]$ to be the least upper bound in $\mathcal{M}_{UI}$ of the set of processes $\{\mathcal{F}_{UI}[\![F^\alpha(Chaos)]\!] \mid \alpha < \gamma\}$. It is established in [Ros88] that this is well-defined.

Infinite Timed Failures

The information ordering on behaviours is defined as follows:

$$(s', \aleph') \preceq (s, \aleph) \quad \Leftrightarrow \quad \exists s'' \bullet s = s'^\frown s'' \wedge \aleph' \subseteq \aleph \lhd begin(s'')$$

We formally define $\mathcal{M}_{TI}$ to be those subsets S of $T\Sigma^{\omega}_{\leq} \times IRSET$ satisfying axioms 1–3 given below, and axiom 4 to follow.

1. $(\langle\rangle, \{\}) \in S$
2. $(s, \aleph) \in S \wedge (s', \aleph') \preceq (s, \aleph) \Rightarrow (s', \aleph') \in S$
3. $(s, \aleph) \in S \Rightarrow$

$$\begin{array}{lll} \exists \aleph' \in IRSET \bullet & \aleph \subseteq \aleph' \wedge (s, \aleph') \in S \wedge \forall (t, a) \in \mathbf{R}^+ \times \mathbf{\Sigma} \bullet & \\ (C1) & (t, a) \notin \aleph' \Rightarrow (s \lhd t^\frown\langle(t, a)\rangle, \aleph' \lhd t) \in S & \\ & \wedge & \\ (C2) & (t > 0 \wedge \neg \exists \varepsilon > 0 \bullet ((t - \varepsilon, t) \times \{a\} \subseteq \aleph')) & \\ & \qquad \Rightarrow (s \lhd t^\frown\langle(t, a)\rangle, \aleph' \lhd t) \in S & \end{array}$$

Axioms 1 and 2 require that an element of $\mathcal{M}_{TI}$ must be a non-empty downward closed set of behaviours. Axiom 3 requires that on every execution, timed events must be either possible or refusible.

A set of behaviours T is finitely variable if for every time t, the set $T \lhd t$ is a complete partial order under $\preceq$. A set of behaviours T is closed if

$$T \quad = \quad \overline{T} \quad = \quad \{(s, \aleph) \mid \forall t \bullet (s, \aleph) \lhd t \in T\}$$

Let $\mathcal{CL}$ be the set of finitely variable closed sets of behaviours satisfying axioms 1–3. Then axiom 4 states that

4. $S = \sqcap\{Q \in \mathcal{CL} \mid S \sqsubseteq Q\}$

The semantic function $\mathcal{F}_{TI}$

The semantic function

$$\mathcal{F}_{TI} : CSP \longrightarrow \mathcal{M}_{TI}$$

is defined by the following set of equations:

$$\begin{array}{rcl} \mathcal{F}_{TI}[\![Chaos]\!] & \widehat{=} & \{(s, \aleph) \mid s \in T\Sigma^{\omega}_{\leq} \wedge \aleph \in IRSET\} \\ \mathcal{F}_{TI}[\![Stop]\!] & \widehat{=} & \{(\langle\rangle, \aleph) \mid \aleph \in IRSET\} \\ \mathcal{F}_{TI}[\![Skip]\!] & \widehat{=} & \{(\langle\rangle, \aleph) \mid \checkmark \notin \sigma(\aleph)\} \\ & & \cup \\ & & \{(\langle(t, \checkmark)\rangle, \aleph) \mid t \geq 0 \wedge \checkmark \notin \sigma(\aleph \uparrow [0, t))\} \end{array}$$

$$\begin{aligned}
\mathcal{F}_{TI}[\![P\,;Q]\!] \;\widehat{=}\;& \{(s,\aleph) \mid \surd \notin \sigma(s) \;\wedge \\
&\quad (s, \aleph \cup ([0, end(s,\aleph)) \times \{\surd\})) \in \mathcal{F}_{TI}[\![P]\!] \\
&\quad \vee \\
&\quad s = s_P{}^\frown s_Q \wedge \surd \notin \sigma(s_P) \;\wedge \\
&\quad (s_Q, \aleph) - t \in \mathcal{F}_{TI}[\![Q]\!] \;\wedge \\
&\quad (s_P{}^\frown\langle(t,\surd)\rangle, \aleph \blacktriangleleft t \cup ([0,t) \times \{\surd\})) \in \mathcal{F}_{TI}[\![P]\!]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_{TI}[\![P \overset{t_0}{\triangleright} Q]\!] \;\widehat{=}\;& \{(s,\aleph) \mid begin(s) \le t_0 \wedge (s,\aleph) \in \mathcal{F}_{TI}[\![P]\!]\} \\
& \cup \\
& \{(s,\aleph) \mid begin(s) \ge t_0 \wedge (\langle\rangle, \aleph \blacktriangleleft t_0) \in \mathcal{F}_{TI}[\![P]\!] \\
&\quad\quad \wedge \\
&\quad\quad (s,\aleph) - t_0 \in \mathcal{F}_{TI}[\![Q]\!]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_{TI}[\![P \,\Box\, Q]\!] \;\widehat{=}\;& \{(\langle\rangle,\aleph) \mid (\langle\rangle,\aleph) \in \mathcal{F}_{TI}[\![P]\!] \cap \mathcal{F}_{TI}[\![Q]\!]\} \\
& \cup \\
& \{(s,\aleph) \mid s \ne \langle\rangle \wedge (s,\aleph) \in \mathcal{F}_{TI}[\![P]\!] \cup \mathcal{F}_{TI}[\![Q]\!] \\
&\quad\quad \wedge \\
&\quad\quad (\langle\rangle, \aleph \blacktriangleleft begin(s)) \in \mathcal{F}_{TI}[\![P]\!] \cap \mathcal{F}_{TI}[\![Q]\!]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_{TI}[\![a : A \longrightarrow P_a]\!] \;=\;& \{(\langle\rangle,\aleph) \mid A \cap \sigma(\aleph) = \{\}\} \\
& \cup \\
& \{(\langle(t,a)\rangle{}^\frown(s+t), \aleph) \mid \\
&\quad\quad\quad a \in A \wedge t \ge 0 \wedge A \cap \sigma(\aleph \blacktriangleleft t) = \{\} \\
&\quad\quad\quad \wedge (s, \aleph - t) \in \mathcal{F}_{TI}[\![P(a)]\!]\}
\end{aligned}$$

$$\mathcal{F}_{TI}[\![\textstyle\sqcap_{i \in I} P_i]\!] \;\widehat{=}\; \textstyle\bigcup_{i \in I} \mathcal{F}_{TI}[\![P_i]\!]$$

$$\begin{aligned}
\mathcal{F}_{TI}[\![P \;{}_A\|_B\; Q]\!] \;\widehat{=}\;& \{(s,\aleph) \mid \exists \aleph_P, \aleph_Q \bullet \\
&\quad\quad\quad \aleph \upharpoonright (A \cup B) = (\aleph_P \upharpoonright A) \cup (\aleph_Q \upharpoonright B) \\
&\quad\quad\quad \wedge s = s \upharpoonright (A \cup B) \\
&\quad\quad\quad \wedge (s \upharpoonright A, \aleph_P) \in \mathcal{F}_{TI}[\![P]\!] \\
&\quad\quad\quad \wedge (s \upharpoonright B, \aleph_Q) \in \mathcal{F}_{TI}[\![Q]\!]\,\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_{TI}[\![P \;|\!|\!|\; Q]\!] \;\widehat{=}\;& \{(s,\aleph) \mid \exists s_P, s_Q \bullet \; s \in s_P \;|\!|\!|\; s_Q \;\wedge \\
&\quad\quad\quad\quad\quad\quad\quad\; (s_P, \aleph) \in \mathcal{F}_{TI}[\![P]\!] \;\wedge \\
&\quad\quad\quad\quad\quad\quad\quad\; (s_Q, \aleph) \in \mathcal{F}_{TI}[\![Q]\!]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_{TI}[\![P \setminus A]\!] \;\widehat{=}\;& \{(s \setminus A, \aleph) \mid (s, \aleph \cup ([0,\infty) \times A) \in \mathcal{F}_{TI}[\![P]\!]\} \\
\mathcal{F}_{TI}[\![f(P)]\!] \;\widehat{=}\;& \{(f(s), \aleph) \mid (s, f^{-1}(\aleph)) \in \mathcal{F}_{TI}[\![P]\!]\} \\
\mathcal{F}_{TI}[\![f^{-1}(P)]\!] \;\widehat{=}\;& \{(s,\aleph) \mid (f(s), f(\text{[illegible]} \in \mathcal{F}_{TI}[\![P]\!]\}
\end{aligned}$$

The least fixed point is given by

$$\mathcal{F}_{TI}[\![\mu X \circ F(X)]\!] \;=\; \textstyle\bigcap_\alpha \mathcal{F}_{TI}[\![F^\alpha(Chaos)]\!]$$

where α ranges over all ordinals; and for limit ordinals γ, we define the semantics $\mathcal{F}_{TI}[\![F^{\gamma}(Chaos)]\!]$ to be the least upper bound in $\mathcal{M}_{TI}$ of $\{\mathcal{F}_{TI}[\![F^{\alpha}(Chaos)]\!] \mid \alpha < \gamma\}$. It is established in [MRS92] that this is well-defined.

Axiomatising Real-Timed Processes

Liang Chen*

Centre for Communications Research
University of Bristol
Queen's Building
Bristol BS8 1TR, Great Britain
chen@uk.ac.bristol.comms-research

Abstract. In this paper, we present a relativised compositional proof system for real-timed processes. The proof system allows us to derive statements of the form $A \vdash E = F$, where processes E, F may contain free time variables and A is a formula of the first order theory of time domain. The formula $A \vdash E = F$ means that A is a condition for process E to be bisimilar to process F. The proof system is sound and is independent of the choice of time domain, allowing time to be discrete or dense. It is complete for finite terms, i.e. terms without recursion, over dense time domains. It is also shown complete for a sublanguage over discrete time domains. We discuss how to restrict occurrences of time variables to obtain the sublanguage. We finally discuss extensions of the proof system for recursively defined processes.

1 Introduction

Process algebras, such as CCS [Mil80, Mil89], CSP [Hoa85] and ACP [BK85], are structured description languages for concurrent systems and have a variety of well developed semantics theories and verification methods. However, none of them consider temporal aspects of systems. Instead they deal with the quantitative aspects of time of systems in a qualitative way. There are many systems and applications for which purely qualitative specification and analysis are inadequate. The examples are real-time systems, such as the fault tolerant systems and safety critical systems, in which the interactions with the environments must satisfy some time constraints.

Recently there are some attempts of introducing real time in well developed process algebras [BB91, CAM90, Che92a, MT90, RR88, Wan91]. In [Che92a], we have proposed a timed calculus, Timed CCS, which is an extension of Milner's CCS with time. We make no assumption about the underlying nature of time, allowing time to be discrete or dense. The time variables in the language allow us to express a notion of time dependency which says that time for some actions depends on the happening time of their previous actions. For example, in process $a(t)_0^{15}.\, b(s)_0^{15-t}.\, nil$, the time for action b depends on the happening time of action a. For different happening time of a, the time for b is different. In [Che91a, Klu91], sound and complete proof systems for Timed CCS and a restricted language of ACPρI have

* The author is supported by grant GR/G54399 of the Science and Engineering Research Council of the UK. Most of the work was done when the author was in LFCS, University of Edinburgh.

been proposed. However the proof systems are based on some powerful infinite rules. If in Timed CCS, the corresponding rule has a form

$$\frac{\forall u.v \leq u \leq w \rightarrow E\{u/t\} = F\{u/t\}}{\alpha(t)_v^w.E = \alpha(t)_v^w.F}$$

Since time may be dense, the proof systems , although sound and complete, are only of theoretical interest.

In [Che92a], we use a notation $A \models E \sim F$ to represent that formula A is a condition for process E to be bisimilar to process F. We say A is a condition for E to be bisimilar to F if for any time instants $u_1, \cdots, u_n$, $A\{u_1/t_1, \cdots, u_n/t_n\}$ implies $E\{u_1/t_1, \cdots, u_n/t_n\} \sim F\{u_1/t_1, \cdots, u_n/t_n\}$, where A, E and F contain at most free time variables $t_1, \cdots, t_n$. In this paper, we present a relativised compositional proof system in which we derive statements of the form $A \vdash E = F$. The formula $A \vdash E = F$ means that it is provable that A is a condition for E to be bisimilar to F. The proof system is sound and is independent of the choice of time domain, allowing time to be discrete or dense. It is complete for finite processes over dense time domains. It is also shown complete for a sublanguage over discrete time domains. We discuss how to extend the proof system with some form of inductive rule for the proofs of recursively defined processes. There is no infinite proof rule in the proof system and therefore it is realistic and hopefully useful.

We mainly focus on the finite terms, i.e. those without recursions. In section 2, we give a formal description of the syntax and semantics of a simple real-time calculus. We define strong bisimulation for timed processes. In section 3, we present the proof system and show by an example how it works. In section 4, we show soundness of the proof system. We also show completeness of the proof system over dense time domains and completeness for a sublanguage over discrete time domains. We discuss how to restrict occurrences of time variables to obtain the sublanguage. Finally, we discuss in section 5 extensions of the proof system with inductive rules for the proof of recursively defined processes. We show by an example how an extended proof system works for recursively defined processes.

All proofs are omitted and can be found in [Che92b].

2 The Language

We only consider here a simple timed calculus, a sublanguage of Timed CCS [Che92a]. This is done to facilitate an elegant presentation of the key ideas of the paper. There is no difficulty in extending the ideas to include both restriction and relabelling.

2.1 The Syntax

To give a formal description of the simple timed calculus, we presuppose a set Λ, ranged over by a, b, of atomic actions not containing τ. Let Act $= \Lambda \cup \{\tau\}$, ranged over by α, β. As in CCS, Λ can be partitioned into Γ, the set of names, and $\overline{\Gamma} = \{\bar{a} \mid a \in \Gamma\}$, the set of co-names, with the provision that $\bar{\bar{a}} = a$. a and $\bar{a}$ are called complementary actions which form the basis of communications in our language, analogous to CCS. We also presuppose an infinite set V_t of time variables, ranged over by t, s, r. Let the time domain be $(\mathcal{T} \cup \{\infty\}, \leq)$, where $\mathcal{T}$ contains a least element 0 to represent the starting time and $\leq$ is a linear order over $\mathcal{T}$. Note that we make no assumption about the underlying nature of time, allowing $\mathcal{T}$ to be $\aleph$, the set of natural numbers, or $\Re^{\geq 0}$, the set of the non-negative reals. We introduce ∞ to represent infinite time, where $\infty \notin \mathcal{T}$. Our time expressions, ranged over by e, f, g, are defined as follows:

Definition 2.1

1 *for any $u \in \mathcal{T}$ and $t \in V_t$, u and t are time expressions;*
2 *for every $u \in \mathcal{T}$ and time expression e, $u \times e$ is a time expression; and*
3 *if e and f are time expressions, then $e + f$, $e \dot{-} f$, $max(e, f)$ and $min(e, f)$ are all time expressions, where $\dot{-}$ is the conditional subtraction, i.e. $e' \dot{-} e = e' - e$ whenever $e \leq e'$ and $e' \dot{-} e = 0$ whenever $e > e'$.*

Remark The decidability result of [Che92a] justifies our decision on the choices of time expressions.

By convention, for any time expression e, we have $e \leq \infty$, $\infty \circ e = \infty$, $max(e, \infty) = \infty$ and $min(e, \infty) = e$, where $\circ$ is $+$ or $\dot{-}$. We will write $e' - e$ in place of $e' \dot{-} e$ whenever $e \leq e'$.

The process expressions of the language are defined by the following BNF expressions.

$$E ::= \delta \mid nil \mid (e)E \mid \alpha(t)_e^{e'}.E \mid E + F \mid E \mid F$$

where e is a time expression and e' is a time expression or $e' = \infty$.

Note that $(\infty)E$ is not a process, but we allow us to write $(\infty)\delta$ as syntactically identical to nil.

Process δ is a dead process which neither performs any actions, nor idles. Process nil cannot do any action, but idles any time. Time prefix $(e)E$ [2] will behave as process E after a delay of time e. Action prefix $\alpha(t)_e^{e'}.E$ represents the process which can perform action α between time e and e' (inclusive), where the time variable t refer to the happening time of action α. Time variables of the language allow us to represent the notion of time dependency. For example, a system which can perform an action a followed by an action b, where a can occur at any time and if a occurs after a delay of time t then b must occur within another t time, can be expressed as a process $a(t)_0^{\infty}.b(s)_0^t\, nil$ of the language. Summation $E+F$ represents choice between processes E and F. The choice is made at the time of the first action of E or F, or at time when only one process can idle. In the later case, the process which cannot delay is dropped from the future computation. Process $E \mid F$ represents the parallel composition of processes E and F. Each of them may perform actions independently or they may synchronise on complementary actions which represent communications between them. Parallel composition is synchronous with respect to time proceeding, i.e. the parallel composition $E \mid F$ can delay time u only when both E and F can.

The action prefix operator $\alpha(t)_e^{e'}$ in $\alpha(t)_e^{e'}.E$ binds all free occurrences of time variable t in E. This gives us, in the usual sense, the notions of free and bound occurrences of time variables. We use $fv_t(E)$ to represent the set of all free time

[2] Time prefix $(e)E$ is a derivable operator of Timed CCS [Che92a].

variables occurring in E. A process E is said to be an agent if $fv_t(E) = \emptyset$. Let $\mathcal{P}$ represent the set of agents which is ranged over by P, Q, R.

2.2 The Operational Semantics

In order to define an operational semantics for the simple timed calculus, we use a labelled transition system of the form

$$(\mathcal{P}, \{\xrightarrow{\alpha}_u \cup \longrightarrow_u \mid \alpha \in \text{Act} \wedge u \in \mathcal{T}\})$$

The understanding of transition $P \xrightarrow{\alpha}_u P'$ is that agent P performs action α at time u relative to the previous action and then evolves to P'. The transition $P \longrightarrow_u P'$ means that agent P idles up to time u without any action and then evolves to P'.

To define the transition rules, we first define Moller and Tofts' maximal delay time of processes before any actions [MT90].

Definition 2.2

(1) $\lvert\delta\rvert_{\mathcal{T}} = 0$	(4) $\lvert\alpha(t)_e^{e'}.E\rvert_{\mathcal{T}} = e'$
(2) $\lvert nil\rvert_{\mathcal{T}} = \infty$	(5) $\lvert E + F\rvert_{\mathcal{T}} = max(\lvert E\rvert_{\mathcal{T}}, \lvert F\rvert_{\mathcal{T}})$
(3) $\lvert(e)E\rvert_{\mathcal{T}} = e + \lvert E\rvert_{\mathcal{T}}$	(6) $\lvert E \mid F\rvert_{\mathcal{T}} = min(\lvert E\rvert_{\mathcal{T}}, \lvert F\rvert_{\mathcal{T}})$

Table 1 presents transition rules of the language. The rules are presented in natural deduction style which are read as follows: if the transition or transitions above the inference line can be inferred, then we can infer the transition below the line. The operational semantics of the language is then given by the least transition relations $\xrightarrow{\alpha}_u$ and $\longrightarrow_u$, where $\alpha \in \text{Act}$ and $u \in \mathcal{T}$, defined in Table 1.

2.3 Strong Equivalence

We do not wish to distinguish agents which, in some sense, have the same behaviours. The notion of bisimulation between agents captures the idea of having the same behaviours. We say two agents are not equivalent if a distinction can be detected by an experimenter who interacts with each of them.

Definition 2.3 *A binary relation $\mathcal{S}$ over agents is a strong $\mathcal{T}$-bisimulation if $(P, Q) \in \mathcal{S}$ implies that for all $\alpha \in \text{Act}$ and $u \in \mathcal{T}$*

(1) *if $P \xrightarrow{\alpha}_u P'$, then there is a Q' such that $Q \xrightarrow{\alpha}_u Q'$ and $(P', Q') \in \mathcal{S}$;*
(2) *if $Q \xrightarrow{\alpha}_u Q'$, then there is a P' such that $P \xrightarrow{\alpha}_u P'$ and $(P', Q') \in \mathcal{S}$; and*
(3) $\lvert P\rvert_{\mathcal{T}} = \lvert Q\rvert_{\mathcal{T}}$.

We say two agents P and Q are strongly bisimilar, denoted by $P \sim Q$, if there is a strong $\mathcal{T}$-bisimulation $\mathcal{S}$ such that $(P, Q) \in \mathcal{S}$.

Definition 2.4 *For any processes E and F which contain at most time variables $t_1, \ldots, t_n$, we say $E \sim F$ if for any $u_1, \cdots, u_n \in \mathcal{T}$ we have*

$$E\{u_1/t_1, \cdots, u_n/t_n\} \sim F\{u_1/t_1, \cdots, u_n/t_n\}$$

$$
\begin{array}{ll}
\dfrac{}{nil \longrightarrow_u nil} & \dfrac{}{\alpha(t)^w_v . E \xrightarrow{a}_u E\{u/t\}} \quad v \leq u \leq w \\[2ex]
\dfrac{}{(v)P \longrightarrow_u (v-u)P} \quad u \leq v & \dfrac{}{\alpha(t)^w_v . E \longrightarrow_u \alpha(t)^{w-u}_{v-u} . (E\{u+t/t\})} \quad u \leq w \\[2ex]
\dfrac{P \longrightarrow_u P'}{(v)P \longrightarrow_{u+v} P'} & \dfrac{P \xrightarrow{a}_u P'}{(v)P \xrightarrow{a}_{u+v} P'} \\[2ex]
\dfrac{P \longrightarrow_u P'}{P+Q \longrightarrow_u P'} \quad |Q|_T < u & \dfrac{P \longrightarrow_u P'}{Q+P \longrightarrow_u P'} \quad |Q|_T < u \\[2ex]
\dfrac{P \xrightarrow{a}_u P'}{P+Q \xrightarrow{a}_u P'} & \dfrac{P \xrightarrow{a}_u P'}{Q+P \xrightarrow{a}_u P'} \\[2ex]
\dfrac{P \longrightarrow_u P' \qquad Q \longrightarrow_u Q'}{P+Q \longrightarrow_u P'+Q'} & \\[2ex]
\dfrac{P \longrightarrow_u P' \qquad Q \longrightarrow_u Q'}{P \,|\, Q \longrightarrow_u P' \,|\, Q'} & \dfrac{P \xrightarrow{a}_u P' \qquad Q \xrightarrow{\bar{a}}_u Q'}{P \,|\, Q \xrightarrow{\tau}_u P' \,|\, Q'} \\[2ex]
\dfrac{P \xrightarrow{a}_u P' \qquad Q \longrightarrow_u Q'}{P \,|\, Q \xrightarrow{a}_u P' \,|\, Q'} & \dfrac{P \longrightarrow_u P' \qquad Q \xrightarrow{a}_u Q'}{P \,|\, Q \xrightarrow{a}_u P' \,|\, Q'}
\end{array}
$$

Table 1. Operational Semantics

The relation $\sim$ itself is a strong $\mathcal{T}$-bisimulation, the largest strong $\mathcal{T}$-bisimulation. It is an equivalence relation, called the strong equivalence. Moreover it is a congruence relation.

To define a characteristic formula $WC(E, F)$ for processes E and F, we first introduce a notion of normal form.

Definition 2.5 *A process $E \equiv \sum_{i \in I} a_i(t_i)^{e'_i}_{e_i} . E_i + (e)\delta$ is in normal form if for any $i \in I$, $e'_i \leq e$ and E_i is also in normal form.*

We identify those formulae which are logically equivalence, i.e. two formulae A and B are identical if $A \leftrightarrow B$. Clearly, for any process E there is a normal form E' such that $E \sim E'$. The characteristic formula $WC(E, F)$ of E to be bisimilar to F is defined as follows:

Definition 2.6 *For any processes E and F, let*

$$E' \equiv \sum_{i \in I} \alpha_i(t_i)^{e'_i}_{e_i} . E_i + (e)\delta \qquad \textit{and} \qquad F' \equiv \sum_{j \in J} \beta_j(s_j)^{f'_j}_{f_j} . F_j + (f)\delta$$

be normal forms which satisfy $E \sim E'$ and $F \sim F'$. We define

$$WC(E,F) \overset{\text{def}}{=} (e = f) \wedge$$

$$\bigwedge_{i \in I} (\forall t (e_i \leq t \wedge t \leq e'_i \rightarrow \bigvee_{\substack{\alpha_i = \beta_j \\ j \in J}} (f_j \leq t \wedge t \leq f'_j \wedge WC(E_i\{t/t_i\}, F_j\{t/s_j\}))))) \quad \wedge$$

$$\bigwedge_{j \in J} (\forall s (f_j \leq s \wedge s \leq f'_j \rightarrow \bigvee_{\substack{\alpha_i = \beta_j \\ i \in I}} (e_i \leq s \wedge s \leq e'_i \wedge WC(E_i\{s/t_i\}, F_j\{s/s_j\}))))$$

It is easy to see that the characteristic formula $WC(E,F)$ of E to be bisimilar to F is well defined. Moreover we have the following property.

Proposition 2.7 *For any processes E and F, $WC(E,F)$ if and only if $E \sim F$.*

3 A Proof System

We have shown that for any processes E and F, there is a characteristic formula $WC(E,F)$ such that $E \sim F$ if and only if $WC(E,F)$. We say a formula A is a condition for E to be bisimilar to F if and only if $WC(E,F) \longrightarrow A$. We use a notation $A \models E \sim F$ to represent that the formula A is a condition for process E to be bisimilar to process F. In this section, we describe a relativised compositional proof system in which we derive statements of the form $A \vdash E = F$.

To simplify the presentation of the proof system, we first introduce a notion of time shift $e \gg E$, which is a relative version of that of [BB91].

Definition 3.1 *For any time expression e and process E, the time shift $e \gg E$ is inductively defined as follows:*

(1) $e \gg \delta \overset{\text{def}}{=} \delta$ (4) $e \gg (E + F) \overset{\text{def}}{=} e \gg E + e \gg F$

(2) $e \gg nil \overset{\text{def}}{=} nil$ (5) $e \gg (E \mid F) \overset{\text{def}}{=} e \gg E \mid e \gg F$

(3) $e \gg (\alpha(t)^{f'}_{f}.E) \overset{\text{def}}{=} \alpha(t)^{f' \dot{-} e}_{(f \dot{-} e)+(max(e,f')-f')}.(E\{e+t/t\})$

Note the subtlety in the definition for $e \gg (\alpha(t)^{f'}_{f}.E)$. It ensures that the lower bound $(f \dot{-} e) + (max(e,f') - f') > 0$ whenever $e > f'$.

Table 2 contains all axioms and Table 3 contains all proof rules. The proof rules are in the form

$$\frac{S_1 \ldots S_n}{S}$$

where $S_1, \ldots, S_n, S$ are statements. The rule can be read as: if all premises $S_1, \ldots, S_n$ can be derived, then the conclusion S can be derived.

We say $A \vdash X = Y$ is derivable if it can be derived from the axioms in Table 2 by using the proof rules in Table 3. For convenience, in the sequel, we write $A \vdash X = Y$ to assert that $A \vdash X = Y$ is derivable. We also write $\vdash E = F$ in place of $true \vdash E = F$.

$\textbf{false} \vdash X = Y$ $\qquad$ $\textbf{true} \vdash X = X$

$\textbf{true} \vdash X = X + \delta$ $\qquad$ $\textbf{true} \vdash X = X + X$

$\textbf{true} \vdash X + Y = Y + X$ $\qquad$ $\textbf{true} \vdash X + (Y + Z) = (X + Y) + Z$

$\textbf{true} \vdash X = (0)X$ $\qquad$ $\textbf{true} \vdash (e)(e')\delta = (e + e')\delta$

$\textbf{true} \vdash (e)(X + Y) = (e)X + (e)Y$ $\qquad$ $\textbf{true} \vdash \alpha(t)_e^{e'}.X = \alpha(t)_e^{e'}.X + (max(0, e'))\delta$

$e' < e \vdash \alpha(t)_e^{e'}.X = (max(0, e'))\delta$

$e \leq s \leq e' \vdash \alpha(t)_e^{e'}.X = \alpha(t)_e^{s} X + \alpha(t)_s^{e'}.X$

$\textbf{true} \vdash \alpha(t)_e^{e'}.X = \alpha(s)_e^{e'}.X\{s/t\}$ $\qquad$ **s is free for t in X**

$\textbf{true} \vdash (e)(\alpha(t)_f^{f'}.X) = \alpha(t)_{f+e}^{f'+e}.X\{t - e/t\}$ $\qquad$ $t \notin fv(e)$

Let $\quad X \equiv \sum_{i \in I} \alpha_i(t_i)_{e_i}^{e'_i}.X_i + (e)\delta$

and

$$Y \equiv \sum_{j \in J} \beta_j(s_j)_{f_j}^{f'_j}.F_j + (f)\delta$$

be normal forms, then

$$\begin{aligned}\textbf{true} \vdash X \mid Y = &\sum_{i \in I} \alpha_i(r_i)_{e_i}^{min(e'_i, f)}.(X_i\{r_i/t_i\} \mid r_i \gg Y)\\ &+ \sum_{j \in J} \beta_j(r'_j)_{f_j}^{min(f'_j, e)}.(r'_j \gg X \mid Y_j\{r'_j/s_j\})\\ &+ \sum_{i \in I \& j \in J \& \alpha_i = \bar{\beta}_j} \tau(r_{ij})_{max(e_i, f_j)}^{min(e'_i, f'_j)}.(X_i\{r_{ij}/t_i\} \mid Y_j\{r_{ij}/s_j\})\\ &+ (min(e, f))\delta\end{aligned}$$

where for any $i \in I$, $j \in J$, r_i, r'_j and r_{ij} are fresh time variables

Table 2. Axioms

1	$\dfrac{A \vdash X = Y \qquad A \vdash Y = Z}{A \vdash X = Z}$	
2	$\dfrac{A \vdash X = Y \qquad B \vdash X = Y}{A \vee B \vdash X = Y}$	
3	$\dfrac{B \vdash X = Y}{A \vdash X = Y} \qquad A \rightarrow B$	
4	$\dfrac{A \vdash X = Y}{A \wedge e = f \vdash (e)X = (f)Y}$	
5	$\dfrac{A \wedge e \leq t \leq e' \vdash X = Y}{A \wedge e = f \wedge e' = f' \vdash \alpha(t)_e^{e'}.X = \alpha(t)_f^{f'}.Y}$	$t \notin \mathrm{fv}(A)$
6	$\dfrac{A \vdash X = Y \qquad A \vdash X' = Y'}{A \vdash X + X' = Y + Y'}$	
7	$\dfrac{A \vdash X = Y \qquad B \vdash X = Z}{A \vee B \vdash X + Y + Z = Y + Z}$	

Table 3. Proof Rules

Lemma 3.2

(1) *If* $A \vdash X = Y$ *and* $t \notin \mathrm{fv}(A)$, *then* $A \wedge (e \leq t \leq e') \vdash \alpha(t)_e^{e'}.X = \alpha(t)_e^{e'}.Y$.

(2) *If* $A \vdash X = Y$ *and* $t \notin \mathrm{fv}(A)$, *then* $A \wedge e = f \wedge e' = f' \vdash \alpha(t)_e^{e'}.X = \alpha(t)_f^{f'}.Y$.

Remark Rules (1) and (2) of Lemma 3.2 are equipotent with the proof rule 5. In fact they were the version which I first proposed. Thanks to Faron Moller for suggesting the present proof rule 5.

Now we consider a simple example and show how the proof system works. In Section 5, we will consider a more interesting example and show how the proof system also works for recursively defined processes.

Example Let $E \equiv b(s)_0^{t-(2 \dot{-} t)}.\delta$, $F \equiv b(s)_0^{2t-2}.\delta$ and $G \equiv b(s)_0^{t}.\delta$, we show that

$$\vdash a(t)_1^{10}.(E + F + G) = a(t)_1^{10}.(F + G)$$

is derivable.

$$\dfrac{\vdash \delta = \delta}{\dfrac{t - (2 \dot{-} t) = 2t - 2 \vdash b(s)_0^{t-(2\dot{-}t)}.\delta = b(s)_0^{2t-2}.\delta}{t \leq 2 \vdash b(s)_0^{t-(2\dot{-}t)}.\delta = b(s)_0^{2t-2}.\delta} \qquad t \leq 2 \rightarrow (t - (2 \dot{-} t) = 2t - 2)}$$

and

$$\dfrac{\vdash \delta = \delta}{\dfrac{t - (2 \dot{-} t) = t \vdash b(s)_0^{t-(2\dot{-}t)}.\delta = b(s)_0^{t}.\delta}{t > 2 \vdash b(s)_0^{t-(2\dot{-}t)}.\delta = b(s)_0^{t}.\delta} \qquad t > 2 \rightarrow (t - (2 \dot{-} t) = t)}$$

By rule 7

$$t \leq 2 \vee t > 2 \vdash E + F + G = F + G$$

and

$$\dfrac{\dfrac{t \leq 2 \vee t > 2 \vdash E + F + G = F + G}{1 \leq t \leq 10 \vdash E + F + G = F + G} \qquad 1 \leq t \leq 10 \rightarrow (t \leq 2 \vee t > 2)}{\vdash a(t)_1^{10}.(E + F + G) = a(t)_1^{10}.(F + G)}$$

4 Soundness and Completeness

In this section, we show that the proof system is sound, i.e. whenever we have a derivation of the form $A \vdash E = F$, then formula A is a condition for E to be bisimilar to F. The soundness is independent of the choice of time domain. We also show that the proof system is complete for processes over dense time domains, but only complete for a sublanguage over discrete time domains.

4.1 Soundness

The soundness of the proof system is shown by the following proposition.

Proposition 4.1 (Soundness) *If $A \vdash E = F$, then $A \models E \sim F$.*

Note that the side condition of rule 5 is important, as otherwise the rule is invalid. As an example, let the formula A be $1 \leq t \leq 5$, the processes X and Y be $b(s)_0^{10-t}.nil$ and $b(s)_0^{t}.nil$, respectively. Clearly we have $A \wedge 5 \leq t \leq 10 \models X \sim Y$, but $A \not\models a(t)_5^{10}.X \sim a(t)_5^{10}.Y$.

4.2 Completeness in Dense Time Domains

In this section, we assume time to be dense. For example, we can assume the time domain is $(\Re^{\geq 0} \cup \{\infty\}, \leq)$. We first show that for any processes E and F, the weakest condition for E to be bisimilar to F can be written in a disjunctive normal form.

Lemma 4.2 *For any processes E and F which contain at most free time variables $t_1, \ldots, t_n$, $WC(E,F)$ can be written in a disjunctive normal form*

$$\bigvee_{i \in I} e_1^i \leq t_1 \leq f_1^i \wedge \cdots \wedge e_n^i(t_1, \ldots, t_{n-1}) \leq t_n \leq f_n^i(t_1, \ldots, t_{n-1})$$

for some finite I, where time expressions $e_k^i(t_1, \ldots, t_{k-1})$ and $f_k^i(t_1, \ldots, t_{k-1})$ ($k = 1, \cdots, n$) contain at most variables $t_1, \ldots, t_{k-1}$.

Remark For discrete time domains, the lemma in general does not hold. For example, the formula $3t = 5s$ cannot be written in the required form. In the next section, we will show how to restrict occurrences of time variables to retain the lemma for discrete time domains.

Proposition 4.3 *For any processes E and F, $WC(E,F) \vdash E = F$.*

Corollary 4.4 *$A \models E \sim F$ implies $A \vdash E = F$.*

Corollary 4.5 (Completeness) *For any processes E and F, $E \sim F$ implies $\vdash E = F$.*

4.3 Completeness in Discrete Time Domains

In this section, we assume time to be discrete, e.g. the time domain is $(\aleph \cup \{\infty\}, \leq)$, where $\aleph$ is the set of natural numbers.

As shown in the last section, Lemma 4.2 in general does not hold for discrete time domains. It is not known whether the proof system is complete for processes over discrete time domain $(\aleph \cup \{\infty\}, \leq)$. Consider processes $E \equiv a(t)^6_6.\, b(s)^{10}_{10}.(3s)\delta$ and $F \equiv a(t)^6_6.\, b(s)^{10}_{10}.(5t)\delta$. Clearly we have $\models E \sim F$, but $\vdash E = F$ is not derivable by using the above proposed technique. However, if we restrict the occurrences of time variables, we can still retain the lemma.

Notation Let e be a time expression and S be a set of time expressions, $e + S$ represents the set $\{e + f \mid f \in S\}$ of time expressions.

Definition 4.6 *For any process E, the set of time expressions $Exp_1(E)$ of E is inductively defined as follows:*

$$\begin{array}{ll} Exp_1(\delta) = \{0\} & Exp_1(a(t)^{e'}_{e}.E) = \{e, e'\} \\ Exp_1(nil) = \{0\} & Exp_1(E + F) = Exp_1(E) \cup Exp_1(F) \\ Exp_1((e)E) = e + Exp_1(E) & \end{array}$$

Definition 4.7 *For any process E, the sets of time expressions $Exp_2(E)$ and $Exp_3(E)$ of E are defined as follows:*

$$\begin{array}{ll}
Exp_2(\delta) = \emptyset & Exp_3((e)E) = Exp_1((e)E) \cup Exp_2(E) \\
Exp_3(\delta) = \emptyset & Exp_2(a(t)_e^{e'}.E) = Exp_3(E) \\
Exp_2(nil) = \emptyset & Exp_3(a(t)_e^{e'}.E) = Exp_1(a(t)_e^{e'}.E) \cup Exp_3(E) \\
Exp_3(nil) = \emptyset & Exp_2(E+F) = Exp_2(E) \cup Exp_2(F) \\
Exp_2((e)E) = Exp_2(E) & Exp_3(E+F) = Exp_3(E) \cup Exp_3(F)
\end{array}$$

Proposition 4.8 *For any normal form E, where $E \equiv \sum_{i\in I} a_i(t_i)_{e_i}^{e_i'}.E_i + (e)\delta$, we have*

$$Exp_3(E) = \bigcup_{i\in I} Exp_3(E_i) \cup \{e+0, e_i, e_i' \mid i \in I\}$$

For any set of time expressions S, let $Basic(S)$ be the set of time expressions resulted by eliminating *max* and *min* in S by the following procedure:

1 Let $Basic(S)$ be S.
2 If $max(e,f) \in Basic(S)$ or $min(e,f) \in Basic(S)$, then replace $max(e,f)$ or $min(e,f)$ by e and f in $Basic(S)$.
3 If $e \circ max(f,f') \in Basic(S)$, or $max(f,f') \circ e \in Basic(S)$, replace them by $e \circ f$ and $e \circ f'$, where $\circ$ is $+$ or $\dot{-}$.
4 If $e \circ min(f,f') \in Basic(S)$, or $min(f,f') \circ e \in Basic(S)$, replace them by $e \circ f$ and $e \circ f'$, where $\circ$ is $+$ or $\dot{-}$.
5 Repeat steps 2 to 4 until there is no occurrence of *min* and *max* in $Basic(S)$.

For any set of time expressions S, we say S only contains time expressions which have single occurrences of the same time variables if for any time expression e of $Basic(S)$, a time variable t occurs in e implies that e satisfies one of the following conditions:

(1) $e = f \circ t$ or $e = t \circ f$ for some time expression f, where $t \notin \mathrm{fv}(f)$, and $\circ$ is $+$, $-$, or $\dot{-}$.
(2) $e = e_1 \circ e_2$ for some time expressions e_1 and e_2 such that $t \notin \mathrm{fv}(e_2)$ and e_1 satisfies one of the two conditions, or $t \notin \mathrm{fv}(e_1)$ and e_2 satisfies one of the two conditions, where $\circ$ is $+$ or $\dot{-}$.

Let $\mathcal{E}'$ be a set of all processes such that for any $E \in \mathcal{E}'$, $Basic(Exp_3(E))$ only contains time expressions which have single occurrences of the same time variables. The sublanguage of $\mathcal{E}'$ is still very rich. In fact, we have the following property:

Proposition 4.9 *For any processes E and F, if $E, F \in \mathcal{E}'$, then $E + F$ is still in $\mathcal{E}'$. Also if $\{e, e'\}$ only contains time expressions which only have a single occurrence of the same time variables and $E \in \mathcal{E}'$, then $\alpha(t)_e^{e'}.E \in \mathcal{E}'$.*

Now we can show that for the processes of $\mathcal{E}'$, Lemma 4.2 of the last section still holds.

Lemma 4.10 *For any processes E and F, where $E \in \mathcal{E}'$ and $F \in \mathcal{E}'$, $WC(E,F)$ can be written in the disjunctive normal form*

$$\bigvee_{i\in I} e_1^i \le t_1 \le f_1^i \wedge \cdots \wedge e_n^i(t_1,\ldots,t_{n-1}) \le t_n \le f_n^i(t_1,\ldots,t_{n-1})$$

for some n and I, where I is finite, and $e_k^i(t_1,\ldots,t_{k-1})$, $f_k^i(t_1,\ldots,t_{k-1})$ ($k = 1,\cdots,n$) contain at most variables $t_1,\ldots,t_{k-1}$.

Proposition 4.11 *For any processes E and F of $\mathcal{E}'$, $WC(E,F) \vdash E = F$.*

Corollary 4.12 (Completeness) *For any processes E and F of $\mathcal{E}'$, if $E \sim F$, then $\vdash E = F$*

5 Proofs of Recursively Defined Processes

Up to now, we have only considered finite processes. However in [Che92a] we also allow recursively defined processes. For example, $\mu X.E$ represents an infinite process defined by an equation $X = E$. The operational rules for process $\mu X.E$ are:

$$\frac{E\{\mu X.E/X\} \longrightarrow_u P}{\mu X.E \longrightarrow_u P} \qquad \textit{and} \qquad \frac{E\{\mu X.E/X\} \xrightarrow{\alpha}_u P}{\mu X.E \xrightarrow{\alpha}_u P}$$

We say a process E is weakly guarded if every process variable of E is weakly guarded in E, where X is weakly guarded in E if every occurrence of X is in some subterm of form $\alpha(t)_e^{e'}.F$ of E. For example, the process $a(t)_0^{\infty}.X + b(s)_1^{10}.nil$ is weakly guarded. However the process $a(t)_2^{10}.X + X$ is not weakly guarded as the second occurrence of X is not guarded in it.

In this section, we consider proofs of recursively defined processes. We show by an example how the proof system also works for recursively defined processes. To do so, the proof system needs to be augmented with an axiom

$$\textit{true} \vdash \mu X.E = E\{\mu X.E/X\}$$

and some form of induction. We choose a very simple form of induction, namely *Unique Fixpoint Induction*:

$$\frac{\textit{true} \vdash P = E\{P/X\}}{\textit{true} \vdash P = \mu X.E}$$

The soundness of the axiom can be proved by showing an appropriate $\mathcal{T}$-bisimulation. For a weakly guarded process E, the soundness of the above inductive rule follows from the property of unique solution of weakly guarded equations up to strong bisimulation [Che92b]. However the inductive rule, in general, is not valid for a process E which is not weakly guarded. For example, if $E \equiv X$, then for any process P we have $P \sim X\{P/X\}$ and clearly $a(t)_0^{10}.\,nil \not\sim \mu X.X$. Also for every agent P, we have $P + a(t)_0^{10}.\delta \sim (a(t)_0^{10}.\delta + X)\{(P + a(t)_0^{10}.\delta)/X\}$, but $P + a(t)_0^{10}.\delta \not\sim \mu X.a(t)_0^{10}.\delta + X$ when we have $P \equiv b(s)_0^{10}.\delta$.

Now we consider processes

$$P \equiv \mu X.a(t)_1^{10}.(b(s)_0^{t-(2\dot{-}t)}.X + b(s)_0^{2t-2}.X + b(s)_0^{t}.X)$$

and

$$Q \equiv \mu X.a(t)_1^{10}.(b(s)_0^{2t-2}.X + b(s)_0^{t}.X)$$

and show how to derive

$$\vdash P = Q$$

in the extended proof system.

Since Q is recursively defined and weakly guarded, by the above induction rule we only need to show that

$$\vdash P = a(t)_1^{10}.\,(b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P)$$

However we have

$$\vdash P = a(t)_1^{10}.\,(b(s)_0^{t-(2\dot{-}t)}.\,P + b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P)$$

By rule 1 we only need to show

$$\vdash a(t)_1^{10}.\,(b(s)_0^{t-(2\dot{-}t)}.\,P + b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P) = a(t)_1^{10}.\,(b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P)$$

Clearly

$$\vdash P = P$$

$$\vdash b(s)_0^{2t-2}.\,P = b(s)_0^{2t-2}.\,P$$

and

$$\vdash b(s)_0^{t}.\,P = b(s)_0^{t}.\,P$$

By rule 5, we have

$$t \geq 2 \vdash b(s)_0^{t-(2\dot{-}t)}.\,P = b(s)_0^{t}.\,P$$

and

$$t \leq 2 \vdash b(s)_0^{t-(2\dot{-}t)}.\,P = b(s)_0^{2t-2}.\,P$$

By rule 7, we have

$$t \leq 2 \vee t \geq 2 \vdash b(s)_0^{t-(2\dot{-}t)}.\,P + b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P = b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P$$

Since $1 \leq t \leq 10 \rightarrow t \leq 2 \vee t \geq 2$, by the rule 3 we have

$$1 \leq t \leq 10 \vdash b(s)_0^{t-(2\dot{-}t)}.\,P + b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P = b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P$$

By rule 5, we have the result

$$\vdash a(t)_1^{10}.\,(b(s)_0^{t-(2\dot{-}t)}.\,P + b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P) = a(t)_1^{10}.\,(b(s)_0^{2t-2}.\,P + b(s)_0^{t}.\,P)$$

Remark Even untimed CCS is Turing-powerful [Mil89] and therefore no effective complete proof system can exist. By adding sufficiently powerful inductive methods for handing recursively defined processes, we would have a complete (and therefore ineffective) proof system for reasoning about real-timed processes.

6 Conclusion

In this paper, we propose a relativised compositional proof system for real-time processes. The proof system is sound and is independent of the choice of time domain, allowing time to be discrete or dense. We show that the proof system is complete for finite processes over dense time domain, but only complete for a sublanguage over discrete time domain. We discuss how to restrict the definition of time expressions to get the sublanguage. Moreover, the proof system has no infinite rules and therefore is realistic and hopefully useful.

In [ACM92], we present a timed semantics for Milner's CCS, which in fact is a partial order or true concurrency semantics. As a result, we develop a partial order or true concurrency semantics for CCS based on an interleaving approach. The proof system discussed here can also be used for a partial order or true concurrency semantics of CCS.

Although the proof system is presented for Timed CCS, the approach can also be used for some other work. As an example, the approach can be used for the restricted language of Baeten and Bergstra's $ACP\rho I$ discussed in [Klu91] (restricting to those prefixed integrations and not allowing general integration). As discussed in [Che92b], the restricted language of $ACP\rho I$ just corresponds to Timed CCS.

Recently, Hennessy [Hen91] has independently developed a proof system for reasoning about value-passing processes. The main idea of his proof system is to separate reasoning about the data from reasoning about process behaviour. In his proof system, we derive statements of the form

$$Ass \vdash P \leq Q$$

where Ass is a list of assumptions about data expressions. This statement means that whenever these assumptions are true then the process P is semantically less than or equal to the process Q.

There is a simple proof system pointed out by Kim Larsen which consists of a single rule

$$\frac{WC(E,F)}{E=F}$$

The proof system is sound and complete for finite processes. The soundness and completeness is independent of the choice of the time domain.

Acknowledgement: I would like to thank S. Anderson, K. Larsen, F. Moller, A. Munro, J. Power and P. Sewell for many helpful discussions and constructive suggestions. I would also like to thank anonymous referees for their helpful comments.

References

[ACM92] S. Anderson, L. Chen & F. Moller, *Observing Causality in Real-Timed Calculi*, Preliminary Draft, LFCS, University of Edinburgh, 1992

[BB91] J.C.M. Baeten & J.A. Bergstra, *Real Time Process Algebra*, Formal Aspects of Computing, Vol 3, No 2, pp142-188, 1991

[BK85] J.A. Bergstra & J.W. Klop, *Algebra of Communicating Processes with Abstraction*, Theoretical Computer Science 37, pp 77-12, 1985

[Che91a] L. Chen, *Specification and Verification of Real-Time Systems*, Note, 1991

[Che91b] L. Chen, *Decidability and Completeness in Real-Time Processes*, Technical Report ECS-LFCS-91-185, Edinburgh University, 1991

[Che92a] L. Chen, *An Interleaving Model for Real-Time Systems*, Proc. of Logical Foundations of Computer Science, Lecture Notes in Computer Science 620, pp 81-92, 1992

[Che92b] L. Chen, *Timed Processes: Models, Axioms and Decidability*, Ph.D Thesis, University of Edinburgh, 1992

[Che93] L. Chen, *A Model for Real-Time Process Algebras*, Proc. MFCS'93, Lecture Notes in Computer Science, 1993

[CAM90] L. Chen, S. Anderson & F. Moller, *A Timed Calculus of Communicating System*, Technical Report ECS-LFCS-90-127, University of Edinburgh, 1990

[DDM89] P. Degano, R. De Nicola & U. Montanari, *Partial Orderings Descriptions and Observations of Nondeterministics Concurrent Processes*, Lecture Notes in Computer Science 354, pp 438-466, 1989

[Hen88] M. Hennessy, *Axiomatising Finite Concurrent Processes*, SIAM J. Comput. Vol 17, No 5, pp 997-1017, 1988

[Hen91] M. Hennessy, *A Proof System for Communicating Processes with Value-Passing*, Formal Aspects of Computing, Vol. 3, No. 4, pp 346-366, 1991

[Hoa85] C.A.R. Hoare, **Communicating Sequential Processes**, Prentice-Hall international, 1985

[Klu91] A.S. Klusener, *Completeness in Real Time Process Algebra*, Proceedings of CONCUR'91, Lecture Notes in Computer Science 527, pp 96-110, 1991

[Mil80] R. Milner, *A Calculus of Communicating systems*, Lecture Notes in Computer Science 92, Springer-verlag, 1980

[Mil89] R. Milner, **Communication and Concurrency**, Prentice-Hall international, 1989

[MT90] F. Moller & C. Tofts, *A Temporal Calculus of Communicating System*, Lecture Notes in Computer Science 458, pp 401-415, 1990

[RR88] R. Reed & A. W. Roscoe, *A Timed Model for Communicating Sequential Processes*, Theoretical Computer Science, 58, pp 249-261, 1988

[Wan91] Y. Wang, *CCS + Time = an Interleaving Model for Real Time Systems*, Proc. of ICALP'91, Lecture Notes in Computer Science, 1991

A Predicative Semantics for the Refinement of Real-Time Systems

David Scholefield, Hussein Zedan, He Jifeng†

Formal Systems Research Group
Department of Computer Science
University of York, Heslington, York (UK)
†Programming Research Group
Oxford University, Keble Road, Oxford (UK)

Abstract. A formal framework for a calculus of real-time systems is presented. Specifications and program statements are combined into a single language called TAM (the Temporal Agent Model), that allows the user to express both functional and timing properties. A specification-oriented semantics for TAM is given, along with the definition of a refinement relation and a calculus which is sound with respect to that relation. A simple real-time program is also developed using the calculus.

1 Introduction

In most formal development methods there are at least two languages involved, one for the specification task, and one for the design task (often the translation to implementation is ignored, or considered to be trivial). However, an inherent problem with such a 'multi language' approach is the lack of method by which suitable designs are arrived at. A combination of experience and guess-work must be used in order to formulate a design, and then verification – a time consuming task – is undertaken. If the verification fails then the design task is undertaken again. This cycle is undergone repeatedly until verification is achieved.

To overcome this problem we have developed the 'Temporal Agent Model' (TAM) which is a theory centered around a wide-spectrum language in which both specifications and executable programs can be intermixed. A real-time functional specification in TAM is transformed step-by-step into a mixed program containing both specification fragments and executable code. Such transformations continue until a completely executable program is produced which is guaranteed correct with respect to the original specification. The program may then be analysed by run-time schedulability and allocation tools in the usual manner, and executed.

The paper introduces extensions to first-order predicate logic to cover time, a wide spectrum language with a specificational semantics, a refinement calculus, and an example of program development.

2 The TAM Philosophy

TAM aims to be a *realistic* software development method for real-time systems. It has striven to support a computational model which is amenable both to analysis by run-time execution environment software, and to efficient implementation. In doing so, TAM has not shared any of the simplifying assumptions that other techniques promote, e.g. the maximum parallelism hypothesis (there exist an infinite number of resources available to the program) [7], and the instantaneous communication assumption promoted by many real-time process algebras [4] [8].

The trade-off is that TAM can often appear complex, both in the syntax it provides for specifications, and in the discharging of proof obligations during the verification process. Thus the learning curve for TAM is very steep, but we believe that the eventual pay-off is worth the extra effort: the TAM language not only provides a method for verifying real-time and functional correctness of programs, but also provides a language of great flexibility for discussing general issues in real-time system design. This latter point has been demonstrated in publications by researchers in fields which are not mainstream real-time (for example see [6]).

The TAM theory has also been designed to support a specific development method. Many so-called formal methods only consist of a notation, and not a method which enables the user to carry out a specific list of steps in order to arrive at a correct implementation. The TAM method can be summarised as follows:

- Step 1 – the user describes timing and functional requirements in a specification language based upon simple extensions to first-order predicate logic. The specification also defines the interface between the system and the environment.
- Step 2 – the TAM theory provides a set of laws which enables the user to gradually replace parts of the specification with executable code which is guaranteed to be correct with respect to the specification. This process is known as *step-wise refinement*.
- Step 3 – eventually only executable code remains, and this is analysed by schedulability and allocation tools, compiled, and then executed.

The executable language provides real-time syntactic constructs such as deadlines and timeouts, as well as more conventional constructs such as assignment, loops, concurrent composition, communication and conditionals. There is no provision of any syntax to describe the behaviour of the resources used when the program is executed e.g. there is no syntax to describe which processor each concurrent agent should execute on. This is because we believe that the run-time execution support tools such as schedulers and task allocators, and not the programmer, should provide information such as the placement of agents: programs should be independent of such concerns. If this were not the case then the verification process would have to deal with much more complex issues such as scheduling correctness and task placement, and this would prove infeasible in any realistic computational model.

The TAM software development method therefore results in a number of concurrent agents, which are descriptions of *tasks*, and which include information such as deadlines, delays, precedence constraints etc. We then expect the run-time execution environment tools to place those agents, and decide upon their release times (within the bounds defined by the timing information inherent in the syntactic description of the agents) so as to make the schedulability test succeed. This approach makes the verification manageable, but has the drawback that the scheduler may not be able to find a schedule for the given set of agents, and the development process may be forced to backtrack to ensure that a different set of agents is produced. We are currently investigating ways in which tools such as the scheduler can produce guidance to the refinement process at a very early stage so as to avoid backtracking.

The TAM theory also aims to provide a language which supports the ways in which software engineers already produce software, rather than forcing them to change development practices to suit a particular formal approach. For this reason we provide a *wide-spectrum* language which has a syntax that supports both conventional real-time programming constructs (a kind of Pascal with deadlines, delays, timeouts, and timestamps), and a specification construct in which requirements may be written. The specification construct forms a normal part of the language and may be freely intermixed with other code. A program may then contain assertions on what the programmer requires, as well as algorithms which describe how requirements are going to be met. This language directly supports the step-wise refinement process as well as allowing for a less strict method in which some parts of systems can be specified and refined, and some parts can be written directly in code.

The TAM theory views a real-time system as a set of concurrently executing agents, each with deadline, release offsets, and period or release event. Agents communicate via shared variables called *shunts*. Shunts are time-stamped with the time of the most recent write (the programmer does not have to worry about writing the time-stamp as it is assumed that the run-time execution environment will perform this task). Shunts may only be written to by a single agent throughout the lifetime of the system (although they may be read by many). Shunts are assumed to be non-blocking on reading and writing, and therefore an agent does not have to wait for a partner in order to read or write a shunt. Agents also have local protected state. The values found in shunts and variables at the start of the system execution are nondeterministic. All agents are assumed to be terminating.

The use of timestamps in the shunts enables the user to reason about the freshness of data, and this, we believe, is one of the most important issues in real-time software design. Timestamps also provide the basic building block of real-time requirements specifications: we discuss this in detail in the next section.

3 The TAM Real-Time Logic

3.1 Overview

The TAM real-time logic is used both as a language in which to express requirements specifications, and as a formalism in which to define the semantics of the wide-spectrum language used in the TAM theory. It is constructed from conservative extensions to first-order predicate logic, and this enables the developer to use the standard first-order proof system. The logic formalises the notion of a *timed variable* which is the notation used to represent real-time program variables and shunts. Time is represented by positive integers, and a timing function is used to represent the values found in variables and shunts at a specific time. Specifications are therefore constraints on the relationship between time-stamps and values found in shunts during the lifetime of the system. Additional free variables are also provided which represent the release and termination time of the system; these variables may be predicated over in the usual way and therefore provide a mechanism for specifying duration.

The timing function is denoted '@', and is defined over pairs containing the name of a variable or shunt, and a time: thus the term '$@(X, 3)$' represents the value found in the variable (shunt) X at time=3. We usually write the term with '@' as an infix function. The projection function '$.ts$' and '$.v$' are also used to refer to the time-stamp and and value found in a shunt respectively, so we can write $s.v@t$ (the value found in shunt s at time=t), and $s.ts@t$ (the timestamp found in shunt s at time=t). The two free variables t_α and t_ω are used to denote the release time of the system, and termination time of the system respectively.

Example

Consider a simple real-time system which within 10 time units reads the integer value from a shunt called *in*, calculates the square of the number, and outputs the value to a shunt called *out*. It is assumed that the behaviour of the shunt *in* is constrained by the environment, but that the shunt *out* is entirely under the control of the system we are specifying. The liveness and timeliness property for this system is captured in the following formula:

$$\exists\sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.v@t_\omega = (in.v@\sigma)^2 \ \wedge \ t_\omega \leq t_\alpha + 10)$$

Of course it is also important that no other value is written to the shunt *out* during the execution of the system. and so we provide a safety requirement that asserts that during the execution of the system, there is only one write to the shunt *out* (*we do this by counting the number of time-stamps* which appear in *out* that are different to the time-stamp found at time=t_ω):

$$\#\{n | \exists\sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.ts@\sigma = n) \wedge n \neq out.ts@t_\omega\} \ = \ 1$$

The specification formed from the conjunction of these two formulae is much more complex than specifications commonly written for transformational systems. This is because we have to concern ourselves with the values found in the shunt *out* during the lifetime of the system rather than just at the start and end of execution; of course this is true of any specification language for reactive systems.

3.2 The TAM Logic Language

The real-time TAM logic is a multi-sorted. first-order. predicate logic, and is made up of the following symbols:

- The truth symbols *true* and *false*
- A set of variable symbols $x, y, z, \ldots$
- The duration symbols t_α and t_ω
- A set of computation variables $\mathbf{Name_V}$
- A set of computation variable variables $a, b, c, \ldots$
- A set of shunt names $\mathbf{Name_S}$
- A set of shunt name variables $s, s', \ldots$
- A set of constant symbols $\mathbf{C}$

- A set of function symbols $+, -, \times, ..$
- The timing function symbol '@' and the projection function symbols '$.ts@$' and '$.v@$'
- A set of predicate symbols $<, >, =, P, Q, ...$
- The propositional connectives $\wedge$ and $\neg$
- The universal quantifier $\forall$, and existential quantifier $\exists$
- Right and left parenthesis (,)

There are three sorts of terms, $\mathcal{N}$-terms, $Name_v$-terms and $Name_s$-terms. These terms are constructed as follows:

- The constant symbols in **C** are terms of sort $\mathcal{N}$
- The variable symbols are terms of sort $\mathcal{N}$
- The computation variable variables form terms of sort $Name_v$
- The shunt variables form terms of the sort $Name_s$
- The symbols in $\mathbf{Name_V}$ are terms of sort $Name_v$
- The symbols in $\mathbf{Name_S}$ are terms of sort $Name_s$
- If f is a function of arity=n and $t_1, .., t_n$ are terms of sort $\mathcal{N}$ then $f(t_1, .., t_n)$ is a term of sort $\mathcal{N}$
- If x is a term of sort $Name_v$ and t is a term of sort $\mathcal{N}$, then $@(x, t)$ is a term of sort $\mathcal{N}$
- If x is a term of sort $Name_s$ and t is a term of sort $\mathcal{N}$, then $.ts@(x, t)$ is a term of sort $\mathcal{N}$
- If x is a term of sort $Name_s$ and t is a term of sort $\mathcal{N}$, then $.v@(x, t)$ is a term of sort $\mathcal{N}$
- The duration symbols t_α and t_ω are terms of sort $\mathcal{N}$

Formulae are defined as follows:

- If P is an arity=n predicate, and $t_1, ..., t_n$ are terms of the appropriate sort, then $P(t_1, .., t_n)$ is a formula
- If Φ and Ψ are formulae, then $\Phi \wedge \Psi$ is a formula
- If Φ is a formula then $\neg\Phi$ is a formula
- If x is a variable symbol which occurs free and of sort $Name_s$ in Φ, then $\forall x : Shunt(\Phi)$ is a formula (similarly for *exists*)
- If x is a variable symbol which occurs free and of sort $Name_v$ in Φ, then $\forall x : Var(\Phi)$ is a formula (similarly for $\exists$)
- If x is a variable symbol which occurs free and of sort $\mathcal{N}$ in Φ, then $\forall x : \mathcal{N}(\Phi)$ is a formula (similarly for $\exists$)

3.3 The Meaning of Real-Time TAM Logic Formulae

We define three sorts in our domain of interest: the set of positive integers $\mathcal{N}$, a set of text strings $\mathbf{Strings_S}$ and a set of strings $\mathbf{Strings_V}$. Functions and predicates are given their usual interpretation, and the terms are interpreted as follows:

- Each element in the set **C** is assigned an element in $\mathcal{N}$

- Each element in the set $\mathbf{Name_S}$ is assigned a unique text string in $\mathbf{Strings_S}$ which corresponds the the symbol found in name (e.g. $s \in Name$, is assigned the string 's')
- Each element in the set $\mathbf{Name_V}$ is assigned a unique text string in $\mathbf{Strings_V}$
- The duration symbols are assigned values from $\mathcal{N}$.
- The function '@' is assigned a value from the function space $[[\mathbf{Strings_S} \cup \mathbf{Strings_V} \times \mathcal{N}] \rightarrow \mathcal{N}]$
- The functions '$.ts$@' and '$.v$@' are each assigned a value from the function space $[[\mathbf{Strings_S} \times \mathcal{N}] \rightarrow \mathcal{N}]$

Given an interpretation $\mathcal{I}$ with the structure defined above, then we define satisfaction ($\models$) as follows:

$\mathcal{I} \models \Phi \wedge \Psi$	iff $\mathcal{I} \models \Phi$ and $\mathcal{I} \models \Psi$
$\mathcal{I} \models \neg\Phi$	iff not $\mathcal{I} \models \Phi$
$\mathcal{I} \models P(t_1, \ldots t_n)$	iff $\mathcal{I}[P](\mathcal{I}[t_1], \ldots \mathcal{I}[t_n])$
$\mathcal{I} \models \forall x : Shunt(\Phi)$	iff for every element s in $\mathbf{Strings_S}$ we have $\mathcal{I} \models \Phi[s/x]$
$\mathcal{I} \models \forall x : Var(\Phi)$	iff for every element v in $\mathbf{Strings_V}$ we have $\mathcal{I} \models \Phi[v/x]$
$\mathcal{I} \models \forall x : \mathcal{N}(\Phi)$	iff for every element n in $\mathcal{N}$ we have $\mathcal{I} \models \Phi[n/x]$
$\mathcal{I} \models \exists x : \mathcal{T}(\Phi)$	iff $\mathcal{I} \models \neg\forall x : \mathcal{T}(\neg\Phi)$ where $\mathcal{T}$ is $\mathcal{N}$, $Shunt$, or Var

We also assume the usual shorthand notation for defining disjunction, implication, and existential quantification. We use the '@' in infix form, and write $.ts@(s,t)$ as $s.ts@t$ and similarly for '$.v$@'. In addition we shall use the notation $s@n = s@m$ as a shorthand notation for $s.v@n = s.v@m \ \wedge \ s.ts@n = s.ts@m$, and the notation $s@n = (x, y)$ for $s.ts@n = x \ \wedge \ s.v@n = y$. We also assume the usual notation for indexed (finite) conjunction and disjunction.

3.4 Axioms

We can rely on the fact that writing to a shunt will cause the timestamp in the shunt to update appropriately.

(Freshness) $\forall s : Shunt(\forall t : \mathcal{N}(s.v@t \neq s.v@t - 1 \ \Rightarrow \ s.ts@t = t))$

We can also rely on the termination time of a system to be at, or after, the release time.

(Duration) $t_\omega \geq t_\alpha$

4 The TAM Language

The TAM language contains both a specification statement syntax, and a syntax for an imperative style real-time programming language, which we shall refer to as the *concrete* syntax. The major difference between the specification statement and the concrete syntax is that variables and shunts are referred to in the concrete syntax without reference to the timing function – the semantics are responsible for the association between the variables and shunts at the concrete level and the timed variables and shunts in the underlying logic. The syntax can be defined in terms of agents by the following table:

Agent form	Name
$w : \Phi$	Specification
$x := e$	Assignment
$(x, y) — s$	Input
$x — s$	Output
$\mathcal{A}; \mathcal{B}$	Sequence
$\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i$	Conditional
$\mathcal{A} \vert \mathcal{B}$	Concurrent
$[S]\mathcal{A}$	Deadline
$\mathcal{A}/s$	Restriction
$(x)\mathcal{A}$	Local
$\mathcal{A} \rhd^n_s \mathcal{B}$	Signal
$\mu_n \mathcal{A}$	Iteration

Where w is a set of shunt and variable names, Φ is a real-time TAM logic formula, e is some term on computation variables which evaluates to a value in $\mathcal{N}$, x and y are terms of sort $Name_v$, s is a term of sort $Name_s$. $\mathcal{A}$ and $\mathcal{B}$ are agents, I is some finite indexing set, g_i are boolean expressions (predicates) on shunt names and computational variable names, S is a set of values from $\mathcal{N}$, and n is a value from $\mathcal{N}$

The semantics and well-formedness conditions can be informally described as follows:

Specification

The user of the TAM theory is expected to keep account of the environment of each agent, i.e. the set of variable and shunt names which may be written to by each agent. The frame 'w' in the specification agent denotes those variables and shunts which are in the environment of the specification and which may have their value changed by an agent which is a valid refinement of the specification. Thus, any variable or shunt which is in the agent's environment, but which is not in the frame, can be assumed to remain stable during the execution of the agent.

The real-time TAM logic formula Φ provides a specification of the behaviour of the system which will eventually result from the refinement of $w : \Phi$. This is usually achieved by constraining the values and timestamps found in the shunts that appear in w, and by predicating over the values of t_α and t_ω.

Assignment

The expression ϵ is a term within which variables may be referred to without reference to the timing function. This is because the application of the timing function with the time equal to the release time of the assignment agent will be assumed. Thus the agent form:

$$x := x + y$$

(where x and y are computation variable names) will be translated by the semantics into the constraint:

$$x@t_\omega = x@t_\alpha \; + \; y@t_\alpha$$

Note that because the variables x and y must belong to the assignment agent, and no concurrently executing agent can write to those variables it does not matter when the values are read. thus reading them at time=t_α is simply a notational convenience. The assignment statement will also take some time in which to execute, and *no* assignment will be instantaneous (even assignments of the form $x := x$).

Input

The input statement reads the timestamp and value from a shunt at the same time. The timestamp is read into the left variable, and the value into the right. The read is asynchronous i.e. it does not need a partner (doing the writing) with which to synchronise. The reading cannot be instantaneous. The read occurs sometime between the release and termination of the read agent, but the user can not depend upon a particular instant (unless he constrains the reading further by deadlines and delays etc). If there has been no write to the shunt before the read takes place then the value and timestamp can have any value from $\mathcal{N}$.

Output

The output statement writes the value given into the shunt. The value can be any positive integer constant, or the value found in a computational variable. The write occurs sometime between the release time and termination time of the write agent, but the user can not depend upon a specific time (unless he specifically constrains the write with deadlines and delays etc). The run-time system is responsible for writing the current time as a timestamp into the shunt at the same time as the value is written. The writing is asynchronous in that the output agent does not have to wait for a partner (doing the reading) in order to write to the shunt. The shunt is assumed to have been written to before the output agent terminates.

Sequence

The termination time of the first agent becomes the release time of the second. Thus sequencing defines a precedence relation not a physical operation, and the transfer of control is assumed to occur instantaneously. The final instant of the first agent is also the first instant of the second, and this initially suggests the scheduling problem of the 'dangling drop' (i.e. the extreme case where the first task is pre-empted at the point where it has actually completed all of its computation, but has not yet told the run-time system; technically the deadline of the first task may be missed even though it has completed all of its useful work). This however is not a problem as an idle delay tick may be inserted at the release time of the second agent. This is consistent with the semantics: no agent may do any communication on its release instant and so it can be assumed to idle.

Conditional

Each of the boolean expressions are evaluated, and the agent corresponding to a true value will be executed immediately. The expressions can be any predicate (with shunts and variable names occurring untimed). If none of the boolean expressions are true then the agent terminates immediately, and if more than one is true then a non-deterministic choice is made between them. The semantics do not assume that the evaluation of the conditionals requires any computational resources, and this enables the user to encode idle polling. However, care needs to be taken during refinement to ensure that computationally expensive evaluation is not constrained into an infeasibly small interval: this is discussed in more detail in section 6. As with the assignment, the values at the release time of the agent are used (i.e. shunt values at the release time of the conditional agent are used – this must also be taken into account when conditional evaluation is considered). We use the notation $g \Rightarrow \mathcal{A}$ when only one conditional is present.

Concurrency

Concurrency is distributed and so the concurrent composition terminates when both agents have terminated. Concurrent agents should not write to the same shunt, and should not refer to the same computation variables. An attempt to do so may result in the system aborting with unpredictable results. We use the shorthand notation $\prod_{i \in I} \mathcal{A}_i$ for indexed concurrency. Concurrency should be seen as a declaration of the lack of precedence constraints between two agents: if those agents are representing tasks then the concurrency operator declares that those two tasks can be independently scheduled.

Deadline

It is assumed that the duration of the agent (the difference between release and termination time) is equal to one of the values found in the set S. Thus the deadline agent forms a constraint upon the run-time execution environment that the refinement calculus and the system developer can depend upon being met.

Restriction

The shunt in the restriction becomes 'hidden' from the rest of the system and may only be written and read by the agent specified. This operator becomes useful when program equivalence is being proven: it is possible to prove that a program which uses a number of concurrently executing agents, communicating via hidden shunts, is equivalent to an agent with no apparent internal structure.

Local
The variable in the local declaration becomes 'hidden' and the rest of the system (i.e. the sequential agents which follow the agent with the hidden variable) may not read it or write to it. We use the shorthand notation $(x, y)\mathcal{A}$ in place of $(x)(y)\mathcal{A}$ etc. for clarity.

Signal
The given shunt is treated as a signal, and is monitored from the release time for the number of time units specified. If the shunt is written to in that interval then the agent on the right is released with a release time equal to that of the first write to the shunt, otherwise the agent on the left is released at the end of the interval.

Iteration
The specified agent will be executed in sequence the given number of times. This agent is simply used as a shorthand for long sequences of task executions.

4.1 Semantics

We start by defining some useful predicates on shunts and variables. The predicate '*stable*' asserts that the shunt s will not be changed during the given interval:

Definition Stable (shunts) $stable(s, n, m) =_{def} \bigwedge_{\sigma \in (n,m]} s @ \sigma = s @ (\sigma - 1)$

Similarly for variables:

Definition Stable (variables) $stable(x, n, m) =_{def} x @ m = x @ n$

In addition, the definitions for *stable* are extended to sets of variables or shunts.

The predicate *write* asserts that a given value is written to a shunt within an interval, and that the shunt remains stable at all other times within the interval:

Definition Write $write(x, s, n, m) =_{def}$

$$\bigvee_{\sigma \in (n,m]} stable(s, n, \sigma - 1) \wedge s @ \sigma = (\sigma, x @ n) \wedge stable(s, \sigma, m)$$

We also define an operator for dividing formulae into time consecutive subformulae:

Definition Chop. Given two timed logic formulae $\mathcal{A}$ and $\mathcal{B}$, then,

$$\mathcal{A} \frown \mathcal{B} =_{def} \exists m : \mathcal{N}(m \in [t_\alpha, t_\omega] \wedge \mathcal{A}[m/t_\omega] \wedge \mathcal{B}[m/t_\alpha])$$

The semantics of an agent are now given by a timed logic formula. The specification statement is defined in this manner also, giving a natural interpretation for a refinement relation. Note that we use the notation $\vec{w}$ to denote those variables and shunts which are owned by the specification agent, but which do not appear in the frame, and the notation Π to denote the agent $\emptyset : true$ (a particularly useful agent). Figure 1 gives the definition of the semantics. We also assume that the set of variables and shunts owned by the agent $\mathcal{A}|\mathcal{B}$ is partitioned into disjoint environments for $\mathcal{A}$ and $\mathcal{B}$; the identification of environments with agents is informal, and it is the responsibility of the refiner to decide upon suitable partitions[1].

$$\llbracket w : \Phi \rrbracket =_{def} stable(\bar{w}, t_\alpha, t_\omega) \wedge \Phi \qquad \llbracket x := e \rrbracket =_{def} \llbracket \{x\} : (t_\alpha < t_\omega \wedge x@t_\omega = e@t_\alpha) \rrbracket$$

$$\llbracket x \rightarrow s \rrbracket =_{def} \llbracket \{s\} : write(x, s, t_\alpha, t_\omega) \rrbracket \qquad \llbracket \mathcal{A}/s \rrbracket =_{def} \exists x : Shunt(\llbracket \mathcal{A} \rrbracket[x/s])$$

$$\llbracket (x, y) \leftarrow s \rrbracket =_{def} \llbracket \{x, y\} : t_\alpha < t_\omega$$

$$\wedge \exists m : \mathcal{N}(m \in (t_\alpha, t_\omega] \wedge x@t_\omega = s.ts@m \wedge y@t_\omega = s.v@m) \rrbracket$$

$$\llbracket (x)\mathcal{A} \rrbracket =_{def} \exists y : Var(\llbracket \mathcal{A} \rrbracket[y/x])$$

$$\llbracket \mathcal{A}|\mathcal{B} \rrbracket =_{def} (\llbracket \mathcal{A} \rrbracket \frown \llbracket \Pi \rrbracket) \wedge (\llbracket \mathcal{B} \rrbracket \frown \llbracket \Pi \rrbracket)$$

$$\llbracket [S]\mathcal{A} \rrbracket =_{def} \llbracket \mathcal{A} \rrbracket \wedge t_\omega - t_\alpha \in S \qquad \llbracket \mathcal{A} : \mathcal{B} \rrbracket =_{def} \llbracket \mathcal{A} \rrbracket \frown \llbracket \mathcal{B} \rrbracket$$

$$\llbracket \bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \rrbracket =_{def} (((\bigwedge_{i \in I} \neg g_i @t_\alpha) \wedge \llbracket \Pi \rrbracket) \vee \bigvee_{i \in I} (g_i @t_\alpha \wedge \llbracket \mathcal{A}_i \rrbracket)))$$

$$\llbracket \mathcal{A} \rhd_n^s \mathcal{B} \rrbracket =_{def} \text{Nowrite} \frown \llbracket \mathcal{A} \rrbracket \ \vee \ \text{Input} \frown \llbracket \mathcal{B} \rrbracket$$

$$\llbracket \mu_{n+1} \mathcal{A} \rrbracket =_{def} (\llbracket \mathcal{A} \rrbracket) \frown \llbracket \mu_n \mathcal{A} \rrbracket \qquad \llbracket \mu_0 \mathcal{A} \rrbracket =_{def} \llbracket \Pi \rrbracket$$

$$\text{Nowrite} =_{def} \llbracket \emptyset : t_\omega = t_\alpha + n \ \wedge \ s.ts@t_\alpha < t_\alpha \ \wedge \ stable(s, t_\alpha, t_\omega) \rrbracket$$

$$\text{Input} =_{def} \llbracket \emptyset : t_\omega \in [t_\alpha, t_\alpha + n] \ \wedge \ s.ts@t_\omega = t_\omega \ \wedge \ \forall m \in [t_\alpha, t_\omega)(s@m = s@t_\alpha) \rrbracket$$

Fig. 1. TAM Semantics

5 Standard Agent Definitions

In a number of TAM publications, definitions for agents which capture real-time behaviour succinctly have been proposed, these definitions have been syntactic sugar for complex concrete agent forms. In this section we define the standard agent forms for periods, delays, finding the current time (within bounds), and managing mutual exclusion.

[1] The partitioning is almost always obvious as the frame of any specification agent dictates the minimum contents of the specification agent's environment. Those variables or shunts which are not changed by either of the two concurrently executing agents may appear in either environment.

5.1 Minimum Delays

A delay agent does not change any variable or shunt value, and guarantees not to terminate for a minimum duration (i.e. models an idle delay). Given a duration $n \in \mathcal{N}$, then we define the delay agent as:

$$\delta n =_{def} \emptyset : t_\omega \geq t_\alpha + n$$

A specific length delay (which might be used to model task offsets for instance) can be defined by:

$$\Delta n =_{def} [\{n\}]\delta n$$

This agent will guarantee to terminate at exactly n time units after release.

5.2 The Current Time

Consider the following agent which writes to a private shunt and then reads the timestamp which was just written:

$$(0 - s; (ts, x) - s)/s; \mathcal{A}$$

The value found in the variable ts will provide a *lower bound* only on the current time. This is because the agent may have been pre-empted between the writing of the shunt and the reading, or between the reading and the use of the timestamp in agent $\mathcal{A}$.

However, consider the following agent which performs the same task, but within a tight deadline:

$$[m]((0 - s; (ts, x) - s; \mathcal{A})/s)$$

The user knows that in the agent $\mathcal{A}$, the value found in the variable ts will have a bound on freshness, i.e. he will know that the current time is somewhere between ts and $ts + m$.

5.3 Specific Deadlines

We overload the deadline operator to define a more restricted deadline. We use the notation $[n]\mathcal{A}$ (where n is a single positive integer value) to denote the agent $[\,[0..n]\,]\mathcal{A}$.

5.4 Periodic Agents

The periodic agent can now be defined. Given an agent $\mathcal{A}$, a period=T, a deadline=D, and a number of periods=n, then:

$$Period(\mathcal{A}, T, D, n) = \mu_n([T]([D]\mathcal{A}|\delta T))$$

In this definition we assume that $D \leq T$. if we remove this constraint then we have a more general periodic agent:

$$Period(\mathcal{A}, T, D, n) \;=\; \prod_{i \in [1..n]} \Delta(T \times i); [D]\mathcal{A}$$

But note that in overlapping agents we have to consider certain constraints on the possibility in the overlap of writing to the same shunt, this is discussed in more detail in section 7.

5.5 Critical Sections

Consider the case when two (or more) agents wish to write to the same shunt (i.e. share the same resource), and the system has to enforce mutual exclusion. We can model this by attaching a *semaphore-dispatch* agent to the shared shunt which communicates to the requesting tasks via three sets of shunts: Req_1, Req_2 are written to by the requesting agents when they require exclusive access to the shunt. Gnt_1, Gnt_2 are used to grant access (the semaphore-dispatch agent writes to the shunt of the agent requesting access), and Rel_1, Rel_2 are used to release the semaphore and are written by the requesting agent when it has finished. We also assume a priority of agent 1 over agent 2 if two requests for access are made at the same time. The semaphore-dispatch agent is assumed to grant up to n semaphores (or ticks for which the semaphore is not granted) before it removes the resource from the system. We also place a bound on the waiting time for the release signal of m time units (in case an agent 'hangs' without releasing): this is acceptable only if we choose an m which is greater than the deadline of both requesting agents.

We can write the semaphore-dispatch agent as follows:

$$\mu n((\Delta 1 \rhd_0^{Req_2} [1](0 - Gnt_2); R(2)) \rhd_0^{Req_1} [1](0 - Gnt_1); R(1))$$

$$R(n) =_{def} \Delta 0 \rhd_m^{Rel_n} \Delta 0$$

The behaviour of this agent can be read as follows. The first signal to be tested is that for Req_1 (the request for access from agent 1). the test will only be for a single instant in time. If the agent is signalling then the grant shunt will be written to within one tick, and the semaphore-dispatch agent then behaves like $R(1)$ which is waiting for the release from agent 1. When the release comes (or the signal times out) the semaphore-dispatch agent returns to the waiting state. If the signal from agent 1 is not written then the signal from agent 2 is tested. if it is being written then the same behaviour occurs as for agent 1. if it is not being written then the semaphore-dispatcher waits for the next tick and starts the signal monitoring again. Note that semaphore requests which occur during the wait for a release signal will be ignored, and thus signalling must be repeated by the requesting agent periodically. Also, the grant signal time is bounded by the deadline on the grant write (in this instance this is a single tick).

Note that this semaphore-dispatcher method of dealing with mutual exclusion is only one of a possible number of solutions to the problem. Consider the solution whereby requests are not lost, and are guaranteed to be serviced when the resource next becomes free. This is achieved by the semaphore-dispatcher updating local variables rt_1 and rt_2 with the timestamp of the most recent request of each agent:

$$\begin{aligned}
&(x, y, rt_1, rt_2)(\ [1]((rt_1, x) - Req_1 | (rt_2, y) - Req_2): \\
&\qquad \mu_n\ (\ Req_1.ts > rt_1 \Rightarrow P(1) \\
&\qquad\quad \sqcup \\
&\qquad\quad Req_2.ts > rt_2 \Rightarrow P(2) \\
&\qquad\quad \sqcup \\
&\qquad\quad Req_1.ts = rt_1 \wedge Req_2.ts = rt_2 \Rightarrow \Delta 1 \\
&\qquad))
\end{aligned}$$

$$P(1) =_{def} [1]((rt_1, x) - Req_1); [1](0 - Gnt_1); (\Delta 0 \triangleright_m^{Rel_1} \Delta 0)$$

$$P(2) =_{def} [1]((rt_2, y) - Req_2); [1](0 - Gnt_2); (\Delta 0 \triangleright_m^{Rel_2} \Delta 0)$$

In this solution there is idle waiting on the release signals, and on the request signals (in this special instance – where the conditions are expressions on shunt timestamps – we can assume that a conditional agent which does nothing but execute an idle delay is implemented by an agent which idles until the condition holds). Note that again, the deadlines on grant signals could be slackened if necessary.

6 The Refinement Calculus

The aim of a refinement calculus is to provide a set of syntactic rewrite rules which enables the software developer to transform a requirements specification into an executable program. The calculus must ensure that any program resulting from the application of laws must be correct with respect to the original specification. Correctness is defined in terms of a refinement relation, and individual refinement laws are proven sound with respect to this relation.

We define a refinement relation as follows:

Definition Refinement. Given two agents $\mathcal{A}$ and $\mathcal{B}$, then $\mathcal{B}$ refines $\mathcal{A}$ (written $\mathcal{A} \sqsubseteq \mathcal{B}$) exactly when $[\![\mathcal{B}]\!] \Rightarrow [\![\mathcal{A}]\!]$.

Refinement can be seen as a lessening of nondeterminism, i.e. given a specification which lists a number of acceptable alternatives (which might be inherent in the underspecification of a system), then a program which guarantees at least one of those alternatives is a valid refinement.

It is clear that the refinement relation is a partial order (a property inherited from the implication connective), and this will allow us to perform refinement steps of any granularity without affecting the resulting program.

If we return to our early example of a system specification, we are now in a position to prove the refinement:

$$\begin{aligned}
\{out\} :& \exists \sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.v@t_\omega = (in.v@\sigma)^2 \wedge t_\omega \leq t_\alpha + 10) \\
& \wedge \#\{n | \exists \sigma \in \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.ts@\sigma = n) \wedge n \neq out.ts@t_\omega\} = 1
\end{aligned}$$

$$\sqsubseteq \ (x, y)[10]((x, y) - in; y^2 - out)$$

The proof sketch is as follows:

$$
\begin{aligned}
&[\![(x,y)[10]((x,y) - in; y^2 - out)]\!] \\
&\quad = \exists x, y : Var(\\
&\qquad \exists m : \mathcal{N}(m \in [t_\alpha, t_\omega] \wedge \bigvee_{\sigma \in (t_\alpha, m]} (x @ t_\omega = in.ts @ \sigma \wedge y @ t_\omega = in.v @ \sigma) \\
&\qquad\qquad \wedge stable(out, t_\alpha, m) \\
&\qquad\qquad \wedge write(y @ m^2, out, m, t_\omega) \wedge t_\omega \leq t_\alpha + 10)
\end{aligned}
$$

We can see that:

$$
\begin{aligned}
&stable(out, t, m) \wedge write(y @ m^2, out, m, t) \Rightarrow \\
&\quad \#\{n | \exists \sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.ts @ \sigma = n) \wedge n \neq out.ts @ t_\omega\} \; = \; 1
\end{aligned}
$$

by the definition of *stable* and *write*. The liveness property is guaranteed by the theorem:

$$
\begin{aligned}
&\exists x, y : Var(\exists m : \mathcal{N}(m \in [t_\alpha, t_\omega] \wedge \bigvee_{\sigma \in (t_\alpha, m]} (x @ t_\omega = in.ts @ \sigma \wedge y @ t_\omega = in.v @ \sigma) \\
&\qquad\qquad \wedge write(y @ m^2, out, m, t_\omega) \wedge t_\omega \leq t_\alpha + 10) \\
&\Rightarrow \exists \sigma : \mathcal{N}(\sigma \in [t_\alpha, t_\omega] \wedge out.v @ t_\omega = (in.v @ \sigma)^2 \wedge t_\omega \leq t_\alpha + 10) \qquad \Box
\end{aligned}
$$

It would be infeasible to prove refinements of any reasonable sized program in this manner, the complexity of such a task would ensure that mistakes would be made. Instead we provide a calculus of refinement laws which have much simpler proof obligations. We define a subset of these laws below, each law is labeled so that it can be referred to in refinement proofs.

We start with a refinement law schema which arises from the semantic definitions.

RS (refinement schema) If $[\![\mathcal{A}]\!] =_{def} \Phi$ then $\Phi \sqsubseteq \mathcal{A}$

We now define the laws of the calculus by construct, we use the equality symbol '=' to denote that refinement is valid in both directions.

Specification

SR.1 $w : \Phi \;\; = \;\; w : \Phi[x @ t_\alpha / x @ t_\omega]$ (if $x \notin w$)

SR.2 $w : \Phi \;\; = \;\; w : \Phi[x @ t_\omega / x @ t_\alpha]$ (if $x \notin w$)

SR.3 $w \cup \{s\} : \Phi \; \wedge \; stable(s, t_\alpha, t_\omega) \;\; = \;\; w - s : \Phi$

(if s not in Φ)

SR.4 $w : \Phi \;\; \sqsubseteq \;\; w : \Phi'$ (if $\Phi' \Rrightarrow \Phi$)

Restriction

For any variable x:

$VR.1\ (x)\mathcal{A} \ = \ \mathcal{A}$ (if x not in $\mathcal{A}$)

$VR.2\ (x)((y)\mathcal{A}) \ = \ (y)((x)\mathcal{A})$

$VR.3\ w : (\exists y : Var(\Phi)) \ = \ (x)(w \cup \{x\} : \Phi[x/y])$ (if $x \notin w : \Phi$)

$VR.4\ w : \Phi \ \sqsubseteq \ (x)w \cup \{x\} : \Phi$

(if x is new unique computational variable)

Sequential

SE.1 $P;(Q;R) \ = \ (P;Q);R$

Deadline

DE.1 $[S]\mathcal{A} \sqsubseteq [S']\mathcal{A}$ (if $S' \subseteq S$)

Delay

DL.1 $\delta n;\delta m \ = \ \delta n+m$ DL.2 $\Delta n;\Delta m \ = \ \Delta n+m$

DL.3 $\Delta n;\amalg \ = \ \delta n$ DL.4 $\delta 0 \ = \ \amalg$

Concurrent

CR.1 $\mathcal{A}|\mathcal{B} \ = \ \mathcal{B}|\mathcal{A}$ CR.2$\mathcal{A}|(\mathcal{B}|\mathcal{C}) \ = \ (\mathcal{A}|\mathcal{B})|\mathcal{C}$

Conditional

GR.1 $\bigsqcup_{i \in I} g_i \Rightarrow \mathcal{A}_i \ \sqsubseteq \ \mathcal{A}_j$ (if $g_j \equiv true$)

Signal

For any shunt s and time n:

TR.1 $(\mathcal{A} \triangleright^s_n \mathcal{B})|(\mathcal{C} \triangleright^s_n \mathcal{D}) \ = \ (\mathcal{A}|\mathcal{C}) \triangleright^s_n (\mathcal{B}|\mathcal{D})$

TR.2 $\mathcal{A} \triangleright^s_{n+1} \mathcal{B} \ = \ (\mathcal{A} \triangleright^s_n \mathcal{B}) \triangleright^s_1 \mathcal{B}$

TR.3 $\mathcal{A} \triangleright^s_n (\mathcal{C} \triangleright^s_0 \mathcal{B}) \ = \ \mathcal{A} \triangleright^s_n \mathcal{B}$

Iteration

IR.1 $\mu_{n+1}\mathcal{A} \;=\; \mathcal{A};\mu_n\mathcal{A};\amalg \;=\; \mu_n\mathcal{A};\mathcal{A};\amalg$

Of course there are many such useful rules, and the reader can probably think of many more. The refinement calculus needs to be proven sound with respect to the definition of the refinement relation, and the proof is presented in [10]. In addition, the refinement relation needs to be proven monotonic, i.e.

Monotonicity Given that $\mathcal{A} \sqsubseteq \mathcal{B}$, then for any context $\mathcal{C}[\_]$, we have $\mathcal{C}[\mathcal{A}] \sqsubseteq \mathcal{C}[\mathcal{B}]$.

This property enables the user of the calculus to remove an agent from its context, refine it isolation, and replace the refined agent back into the same context without requiring them to proving that the new composed agent remains a valid refinement. The proof of this theorem is given in [9].

7 Postscript: A Few Notes on 'Sensible' Refinements

It is possible to refine any agent by the specification $w : \mathit{false}$ (for any frame w), this is because:

$$\begin{aligned}[\![w : \mathit{false}]\!] &= \mathit{stable}(\bar{w}, t_\alpha, t_\omega) \wedge \mathit{false}\\ &= \mathit{false}\\ &\Rightarrow [\![\mathcal{A}]\!] \text{ (any } \mathcal{A}) \;\Box\end{aligned}$$

The specification $w : \mathit{false}$ is often referred to as the miraculous specification (see [5] for example), and usually arises as a result of an inconsistent specification at an earlier stage in refinement. However,other kinds of refinement can result in undesired agents as well. Consider the following refinement:

$$\begin{aligned}&\{out\} : \mathit{write}(1, out, t_\alpha, t_\omega)\\ &\quad\sqsubseteq \{out\} : \mathit{write}(1, out, t_\alpha, t_\omega) \wedge \mathit{write}(1, out, t_\alpha, t_\omega) \qquad \text{(by `strengthen')}\\ &\quad\sqsubseteq \{out\} : \exists m : \mathcal{N}(m \in [t_\alpha, t_\omega] \wedge \mathit{write}(1, out, t_\alpha, m) \wedge \mathit{stable}(out, m, t_\omega))\\ &\qquad\qquad \wedge \exists l : \mathcal{N}(l \in [t_\alpha, t_\omega] \wedge \mathit{write}(1, out, t_\alpha, l) \wedge \mathit{stable}(out, l, t_\omega))\\ &\qquad\qquad \text{(by `strengthen')}\\ &\quad\sqsubseteq \{out\} : \mathit{write}(1, out, t_\alpha, t_\omega) | \{out\} : \mathit{write}(1, out, t_\alpha, t_\omega) \text{ (by } \mathbf{RS})\\ &\quad\sqsubseteq 1 - out | 1 - out \text{ (by } \mathbf{RS})\end{aligned}$$

In the final concurrent agent we expect that an implementation would be able to write the value to the shunt *out* twice, and *independently* - but the semantics of the agent dictate that the two agents must write at the same instant in time (otherwise the stability constraints inherent in the definition of the predicate *write* would be contradicted). It is important to realise that the refinement is not incorrect, but that in this case the informal understanding of the meaning of the concurrent agent is not supported by the actual semantics[2].

In order to make sure that our informal view is supported as much as is feasible by the refinement calculus we can introduce a number of heuristics. For example we can assert the following heuristic rule:

Separation Constraint The refinement $w : \Phi \sqsubseteq w_{\mathcal{A}} : \Phi_{\mathcal{A}} | w_{\mathcal{B}} : \Phi_{\mathcal{B}}$ is acceptable only when $w_{\mathcal{A}} \cup w_{\mathcal{B}} = w$ and when $w_{\mathcal{A}} \cap w_{\mathcal{B}} = \emptyset$

This heuristic would have enabled the developer to reject the above refinement in favour of a more 'sensible' alternative. Of course we could have defined the original refinement law for concurrency with this constraint enforced, but we believe that the underlying theory should be as flexible as possible, and that such high-level pragmatics should be defined at as late a stage as possible.

Another instance of a 'sensible' refinement heuristic is that of allowing time for the evaluation of conditionals. The semantics do not assume that the conditionals require evaluation time, and this enables the user to define programs which block on events. Consider the following agent:

$$\mu_n(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1)$$

This agent is waiting for the value in the shunt s to be set to '1': if the value is not currently set to '1' then the agent idle waits for one tick and then tries again. When the value is set to 1 the agent $\mathcal{A}$ will be released. Note that if the owner of the shunt s does not change the value in s then the agent $\mathcal{A}$ might be released many times (although they will be precedence constrained).

We can see that one of the properties of this agent is that if the shunt s at the release time does not have the value 1, and the shunt remains stable, then the agent will reduce to an idle delay, i.e. we can assert the theorem:

$$\mu_n(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) \sqsubseteq \delta n \quad (\text{if } s.v @ t_\alpha \neq 1 \wedge stable(s, t_\alpha, t_\alpha + n))$$

Proof (by induction)

(Base case where n=0)

(1) $\mu 0(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) = \amalg$ (by def Iteration)
(2) $= \delta 0$ (by DL.4)

(Inductive step)

If $\mu_n(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) \sqsubseteq \delta n$

then $\mu_n + 1(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) \sqsubseteq \delta n + 1$

[2] This is an excellent example of why it is important that any compiler for TAM – or any other formal development language – must also be proved correct: it is not good enough that we believe that the compiler supports our informal understanding of the semantics of the language.

(1) $\mu_n + 1(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1)$
$= \mu_n(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1);(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1);\mathbb{T}$ (by IR.1)
(2) $\delta n + 1 = \delta n;\delta 1$ (by DL.1)
(3) $(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1) \sqsubseteq \Delta 1$ (by def Conditional and stability constraint)
(4) $(s.v = 1 \Rightarrow \mathcal{A} \sqcup s.v \neq 1 \Rightarrow \Delta 1);\mathbb{T} \sqsubseteq \Delta 1;\mathbb{T}$ (by monotonicity)
(5) $\Delta 1;\mathbb{T} \sqsubseteq \delta 1$ (by 1.3 and DL.3)

and so by monotonicity the inductive step holds □

Thus the implementation would be sensible. However, it would be difficult to implement a conditional which relied on computation, and also relied on a very short deadline. Consider the agent:

$$[m](x^y > z \Rightarrow (\delta m|\mathcal{A}))$$

which gives a deadline of m time units, all of which is required for the execution of the agent $\mathcal{A}$: thus, the conditional must be evaluated instantaneously. This may be implementable by using some of the time given to the agent $\mathcal{A}$, but this certainly should not be relied upon.

8 Conclusion

TAM is unique in providing a wide-spectrum development language for real-time systems in which abstract specifications can be refined down to concrete executable programs. Wide-spectrum languages for non real-time systems have been studied extensively, for example in the SETL language [11], and the CIP project [3], wide-spectrum languages based upon predicate logic are given transformation rules which allow refinement in a manner similar to TAM.

The utility of a wide-spectrum language can be clearly seen in the refinement method used by Morgan in his calculus [5]. In this language, the concrete syntax is provided by an extended version of Dijkstra's Guarded Command Language [2]. The abstract specification syntax is provided by a statement form:

$$w : [pre, post]$$

where 'w' (called the 'frame') defines the scope of the specification, i.e. those state variables which may be changed by the behaviour defined by the specification, and 'pre' and '$post$' are first-order predicate logic formulae which describe the relationship between the program state before the 'execution' of the specification statement, and after the termination of the specification statement respectively. The specification statement can therefore be viewed as a description of the minimum requirements on the behaviour of any concrete statement which may replace it during refinement.

Similarly, in Back and Wright's wide-spectrum language [1], the concrete code is a version of Dijkstra's Guarded Command Language and a statement, called an *assert statement*, is denoted $\{b\}$, where b is a formula on the local state. The assert statement will terminate correctly if the local state satisfies the formulae when 'executed', and will abort otherwise.

The common factor of both Morgan and Back and Wright's languages is that they are transformational: they describe computations which have all input data available at the start of execution, and provide the result at the time of termination. This restriction provides the basis for the 'shape' of Morgan's specification statement – it describes a relationship between initial and final states. In real-time systems we are interested in *reaction*, i.e. input and output during the execution of an agent. In addition, we are interested in the time at which the inputs and outputs occur; our specification statement for real-time systems reflects these requirements.

Clearly there are many facets of the language yet to investigate, both in the existing constraints of the theory, and in the possibility of extending the theory to deal with issues such as non-termination, and type refinement, etc. The standard offered in this paper should provide a firm foundation for these experiments.

Acknowledgments

The authors would like to thank the Real-Time research group at York for their helpful comments, and in particular Ken Tindell for his reference to the 'dangling drop' problem of pre-emption, and discussions on the expressiveness of the computational model.

References

1. R.J.R. Back and J von Wright. Refinement concepts formalised in higher order logic. *BCS-FACS*, 2(3):247–272, 1990.
2. E.W. Dijkstra. Guarded commands, non-determinacy, and formal languages. *Communications of the ACM*, 18(8), August 1975.
3. The CIP Language Group. The munich project cip. vol1. *LNCS*, 183, 1985.
4. Faron Moller and Christopher M N Tofts. A temporal calculus of communicating systems. Technical Report ECS-LFCS-89-104, University of Edinburgh, December 1989.
5. C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
6. M. Portman and H. S. M. Zedan. The development of imprecise real-time systems. *Systems and Software*, (to appear 1993).
7. A. Salwicki and T. Muldner. *On the Algorithmic Properties of Concurrent Programs*, volume LNCS 125. Springer-Verlag, 1981.
8. S. Schneider. *Correctness and Communication in Real-Time Systems*. PhD thesis, Oxford University, 1990.
9. D. J. Scholefield. *A Refinement Calculus for Real-Time Systems*. PhD thesis, University of York, 1992.
10. D. J. Scholefield, H. S. M. Zedan, and J. He. Real-time refinement: Semantics and application. In *Proceedings of MFCS 93, Gdansk*. Springer-Verlag LNCS 711, October 1993.
11. E. Schonberg and D. Shields. From prototype to efficient implementation: a case study using setl and c. Technical report, 1985.

Compositional Process Semantics of Petri Boxes[1]

Eike Best[2] and Hans-Günther Linde-Göers[2]

Abstract

The Petri Box algebra defines a linear notation to express a structured class of Petri nets which can be seen as a modification and generalisation of Milner's CCS. The calculus has been designed as an intermediate stage in the compositional translation of higher level concurrent programming notations into Petri nets. This paper defines the notion of a 'Box process' intended to capture the (Petri net) partial order semantics of the Box algebra. The main result is the equivalence of the direct compositional semantics so defined, and the indirect non-compositional semantics which uses processes of Petri nets, for a class of expressions.

1 Introduction

The Petri Box Calculus (PBC [5]), which has been developed in the Esprit Basic Research Action DEMON, is a blend which is partially derived from existing calculi (notably, Milner's CCS [23]) and is partially novel. It was designed to satisfy two requirements. Firstly, it should be firmly based on a Petri net semantics, and secondly, it should be oriented towards easing the compositional definition of the semantics of various concurrent programming languages such as occam [21], including all data aspects; it has been discussed in [6, 18] how this can be achieved.

Compared with CCS, the PBC features a different synchronisation operator and a refinement operator. Moreover, the PBC is not prefix-driven but, on the contrary, treats entry and exit points of processes symmetrically. As a consequence, the sequence operator is basic and the recursion operator is much more general and not limited to tail-end recursion.

Up to the present time, there have been various developments concerning the Box calculus; they have been chiefly oriented towards its static aspects. In [5], a number of static equivalences that can be derived as a consequence of the static semantics have been established; in [4], static (denotational) definitions have been given for refinement and recursion; in [12], the S-invariant covering of Boxes has been investigated.

This present paper and its companion papers [7, 20], by contrast, address the dynamic aspects of Box expressions. The aim of the present paper is to make a domain out of the set of (Petri net) processes and to define a compositional

[1]Work done within the Esprit Basic Research Action 3148 DEMON (Design Methods Based on Nets) and the Working Group 6067 CALIBAN (Causal Calculi Based on Nets).

[2]Institut für Informatik, Universität Hildesheim, Marienburger Platz 22, D-31141 Hildesheim, {e.best,linde}@informatik.uni-hildesheim.de

process semantics of Box expressions on this domain. The ground set is defined as the sets of (equivalence classes of) processes, denoting all possible concurrent runs of an expression. The operations on the domain mirror the Box expression operators. The main result establishes the consistency of the Box process semantics (the 'direct' semantics) and the set of processes that can be obtained indirectly by first deriving the Box of an expression and then the processes of this Box using the standard net theoretical notion [3, 17] (the 'indirect' semantics).

The operations we shall define on Box processes have been inspired by prior work such as that of Cherkasova and Kotov [10] and Pratt [25]. The specific form of our operations, which significantly differ from the ones used in the cited papers, has been motivated by [5] and other work on the Box Calculus. This work is also pertinent to a large body of recent work on giving Petri net semantics of existing process calculi such as CCS, CSP [19] or ACP [1] (for instance, [2, 9, 10, 11, 14, 15, 22, 24, 26]).

The organisation is as follows. Section 2 explains the syntactic domain of Box expressions. Section 3 defines the basic elements of the semantic domains we are going to consider: labelled nets, labelled causal nets, Petri Boxes and Box processes. Section 4 defines the first semantic domain, namely the domain of Box processes. Section 5 describes the second semantic domain, the domain of Petri Boxes. Section 6 deals with consistency between the direct semantics and the indirect semantics. The main result of section 6 establishes the equivalence of these two notions. Section 7 contains concluding remarks.

2 The Syntactic Domain: Box Expressions

Action names and variable names are the basic constituents of the Box expression algebra. We assume a set of action names, A, to be given. On A, we assume a conjugation bijection to be defined: $\hat{}: A \rightarrow A$ with $\hat{a} \neq a$ and $\hat{\hat{a}} = a$ for all $a \in A$. The set $\mathcal{L}$ of finite multisets over A is called the set of communication labels. Elements α of the set $\mathcal{L}$ may serve as the labels of transitions and events. The function $\hat{}$ can be extended to any multiset μ over A by element-wise application. When α is a singleton set $\{a\}$, we omit the enclosing set brackets if unambiguity is ensured. We use capital letters X, Y etc. to denote Box expression variables which are used for refinement and recursion. Let $\mathcal{V}$ denote the set of such variable names. Elements of $\mathcal{V}$ may also serve as transition or event labels.

Using these conventions, Table 1 defines the Box expression syntax which we consider in this paper - which is the full syntax considered in [5] except for scoping and relabelling. Scoping is a derived operator from synchronisation and restriction (and its semantics follows accordingly). Relabelling is omitted here since its treatment complicates the formalism but presents no specific difficulties.

E ::=	{Basic Box Expression}	$\alpha \mid$	multiaction
	{Sequential Constructs}	$E; E \mid$	sequence
		$E \,\square\, E \mid$	choice
		$[E * E * E] \mid$	iteration
	{Concurrent Constructs}	$E \| E \mid$	concurrent composition
		E **sy** $a \mid$	synchronisation
		E **rs** $a \mid$	restriction
	{Hierarchical Constructs}	$X \mid$	variable
		$E[X \leftarrow E] \mid$	refinement
		$\mu X.E$	recursion

Table 1: The (slightly) reduced Box expression syntax

3 Basic Semantic Definitions

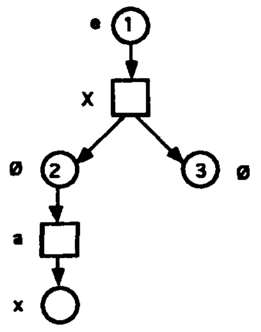
3.1 Labelled nets and renaming equivalence

A labelled Petri net is a quadruple $\Sigma = (S, T, W, \lambda)$, where (S, T, W) is an arc-weighted Petri net with places S, transitions T, weight

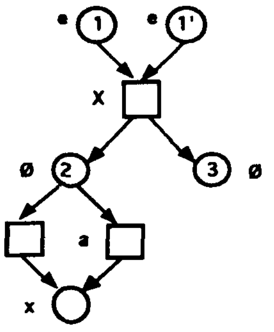
$$W: ((S \times T) \cup (T \times S)) \rightarrow \mathbf{N},$$

place labelling $\lambda: S \rightarrow \{e, \emptyset, x\}$ and transition labelling $\lambda: T \rightarrow \mathcal{L} \cup \mathcal{V}$. The pre-set (post-set) of an element $x \in S \cup T$ is defined by ${}^\bullet x = \{y \mid W(y,x) > 0\}$ (respectively, $x^\bullet = \{y \mid W(x,y) > 0\}$).
We require T-restrictedness, i.e., $\forall t \in T: {}^\bullet t \neq \emptyset \neq t^\bullet$.
The labellings indicate the interfaces of nets, which are relevant for their composition. The places s with $\lambda(s) = e$ are called entry places and denoted by ${}^\bullet\Sigma$. The places with $x \in \lambda(s)$ are called exit places and denoted by $\Sigma^\bullet$. All places with label $\lambda(s) = \emptyset$ are called internal. Similarly, all transitions t with $\lambda(t) = \emptyset$ are called internal or silent. The transitions t with $\lambda(t) \neq \emptyset$ are called interface transitions. There are two types of interface transitions: communication transitions ($\lambda(t) \in \mathcal{L}\backslash\{\emptyset\}$) and hierarchical transitions ($\lambda(t) \in \mathcal{V}$).
A labelled causal Petri net $\eta = (B, E, F, \lambda')$ is a labelled net which satisfies $|{}^\bullet b| \leq 1 \geq |b^\bullet|$ for all $b \in B$ and $F: ((B \times E) \cup (E \times B)) \rightarrow \{0, 1\}$. F can equivalently[3] be viewed as a relation $F \subseteq ((B \times E) \cup (E \times B))$. Elements of B and E are called conditions and events, respectively. A B-cut of η is a maximal set of mutually incomparable elements of B, with respect to the partial order $\prec = F^+$.
Figure 1(i) shows a causal net with four places (named 1, 2, 3, and an unnamed one) labelled, respectively, by e, $\emptyset$, $\emptyset$, and x; and two (unnamed) transitions labelled by X ($\in \mathcal{V}$) and $\{a\}$ ($\in \mathcal{L}$), respectively.

[3]With $F(x,y) = 0$ (resp. 1) iff $(x,y) \notin$ (resp. $\in$)F.



(i) A labelled causal net



(ii) A labelled net which is ρ-equivalent to (i)

Figure 1: Illustration of the basic definitions

The names of places and transitions are only interesting for the level of basic operations on causal nets. For the semantics of Box expressions, they are irrelevant because on this level, only the interface expressed by the labellings and the interconnection structure expressed by the underlying net matters. For the latter purpose, we can call two net elements equivalent if they have the same labels and the same environments.

Two places s and s' in a labelled net duplicate each other if $\lambda(s) = \lambda(s')$ and for all $t \in T$, both $W(s,t) = W(s',t)$ and $W(t,s) = W(t,s')$. The duplication of transitions is defined similarly. In a labelled causal net two different duplicating elements are conditions; this follows from the fact that no condition may be branched. Two labelled nets $\Sigma_1 = (S_1, T_1, W_1, \lambda_1)$ and $\Sigma_2 = (S_2, T_2, W_2, \lambda_2)$ will be called renaming equivalent iff there is a sort-preserving relation $\rho \subseteq (S_1 \times S_2) \cup (T_1 \times T_2)$ such that ρ is (both ways) surjective on places; ρ is (both ways) surjective on transitions; ρ is arc-(weight-)preserving; ρ is label-preserving; and ρ is bijective on hierarchical transitions. Figure 1(ii) shows a labelled net which is ρ-equivalent to the one shown in Figure 1(i). As a consequence of the non-branching of conditions, any two ρ-equivalent labelled causal nets have the same number, or cardinality, of events.

3.2 Boxes and Box processes

Definition 3.1 *Boxes*

A Petri Box B is a ρ-equivalence class $B = [\Sigma]$, such that $\Sigma = (S, T, W, \lambda)$ is a labelled net satisfying:

(i) At least one entry place: ${}^\bullet\Sigma \neq \emptyset$.

(ii) At least one exit place: $\Sigma^\bullet \neq \emptyset$.

(iii) No arcs into entry places: $\forall s \in {}^\bullet\Sigma \; \forall t \in T\colon W(t,s) = 0$.

(iv) No arcs out of exit places: $\forall s \in \Sigma^{\bullet}\ \forall t \in T\colon W(s,t) = 0$. ■ 3.1

Since the properties (i)-(iv) are independent of the choice of a representative of $[\Sigma]$, this definition is good. In fact, similar properties of representative independence hold for all the subsequent definitions whenever they involve Boxes. We shall refrain from mentioning this explicitly; but as a consequence, we may allow ourselves some freedom in referring to Boxes and their representatives, not always making a clear distinction (for instance, 'the transition t of a Box B' is nominally not defined, but can be understood as referring to the transition t of some representative of B).

Definition 3.2 *Box Processes*

A Box process is an equivalence class $\pi = [\eta]$, where renaming equivalence is restricted to labelled causal nets, such that $\eta = (B, E, F, \lambda')$ is a labelled causal net satisfying[4]:

(a) $\text{Min}(\eta)$ is a B-cut.

(b) For all $b \in B$: $\lambda'(b) = e$ iff $b \in \text{Min}(\eta)$.

(c) For all $b \in B$: $\lambda'(b) = x \Rightarrow b \in \text{Max}(\eta)$.

π is, moreover, called complete if η satisfies:

(d) $\text{Max}(\eta)$ is a B-cut and $\forall b \in \text{Max}(\eta)\colon \lambda'(b) = x$. ■ 3.2

The reason for the asymmetry in clauses (b,c) is that we allow nonterminating processes; terminating ones are captured by clause (d) which restores the symmetry of the definition. For the same reason, we do not require property (ii) of Boxes. The properties (i), (iii) and (iv) of Boxes are automatically satisfied by (a,b,c) and the special properties of labelled causal nets; property (ii) is additionally satisfied if (d) holds. It is not hard to see that properties (a)-(d) are robust with respect to ρ-equivalence; thus this definition is again good. We say that π is finite if the representatives of π have finitely many events; according to the above remark, if this is true for one representative then it is true for all of them, so that the notion is well-defined.

4 Domain 1: the Box Process Algebra

We define the Box process algebra by giving its domain, i.e., the set of possible elements (section 4.1); its basic elements (section 4.1); and the operations on the domain (section 4.2). In section 4.3, we use this algebra to define the process semantics of Box expressions.

[4]The Minima and Maxima being defined with respect to the partial order $\prec = F^{+}$.

4.1 The domain $\mathcal{SBP}$ and its basic elements

The elements of the Box process algebra are sets of Box processes (called SBPs in the sequel):

$$\Pi = \{\, \pi \mid \pi \text{ is a Box process and if } \pi \text{ is complete then it is finite}\}.$$

The interpretation will be that each SBP represents a set of possible executions of a Box expression. Let $\mathcal{SBP}$ denote the domain of all SBPs.

We need one basic SBP for every finite communication label $\alpha \in \mathcal{L}$, namely $\Pi_\alpha = \{[\eta_0], [\eta_1]\}$, where $\eta_0 = (\{b\}, \emptyset, \emptyset, \lambda_0')$ and $\eta_1 = (\{b_1, b_1'\}), \{e_1\}, F_1, \lambda_1')$ with $\lambda_0'(b) = e$, $\lambda_1'(b_1) = e$, $\lambda_1'(b_1') = x$, $\lambda_1'(e_1) = \alpha$ and $F_1 = \{(b_1, e_1), (e_1, b_1')\}$. Thus, η_0 is the initial process that describes 'no action as yet', while η_1 is the process that corresponds to a complete execution of α. For every variable X, we need a similar set of two processes called Π_X and defined as above, except that $\lambda_1'(e_1) = X$.

4.2 Operations on Box process sets

In this section - the core of this paper - we define set union, constituent union, concatenation, iteration, synchronisation, restriction, refinement and recursion on the domain $\mathcal{SBP}$. This relies on a few auxiliary Petri net changing operations: $\oplus$ denotes the addition (with the correct connections, which will always be clear from the context) of a set of places or transitions, the labellings of which have to be given as a parameter to the $\oplus$ operation; $\otimes$ is defined between subsets of the elements of two nets and denotes the formation of the symmetric Cartesian product yielding sets $\{x, y\}$ where x is from the first subset and y is from the second subset; $\ominus$ denotes the removal of places or transitions together with their interconnections.

The domain $\mathcal{SBP}$ has been defined in such a way that all its complete elements are finite. This necessitates a proof of the fact that none of the operations we are about to define lead out of that domain. In [8] these proofs are given.

4.2.1 Union of SBPs

The first operation we define is plain set union; it is binary and creates a new SBP $\Pi_1 \cup \Pi_2$ out of two given SBPs Π_1 and Π_2. Being a set theoretical operation, it can be extended straightforwardly to more than two arguments, yielding the union $\bigcup_{i=1}^{\infty} \Pi_i$ for a set of arguments $\{\Pi_1, \Pi_2, \ldots\}$.

4.2.2 Disjoint union of constituents of SBPs

The second operation we define is disjoint union (or juxtaposition) of constituents. Let Π_1 and Π_2 be SBPs. Then

$$\Pi_1 \| \Pi_2 = \{\pi_1 \| \pi_2 \mid \pi_1 \in \Pi_1 \wedge \pi_2 \in \Pi_2\},$$

where for $\pi_1 = [\eta_1]$, $\pi_2 = [\eta_2]$ (and η_1 and η_2 are w.l.o.g. disjoint[5]) we define:

$$\pi_1 \| \pi_2 = [\eta_1 \sqcup \eta_2]$$

where $\sqcup$ denotes the union of labelled causal nets. Since the operation $\|$ differs from the set union of SBPs and will be used to describe concurrent composition, we have used the symbol $\|$ instead of $\cup$ or $\sqcup$.

4.2.3 Concatenation of SBPs

The third operation we define is concatenation. For an SBP Π let Π^C denote the set of complete elements of Π. Π_{-x} will denote the set of processes of Π such that all x-labels of conditions are changed into $\emptyset$-labels.

Let Π_1 and Π_2 be two SBPs. We define

$$\Pi_1;\Pi_2 = (\Pi_1)_{-x} \cup \{\pi_1;\pi_2 \mid \pi_1 \in \Pi_1^C \wedge \pi_2 \in \Pi_2\}.$$

For $\pi_1 = [\eta_1]$, $\pi_2 = [\eta_2]$ (η_1 and η_2 w.l.o.g. disjoint) we put $\pi_1;\pi_2 = [\eta_1;\eta_2]$ and

$$\eta_1;\eta_2 \;=\; (\eta_1 \sqcup \eta_2) \oplus (\mathrm{Max}(\eta_1) \otimes \mathrm{Min}(\eta_2), l) \ominus (\mathrm{Max}(\eta_1) \cup \mathrm{Min}(\eta_2))$$

where $l(\{b_1, b_2\}) = \emptyset$ for any $\{b_1, b_2\} \in \mathrm{Max}(\eta_1) \otimes \mathrm{Min}(\eta_2)$.

The formula for $\eta_1;\eta_2$ means that η_1 and η_2 are first juxtaposed ($\eta_1 \sqcup \eta_2$, together with their disjointness); the exit conditions of η_1 (the same as $\mathrm{Max}(\eta_1)$ because of the completeness of η_1) are multiplied with the entry conditions of η_2, yielding a new set of places that are connected in the appropriate way; and the exit conditions of η_1 as well as the entry conditions of η_2 are removed. The new places get the $\emptyset$ label. This construction - which for the finite case coincides with that in [10] - reappears later when refinement is defined; we shall give an example at that point.

In general, $\pi_1;\pi_2$ is complete iff π_2 is complete [8]. One may define a natural prefix relation $\preceq$ on labelled causal nets: $\eta_1 \preceq \eta_2$ if there is a cut (a maximal set of concurrent elements) in η_2 such that η_1 lies below or equal that cut. In particular, for all $i \geq 1$, $\eta_1;\ldots;\eta_i \ominus \mathrm{Max}(\eta_i) \preceq \eta_1;\ldots;\eta_i;\eta_{i+1}$. Also, ; is an associative operation [8]. Therefore, the infinite sequence

$$\pi_1;\pi_2;\pi_3;\ldots = \bigsqcup_{i=1}^{\infty} (\eta_1;\ldots;\eta_i \ominus \mathrm{Max}(\eta_i))$$

is well defined.

[5]'W.l.o.g.' because of renaming equivalence.

4.2.4 Iteration of SBPs

The iterative construct $[E_1 * E_2 * E_3]$ has the meaning that E_1 is an initial Box expression that may be executed once, after which zero or more repetitions of the body E_2 may occur, after which exactly one execution of E_3, the terminal expression, can complete the execution of the entire expression; but 'once E_1 and then infinitely often E_2' is also possible.

Using union and concatenation, we may now define an iteration operator on SBPs. Let Π_1, Π_2 and Π_3 be SBPs.

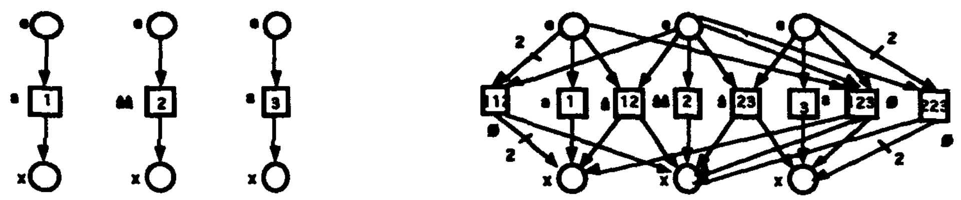
$$\begin{aligned}
\Pi^{(1)} &= \Pi_1 \\
\Pi^{(i+1)} &= \Pi^{(i)};\Pi_2 \quad (\text{for } i \geq 1) \\
\Pi^{*} &= \textstyle\bigcup_{i=1}^{\infty}(\Pi^{(i)};\Pi_3) \\
\Pi^{\omega} &= \{\pi_1;\pi_2;\pi_3;\ldots \mid \pi_1 \in \Pi_1, \pi_j \in \Pi_2, j \geq 2\} \\
\Pi_1 * \Pi_2 * \Pi_3 &= \Pi^{*} \cup \Pi^{\omega}.
\end{aligned}$$

4.2.5 Synchronisation of SBPs

Synchronisation of a Box expression does what is often viewed as an integral feature of concurrent composition, that of effecting the synchronisation over labels. In terms of the Box process semantics, it adds processes to the already existing ones according to certain criteria related to the communication labels of events. In terms of the Box semantics, it adds transitions to an already existing Box according to the same criteria applied to the labels of transitions, rather than events.

The main idea is a 'repetition' of the basic CCS idea, which calls for the synchronisation of sets of transitions over pairs $(a, \hat{a})$ of labels; to take away the labels that effect the synchronisation; and to keep all the other labels. Due to the presence of multisets of labels, the transitions resulting from a synchronisation may carry labels that can continue to lead to further synchronisations. In general, as explained in [5], the multisets of transitions that may synchronise have an underlying tree formed by pairs $(a, \hat{a})$. In CCS, this tree always contains only two nodes and a single arc since the multisets considered there are either empty (τ-action) or singletons; but in the Box expression algebra, the synchronisation tree may be arbitrary.

For instance, consider the expression $(\{a\}\|\{\hat{a},\hat{a}\}\|\{a\})$ sy a, a representative of whose Box is shown in Figure 2(ii) (derived from the Box in 2(i)). The idea in this example is that the first subexpression $\{a\}$ can synchronise with the second subexpression $\{\hat{a},\hat{a}\}$ (through sy a, using one of the $\hat{a}$'s that exist in the second subexpression) and the second subexpression can also synchronise with the third subexpression $\{a\}$ (through sy a, using the other $\hat{a}$ of the second subexpression), yielding a 3-way $\emptyset$-labelled synchronisation. The set of transitions describing this synchronisation is $\tau = \{1,2,3\}$. The fact that this set of transitions can be synchronised together, can be described by the formula $c(\tau) \geq |\tau| - 1$ where $c(\tau)$ counts the minimum number of a's and $\hat{a}$'s in τ [5].



(i) $Box(\{a\}\|\{\hat{a},a\}\|\{\hat{a}\})$ (ii) $Box((\{a\}\|\{\hat{a},a\}\|\{\hat{a}\})$ sy $a)$

Figure 2: An example of multi-way synchronisation

The next definition translates the synchronisation operation just described into an operation on Box process sets. Let Π be an SBP and $a \in A$ an action name. We define

$$\Pi \textbf{ sy } a = \bigcup_{\pi \in \Pi} \pi \textbf{ sy } a.$$

For the definition of π sy a, assume that $\pi = [\eta]$ with $\eta = (B, E, F, \lambda')$. Let E^a be the set of events of η that carry an a or an $\hat{a}$ in their label. Let $\tau \subseteq E^a$ be some finite nonempty set of events of E^a. Define $c(\tau)$ as the minimum of the sum of a names and the sum of $\hat{a}$ names in $\lambda'(\tau)$. Now, let $\mathcal{E} \subseteq \{\tau \subseteq E^a \mid c(\tau) \geq |\tau| - 1\}$ be some set of subsets of E^a, called a synchronisation set of events. With this we define

$$\eta \textbf{ sy}_{\mathcal{E}} a \;=\; \eta \;\oplus\; (\mathcal{E}, l) \;\ominus\; (\bigcup_{\tau \in \mathcal{E}} \tau)$$

where $l(\tau)$ comprises the multiset sum of the labels in τ, minus $|\tau| - 1$ times the pair $\{a, \hat{a}\}$ (because that is just the set of pairs that have effected the synchronisation). The net $\eta \textbf{ sy}_{\mathcal{E}} a$ may not be a causal net, because the constituent events of the sets τ may form a cycle. Therefore, we restrict the synchronisation sets under consideration and define

$$\pi \textbf{ sy } a = \{[\eta \textbf{ sy}_{\mathcal{E}} a] \mid \mathcal{E} \text{ is a synchronisation set of events such that } \eta \textbf{ sy}_{\mathcal{E}} a \text{ is a causal net}\}.$$

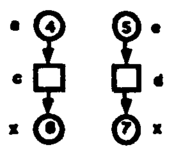
This definition guarantees that Π sy a is indeed an SBP. In general, synchronisation is conservative on complete processes; i.e., π sy a is complete iff π is complete [8].

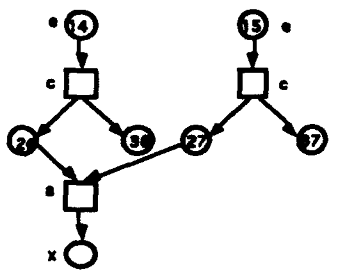
4.2.6 Restriction of SBPs

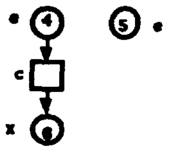
Restriction of an SBP over an action name a removes those processes that contain an event labelled with a or $\hat{a}$. Let Π be an SBP and a an action name. Moreover let E^a be the set of events of a causal net $\eta = (B, E, F, \lambda')$ that contain an a or an $\hat{a}$ in their label. We define

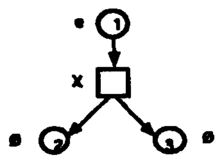
$$\Pi \textbf{ rs } a = \Pi \setminus \{[(B, E, F, \lambda')] \in \Pi \mid E^a \neq \emptyset\}.$$

4.2.7 Refinement of SBPs


(i) A complete process of $E_2 = (c\|d)$


(ii) Refining X in 1(i) by (i)


(iii) A non-complete process of $E_2 = (c\|d)$


(iv) Another concrete process of $E_1 = X;(a\|b)$

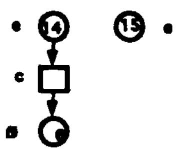

(v) Refining X in (iv) by (iii)

Figure 3: Illustration of the definition of refinement

We define the processes of $\Pi_1[X \leftarrow \Pi_2]$ by taking the processes of Π_1 and refining all X-labelled events in them by processes of Π_2. Figures 1(i) and 3 depict one of the possible pitfalls in this approach. The example concerns the Box expression $E_1[X \leftarrow E_2]$, with $E_1 = X;(a\|b)$ and $E_2 = (c\|d)$. Figure 1(i) depicts a representative process of E_1, Figure 3(i) shows a process of E_2 and Figure 3(ii) the result of refining the event labelled X in the first process by the second process. If the refining process is complete, the definition is straightforward. However, it makes no sense to allow the X in 1(i) to be refined by the non-complete process shown in Figure 3(iii), since the occurrence of event a shows that X has been 'completed'. By contrast, in Figure 3(iv) - another process of E_1 - we will have to allow the refinement of the X-labelled event into the process of 3(iii), yielding the result shown in Figure 3(v); otherwise, not all processes of the refined expression would be generated in this compositional way. It is not hard to characterise the events that may be refined by non-complete processes: they are the maximal ones; in Figure 1(i), the X-labelled event is not maximal while in 3(iv), it is. The definition is structured accordingly. Let Π_1 and Π_2 be two SBPs. Let $\pi_1 \in \Pi_1, \pi_2 \in \Pi_2$ and $\pi_1 = [\eta_1], \pi_2 = [\eta_2]$. For

$\eta_1 = (B_1, E_1, F_1, \lambda_1)$ we define

$$\begin{aligned} E^X_{1m} &= \{e \in E_1 \mid \lambda_1(e) = X \text{ and } e \text{ is maximal}\} \\ E^X_1 &= \{e \in E_1 \mid \lambda_1(e) = X\}. \end{aligned}$$

In the following, (a) determines the proper relationship between e of η_1 and η_2 for a refinement of the former by the latter to be possible, and also gives the definition; (b) appeals to [4] to generalise this definition to simultaneous refinement; (c), finally, gives the general definition of $\Pi_1[X \leftarrow \Pi_2]$.

(a) This assumes that if $e \in E^X_1 \setminus E^X_{1m}$ then η_2 is complete. We define

$$\begin{aligned} \eta_1[e \leftarrow \eta_2] &= (\eta_1 \sqcup \eta_2) \;\oplus\; ({}^\bullet e \otimes {}^\bullet\eta_2, l) \\ &\qquad\oplus\; (e^\bullet \otimes \eta_2^\bullet, l) \\ &\qquad\ominus\; ({}^\bullet e \cup e^\bullet \cup {}^\bullet\eta_2 \cup \eta_2^\bullet) \;\ominus\; \{e\} \end{aligned}$$

where $l(\{b_1, b_2\}) = \lambda_1(b_1)$ for new conditions $\{b_1, b_2\}$ with $b_1 \in B_1$.

(b) We now appeal to the results of [4] which show that the refinement of e by η_2 and the refinement of e' by η_2' (both e and e' being different events of the same process η_1) commute; and, more generally still, that one may extend the whole definition to the simultaneous refinement

$$\eta_1[e^{(i)} \leftarrow \eta_2^{(i)}, i \in I],$$

of a whole (possibly empty[6] or possibly infinite) set of events $\{e^{(i)} \mid i \in I\} \subseteq E^X_1$ - provided each individual refinement $\eta_1[e^{(i)} \leftarrow \eta_2^{(i)}]$ is valid according to (a).

(c) Finally, we define generally:

$$\begin{aligned} \Pi_1[X \leftarrow \Pi_2] = \{\, [\eta_1[e^{(i)} \leftarrow \eta_2^{(i)}, i \in I]] \mid & \{e^{(i)} \mid i \in I\} = E^X_1 \text{ and} \\ & [\eta_1] \in \Pi_1 \text{ and } [\eta_2^{(i)}] \text{ is some element of } \Pi_2 \,\} \end{aligned}$$

That is, we require that *all* X-labelled events of η_1 be replaced by processes from Π_2. This may introduce new X-labelled events if the processes from Π_2 contain such events.

4.2.8 Recursion of SBPs

In accordance with one of the central ideas behind the Petri Box semantics, we shall interpret recursion as the limit of successive refinements. In the static Box semantics, the Box associated to, say, $\mu X.(a; X; b)$ is symmetric in a and b; it has as many a-labelled transitions as it has b-labelled transitions even though only the former, but not the latter, can be executed. In the Box process semantics, this symmetry must be broken: the processes of $\mu X.(a; X; b)$ may

[6]In which case the refinement changes nothing.

contain arbitrarily many a-labelled events, but none of them may contain any b-labelled event. The if...then premise of part (a) of the definition of refinement is the technical means by which this is achieved.

Let Π be an SBP. Define a sequence of SBPs as follows[7]:

$$\begin{aligned}\Pi_0 &= \{[\eta_0]\}\\ \Pi_{j+1} &= \Pi[X \leftarrow \Pi_j]\\ \mu X.\Pi &= \{[\bigsqcup_{j=0}^{\infty}\eta_j] \mid \eta_0 \preceq \eta_1 \preceq \ldots, [\eta_j] \in \Pi_j\}.\end{aligned}$$

Note first that none of the processes in Π_j, and hence none of the processes in $\mu X.\Pi$ either, may contain any X-labelled events.

Figure 4 illustrates this definition[8]. The process $[\eta_0]$ is in Π_0 by definition. The other processes arise out of each other as follows:

$$\begin{aligned}\eta_1 &= \gamma_1[\emptyset \leftarrow \eta_0]\\ \eta_2 &= \gamma_2[e \leftarrow \eta_1]\\ \eta_3 &= \gamma_2[e \leftarrow \eta_2],\end{aligned}$$

and the infinite process arises as the union of the finite ones. Note that the X-labelled event in γ_3 cannot be refined at all, because none of the processes in Π_j is complete.

Other interesting expressions on which this definition can be checked are $\mu X.X$ and $\mu X.(a\|X)$ and $\mu X.(a \,\square\, a\|X)$. The process set $\Pi(\mu X.X)$ has only one process with an e-labelled condition[9]. The second process set $\Pi(\mu X.(a\|X))$ has an isolated e-labelled condition in every process. It also has an infinite process with an isolated e-labelled condition, which can be obtained as a sequence of prefix-related processes all of which (including the very first one) have *infinitely many* isolated e-labelled conditions. In this way, the definition is consistent with the one (on Boxes) given in [4]. The third process set $\Pi(\mu X.(a \,\square\, a\|X))$ has infinitely many *terminating* finite processes, but again no infinite terminating process; this comes from the fact that in the definition we use prefix ordering (rather than the weaker relation of net inclusion, which is also a partial order on labelled causal nets).

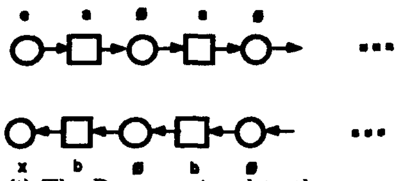
4.3 Process semantics of Box expressions

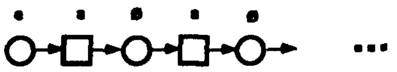
For each term E of the Box algebra defined in section 2, we are now able to define a function Π such that $\Pi(E)$ gives the set of partial executions (Box processes) of E. Let E, E_1, E_2, E_3 be Box expressions, $a \in A$ an action name and α a communication label.

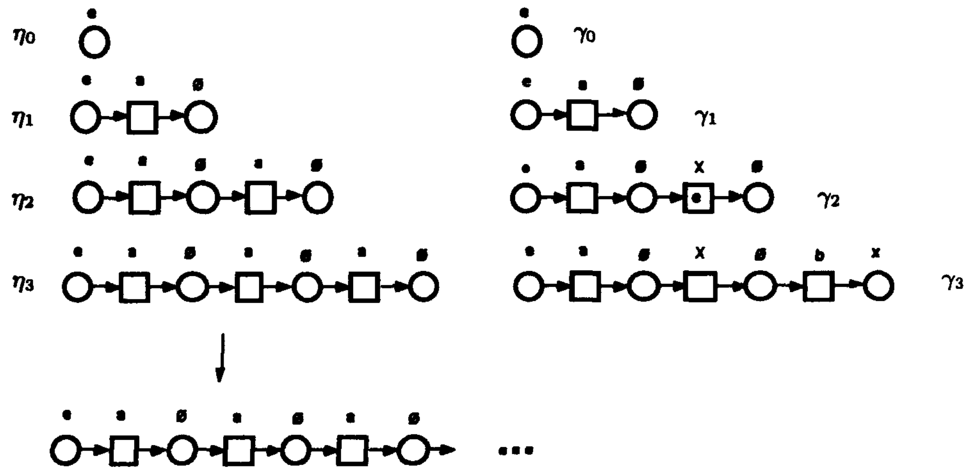
[7]The net η_0 consists of an isolated e-labelled condition, as defined in section 4.1.

[8]In parts (i) and (ii) of this figure, the semantics of section 4.3 is used.

[9]While the Box of $\mu X.X$ (section 5) consists of two isolated conditions, one of which is e-labelled and one of which is x-labelled.


(i) The Box associated to the expression $\mu X.(a; X; b)$


(ii) A process of the Box shown in (i)



(iii) An application of the definition to obtain the process (ii)

Figure 4: Illustration of the definition of recursion

$$\begin{array}{lcl}
\Pi(\alpha) & = & \Pi_\alpha \\
\Pi(E_1;E_2) & = & \Pi(E_1);\Pi(E_2) \\
\Pi(E_1 \,\square\, E_2) & = & \Pi(E_1) \cup \Pi(E_2) \\
\Pi([E_1 * E_2 * E_3]) & = & \Pi(E_1) * \Pi(E_2) * \Pi(E_3) \\
\Pi(E_1 \| E_2) & = & \Pi(E_1) \| \Pi(E_2) \\
\Pi(E \,\mathbf{sy}\, a) & = & (\Pi(E)) \,\mathbf{sy}\, a \\
\Pi(E \,\mathbf{rs}\, a) & = & (\Pi(E)) \,\mathbf{rs}\, a \\
\Pi(X) & = & \Pi_X \\
\Pi(E[X \leftarrow E']) & = & \Pi(E)[X \leftarrow \Pi(E')] \\
\Pi(\mu X.E_0) & = & \mu X.(\Pi(E_0)).
\end{array}$$

5 Domain 2: the Box Algebra

We may define the Box algebra in a similar manner as before by defining its domain and its basic elements (section 5.1); and its operations (section 5.2). We apply the approach of [4] which allows us to be relatively brief, because many operations can be based on refinement. However, we can only outline the definitions; for details, the reader is referred to [4, 5].

5.1 The domain $\mathcal{B}$ and its basic elements

The elements of the Box algebra are the Petri Boxes $B = [\Sigma]$. Let $\mathcal{B}$ denote the set of Boxes. Let $\alpha \in \mathcal{L}$. The Box $Box(\alpha)$ which offers the communication possibilities of α in a single transition is defined as follows: $Box(\alpha) = [(\{s_1, s_2\}, \{t\}, \{(s_1, t), (t, s_2)\}, \lambda)]$ with $\lambda(s_1) = e$, $\lambda(s_2) = x$ and $\lambda(t) = \alpha$. A special case is $Box(\emptyset)$, which is analogous to CCS's silent action τ. The Box $Box(X)$, for a variable X, is defined similarly.

5.2 Operations on Boxes

5.2.1 Refinement

Let $E[X \leftarrow E']$ be a Box expression with refinement. Let $B = Box(E)$ and $B' = Box(E')$. Any representative of B may contain transitions with labels X. The natural semantics of the operator $E[X \leftarrow E']$ corresponds to the refinement of such transitions.

The basic idea behind transition refinement is as follows [4, 16]. Let t be an X-labelled transition in some representative Σ of a Box B, and let Σ' be a representative of a Box B such that t is to be refined by Σ'. By our basic properties of labelled nets and of Boxes, we have that t has at least one pre-place and Σ' has at least one e-labelled place. Hence the product ${}^\bullet t \otimes {}^\bullet\Sigma'$ is not empty and can be used in the refined system; a similar remark is true for the post-places $t^\bullet$ of t and the exit places of Σ'.

This basic idea works directly if there is a single X-labelled transition t whose pre-places and post-places are disjoint. [4] describes a significantly generalised construction which works if there are arbitrarily many X-labelled transitions (including the case of infinitely many); if X-labelled transitions are contained in side conditions; for simultaneous refinement $B[X \leftarrow B', Y \leftarrow B'']$ (meaning: refine all X-labelled transitions by B', and simultaneously, all Y-labelled transitions by B''), and generalisations thereof.

5.2.2 Sequence, choice, concurrent composition and iteration

These Box operations may be based on simultaneous refinement. Let B_1, B_2 and B_3 be Boxes. We define

$$\begin{array}{lcl}
B_1;B_2 & = & B_;[X \leftarrow B_1, Y \leftarrow B_2] \\
B_1 \Box B_2 & = & B_{\Box}[X \leftarrow B_1, Y \leftarrow B_2] \\
B_1 \| B_2 & = & B_{\|}[X \leftarrow B_1, Y \leftarrow B_2] \\
[B_1 * B_2 * B_3] & = & B_*[X \leftarrow B_1, Y \leftarrow B_2, Z \leftarrow B_3],
\end{array}$$

where $B_;$, $B_{\Box}$, $B_{\|}$ and B_* are the Boxes shown in Figure 5.

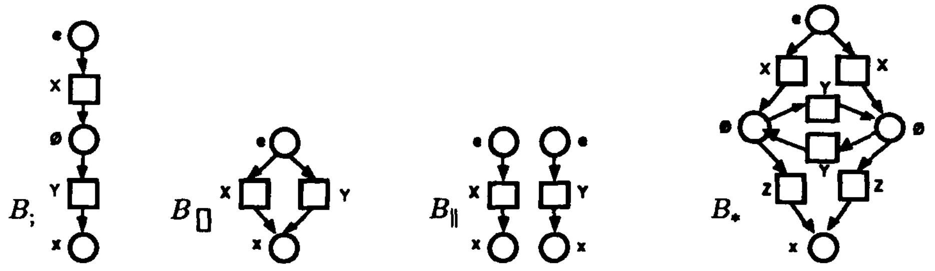


Figure 5: The basic Boxes for sequence, choice, composition and iteration

The translation of iteration using the Box B_* shown in Figure 5 ensures, by the results of [12], that the semantics is 1-safe. An alternative translation that may come to mind (an X-labelled transition followed by a side transition labelled Y followed by a Z-labelled transition) violates 1-safeness in cases like $[a*(b\|c)*d]$. Indeed, under this alternative translation, the main result of section 6.2 becomes wrong.

5.2.3 Recursion

Let $\mu X.E$ be a recursive expression. The intended meaning is that at the free X's, 'E is called recursively'. Corresponding to the fact that a particular kind of syntactic substitution is the natural way of describing this recursive call [1, 23], at the semantic level the semantics translates into a succession of refinements.

Since refinements define a function on labelled nets, the whole approach leads to the employment of fixpoints [4].
The basic idea is the following. First, we construct some representative Σ of $Box(E)$, the Box associated with the body of the recursive expression. Next, we define an initial Box representative Σ_0 with a nonempty set of entry places and a nonempty set of exit places and nothing else (no internal places, no transitions); the equivalence class of Σ_0 is called a **stop** Box, as it cannot terminate (and cannot, in fact, change its initial state). Then, we iterate as follows:

$$\Sigma_{i+1} = \Sigma[X \leftarrow \Sigma_i].$$

As an example, we derive the Box shown in Figure 4(i) which corresponds to the expression $\mu X.(a;X;b)$. First, a three-transition representative Σ of the body $Box(a;X;b)$ may be constructed. Then, as the zero'th approximation, we may define a **stop** representative with only two places, one entry place and one exit place. The X-transition of Σ may be refined by this latter net, yielding the first approximation with 2 transitions. The X-transition of Σ may be refined again by the result, yielding the second approximation with 4 transitions, and so forth. The limit yields the representative shown in Figure 4(i).

5.2.4 Synchronisation and restriction

The effect of the synchronisation operation B **sy** a has already been described in section 4.2.5. The formal definition consists of adding transitions which correspond to multisets of existing transitions such that the minimum-formula is satisfied, which is a criterion for the multiset in question to be a valid synchronisation set. Restriction B **rs** a has an opposite effect; it removes all transitions that have an a or a $\hat{a}$ in their label.
Notice that these operations are quite different from those of sections 4.2.5 and 4.2.6; here, they add (or remove) transitions whilst there, they add (or remove) processes.

5.3 Box semantics of Box expressions

With these definitions, the Box semantics $Box(E)$ of a Box expression E can be given by a homomorphism that maps expressions and their operators into Boxes and their operations. The table giving this definition is analogous to that of section 4.3 and will be omitted here.

6 Consistency

Let $B = Box(E)$ be a Box with a representative $\Sigma = (S, T, W, \lambda)$. The standard initial marking of Σ is the marking that puts one token on each e-labelled place of Σ and zero tokens on all other places; let us denote by $ST(\Sigma)$ the place/transition system so defined. The results of [12] imply that $ST(\Sigma)$ is a 1-safe marked net[10].

[10]Because B derives from an expression; otherwise 1-safeness is not guaranteed.

Standard Petri net theory [3, 17] allows us to associate a set of processes (causal nets labelled with places from S and transitions from T, and hence, by proxy, also with elements of $\{e, \emptyset, x\} \cup \mathcal{L} \cup \mathcal{V}$) to $ST(\Sigma)$.

Two questions arise naturally: (1) If Σ and Σ' are two representatives of B, what is the relation (if any) between their processes? (2) What (if any) is the relation between the processes of representatives of B and the Box processes of B defined in section 4?

6.1 Representative independence of processes

We investigate two labelled nets $\Sigma_1 = (S_1, T_1, W_1, \lambda_1)$ with standard initial marking M_{1e} and $\Sigma_2 = (S_2, T_2, W_2, \lambda_2)$ with standard initial marking M_{2e} which are ρ-related. The first step is to lift the relation ρ from the places of the two nets to their markings. This relies on the observation that all markings (the initial ones and the reachable ones) are duplicate respecting, meaning that duplicate places always carry the same number of tokens. This allows to turn ρ into a bijection on the two reachability graphs of $ST(\Sigma_1)$ and $ST(\Sigma_2)$.

The second step is to consider any two labelled causal nets η_1 and η_2 which are renaming equivalent by means of a relation ρ'. Then ρ defines a bijection both between the events of η_1, η_2 and between their reachable B-cuts (the process equivalent of reachable markings).

The third step is to consider a process $\kappa_1 = (B_1, E_1, F_1, p_1)$ of $ST(\Sigma_1)$ with a function $p_1 : B_1 \cup E_1 \rightarrow S_1 \cup T_1$ which describes which conditions are holdings of which places, and which events are occurrences of which transitions. We may associate to κ_1 a labelled causal net $\eta(\kappa_1) = (B_1, E_1, F_1, p_1 \circ \lambda_1)$ ($p_1 \circ \lambda_1$ is the 'by proxy' labelling of κ_1 inherited from Σ_1). The first main theorem shows that κ_1 is ρ'-equivalent to some process κ_2 of Σ_2, completing a diagram that commutes both in terms of events/transitions and in terms of B-cuts/markings.

Theorem 6.1 *Representative independence*

With the notation as above, there exists a process $\kappa_2 = (B_2, E_2, F_2, p_2)$ of $ST(\Sigma_2)$ and a relation ρ' such that

(a) *the labelled causal nets $\eta(\kappa_1)$, $\eta(\kappa_2)$ associated to κ_1 and (similarly) to κ_2, respectively, are ρ'-related.*

(b) *For all $e_1 \in E_1$ and $e_2 \in E_2$: if $e_2 \in \rho'(e_1)$ then $(p_1(e_1), p_2(e_2)) \in \rho$.*

(c) *For all reachable B-cuts c_1 of κ_1 and c_2 of κ_2: if $c_2 = \rho'(c_1)$ then $p_2(c_2) = \rho(p_1(c_1))$.*

In part (c) of the theorem, we use the notation $p(c)$ to denote the marking corresponding to c via p. Parts (b) and (c) relate back to the fact that ρ' (by part (a)) defines a bijection on events and on B-cuts. This result has a number of corollaries. We state some of them.

Corollary 6.2

For every occurrence sequence of $ST(\Sigma_1)$ ($M_{1e} = M_0$) there exists a corresponding occurrence sequence of $ST(\Sigma_2)$ ($M_{2e} = M_0'$).
The event pomsets of $ST(\Sigma_1)$ and the event pomsets of $ST(\Sigma_2)$ are isomorphic to each other.
If Σ_1 and Σ_2 are two representatives of the same Box B then

$$\{[\eta(\kappa)] \mid \kappa \text{ is a process of } ST(\Sigma_1)\} = \{[\eta(\kappa')] \mid \kappa' \text{ is a process of } ST(\Sigma_2)\}$$

where $[\eta(\kappa)]$ denotes the Box process associated with $\eta(\kappa)$ (Definition 3.2).
■ 6.2

The last part of this corollary allows us to associate a set of Box processes $\Pi(B)$ uniquely to any Box B, by choosing an arbitrary concrete representative Σ of B, evaluating its processes κ, the associated labelled causal nets $\eta(\kappa)$ and forming their ρ-equivalence classes. In particular, we may thus associate a set of Box processes $\Pi(Box(E))$ uniquely to every Box expression E, which may or may not be different from the set $\Pi(E)$; by the next result, it is not.

6.2 Consistency between the indirect and the compositional semantics

Theorem 6.3

Let E be a Box expression. Then $\Pi(E) = \Pi(Box(E))$, wher· $\Pi(E)$ is defined in section 4.3 and $\Pi(Box(E))$ is defined following corollary 6.2.

The proof is by structural induction on the syntax of Box expressions [8].

7 Concluding Remarks

Figure 6 summarises the main ideas and results of this paper. Starting from an expression E one may define its processes $\Pi(E)$ directly (first vertical line). One may also define them indirectly (first horizontal line; second and third vertical line). The auxilia results of section Sct.6.1 state that the indirect definition is representative inaependent. The main result of section Sct.6.2 states that the diagram commutes.

The paper [20] extends the framework, the definitions and the results of this paper to the compositional construction of the branching processes [13] of a Box expression. Work is furthermore in progress to use the process semantics and the branching process semantics in order to define and compare various behavioural equivalence notions that may be defined on the Box expression algebra.

Although the operations we have defined on Box processes have been inspired by prior work such as that of Cherkasova and Kotov [10] and Pratt [25], there are

$$\begin{tikzcd}
\text{Sct.2} & E \arrow[rr, dash, "\text{Sct.5}"] \arrow[ddd, dash, "\text{Sct.4}"'] & & Box(E) \arrow[d, equal, "\text{Sct.3}"] \\
& & & \Sigma \text{---} \Sigma' \arrow[d, equal, "\text{Sct.6.1}"] \\
& & & \text{Processes of } ST(\Sigma) = \text{Processes of } ST(\Sigma') \arrow[d, equal, "\text{Sct.6.1}"] \\
& \Pi(E) \arrow[rr, phantom, "="] \arrow[rr, phantom, "\text{Sct.6.2}"', yshift=-2ex] & & \Pi(Box(E))
\end{tikzcd}$$

Figure 6: Summary of the main results

substantial differences. The paper by Cherkasova and Kotov does not deal with iteration or recursion. Pratt's pomsets are event-based rather than process-based, whence his framework does not lend itself to our purposes. Moreover, our operators are partly different, and the type of result we have aimed for (consistency with standard Petri net semantics) has not been investigated by Pratt. A more extensive discussion of the relationship between our approach and other work can be found in [5].

Acknowledgements

Jon G. Hall has suggested the notion of a duplicate respecting marking; Ludmila Cherkasova participated in the discussions that led to the definitions of section 4; we are particularly indebted to Raymond Devillers and Javier Esparza for their substantial contributions in laying the groundwork for the denotational treatment of refinement and recursion; to Raymond Devillers who has proved the S-invariant covering result which is necessary for the present paper; and again to Raymond Devillers for commenting on a draft of this paper and pointing out two mistakes.

References

[1] J.C.M.Baeten and W.P.Weijland: *Process Algebra.* Cambridge Tracts in Theoretical Computer Science (1990).

[2] E.Best: *Concurrent Behaviour: Sequences, Processes and Axioms.* Seminar on Concurrency, Carnegie-Mellon University, Pittsburgh, PA, July 9-11, 1984 (eds. S.D.Brookes; A.W.Roscoe; G.Winskel) Springer-Verlag Lecture Notes in Computer Science Vol. 197, 221-245 (1985).

[3] E.Best and R.Devillers: *Sequential and Concurrent Behaviour in Petri Net Theory.* Theoretical Computer Science Vol. 55, 87-136 (1987).

[4] E.Best, R.Devillers and J.Esparza: *General Refinement and Recursion Operators for the Petri Box Calculus.* Proceedings of STACS-93, P. Enjalbert et al. (eds.). Springer-Verlag Lecture Notes in Computer Science Vol. 665, 130-140 (1993).

[5] E.Best, R.Devillers and J.G.Hall: *The Box Calculus: a New Causal Algebra with Multi-label Communication.* In: DEMON Special Volume of the Advances in Petri Nets (ed. G.Rozenberg), Springer-Verlag Lecture Notes in Computer Science Vol.609, 21-69 (1992).

[6] E.Best and R.P.Hopkins: $B(PN)^2$ *- A Basic Petri Net Programming Notation.* Proc. PARLE-93, Springer-Verlag LNCS (June 1993).

[7] E.Best, J.Esparza and M.Koutny: *Operational Semantics for the Box Algebra.* Hildesheimer Informatikbericht Nr. 13/93 (1993).

[8] E.Best and H.G.Linde-Göers: *Compositional Process Semantics of Petri Boxes.* Hildesheimer Informatikbericht Nr. 12/93 (1993).

[9] G.Boudol and I.Castellani: *Flow Models of Distributed Computations: Event Structures and Nets.* Rapport de Recherche, INRIA, Sophia Antipolis (July 1991).

[10] L.Cherkasova and V.Kotov: *Descriptive and Analytical Process Algebras.* Advances in Petri Nets 89 (ed. G.Rozenberg), Springer-Verlag Lecture Notes in Computer Science Vol. 424, 77-104 (1989).

[11] P.Degano, R.De Nicola and U.Montanari: *A Distributed Operational Semantics for CCS Based on C/E Systems.* Acta Informatica 26 (1988).

[12] R.Devillers: *Construction of S-invariants and S-components for Refined Petri Boxes.* Proc. of Conf. on Appl. and Theory of Petri Nets, Chicago, Springer-Verlag LNCS (June 1993).

[13] J.Engelfriet: *Branching Processes of Petri Nets.* Acta Informatica Vol.28, 575-591 (1991).

[14] R.J.van Glabbeek and F.V.Vaandrager: *Petri Net Models for Algebraic Theories of Concurrency.* Proc. PARLE 89, Springer-Verlag Lecture Notes in Computer Science Vol.259, 224-242 (1987).

[15] U.Goltz: *Über die Darstellung von CCS-Programmen durch Petrinetze.* (In German.) R.Oldenbourg Verlag, GMD-Bericht Nr.172 (1988). See also: U.Goltz: *On Representing CCS Programs by Finite Petri Nets.* Proc. MFCS-88, Springer-Verlag Lecture Notes in Computer Science Vol.324, 339-350 (1988).

[16] U.Goltz and R.J.van Glabbeek: *Refinement of Actions in Causality Based Models.* Proc. of REX Workshop on Stepwise Refinement of Distributed Systems, Springer-Verlag Lecture Notes in Computer Science, 267-300 (1989).

[17] U.Goltz and W.Reisig: *The Non-sequential Behaviour of Petri Nets.* Information and Control 57(2-3), 125-147 (1983).

[18] R.P.Hopkins, J.Hall and O.Botti: *A Basic-Net Algebra for Program Semantics and its Application to* occam. Advances in Petri Nets 1992. Springer-Verlag Lecture Notes in Computer Science Vol. 609, 179-214 (1992).

[19] C.A.R.Hoare: *Communicating Sequential Processes.* Prentice Hall (1985).

[20] H.G.Linde-Göers: *Compositional Branching Processes of Petri Nets.* Ph.D. Thesis (1993).

[21] D.May: occam. SIGPLAN Notices, Vol.18(4), 69-79 (April 1983).

[22] J.-J.Ch.Meyer and E.P.de Vink: *Pomset Semantics for True Concurrency with Synchronisation and Recursion.* Proc. MFCS-89, Springer LNCS Vol.379 (ed. Kreczmar/Mirkowska) (1989).

[23] R.Milner: *Communication and Concurrency.* Prentice Hall (1989).

[24] E.R.Olderog: *Nets, Terms and Formulas.* Habilitation (1989). Cambridge Tracts in Theoretical Computer Science (1991).

[25] V.Pratt: *Modeling Concurrency with Partial Orders.* Int. Journal of Parallel Programming, Vol.15(1), 33-71 (1986).

[26] D.Taubner: *Finite Representation of CCS and TCSP Programs by Automata and Petri Nets.* Springer-Verlag Lecture Notes in Computer Science Vol. 369 (1989).

On the Specification of Elementary Reactive Behaviour[*]

G. Michele Pinna[†] Axel Poigné
Research Group "Specification of Complex Systems"
System Design Technology Institute
Gesellschaft für Mathematik und Datenverarbeitung (GMD)
Schloß Birlinghoven
D-53757 Sankt Augustin, Germany
e_mail: {pinna,poigne}@gmd.de

Abstract

The paper addresses the specification of reactive behaviour for event-based models. We are concerned with a framework for such specification rather than with a particular style of specifications. Our main observation is that a *logic of events* is needed as well as a *logic of actions*. The former prescribes in fine detail how computations proceed while the latter provides generic scripts for events to happen. The analogy is that of procedures and procedure calls at runtime (= events). We claim that both logics are inherently interrelated, in particular, if "true concurrency" is to be specified.

In order to specify reactive behaviours we propose a logic of actions on top of a new model called *event automata*, focusing on the ingredients that such a specification method should provide.

1 Introduction

A *reactive system* constantly interacts with its environment by reacting to incoming stimuli that may arrive at any stage of the computation. In this paper, we discuss the minimal ingredients to be provided by a specification methodology which is based on "true concurrency" models.

[*]Work partially supported by the German Ministry for Science and Technology, Verbundprojekt KORSO, subproject "Design of Reliable Reactive Systems" (grant No. ITS 900 1A7).

[†]New address: Dipartimento di Matematica, Università degli Studi di Siena, Via del Capitano 15, I-53100 Siena, Italy

The choice of the basic model is crucial. We have introduced *event automata* as a very basic model of reactive behaviour [12]. This model subsumes familiar event-based models such as prime event structures, flow event structures, general event structures as well as geometric automata. These can be accommodated compositionally, meaning that, for instance, synchronization operators can be defined which exactly correspond to those defined for the other models.

The definition of event automata is simple if compared with other event-based models. This is achieved by focusing on behaviour only. Other models *present* behaviour, with behaviour typically being given in terms of configuration spaces. Let us review our basic definition.

Definition 1 *An* event automaton $\mathcal{E} = (E, St, \vdash, ev)$ *consists of*

- *a set* E *of* events,
- *a set* St *of* states *such that* $ev(s) \subseteq E$ *for all* $s \in St$*, and*
- *a transition relation* $s \vdash s'$ *such that* $ev(s') = ev(s) \uplus \{e\}$[1] *for some event* $e \in E$.

Usually, we assume $ev(s)$ *to be finite.*

A state of an event automaton represents a (possibly partial) computation of a concurrent system and a transition relation between states describes how computations proceed.[2] As a minimal requirement, we keep track of the events which have occurred so far. These are recorded by the set $ev(s)$ for every state s. Moreover, every transition corresponds to the occurrence of some event. The definition implies that an event can occur only once in a trace of the system. Throughout the paper we maintain the basic assumption of event-based systems that events are instantaneous and indivisible, and that they occur only once in a computation. We do not subscribe to the other requirement of event structures that an event can only occur once in all computations. An example may highlight the difference.

The automaton in figure 1 cannot be a family of configurations [18] of an event structure: both the events a and b are enabled without preconditions, i.e. $\vdash a$ and $\vdash b$. Hence, by the monotonicity axiom '$Y \vdash a$

[1]We use the notation $X \uplus \{e\}$ to state that $X \cup \{e\}$ is a disjoint union of X and $\{e\}$.

[2]As a remark, reachable states are generated by initial states. We consider as reachable states those $s \in St$ such that $s_{in} \vdash^* s$, where $\vdash^*$ is the transitive and reflexive closure of $\vdash$ and s_{in} is an initial state. Of course we have first to say which states are initial. *Reach*$(\mathcal{E})$ denotes the subautomaton of reachable states.

$$\begin{tikzcd}
 & \{a\} \arrow[dr] & \\
\emptyset \arrow[ur] \arrow[dr] & & \{a,b\} \\
 & \{b\} &
\end{tikzcd}$$

Figure 1: An example where reachability is not monotonic

whenever $X \vdash a$ and $X \subseteq Y$' inherent in all definitions of event structures, there should be a move $\{b\} \rightarrow \{a,b\}$, which one may want to avoid. Inclusion coincides with transition in event structures because of the monotonicity rule. In consequence, every event can occur only once in all computations. Pragmatically, we may say that a and b are in *asymmetric conflict.*

In order to specify reactive behaviour, the following appears to be a minimal set of properties to be captured:

- **Consistency.** A set of events is *consistent* if and only if all the events in this set may coexist together. Inconsistencies may arise if we try to extend a computation with an event which is in *conflict* with another present in the computation. We can then specify conflicts by means of an irreflexive relation[3].

- **Dependency between events.** Dependency between events is usually identified with causality: an event e' *causally depends* on an event e if and only if every computation containing e' contains also e. Since the definition refers to the overall behaviour, causality appears to be more a matter of observation a posteriori than a matter of specification a priori. For specification, the notion of *enabling* is more appropriate: the notation $\phi \vdash e$ states that an event e may occur if the formula ϕ holds.

Specification only in terms of events is often too fine-grained: if we type 20 'a's on a typewriter, we have 20 different events, namely 'typing the i-th a', but all events correspond to the same action 'typing an a'. A system should be specified in terms of *actions* which we consider as being generic scripts for determining the evolution of the system.

The question arises of how to associate events to actions, actions being used for specification while the actual behaviour of a system is defined in

[3]Usually the conflict relation is also symmetric, but then we have problems in specifying a situation where, though the events are in conflict, they may occur in the same computation, provided that a certain order is respected.

terms of events. The nature of an answer depends on the specific computational model, e.g. states having a memory and actions changing the memory with concurrent processes sharing the memory, or communication being achieved by synchronization of events. The variety of models envisaged suggest to look for a *framework* for specification rather than for a specific style of specification. We propose conditions for such a framework in section 4.

We should, however, keep in mind that we ultimately want to specify event-based behaviour. A *logic of actions* should map down to a *logic of events.* Section 3 will analyze the ingredients of the latter. Specifically, we investigate the compositionality of 'event specification' with regard to synchronization operators.

Section 2 restates some results about "pure event automata", and some preliminary conclusions are stated in section 5.

2 Synchronization of Pure Event Automata

A 'logic of events' refers to behaviour only in terms of *events.* Hence we restrict our attention to event automata the states of which are sets of events.

Definition 2 *A* pure event automaton $\mathcal{E} = (E, St, \vdash)$ *consists of*

- *a set E of* events,
- *a set St of* states *such that, for all $X \in St$, X is a finite subset of E, and*
- *a transition relation $X \vdash Y$ such that $Y = X \uplus \{e\}$.*

It is well known that synchronization operators can be defined in terms of *partially synchronous products* and *restriction*[4] [19]. We recall the definitions and results of [12, 13].

Definition 3 *A (partially synchronous) homomorphism $h : \mathcal{E} \rightarrow \mathcal{E}'$ of event automata consists of a partial mapping $h : E \rightarrow E'$ on events such that all the infrastructure is preserved, i.e.*

- *$h(X) \in St'$ if $X \in St$,*
- *h is injective on all states, meaning that, for all $x, y \in X \in St$, if $h(x)$ and $h(y)$ are defined, and $h(x) = h(y)$, then $x = y$, and*

[4]If the events are labeled, then the relabeling operation plays a major rôle.

- *$h(X) \vdash h(X) \cup \{h(e)\}$ if $X \vdash X \uplus \{e\}$, and if $h(e)$ is defined,*

where $h(X) = \{h(x) \in E' : x \in X$ and $h(x)$ is defined$\}$.

Injectivity on states guarantees that every transition coincides with an event, and that $h(X) \vdash h(X) \cup \{h(e)\} = h(X) \uplus \{h(e)\}$. Pure event automata and their morphisms form a category.

The partially synchronous product of two event automata is defined by:

Definition 4 *The product $\mathcal{E}_1 \prod \mathcal{E}_2$ of event automata $\mathcal{E}_1$ and $\mathcal{E}_2$ is defined by $(E, St, \vdash)$ where*

- *$E = E_1 \times_* E_2 = E_1 \cup (E_1 \times E_2) \cup E_2$, and the projections $\pi_i : E \to E_i$ are partial mapping defined by $\pi_i(e_1, e_2) = e_i$, $\pi_i(e_i) = e_i$ if $e_i \in E_i$ for $i = 1, 2$.*
- *$X \in St$ iff $\pi_1(X) \in St_1$ and $\pi_2(X) \in St_2$, and if π_1 and π_2 restricted to states are injective,*
- *$X \vdash X \cup \{e\}$ iff X is a state, and if $\pi_1(X) \vdash \pi_1(X) \uplus \{\pi_1(e)\}$ whenever $\pi_1(e)$ is defined, and if $\pi_2(X) \vdash \pi_2(X) \uplus \{\pi_2(e)\}$ whenever $\pi_2(e)$ is defined.*

Proposition 5 *[12, 13] The product of two pure event automata is a categorical product.*

The restriction operator is defined by[5]

Definition 6 *Let $\mathcal{E}$ be an event automaton and let $h : E' \to E$ be a partial mapping. Then we can construct an event automaton $h^*(\mathcal{E}) = (E', St', \vdash')$ by*

- *$St' = \{X \subseteq E' : h(X) \in St$ and h is injective on $X\}$, and*
- *$X \vdash' X \uplus \{e\}$ if $h(e)$ is undefined, or if $h(X) \vdash h(X) \uplus \{h(e)\}$ otherwise.*

Note that $h : h^*(\mathcal{E}) \to \mathcal{E}$ is an homomorphism of event automata.

Synchronization operations are obtained by a combination of product and restriction. The use of restriction avoids a more complicated direct definition (see [18]).

[5]The definition is slightly more general than necessary in that one usually restricts attention to inclusions.

Definition 7 *Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two event automata and let $\mathcal{A} \subseteq E_1 \times E_2$ be a set (*the synchronization set*). Define $\mathcal{E}_1 \parallel_{\mathcal{A}} \mathcal{E}_2 = \lambda^*(\mathcal{E}_1 \prod \mathcal{E}_2)$ where $\lambda : \mathcal{A} \cup (E_1 \setminus \mathcal{A}_1) \cup (E_2 \setminus \mathcal{A}_2) \to E_1 \times_* E_2$ denotes the embedding with $\mathcal{A}_i = \pi_i(\mathcal{A})$ for $i = 1, 2$.*

The definitions naturally subsume those for (prime, flow) event structures as demonstrated in [13] but as well allow to define synchronization of trace automata (here an admissible state is the set of prefixes of a trace) or Mazurkiewicz traces [7].

3 A Logic of Events

The basic ingredients of a pure event automaton to be specified are the states and the transition relation.

Properties of states may be stated in terms of a simple logic with atomic predicates e which assert that an event e has occurred: a state X satisfies the predicate e, notation $X \models e$, if e is an element of X. Then a state can be characterized by a (finite) conjunction of such atomic formulas where a conjunction is defined by the usual scheme. We may easily introduce more logical infrastructure, for instance $e \Leftrightarrow \neg e'$, which states that e and e' are inconsistent. A candidate for such a logic may be the geometric logic introduced in [4].

Similarly, atomic predicates of the form $\phi \vdash e$ may be used to specify enabling of the event e in that $X \vdash X \cup \{e\}$ whenever $X \models \phi$ and $e \notin X$. We are here not so much concerned with details in style but to stress the distinction between specifications and the automata which satisfy a specification.

Definition 8

- *We assume the existence of some appropriate "logic of events" which allows the statement that a finite set $X \subseteq E$ of events* satisfies *a formula ϕ, in short $X \models \phi$. The formulas are "typed" by E in that the respective set of events is well understood.*
- *A* specification *Spec over events E consists of a set of* state formulas *ϕ, and a set of* event declarations *$\psi \vdash e$ where ϕ, ψ are of type E, and where $e \in E$.*
- *Let $\mathcal{E} = (E, St, \vdash)$ be an event automaton. Then*

  (i) $\mathcal{E} \models \phi$ if, for all states $X \in St$, $X \models \phi$, and

  (ii) $\mathcal{E}, \psi \models e$ if, for all states $X \in St$, $X \vdash X \uplus \{e\}$ whenever $X \models \psi$ and $e \notin X$.

$\mathcal{E}$ satisfies *a specification Spec, notation $\mathcal{E} \models Spec$, if $\mathcal{E} \models \phi$ for the state formulas ϕ in Spec, and if $\mathcal{E}, \psi \models e$ for all event declarations $\psi \vdash e$ in Spec.*

This definition is parametric in the "logic" used. We concentrate on the structure of the specification and on the satisfaction of a given specification, rather then trying to capture all these notions in a logic. The choice of a specific logic is dependent on the nature of the problem to be analyzed.

In general there might be more than one automaton which satisfies a given specification, quite in contrast to the usual approach where "reachable" automata are considered. Enabling, however, provides a certain degree of "liveness" for every automaton satisfying a specification in that an event can happen if it is enabled.

Proposition 9 *For every specification Spec of type E there exists a minimal reachable automaton $Min(Spec)$ of type E such that $Min(Spec) \models Spec$.*

More precisely, for every automaton $\mathcal{E}$ such that $\mathcal{E} \models Spec$ there exists a unique homomorphism $\iota : Min(Spec) \rightarrow Reach(\mathcal{E})$. $Min(Spec)$ is constructed thus: If $\emptyset \not\models \phi$, then this is the empty automaton. Otherwise it is generated by (1) $\emptyset$ is a state, and (2) whenever X is a state, and there exists some event declaration $\psi \vdash e$ such that $X \models \psi$ and $X \uplus \{e\} \models \phi$, then $X \uplus \{e\}$ is a state, and $X \vdash X \uplus \{e\}$.

Some compositionality results may be achieved even on this level of abstraction, provided we add to the logic.

Definition 10 *Let ϕ be a formula of type E', and let $h: E \rightarrow E'$ be a partial mapping. Then $h^*\phi$ is a formula of type E, and, for all $X \subseteq E$, let $X \models h^*\phi$ iff $h(X) \models \phi$ and h is injective on X.*

Proposition 11 *For all event automata $\mathcal{E}'$ of type E', and for all partial mappings $h : E \rightarrow E'$,*

(i) $h^*(\mathcal{E}') \models h^*\phi$ if $\mathcal{E}' \models \phi$, and

(ii) $h^*(\mathcal{E}'), h^*\psi \models e$ if $\mathcal{E}', \psi \models h(e)$, provided that $h(e)$ is defined.

If h is surjective on states, the converse statements hold.

Corollary 12 • *Let h^*Spec denote the specification with state formulas $h^*\phi$ and event declarations $h^*\psi \vdash e$ such that ϕ and $\psi \vdash h(e)$, if $h(e)$ is defined, are in Spec. Then $h^*(\mathcal{E}') \models h^*Spec$ whenever $\mathcal{E} \models Spec$.*

- *If $\mathcal{E} \models h^*Spec$ then $h(\mathcal{E})$ is well-defined, and $h(\mathcal{E}) \models Spec$, where $h(\mathcal{E})$ has states $h(X)$ with X being a state of $\mathcal{E}$, and transitions $h(X) \vdash h(Y)$ if $X \vdash Y$ in $\mathcal{E}$.*

Proposition 13 *Let $X \models \phi \wedge \psi$ if $X \models \phi$ and $X \models \psi$. Define the specification $Spec_1 \prod Spec_2$ to consist of the state formulas $[\pi_1]\phi_1 \wedge [\pi_2]\phi_2$ and the event declarations*

$[\pi_1]\phi_1 \vdash e$ if $\pi_1(e)$ is defined, $\pi_2(e)$ is not defined, and $\phi_1 \vdash \pi_1(e)$ is in $Spec_1$

$[\pi_2]\phi_2 \vdash e$ if $\pi_2(e)$ is defined, $\pi_1(e)$ is not defined, and $\phi_2 \vdash \pi_2(e)$ is in $Spec_2$

$[\pi_1]\phi_1 \wedge [\pi_2]\phi_2 \vdash e$ if both $\pi_1(e)$ and $\pi_2(e)$ are defined, and $\phi_i \vdash \pi_i(e)$ is in $Spec_i$ for $i = 1, 2$.
Then

(i) $\mathcal{E}_1 \prod \mathcal{E}_2 \models Spec_1 \prod Spec_2$ iff $\mathcal{E}_1 \models Spec_1$ and $\mathcal{E}_2 \models Spec_2$, and

(ii) If $\mathcal{E} \models Spec_1 \prod Spec_2$ then $\pi_1(\mathcal{E}) \models Spec_1$ and $\pi_2(\mathcal{E}) \models Spec_2$.

The results support modular specifications involving synchronization operators provided the "logic" satisfies the requirements for $X \models \phi$.

In designing a concrete logic, a possible choice may be thus: for every event $e \in E$, let, ambiguously, e denote a predicate such that $X \vdash e$ iff $e \in X$.[6] We may as well add negation by $X \models \neg\phi$ iff $X \not\models \phi$. Then we can specify inconsistency of events, $e \Leftrightarrow \neg e'$, or *asymmetric conflicts* as mentioned in the introduction: $\neg e \vdash e'$ implies that an occurrence of e prohibits the one of e' (but not vice versa).

As an example we consider a protocol, which is informally specified by the figure 2. Two agents, namely "A" and "B" communicate by means of a channel "K". C stands for "connect", D for "data" and R for "release". The agent "A" may request a connection (C_{req}). The agent "B" responds by events (C_{ind} and C_{rsp}), and "A" receives an acknowledgment (C_{cnf}). The same procedure applies for release. Moreover, the agent "B" may send a data packet (D_{req}), or not, which is received by A (D_{ind}). The data packet should be received by A before the release of the connection is confirmed (R_{cnf}).

We give the specification of the agents and of the channel:

[6]In combination with conjunction, this allows us to specify the enabling relation of arbitrary event structures.

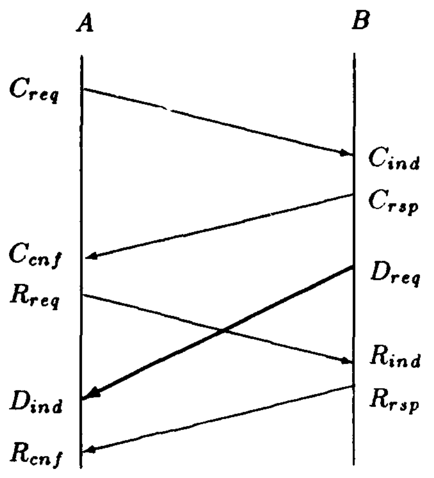


Figure 2: A communication protocol

Agent A	*Agent B*	*Channel K*
Declarations	Declarations	Declarations
$\vdash C_{req}$	$\vdash C_{ind}$	$\vdash C_{req}$
$C_{req} \vdash C_{cnf}$	$C_{ind} \vdash C_{rsp}$	$C_{req} \vdash C_{ind}$
$C_{cnf} \wedge \neg R_{cnf} \vdash D_{ind}$	$C_{rsp} \wedge \neg R_{ind} \vdash D_{req}$	$\vdash C_{rsp}$
$C_{cnf} \vdash R_{req}$	$C_{rsp} \vdash R_{ind}$	$C_{rsp} \vdash C_{cnf}$
$R_{req} \vdash R_{cnf}$	$R_{ind} \vdash R_{rsp}$	$\vdash D_{req}$
		$D_{req} \vdash D_{ind}$
		$\vdash R_{req}$
		$R_{req} \vdash R_{ind}$
		$\vdash R_{rsp}$
		$R_{rsp} \vdash R_{cnf}$

the negation is used to specify that an asymmetric conflict between the sending (receiving) of a data and the release of the connection exists.

We can now synchronize the corresponding events of agent A and the channel K, and then compose with agent B to obtain (modulo renaming) the specification

Protocol
Declarations
$\vdash C_{req}$
$C^K_{req} \vdash C_{ind}$
$C^B_{ind} \vdash C_{rsp}$
$C^A_{req} \wedge C^K_{rsp} \vdash C_{cnf}$
$C^B_{rsp} \wedge \neg R^B_{ind} \vdash D_{req}$
$C^A_{cnf} \wedge D^K_{req} \wedge \neg R^A_{cnf} \vdash D_{ind}$
$C^A_{cnf} \vdash R_{req}$
$C^B_{rsp} \wedge R^K_{req} \vdash R_{ind}$
$R^B_{ind} \vdash R_{rsp}$
$R^A_{req} \wedge R^K_{rsp} \vdash R_{cnf}$

The following notational conventions are used: the superscripts of the predicates indicate the origin, e.g. C^A_{req} refers to the event C_{req} of A.[7] Since only events with the same names are synchronized, we replace e.g. (C^A_{req}, C^K_{req}) by C_{req}. The injectivity conditions for transformed predicates are trivially satisfied. Actually, one might strip off the superscripts altogether because of this specific nature of the example.

Another, more sophisticated example, is Hoare's *trace logic* [5]. Let A be a set of actions. Formulas of Hoare's logic state properties of traces $w \in A^*$. E.g. for the famous vending machine, the formula $\#chocs \leq \#coins$ states that always more coins are inserted than chocolates delivered in an admissible trace (= computation). Traces are events in our terminology.[8] The corresponding event (trace) automata have states $Pref(w)$ being the set of prefixes of the trace w (we refrain from capturing this by a state formula). Trace formulas implicitly define an enabling relation in that $w \vdash wa$ whenever wa is an admissible trace. The "formula" w in $w \vdash wa$ checks for presence of the trace w as maximal trace in a set of events.[9]

We claim no elegance in this translation; one should stick to Hoare's style. But a detail is noteworthy after all: Hoare's logic refers to events and not to actions as basic semantic entities. In fact, more elaborate versions of his logics using *refusal sets* [5] or *ready sets* [9] mix references to events and actions. The "ready set semantics", in particular, compares well to our framework in that $\phi \vdash a$ can be used to declare the action a

[7]C^A_{req} abbreviates the formula $[syn_2][\pi_1][syn_1][\pi_1]C_{req}$ where $[syn_1]$ denotes the transformation synchronizing A and K, and $[syn_2]$ synchronizes the synchronized product of A and K with B. The projection in the middle being used for the construction of the product. The example is extensively discussed in [13].

[8]Note the compliance with the stipulation of events occurring only once in a computation.

[9]Concerning synchronization operators, there are a lot of subtleties involved, which we comment on further below.

to be "ready" if the formula ϕ holds. We resume this theme in the next section.

4 Towards a Logic of Action?

4.1 A Taxonomy of Actions

Specification in terms of events is somewhat limited in that events occur only once in a computation. One would prefer think in terms of *actions* which may occur repeatedly during a run of a system. For instance, if we slightly modify the protocol example to allow several data packets to be sent, we have to introduce two events D_{ind} and D_{req} for data packets, though the events are indistinguishable from an external point of view. Actions relate to events as, for instance, procedures do to run-time calls; actions are generic scripts which, when activated, determine the (next step of) behaviour depending on the present state and, maybe, parameters.

There is little difficulty to add actions. We supply event automata with a labeling mapping $\lambda : E \rightarrow A$ which associate events to the actions they represent. Similarly, we turn the logic of events into a *logic of actions*: we use $\phi \vdash \mathtt{a}$ to stipulate that an event automaton has a move $X \vdash X \cup \{e\}$ if $X \models \phi$ and if there exists an event $e \notin X$ such that $\lambda(e) = \mathtt{a}$, a being an action. Again, the definition is parameterized by satisfaction $s \models \phi$. All this, however, is formal manipulation. The logic needs more sophistication.

Let us consider the behaviour of a stack. There are two actions to modify the stack, a **push** and a **pop** action, and there are some predicates, like $top = i$ or *isempty*. We are allowed to push whenever we are in a legal state, and pop if the stack is not empty. A tentative specification is then given by the following table:

Stack	
States	Declarations
legal	$\vdash$ **push(i)**
	$\neg isempty \vdash$ **pop**

The state formula *legal* asserts that in any feasible computation a **push** action is performed at least as many times as a **pop** action.

We need a more complex notion of satisfaction than just the statement that some event has occurred in order to capture the intended effects of the operations. Let $[\mathtt{a}]\phi$ state that "the observation ϕ holds after execution of the action a" [3]. Then

$$[\mathbf{push(i)}]top = i$$

appears as reasonable but we have to "implement" the predicate $top = i$. Even if we label events with actions, we do not keep track of the ordering in which the events have occurred, which is a necessary prerequisite to compute the proper top of the stack.

We may use traces A^* giving up "true concurrency". In order to define $top = i$, we then apply the equalities $top(\mathbf{v.push(i)}) = i$ and $top(\mathbf{v.push(i).pop.w}) = top(\mathbf{v.w})$ to the longest trace, i.e. the last event.

As an alternative, one might enrich a state by an additional component, a stack of the usual data type variety. If we use an "stack" to refer to this additional component, the behaviour of a stack is fairly well reflected by the formulas

$$stack = s \Rightarrow [\mathbf{push(i)}]stack = push(s, i)$$
$$stack = push(s, i) \Rightarrow [\mathbf{pop}]stack = s$$

One wonders about the distinction between, for instance, the action **push** and the operation *push*. There is very little, if we are only concerned with the behaviour of one stack. But we could try to capture the well known theorem that two stacks plus finite control can be used to simulate a Turing machine[10]. Then the push and top actions of the two stacks need to be synchronized, e.g.

$$top_1 = 0 \vdash \mathbf{pop_1} || \mathbf{push_2}(1)$$

may be used to write a symbol on the Turing tape where || is a synchronization operator on actions (1 denotes a symbol, 0 a blank). We note: *actions may synchronize* or, more generally, *may be visible to other processes*, hence are the ingredients of reactive behaviour, while operations are just functions on some data. By the way, the data type *stack* "freezes" computation history in a rather obvious sense.

There is an asynchronous variant using the communication primitives `c!v` and `c?x` of process algebra:

$$\neg isempty_1; \mathtt{write?x} \vdash \mathbf{pop_1}; \mathtt{ready-to-write!x}$$
$$\mathtt{ready-to-write?x} \vdash \mathbf{push_2(x)}$$

Here the finite control issues the write command. x may either be 0 or 1.

[10]See, for instance, [6].

The example may be sufficient to demonstrate that additional structure is required to support the semantics of actions. Tentatively, there are two types of extensions, namely to enrich the events by some order relation mimicking dependence or temporal precedence, and to add a data component to states. Actions might be classified with regard to the nature and conditions of the changes caused:

- *Imperative actions*[11] the effect of which is independent of whatever has happened beforehand, but the data component of states is affected,
- *History-dependent actions* the effect of which to some extent depends on previous behaviour, and
- *Declarative actions* which do not affect the data component.

In the first variant of our example the action **pop** is clearly history dependent, whereas the action **push** is imperative. The order on events reflects the history. Given an additional data component ("a frozen history"), history-dependent actions may be turned into imperative ones. Communication primitives like `c?x` are declarative since neither a change of a data component is involved nor are they relevant for history if executed. This crude taxonomy is complemented by an orthogonal distinction between

- *hidden actions* which change a *local* state, i.e. the effects can be observed only locally within a process, and
- *visible actions* the occurrence of which are known to other processes which, for instance, can decide or are forced to synchronize.

Whether actions are hidden or visible depends on the specific design. For instance, we may choose to consider only communication primitives as visible, then synchronizations such as $\mathbf{pop}_1||\mathbf{push}_2(1)$ are excluded.

All these distinctions should be supported by a logic of actions.

4.2 A Framework for Dealing with Actions

We do not venture to propose a "definitive" methodology for the specification of reactive behaviour. We rather ask for conditions for a logic of actions which are consistent with the logic of events.

[11] Called Markovian in [2].

We distinguish between states s and the underlying set $ev(s)$ of events as in the definition of event automata as stated in the introduction, for evident reasons. Moreover, we assume existence of some appropriate "logic" where $s \models \phi$ states that $s \in St$ *satisfies* the formula ϕ. All data are "typed" by A, i.e. for each set of actions A, we assume a set of events $Ev(A)$, a set of states $St(A)$, and a set $Form(A)$ of formulas to be given, as well as total mappings $ev_A : St(A) \to Ev(A)$ and $\lambda_A : Ev(A) \to A$ such that $ev(s) \subseteq_{\mathrm{Fin}} Ev(A)$ for every $s \in St(A)$.

Furthermore, we assume that every partial mapping $\sigma: A \to A'$ induces partial mappings $\sigma_{Ev}: Ev(A) \to Ev(A')$ and $\sigma_{St} : St(A) \to St(A')$, and a total mapping $\sigma^*: Form(A') \to Form(A)$. These mappings are supposed to satisfy the following requirements:

- $\sigma_{Ev}(ev_A(s)) = ev_{A'}(\sigma_{St}(s))$ provided $\sigma_{St}(s)$ is defined,
- for every $e \in Ev(A)$, $\sigma(\lambda_A(e) = \lambda_{A'}(\sigma_{Ev}(e))$ provided $\sigma_{Ev}(s)$ is defined, and
- $s \models \sigma^*\phi$ iff $\sigma_{St}(s) \models \phi$.

We refer to such a setup as a "frame" (for sake of a better term). We recast our main definitions.

Definition 14 *An* event automaton $\mathcal{E}$ *of type* A *(henceforth) consists of*

- *a set* $St \subseteq St(A)$ *of* states *with set* $ev(s) \subseteq Fin(E)$ *of events for every* $s \in St$,
- *a transition relation* $s \vdash s'$, *where* $s, s' \in St(A)$, *such that* $ev(s') = ev(s) \uplus \{e\}$ *for some event* $e \in Ev(A)$, *and*

A partially synchronous homomorphism consists of a partial mapping $\sigma : A \to A'$ *such that*

- *for all* $s \in St$, $\sigma_{St}(s) \in St'$,
- σ_{Ev} *is injective on* $ev_A(s)$, *and*
- $\sigma_{St}(s) \vdash_* \sigma_{St}$ *if* $s \vdash s'$

We restate the constructions on event automata with minor changes.

Definition 15 *The (asynchronous) product* $\mathcal{E}_1 \prod \mathcal{E}_2$ *of event automata* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *of type* A_1 *and* A_2, *respectively, is an event automata of type* $A = A_1 \times_* A_2$ *defined by* $(St, \vdash)$ *where*

- $(s_1, s_2, X) \in St$ *if*
    - $s_1 \in St_1$, $s_2 \in St_2$, *and* $X \subseteq Ev(A)$,
    - *for all* $e \in X$, $\pi 1_{Ev}(e) \in ev_{A_1}(s_1)$ *and* $\pi 2_{Ev}(e) \in ev_{A_2}(s_2)$, *and*
    - $\pi 1_{Ev}$ *and* $\pi 2_{Ev}$ *are injective on* X,
- $(s_1, s_2, X) \vdash (s_1', s_2', X \uplus \{e\})$ *if*
    - (s_1, s_2, X) *and* $(s_1', s_2', X \uplus \{e\})$ *are states, and*
    - $s_1 \vdash s_1'$ *or* $s_2 \vdash s_2'$,

Definition 16 *Given an event automaton $\mathcal{E}$ of type A, and given a partial mapping $\sigma : A' \rightarrow A$, then the restriction of $\mathcal{E}$ to A', notation $\mathcal{E}\lceil(\sigma : A' \rightarrow A)$, is the event automata $(St', \vdash')$ of type A' where*

- $(s, \sigma, X) \in St'$ *if*
    - $s \in St$, *and* $X \subseteq Ev(A')$,
    - *for all* $e \in X$, $\sigma_{Ev}(e) \in ev_A(s)$, *and*
    - σ_{Ev} *is injective on* X,
- $(s, \sigma, X) \vdash' (s', \sigma, X \uplus \{e\})$ *if*
    - (s, σ, X) *and* $(s', \sigma, X \uplus \{e\})$ *are states, and*
    - $s \vdash s'$.

Finally we introduce a relabeling operation.

Definition 17 *Given an event automaton $\mathcal{E} = (St, \vdash)$ of type A and a relabeling mapping $\sigma : A \rightarrow A'$, the relabeling of $\mathcal{E}$, notation $\mathcal{E}[\sigma]$, is the event automata $\mathcal{E} = (St', \vdash')$ such that*

- $\sigma_{St}(s) \in St'$ *iff* $s \in S$, *and*
- $\sigma_{St}(s) \vdash \sigma_{St}(s')$ *if* $s \vdash s'$ *and if* $\sigma_{St}(e)$ *for* $e \in ev(s') \setminus ev(s)$.

As such, the definitions make little sense (except for relabeling). Nothing guarantees existence of the respective data. We commit the usual fraud to claim existence of what is needed. For instance, given a partial mapping $h: A \rightarrow A'$, for all states $s' \in St(A')$ and all sets $X \subseteq Ev(A)$, we claim existence of exactly one state s of type A such that such that $ev(s) = X$ and $\lambda_{St}(s) = s'$. If we use (s, λ, X) to denote such a state, restriction is well-defined in a given frame. The requirement is sound in that states (s, λ, X) should differ only in terms of events e, the images of which are undefined w.r.t. h_{Ev}. Similarly, additional requirements can be imposed to recover the definition of products.

Proposition 18 *Assume that, for every pair of states $s_1 \in St(A_1)$ and $s_2 \in St(A_2)$, and every set $X \subseteq Ev(A_1 \times_* A_2)$, there exists a unique state (s_1, s_2, X) such that $\pi_{1_{Ev}}(s_1, s_2, X) = s_1$, $\pi_{2_{Ev}}(s, s_2, X) = s_2$, and $ev(s_1, s_2, X) = X$. Then the construction in 15 defines a (categorical) product of event automata in such a frame.*

Pure event automata satisfy the conditions trivially if we consider events as actions.

Interleaving semantics in terms of traces provides another example. Given a set of labels A let $Ev(A) = A^*$ be the set of traces on this alphabet. We define $St(A) = \{Pref(w) | w \in A^*\}$ as being the prefix-closure of traces. The ev_A is the identity map, and $\lambda_A(\mathbf{wa}) = \mathbf{a}$. We leave open which formulas may be used.

The *asynchronous product* of two sets $T_1 \subseteq A_1^*$ and $T_2 \subseteq A_2^*$ of traces is given by a set $T \subseteq (A_1 \times_* A_2)^*$ such that $w \in T$ iff $\pi_i(w) \in T_i$ and the projections π_i are injective on T. The projection mappings are the canonical extension of those on actions to traces. Various synchronization operators can be defined restricting to suitable sets of labels, for instance the usual CSP operator is obtained as restriction to labels $(\mathbf{a}, \mathbf{a})$ in the product alphabet.

The conditions for the asynchronous product of traces match exactly those for states used in the definition of product automata. The condition on the moves is trivially satisfied.

Similarly, we can set up a frame of *synchronization trees* [8], or a frame of Mazurkiewicz traces [7]. Other choices, are to enrich the set of events $ev_A(s)$ of a state by a partial order representing dependency (as a generalization of traces [17]), or to extend a state by data components in that we allow to access internal data by means of attributes $s.a$. In the latter case, definitions should be given relative to a *signature* which comprises actions as well as attributes. The mathematics of this more general setup will be investigated in [14].

We turn our attention to specifications. We follow the pattern of section 3.

Definition 19 *An* elementary specification *Spec over actions A consists of*

- *a set of* state formulas ϕ, *and*
- *a set of* action declarations $\psi \vdash \mathbf{a}$ *where* ϕ, ψ *are of type* A, *and where* $\mathbf{a} \in A$.

All formulas are of type A.

Definition 20 *Let Spec be a specification and* $\mathcal{E}$ *be an event automaton of type* A. *Then* $\mathcal{E}$ satisfies *Spec, notation* $\mathcal{E} \models Spec$, *if*

(i) $\mathcal{E} \models \phi$ *for all state formulas* ϕ *of Spec, i.e. , for all states* $s \in St$, $s \models \phi$,

(ii) $\mathcal{E}, \psi \models \mathbf{a}$ *for all action declarations* $\psi \vdash \mathbf{a}$, *i.e., for all states* $s \in St$ *such that* $s \models \psi$, *there is some state* $s' \in St$ *such that* $s \vdash s'$ *and* $\lambda(e) = \mathbf{a}$ *(notation* $s \overset{\mathbf{a}}{\vdash} s'$*).*

Proposition 21 *Assuming that all definitions are properly translated we have:*

- $\mathcal{E}'\lceil(\lambda : A \rightarrow A') \models \lambda^* Spec$ *whenever* $\mathcal{E} \models Spec$,
- $\mathcal{E}_1 \prod \mathcal{E}_2 \models Spec_1 \prod Spec_2$ *iff* $\mathcal{E}_1 \models Spec_1$ *and* $\mathcal{E}_2 \models Spec_2$, *and*
- *If* $\mathcal{E} \models Spec_1 \prod Spec_2$ *then* $\lambda_1(\mathcal{E}) \models Spec_1$ *and* $\lambda_2(\mathcal{E}) \models Spec_2$.

We refer to such specifications as *elementary* since they are concerned only with states and with enabling.

4.3 Relating Events and Actions

Every style of specification of reactive behaviour is probably a compromise between specifying in terms of actions and in terms of events. Hoare-style trace logics may serve as a witness. Not surprisingly, most styles of specification make implicit assumptions about the nature of events. Traces or synchronization trees are the structures commonly used for representing events, while equivalences on top determine their nature. The preference is justified. Traces and trees are natural structures to generate from actions, both are well understood, and we have the convenience that the last event (maximal trace, tree) encodes all the history.[12] However, traces or trees do not support "true concurrency" in terms of an operator as, for instance, the gluing of synchronization trees supports "alternative". We

[12]hence support history-dependent actions, according to our taxonomy.

should keep in mind here that the presentation of the semantic concepts in terms of operators is crucial for logics (at least for those with linear notation).

The absence of a "concurrency" operator of handy nature caused the genesis of a long line of true concurrency models, the first being Petri's [10]. Partial orders have been used very early to model *causality* [11, 16], or, if you want, generalize the sequentiality of traces, with *independence*, or concurrency, being a derived notion, but not an algebraic kind of operator. These models do not support "alternative". Event structures combine the three aspects of "sequence", "alternative", and "concurrency", as do Mazurkiewicz traces where sequence and independence are the basic concepts. So the short historical survey takes us right back to the starting point.

To be fair, Vaughn Pratt addressed the question of how to generate partial orders [16], as does recent work based on categorical structures[13] [1]. Still, the work does not provide a logic which is as easy to use and as natural as Hoare's trace logic or the various brands of temporal logics [15], nor does it cover aspects such as history-dependence.

Maybe it is worth considering the communication protocol specified in terms of of events. If more than one data packet is exchanged, then we have several events corresponding to the "actions" D_{req} and D_{ind}. The "action specification" $D_{req} \vdash D_{ind}$ is quite useless. We have to relate each event of sending of a data package to its arrival while the "specification" at best states that some sending of a data package corresponds to some arrival. We may refer to respective events by indexing, in that we consider events $\{D_{req_i} | i > 0\}$ and $\{D_{ind_i} | i > 0\}$. Then the declaration $D_{req_i} \vdash D_{ind_i}$, for all i, will achieve the desired result but we are back on the level of events. We can, however, use the additional information by restricting states to those such that the number of events labeled by D_{req} is greater or equal to the number of events labeled by D_{ind}. If we stipulate that the (action) formula D_{req} holds if the number of D_{req} events is greater than the number of D_{ind} events. The action D_{ind} then enables any of the still missing arrival events. If we want be sure that the channel works in FIFO mode, we need a more structured set of events as well as of states: we assume events to be ordered, for instance by $D_{ind_i} < D_{ind_j}$, if $i < j$ and require that the events in a state respect this order. The latter can be achieved as well by sacrificing auto-concurrency[14]; working

[13]The latter seems to support imperative actions only, according to our nomenclature. This is inherent in the bipartiteness of Petri nets. A marking does not record the events which have previously occurred

[14]meaning that there are concurrent events labeled by the same action.

on Mazurkiewicz traces, we may say that (an action) formula D_{req} is satisfied if the "last" D_{req} is not followed by a D_{ind} (with regard to the order on the traces).

Without exhausting all the possible variations, the example should demonstrate that a precise understanding of the relationship of events and actions is crucial and that there is a variety of choices if true concurrency is taken into account. Hence no universal procedure for relating events and actions can be stated, but some requirements can. Given a set of actions A the corresponding notions of events and states have to be fixed as well as satisfaction. Our logic of actions is based on this infrastructure. Moreover, for every action declaration $\phi \vdash \mathbf{a}$, the effect of the action **a** must be exactly known, namely which event is triggered, and how it relates to the past history.

Formally, for every action declaration

$$\phi \vdash \mathbf{a},$$

a logic of actions must determine a set of event declarations

$$\psi \vdash \mathbf{a}$$

such that, for all states $s \in St(A)$ in the given frame,

$$s \models \psi \text{ if } s \models \phi$$

and

$$\lambda(e) = \mathbf{a}.$$

Moreover

$$\sigma^* \psi \text{ if } \sigma^*_{\mathrm{Ev}} \phi.$$

should hold for the respective ψ's. This guarantees compatibility of the synchronization operators for specifications on the level of actions and of events. This condition completes our requirements for a frame.

The trace logic is an obvious example for a frame. If a trace w satisfies the formula ϕ, then $\phi \vdash \mathbf{a}$ enables a transition to wa. Similarly, Mazurkiewicz traces define a frame though they do not support auto-concurrency. Auto-concurrency is well known to be a major problem.

Our taxonomy of actions further suggests that data components are needed, for instance, for modeling shared memory. Data components might be accessed using attributes for *observation* separating data and events. Then a data logic can be designed quite independently, or rather, a logic of actions may be parameterized by a logic for data.

5 Conclusions

We may summarize as follows:

- There is an elementary style of specification in terms of events which is compositional with regard to synchronization operators.
- Similar results may be achieved for a logic of actions, but
- a framework for specification must clearly state how to relate actions and events.
- Conditions are given for frameworks such that the logics interact appropriately.
- Compositionality of specifications with regard to synchronization operators is established.

We consider our setup as a first step. More work needs to be done: the basic model should be enriched in order to support the various kinds of actions of our taxonomy (which is currently worked out in [14]), and more logical infrastructure must be provided. There are various lines of thought for the latter. For instance, one may introduce formulas ψ_a and extend satisfaction to $(s, s') \models \varphi$ to specify 'one-step' behaviour. The framework is easily extended to cope with such "static semantics". The situation is more complicated with "dynamic" varieties of logic.

Let us use a formula of the form $[\mathrm{a}]\phi$ (where $\phi \in \mathit{Form}(A)$) with satisfaction being defined by: $\mathcal{E}, s \models [\mathrm{a}]\varphi$ if, for all moves $s \overset{\mathrm{a}}{\vdash} s'$, $s' \models \phi$. Then we can prove by a straightforward argument

Proposition 22 *For a partial mapping $\lambda \colon A \to A'$, an automaton $\mathcal{E}'$ of type A', and a formula $\phi \in St(A')$, we have that $\lambda^*(\mathcal{E}'), s \models [a]\lambda^*\phi$ iff $\mathcal{E}', \lambda(s) \models [\lambda(a)]\phi$, provided that $\lambda(a)$ is defined.*

Indeed, the definition of these formulas expressing the "dynamic" of the system strongly depends on the taxonomy we have introduced before.

There may be more dynamic operators, and similar results should hold for them. Since satisfaction depends on the given automata, there seems little hope to give a uniform translation scheme but we note that the pattern for the "dynamic" enabling relation and the "predicate transformation" above is of striking similarity.

References

[1] Brown, C., Gurr, D., "Temporal Logic and Categories of Petri Nets", to appear

[2] Costa, J.F., Sernadas, A., Sernadas, C., "The Semantics of Object-Oriented Specification", in *Proc. of the IS-Core '92 Workshop*, INESC, Lisboa, 1992

[3] Fiadeiro, J., Sernadas, A., *Logics of Modal Terms for System Specification, Journal of Logics and Computation*, Nordwijkerhout, 1990

[4] Gunawardena, J., "Geometric logic, Causality and Event Structures", in *CONCUR'91*, Lecture Notes in Computer Science 527, 1991, pp. 266-280

[5] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1984

[6] Hopcroft, J.E., Ullman, J.D., *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 1979

[7] Mazurkiewicz, A., "Basic Notions of Trace Theory", in *Lecture Notes for the REX Summerschool in Temporal Logic*, Lecture Notes in Computer Science 354, Springer, 1988

[8] Milner, R., *Communication and Concurrency*, Prentice Hall, 1989

[9] Olderog, E.-R., Hoare, C.A.R., *Specification-oriented Semantics for Communicating Processes, Acta Informatica* 23, pp. 9 - 66

[10] Petri, C.A., *Kommunikation mit Automaten*, Institut für Instrumentelle Mathematik, Schriften des IIM, Nr. 2, Bonn 1962
also: Griffiths Air Force Base, Techn. Rep. RADC-TR-65-377, Vol.1, Suppl.1 (English translation), New York, 1966

[11] Petri, C.A., "Concurrency", in *Net Theory and Applications*, Lecture Notes in Computer Science 84, Springer, 1979

[12] Pinna, G.M., Poigné, A., "On the Nature of Events", in *Proc. MFCS'92*, Lecture Notes in Computer Science 629, 1992, pp. 430-441

[13] Pinna, G.M., Poigné, A., "On the Nature of Events", GMD-Arbeitspapier, to appear

[14] Pinna, G.M., Poigné, A., *The Mathematics of Event Automata*, in preparation

[15] Manna,Z. , Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems Specification*, Springer, 1992

[16] Pratt, V., "Modeling Concurrency with Partial Orders", *International Journal of Parallel Programming*, Vol.15, 1986

[17] Rensink, A., "Posets for Configurations!", in *CONCUR'92*, Lecture Notes in Computer Sciences 630, pp. 269-285

[18] Winskel, G., "Event Structures", in *Petri Nets: Applications and Relationships to other Models of Concurrency*, Lecture Notes in Computer Sciences 255, pp. 325-392

[19] Winskel, G., *Categories of Models for Concurrency*, Talk at ESPRIT-BRA CLICS-Meeting, Paris, 1990

A chemical abstract machine for graph reduction
Extended abstract

Alan Jeffrey
COGS, Univ. of Sussex, Brighton BN1 9QH, UK
alanje@cogs.susx.ac.uk

Abstract. Graph reduction is an implementation technique for the lazy λ-calculus. It has been used to implement many non-strict functional languages, such as lazy ML, Gofer and Miranda. Parallel graph reduction allows for concurrent evaluation. In this paper, we present parallel graph reduction as a Chemical Abstract Machine, and show that the resulting testing semantics is adequate wrt testing equivalence for the lazy λ-calculus. We also present a π-calculus implementation of the graph reduction machine, and show that the resulting testing semantics is also adequate.

1 Introduction

The lazy reduction strategy for the λ-calculus investigated by ABRAMSKY (1989) has only two reduction rules:

$$\frac{}{(\lambda x.E)F \to E[F/x]} \quad \frac{E \to E'}{EF \to E'F}$$

This can be compared with the full evaluation strategy of BARENDREGT (1984):

$$\frac{}{(\lambda x.E)F \twoheadrightarrow E[F/x]} \quad \frac{E \twoheadrightarrow E'}{EF \twoheadrightarrow E'F} \quad \frac{F \twoheadrightarrow F'}{EF \twoheadrightarrow EF'} \quad \frac{E \twoheadrightarrow E'}{\lambda x.E \twoheadrightarrow \lambda x.E'}$$

If the full evaluation strategy can terminate, then the lazy evaluation strategy will. For example, if we define:

$$\begin{aligned} \mathsf{K} &= \lambda xy.x \\ \mathsf{I} &= \lambda x.x \\ \mathsf{Y} &= \lambda x.((\lambda y.x(yy))(\lambda y.x(yy))) \end{aligned}$$

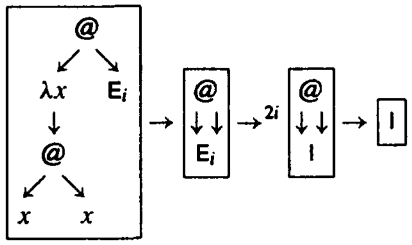
then $\mathsf{YI} \to^{\infty}$ but $\mathsf{KI(YI)} \not\to^{\infty}$, whereas $\mathsf{KI(YI)} \twoheadrightarrow^{\infty}$. However, the lazy evaluation strategy is very inefficient, since it may duplicate arguments when applying a function. For example, if we define:

$$\begin{aligned} \mathsf{E}_0 &= \mathsf{I} \\ \mathsf{E}_{i+1} &= (\lambda x.xx)\mathsf{E}_i \end{aligned}$$

Then $\mathsf{E}_i \twoheadrightarrow^{2i} \mathsf{I}$ but $\mathsf{E}_i \to^{2^{i+1}-2} \mathsf{I}$, that is the lazy strategy can be exponentially worse than the full strategy. Thus, the early functional languages, such as LISP (MCCARTHY *et al.*, 1962) used a strict reduction scheme rather than the lazy reduction scheme.

Graph reduction was introduced by WADSWORTH (1971) as a means of efficiently implementing the lazy reduction strategy. Rather than reducing syntax trees, we reduce syntax *graphs* which allows a more efficient representation of sharing. For example, we can represent the reduction of E_{i+1} as:

This work was supported by SERC project GR/H 16537.



Graph reduction has been used to implement non-strict functional languages such as JOHNSSON's lazy ML (1984), JONES's Gofer (1992) and TURNER's Miranda (1985). It is discussed in PEYTON JONES's textbook (1987).

However, there has been little work in the formal semantics of graph reduction. BARENDREGT *et al.* (1987) have shown that graph reduction is sound and complete with respect to term reduction. LESTER (1989) has shown that the G-machine of AUGUSTSSON (1984) and JOHNSSON (1984) is adequate wrt a denotational model of the lazy λ-calculus. In this paper, we provide an alternative presentation of graph reduction, as a *Chemical Abstract Machine* (CHAM), in the style of BERRY and BOUDOL (1990).

The CHAM was introduced as a way of presenting the operational semantics of parallel languages in a clean fashion. It has been used to give a semantics for MILNER's CCS (1989) and MILNER, PARROW and WALKER's π-calculus (1989).

Here, we shall give a semantics for parallel graph reduction with blocking, as described by PEYTON JONES (1987). We will show that this is an adequate semantics for the lazy λ-calculus, and that it can be implemented in a variant of the π-calculus.

2 The lazy lambda-calculus

The λ-calculus, introduced by CHURCH (1941), has the following syntax:

$$E ::= x \mid EE \mid \lambda x.E$$

where x ranges over an infinite set of variables. This can be given a number of operational semantics, but we shall only look at two of these. We shall call these the *lazy* semantics:

$$\frac{}{(\lambda x.E)F \to E[F/x]} \quad \frac{E \to E'}{EF \to E'F}$$

and the *full* semantics:

$$\frac{}{(\lambda x.E)F \twoheadrightarrow E[F/x]} \quad \frac{E \twoheadrightarrow E'}{EF \twoheadrightarrow E'F} \quad \frac{F \twoheadrightarrow F'}{EF \twoheadrightarrow EF'} \quad \frac{E \twoheadrightarrow E'}{\lambda x.E \twoheadrightarrow \lambda x.E'}$$

Here, $E[F/x]$ is E, with every free occurrence of x replaced by F, up to the usual renaming of bound variables. We can define a variant of MORRIS's testing pre-order (BARENDREGT, 1984, Exercise 16.5.5):

$$E \sqsubseteq F \quad \text{iff} \quad \forall C\,.\,C[F] \to^{\infty} \Rightarrow C[E] \to^{\infty}$$

We can also define a variant of the λ-calculus with recursive declarations and strictness annotations:

$$M ::= x \mid xy \mid \lambda x.M \mid \mathsf{rec}\, x := D \,\mathsf{in}\, M$$
$$D ::= ?M \mid !M$$

Here:

- $\mathsf{rec}\, x := ?M \,\mathsf{in}\, N$ declares x recursively to be M in the context N. For example, a fixed point of f is $\mathsf{rec}\, x := ?f \,\mathsf{in}\, \mathsf{rec}\, y := ?xy \,\mathsf{in}\, y$.
- $\mathsf{rec}\, x := !M \,\mathsf{in}\, N$ is the same, except that x is strict in N, and so evaluation of M can be sparked off as a parallel computation.

We shall let bound variables be α-converted. The free variables of M are $\mathsf{fv}\, M$:

$$\begin{aligned}
\mathsf{fv}\, x &= \{x\}\\
\mathsf{fv}(xy) &= \{x, y\}\\
\mathsf{fv}(\lambda x.M) &= \mathsf{fv}\, M \setminus \{x\}\\
\mathsf{fv}(\mathsf{rec}\, x := D \,\mathsf{in}\, M) &= (\mathsf{fv}\, D \cup \mathsf{fv}\, M) \setminus \{x\}\\
\mathsf{fv}(!M) &= \mathsf{fv}\, M\\
\mathsf{fv}(?M) &= \mathsf{fv}\, M
\end{aligned}$$

There is a translation $|\cdot|$ from the λ-calculus to the λ-calculus with rec:

$$\begin{aligned}
|x| &= x\\
|EF| &= \mathsf{rec}\, x := !|E| \,\mathsf{in}\, \mathsf{rec}\, y := ?|F| \,\mathsf{in}\, xy\\
|\lambda x.E| &= \lambda x.|E|
\end{aligned}$$

Note that in the translation of EF, we know that E will be used, and so it can be evaluated strictly. On the other hand, we do not know if F will be used or not, so it cannot be annotated.

3 The chemical abstract machine

The Chemical Abstract Machine (CHAM) of BERRY and BOUDOL (1990) is a way of presenting the operational semantics of parallel systems. We shall use it to give a semantics for parallel graph reduction of the λ-calculus with rec.

A CHAM gives reductions between *solutions*, which are multisets (or *bags*) of *molecules*. The definition of molecules is specific to each CHAM, but a solution can always be regarded as a molecule. In a solution $\{\!|m_1, \ldots, m_n|\!\}$, the multiset brackets $\{\!|\cdots|\!\}$ are called a *membrane*. Let S range over solutions, and let $S \uplus S'$ be the multiset union of S and S'. Each CHAM has three types of reduction:

- *Heating rules*, of the form $S \rightharpoonup S'$.
- *Cooling rules*, of the form $S \rightharpoondown S'$.
- *Reaction rules*, of the form $S \mapsto S'$.

Heating and cooling rules are always given in pairs $S \rightleftharpoons S'$, whereas reaction rules are irreversible. We shall write $\rightleftharpoons^*$ for the transitive, reflexive, symmetric closure of $\rightleftharpoons$, write $\rightarrow$ for $\rightleftharpoons^* \mapsto \rightleftharpoons^*$, and let $\Rightarrow$ range over $\rightharpoonup$, $\rightharpoondown$ and $\mapsto$. All CHAMs have the following structural rules, where $m[\cdot]$ is a molecule containing precisely one hole:

$$\frac{S \Rightarrow S'}{S \uplus S'' \Rightarrow S' \uplus S''} \qquad \frac{S \Rightarrow S'}{\{\!|m[S]|\!\} \Rightarrow \{\!|m[S']|\!\}}$$

In addition, the CHAMs we shall consider in this paper allow the outermost membrane of any solution to be ignored. This allows us to write $m_1, \ldots, m_n \Rightarrow m'_1, \ldots, m'_{n'}$ for $\{\!|m_1, \ldots, m_n|\!\} \Rightarrow \{\!|m'_1, \ldots, m'_{n'}|\!\}$:

$$\{\!|S|\!\} \rightleftharpoons S$$

The molecules and reduction rules are specific to each CHAM. In the case of the graph reduction CHAM, molecules are defined:

$$m ::= x := D \mid S \mid \nu x.S$$

The free variables of m are fv m:

$$\begin{aligned} \text{fv}(x := D) &= \{x\} \cup \text{fv}\, D \\ \text{fv}\, \{\!|m_1, \ldots, m_n|\!\} &= \text{fv}\, m_1 \cup \cdots \cup \text{fv}\, m_n \\ \text{fv}(\nu x.S) &= \text{fv}\, S \setminus \{x\} \end{aligned}$$

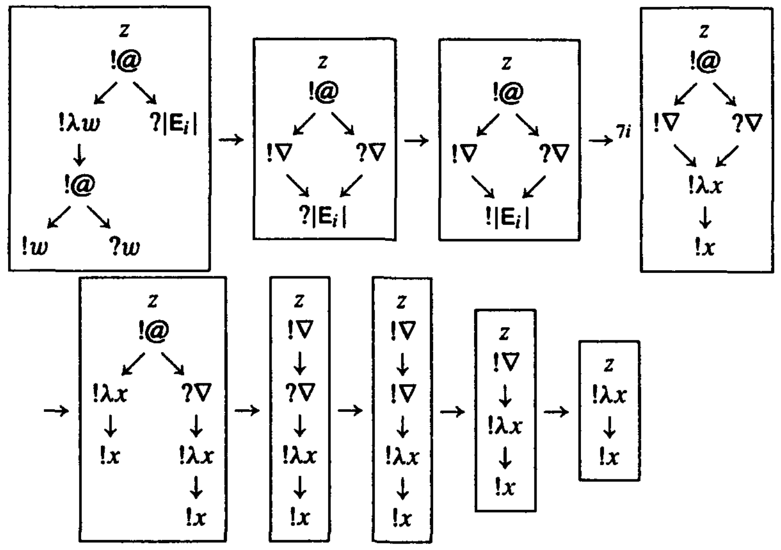
The defined variables of m are dv m:

$$\begin{aligned} \text{dv}(x := D) &= \{x\} \\ \text{dv}\, \{\!|m_1, \ldots, m_n|\!\} &= \text{dv}\, m_1 \cup \cdots \cup \text{dv}\, m_n \\ \text{dv}(\nu x.S) &= \text{dv}\, S \setminus \{x\} \end{aligned}$$

We shall only consider solutions which do not define any variables twice, so in any solution $\{\!|m_1, \ldots, m_n|\!\}$, the defined variables of each m_i are distinct. For example, we do not allow solutions such as $\nu x.\{\!|x := !\lambda w.M, x := !\lambda w.N, y := !xw|\!\}$ which could reduce nondeterministically to $\{\!|y := !M|\!\}$ or to $\{\!|y := !N|\!\}$. If $\tilde{x} = x_1, \ldots, x_n$ then we can write $\nu x.m$ for $\nu x.\{\!|m|\!\}$ and $\nu\tilde{x}.m$ for $\nu x_1 \ldots \nu x_n.m$. Define:

- a molecule is a *positive ion* with *valency* x iff it is $x := ?M$ or $x := !\lambda y.M$.
- a molecule is a *negative ion* with *valency* y iff it is $x := !y$ or $x := !yz$.
- a molecule is *ionic* iff it is a positive or negative ion.
- a solution is *plasmic* iff it is $\{\!|\nu\tilde{y}.\{\!|m_1, \ldots, m_n|\!\}|\!\}$ or $\{\!|m_1, \ldots, m_n|\!\}$ where each m_i is ionic. A plasma is positive (negative) iff it contains only positive (negative) ions.

Plasmas can be regarded as graphs, for example the graph reduction:



is represented by the CHAM reduction:

$$\begin{aligned}
&\nu xy.\{\!|z := !xy, y := ?|\mathsf{E}_i|, x := !|\lambda w.ww||\!\}\\
&\quad\to \nu uvy.\{\!|z := !uv, y := ?|\mathsf{E}_i|, v := ?y, u := !y|\!\}\\
&\quad\to \nu uvy.\{\!|z := !uv, y := !|\mathsf{E}_i|, v := ?y, u := !y|\!\}\\
&\quad\to^{7i} \nu uvy.\{\!|z := !uv, y := !|\mathsf{I}|, v := ?y, u := !y|\!\}\\
&\quad\to \nu uvy.\{\!|z := !uv, y := !|\mathsf{I}|, v := ?y, u := !|\mathsf{I}||\!\}\\
&\quad\to \nu vy.\{\!|z := !v, y := !|\mathsf{I}|, v := ?y|\!\}\\
&\quad\to \nu vy.\{\!|z := !v, y := !|\mathsf{I}|, v := !y|\!\}\\
&\quad\to \nu v.\{\!|z := !v, v := !|\mathsf{I}||\!\}\\
&\quad\to \{\!|z := !|\mathsf{I}||\!\}
\end{aligned}$$

In these diagrams:

- Tagged nodes $x := !M$ are labelled with a !.
- Untagged nodes $x := ?M$ are labelled with a ?.
- Application nodes $x := yz$ are labelled with a @.
- Indirection nodes $x := y$ are labelled with a y, if y is free, and with ∇ otherwise.
- Function nodes $x := \lambda y.M$ are labelled with a λy, and have the graph for $\{\!|z := !M|\!\}$ drawn beneath them, for some fresh variable z.

The most important heating rule allows recursive declarations to become part of a solution, whilst hiding the bound variable. This is only valid when it would not cause the free variable x to become bound by y, which we can achieve by α-converting y first.

$$x := (!\,\mathsf{rec}\,y := D\,\mathsf{in}\,M) \rightleftharpoons \nu y.\{\!|x := !M, y := D|\!\} \qquad (x \neq y)$$

The scope of a hidden variable can migrate, as long as this does not result in variable capture:

$$m, \nu x.m' \rightleftharpoons \nu x.\{\!|m, m'|\!\} \qquad (x \notin \mathrm{fv}\, m)$$

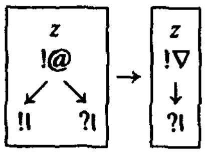
Hidden variables may be α-converted, exchanged and evaporated:

$$\begin{aligned}
\nu x.m &\rightleftharpoons \nu y.(m[y/x]) \qquad (y \notin \mathrm{fv}\, m)\\
\nu xy.m &\rightleftharpoons \nu yx.m\\
\nu x.\{\!|\,|\!\} &\rightleftharpoons \{\!|\,|\!\}
\end{aligned}$$

Finally, we can perform garbage collection on positive plasmas, since a hidden positive plasma can never make any reductions:

$$\nu\tilde{x}.\{\!|\tilde{x} := \tilde{D}|\!\} \rightleftharpoons \{\!|\,|\!\} \qquad (\{\!|\tilde{x} := \tilde{D}|\!\} \text{ is a positive plama})$$

We shall sometimes write $\rightleftharpoons_\gamma$ for this thermal action, and $\rightleftharpoons_{\neq\gamma}$ for any other thermal action. For example, the graph reduction:



can be derived:

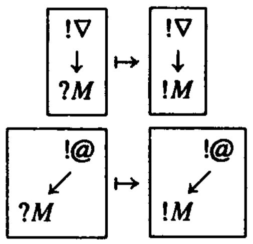
$$\begin{aligned}
&\nu yx.\{\!|x := !\mathsf{I}, y := ?\mathsf{I}, z := !xy|\!\}\\
&\quad\mapsto \nu yx.\{\!|x := !\mathsf{I}, y := ?\mathsf{I}, z := !y|\!\}
\end{aligned}$$

$$
\begin{aligned}
&\rightharpoonup \nu yx.\{\!|x := !\mathsf{I}, \{\!|y := ?\mathsf{I}, z := !y|\!\}|\!\}\\
&\rightharpoonup \nu y.\{\!|\nu x.\{\!|x := !\mathsf{I}|\!\}, \{\!|y := ?\mathsf{I}, z := !y|\!\}|\!\}\\
&\rightharpoonup_{\gamma} \nu y.\{\!|\{\!|y := ?\mathsf{I}, z := !y|\!\}|\!\}\\
&\rightharpoonup \nu y.\{\!|y := ?\mathsf{I}, z := !y|\!\}
\end{aligned}
$$

A reaction can occur whenever one positive and one negative ion with the same valency exist in a solution. Since there are two kinds of positive ion and two kinds of negative ion, there are four reaction rules. The first two allow untagged molecules to become tagged:

$$x := !y, y := ?M \mapsto x := !y, y := !M$$
$$x := !yz, y := ?M \mapsto x := !yz, y := !M$$

These can be drawn:



$$
\begin{aligned}
\{\!|S|\!\} &\rightleftharpoons S\\
x := !\,\mathsf{rec}\,y := D\,\mathsf{in}\,M &\rightleftharpoons \nu y.\{\!|x := !M, y := D|\!\} \qquad (x \neq y)\\
m, \nu x.m' &\rightleftharpoons \nu x.\{\!|m, m'|\!\} \qquad (x \notin \mathrm{fv}\,m)\\
\nu x.m &\rightleftharpoons \nu y.(m[y/x]) \qquad (y \notin \mathrm{fv}\,m)\\
\nu xy.m &\rightleftharpoons \nu yx.m\\
\nu x.\{\!||\!\} &\rightleftharpoons \{\!||\!\}\\
\nu\tilde{x}.\{\!|\tilde{x} := \tilde{D}|\!\} &\rightleftharpoons \{\!||\!\} \qquad (\{\!|\tilde{x} := \tilde{D}|\!\} \text{ is a positive plasma})\\
x := !y, y := ?M &\mapsto x := !y, y := !M\\
x := !yz, y := ?M &\mapsto x := !yz, y := !M\\
x := !y, y := !\lambda w.M &\mapsto x := !\lambda w.M, y := !\lambda w.M\\
x := !yz, y := !\lambda w.M &\mapsto x := !M[z/w], y := !\lambda w.M
\end{aligned}
$$

Table 1. Summary of the graph reduction CHAM

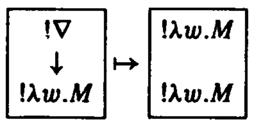
This models the first phase of graph reduction—we search along the spine of a graph, tagging nodes for evaluation. Note that strict reactions do not occur:

$$x := !yz, z := ?M \not\mapsto x := !yz, z := !M$$

If a tagged indirection node points to a function, we can just copy the function. The SKIM (STOYE *et al.*, 1984) and *G*-machine (JOHNNSON, 1984) use this as a method of eliminating indirection nodes. It was shown by LESTER (1989) to be adequate:

$$x := !y, y := !\lambda w.M \mapsto x := !\lambda w.M, y := !\lambda w.M$$

This can be drawn:



If an application node points to a function, it can be β-reduced:

$$x := !yz, y := !\lambda w.M \mapsto x := !M[z/w], y := !\lambda w.M$$

This can be drawn:

$$\begin{tikzcd}
 & & !@ \arrow[dl] \arrow[dr] & \\
 & !\lambda w \arrow[d] & & N \\
 & M \arrow[dl] \arrow[dr] & & \\
w & \cdots & w &
\end{tikzcd}
\quad \mapsto \quad
\begin{tikzcd}
 & & & M \arrow[dl] \arrow[dr] & \\
 & !\lambda w \arrow[d] & \nabla \arrow[dr] & \cdots & \nabla \arrow[dl] \\
 & M \arrow[dl] \arrow[dr] & & N & \\
w & \cdots & w & &
\end{tikzcd}$$

We shall sometimes write $\mapsto_\beta$ for this reaction, and $\mapsto_{\neq\beta}$ for any other reaction. This CHAM is summarized in Table 1.

This CHAM implements the algorithm for parallel graph reduction described by PEYTON JONES (1987). A process is assigned to evaluating a node, which is tagged. It then searches along the spine, tagging each node as it passes. If it reaches a function node which can be β-reduced, it does so. If it reaches a function node which cannot be β-reduced, this is returned as the result. If it reaches a previously tagged application or indirection node, it is blocked until the tagged node is evaluated. For example, in the graph:

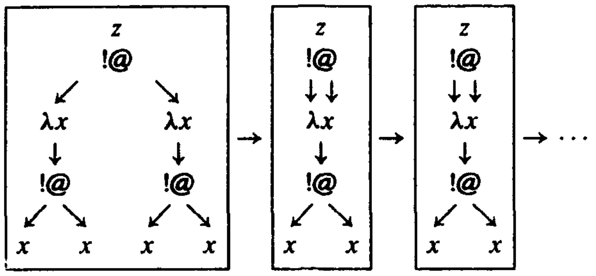
$$\begin{tikzcd}
 & x & & \\
 & !@ \arrow[dl] \arrow[dr] & & \\
!\nabla \arrow[dr] & & !@ \arrow[dl] \arrow[dr] & \\
 & ?M & & ?N
\end{tikzcd}$$

only one process will evaluate M. This is mirrored in the CHAM by the fact that M will only be reduced once. However, this algorithm produces some surprising results with cyclic graphs. The solution $\{\!|y := !\,\mathsf{rec}\,x := !x \text{ in } x|\!\}$ heats to become the plasma $\{\!|\nu x.\{\!|y := !x, x := !x|\!\}\,|\!\}$ and the graph:

$$\begin{tikzcd}
y \\
!\nabla \arrow[d] \\
!\nabla \arrow[loop below]
\end{tikzcd}$$

This has no reductions, because it is negative. This is mirrored in the parallel graph reduction algorithm, since the process evaluating y will discover that the indirection node at x has already been tagged. Thus, it is possible for evaluations to deadlock, when a sequential algorithm would diverge.

Our translation of the λ-calculus will not produce cyclic graphs, although it can still produce divergent terms. For example, the translation of $(\lambda x.xx)(\lambda x.xx)$ has the reductions:



Since $|E|$ is an acyclic graph, we will be able to show that the CHAM semantics for the λ-calculus is adequate. To do this, we define the testing preorder on molecules:

$$m \sqsubseteq m' \quad \text{iff} \quad \forall C \,.\, C[m'] \rightarrow^{\infty} \;\Rightarrow C[m] \rightarrow^{\infty}$$

and show that the CHAM semantics is *adequate*, that is if $(x := ?|E|) \sqsubseteq (x := ?|F|)$ then $E \sqsubseteq F$.

Theorem 1 (adequacy). *If* $(x := ?|E|) \sqsubseteq (x := ?|F|)$ *then* $E \sqsubseteq F$.

Proof. Given in (JEFFREY, 1992). □

However, it is not fully abstract.

Theorem 2. $E \sqsubseteq F$ *does not imply* $(x := ?|E|) \sqsubseteq (x := ?|F|)$.

Proof. Given in (JEFFREY, 1992). □

It is an open problem as to whether the CHAM semantics is fully abstract wrt ABRAMSKY's (1989) λ-calculus with C, and as to whether the canonical semantics for the lazy λ-calculus $D \simeq (D \rightarrow D)_{\perp}$ is adequate wrt the CHAM semantics.

4 The asynchronous pi-calculus

The π-calculus, introduced by MILNER, PARROW and WALKER (1989) is a process algebra in which scope is considered important. MILNER has shown that it can be used to model pointer-structures (1991) and the lazy λ-calculus (1992), which has been further investigated by SANGIORGI (1991).

Since the π-calculus was designed with pointer structures and the λ-calculus in mind, it seems natural to use it to encode a parallel graph reduction algorithm. We shall consider a variant of BOUDOL's asynchronous π-calculus (1992). This has the syntax:

$$P ::= \overline{x}[yz] \mid x(yz).P \mid P \mid P \mid \nu x.P \mid [x = y]P \mid [x \neq y]P \mid A(\tilde{x})$$

Here:

- $\overline{x}[yz]$ is the process which outputs the pair (y, z) along channel x.
- $x(yz).P$ is the process which inputs a pair (y', z') along channel x, then behaves like $P[x'/x, y'/y]$.
- $P \mid Q$ places P and Q in parallel.
- $\nu x.P$ creates a new channel x for use in P.
- $[x = y]P$ acts like P whenever $x = y$, and deadlocks otherwise.
- $[x \neq y]P$ acts like P whenever $x \neq y$, and deadlocks otherwise.
- $A(\tilde{x})$ is a recursive definition, in the style of MILNER (1989). We shall assume an environment of definitions $A(\tilde{x}) \stackrel{\text{def}}{=} P$, where fv $P \subseteq \tilde{x}$.

The CHAM for this variant of the asynchronous π-calculus is given in Table 2, and is very similar to BOUDOL's CHAM for the asynchronous π-calculus (1992). The only new rules are:

- $\{\!|S|\!\} \rightleftharpoons S$, which is missing from BOUDOL's paper. This rule is required to prove the result that for any solution S there is a process P such that $S \rightleftharpoons^* \{\!|P|\!\}$. For example, we cannot show $\{\!|\{\!|\overline{x}[yz]|\!\}|\!\} \rightleftharpoons^* \{\!|\overline{x}[yz]|\!\}$ without this rule.
- $[x = x]P \rightleftharpoons P$ and $[x \neq y]P \rightleftharpoons P$ whenever $x \neq y$, which gives semantics for the conditional operators missing from BOUDOL's paper.
- $A(\tilde{x}) \rightleftharpoons P[\tilde{x}/\tilde{y}]$ whenever $A(\tilde{y}) \overset{\text{def}}{=} P$, which gives semantics for recursive definitions which were not used in BOUDOL's paper.

We can define much of the same vocabulary for this CHAM as we did for the graph reduction CHAM.

$$\begin{array}{rcll}
\{\!|S|\!\} & \rightleftharpoons & S & \\
P \mid Q & \rightleftharpoons & P, Q & \\
\nu x.P & \rightleftharpoons & \nu x.\{\!|P|\!\} & \\
m, \nu x.m' & \rightleftharpoons & \nu x.\{\!|m, m'|\!\} & (x \notin \mathrm{fv}\, m) \\
\nu x.m & \rightleftharpoons & \nu y.(m[y/x]) & (y \notin \mathrm{fv}\, m) \\
\nu xy.m & \rightleftharpoons & \nu yx.m & \\
\nu xx.m & \rightleftharpoons & \nu x.m & \\
[x = x]P & \rightleftharpoons & P & \\
[x \neq y]P & \rightleftharpoons & P & (x \neq y) \\
A(\tilde{x}) & \rightleftharpoons & P[\tilde{x}/\tilde{y}] & (A(\tilde{y}) \overset{\text{def}}{=} P) \\
\overline{x}[yz], x(vw).P & \mapsto & P[y/v, z/w] &
\end{array}$$

Table 2. CHAM for the π-calculus

- A molecule is a positive ion with valency x iff it is $x(yz).P$.
- A molecule is a negative ion with valency x iff it is $\overline{x}[yz]$.
- A molecule is ionic iff it is a positive or negative ion.
- A solution is plasmic iff it is $\{\!|\nu\tilde{x}.\{\!|P_1, \ldots, P_n|\!\}|\!\}$ or $\{\!|P_1, \ldots, P_n|\!\}$ and each P_i is ionic. A plasma is positive (negative) iff it contains only positive (negative) ions.

We can give a translation of each molecule of the graph reduction CHAM into the π-calculus. This uses a special variable $*$, which we shall use to represent a function which is being evaluated, but which has not (yet) been given an argument. The semantics for terms is:

$$\begin{array}{rcll}
[\![x]\!]z & \overset{\text{def}}{=} & \overline{x}[*z] & \\
[\![xy]\!]z & \overset{\text{def}}{=} & \overline{x}[yz] & \\
[\![\lambda x.M]\!]z & \overset{\text{def}}{=} & !z(xy).([x = *][\![\lambda x.M]\!]y \mid [x \neq *][\![M]\!]y) & \\
[\![\mathsf{rec}\, x := D \text{ in } M]\!]z & \overset{\text{def}}{=} & \nu x([\![D]\!]x \mid [\![M]\!]z) & (x \neq z)
\end{array}$$

where MILNER's (1991) *replication* operator is defined:

$$!P \overset{\text{def}}{=} !P \mid P$$

Note that the definition of $[\![\lambda x.M]\!]$ is recursive, which is why we are taking recursion to be primitive, rather than replication. It is not obvious whether one could define a semantics using replication for which there would be a one-to-one correspondence between CHAM reductions and π-calculus reductions. Note also that $[\![\mathsf{rec}\, x := D \text{ in } M]\!]z$ is defined only when $x \neq z$, but we can use α-conversion on x to assure this. The semantics for declarations is:

$$[\![!M]\!]z \overset{\text{def}}{=} [\![M]\!]z$$
$$[\![?M]\!]z \overset{\text{def}}{=} z(xy).(\overline{z}[xy] \mid [\![M]\!]z)$$

The semantics for molecules is:

$$[\![x := D]\!] \overset{\text{def}}{=} [\![D]\!]x$$

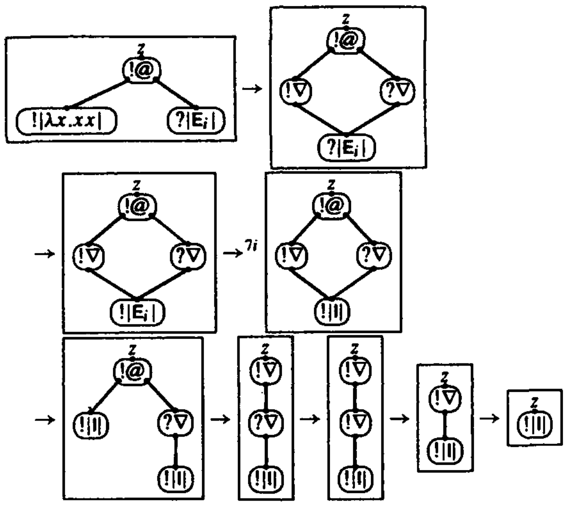


Table 3. A sample graph reduction in the π-calculus

$$[\![\nu x.m]\!] \overset{\text{def}}{=} \nu x.[\![m]\!]$$
$$[\![\{\!|m_1, \ldots, m_n|\!\}]\!] \overset{\text{def}}{=} \{\!|[\![m_1]\!], \ldots, [\![m_n]\!]|\!\}$$

This semantics can be drawn with flow graphs. For example, if we draw:

$$\overset{z}{\textcircled{D}} \quad \text{for} \quad [\![D]\!]z$$

$$\underset{x}{\overset{z}{(!\nabla)}} \quad \text{for} \quad [\![!x]\!]z \qquad \underset{x}{\overset{z}{(?\nabla)}} \quad \text{for} \quad [\![?x]\!]z$$

$$\underset{x\quad y}{\overset{z}{(!@)}} \quad \text{for} \quad [\![!xy]\!]z \qquad \underset{x\quad y}{\overset{z}{(?@)}} \quad \text{for} \quad [\![?xy]\!]z$$

Then the reduction of E_i given in section 1 can be drawn (with some extraneous processes removed to account for garbage collection) in Table 3. This is exactly the same reduction as given in Section 3.

In general, we can show that each CHAM reduction is matched by exactly one π-calculus reduction, and thus that the π-calculus semantics is adequate wrt the CHAM semantics for graph reduction (and so wrt the λ-calculus).

Theorem 3 (adequacy). *If* $[\![S]\!] \sqsubseteq [\![S']\!]$ *then* $S \sqsubseteq S'$.

Proof. Given in (JEFFREY, 1992). □

However, it is not fully abstract.

Theorem 4. *$m \sqsubseteq m'$ does not imply $[\![m]\!] \sqsubseteq [\![m']\!]$.*

Proof. Given in (JEFFREY, 1992). □

SANGIORGI (1991) has investigated λ-calculi semantics for which MILNER's π-calculus translation is fully abstract. It is an open problem as to whether similar results can be shown for the CHAM for graph reduction.

References

ABRAMSKY, S. (1989). The lazy lambda calculus. In TURNER, D., editor, *Declarative Programming*. Addison-Wesley.

AUGUSTSSON, L. (1984). A compiler for lazy ML. In *Proc. ACM Symp. Lisp and Functional Programming*, pages 218–227.

BARENDREGT, H. (1984). *The Lambda Calculus*. North-Holland. Studies in logic 103.

BARENDREGT, H. P., VAN EEKELEN, M. C. J. D., GLAUERT, J. R. W., KENNAWAY, J. R., PLASMEIJER, M. J., and SLEEP, M. R. (1987). Term graph rewriting. In *Proc. PARLE 87*, volume 2, pages 141–158. Springer-Verlag. LNCS 259.

BERRY, G. and BOUDOL, G. (1990). The chemical abstract machine. In *Proc. 17th Ann. Symp. Principles of Programming Languages*.

BOUDOL, G. (1992). Asynchrony and the pi-calculus. INRIA Sophia-Antipolis.

CHURCH, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press.

HUGHES, R. J. M. (1984). *The Design and Implementation of Programming Languages*. D.Phil thesis, Oxford University.

JEFFREY, A. (1992). A chemical abstract machine for graph reduction. Technical report 3/92, University of Sussex.

JOHNNSON, T. (1984). Effecient compilation of lazy evaluation. In *Proc. Sigplan 84 Symp. Compiler Construction*, pages 58–69.

JONES, M. (1992). The Gofer technical manual. Part of the Gofer distribution.

LESTER, D. (1989). *Combinator Graph Reduction: A Congruence and its Applications*. D.Phil thesis, Oxford University.

MCCARTHY, J., ABRAHAMS, P. W., EDWARDS, D. J., HART, T. P., and LEVIN, M. I. (1962). *The Lisp 1.5 Programmers Kit*. MIT Press.

MILNER, R. (1989). *Communication and Concurrency*. Prentice-Hall.

MILNER, R. (1991). The polyadic π-calculus: a tutorial. In *Proc. International Summer School on Logic and Algebra of Specification*, Marktoberdorf.

MILNER, R. (1992). Functions as processes. *Math. Struct. in Comput. Science*, 2:119–141.

MILNER, R., PARROW, J., and WALKER, D. (1989). A calculus of mobile processes. Technical reports ECS-LFCS-89-86 and -87, LFCS, University of Edinburgh.

PEYTON JONES, S. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall.

SANGIORGI, D. (1991). The lazy lambda calculus in a concurrency scenario. Technical Report ECS-LFCS-91-189, LFCS, Edinburgh University.

STOYE, W. R., CLARKE, T. J. W., and NORMAN, A. C. (1984). Some practical methods for rapid combinator reduction. In *Proc. ACM Symp. Lisp and Functional Programming*, pages 159–166.

TURNER, D. (1985). Miranda: A non-strict functional language with polymorphic types. In *Proc. IFIP Conf. Functional Programming Languages and Computer Architecture*. Springer-Verlag. LNCS 201.

WADSWORTH, C. P. (1971). *Semantics and Pragmatics of the Lambda Calculus*. D.Phil thesis, Oxford University.

Lifting Theorems for Kleisli Categories

Philip S. Mulry [*]
Department of Computer Science
Colgate University
Hamilton, NY 13346.
e-mail phil@cs.colgate.edu

Abstract

Monads, comonads and categories of algebras have become increasingly important tools in formulating and interpreting concepts in programming language semantics. A natural question that arises is how various categories of algebras for different monads relate functorially. In this paper we investigate when functors between categories with monads or comonads can be lifted to their corresponding Kleisli categories. Determining when adjoint pairs of functors can be lifted or inherited is of particular interest. The results lead naturally to various applications in both extensional and intensional semantics, including work on partial maps and data types and the work of Brookes/Geva on computational comonads.

1 Introduction

Monads, comonads and categories of algebras have become increasingly important tools in formulating and interpreting concepts in

[*]This research was partially supported by NSF Grants CCR-9002251, CCR-9203106, INT-9113406 and a Colgate University Picker Fellowship.

programming language semantics. The existence of reflections and coreflections on various categories of cpos and domains, for example, has proven to be a special case of comparison functors between Kleisli and Eilenberg-Moore categories of algebras while initial algebras for various monads are routinely used to interpret recursive data types. Final coalgebras and invariant objects have also played important roles, such as describing PER semantics, algebraic completeness and fixed point semantics [CP], [F], [FMRS], [Mu1], [Mu2].

In refining such work the particular role played by special monads has been profitably emphasized. For example the existence of partial map classifier(*pmc*) monads in more general settings than a topos was addressed in [Mu3] in order to connect the notions of *pmc* and partial cartesian closed category(*pccc*). Additionally Kleisli categories for strong monads have been utilized in providing a general interpretation for abstract programming languages [Mo]. In a different direction Kleisli categories of comonads, particularly computational comonads, have been used to describe an intensional semantics in which comonads represent different possible notions of computation[BG]. Despite the success in utilizing different monads and comonads in separate semantic contexts, little work to date has examined how the corresponding categories of algebras might relate functorially and what effect this has on the associated semantics.

In this paper we address this question by considering when functorial processes on categories with associated monads can be lifted to their corresponding Kleisli categories. We also investigate conditions under which adjoints may be either lifted or inherited. A key step is the observation that the presence of a lifting is equivalent to the existence of a natural transformation satisfying certain equations. Equally important are the wide variety of examples that arise in this setting. We also find that some timely issues gain a new perspective when examined from this viewpoint. For example the notion of monadic strength, which plays an important role in calculations for computational lambda calculi, is a special case of the required existence of a natural transformation. Conditions enabling cartesian closure for semantic categories also naturally arise through the lifting of adjoints as do interpretations of partial data types. Examples such as Moggi's extension construction $\hat{()}$, as well as Brookes and Geva's foundational work on intensional semantics can be interpreted and generalized in this setting as well.

Some acknowledgements are in order. It would be amiss if I did not mention the hospitality of LFCS, Edinburgh, where much of the early work on this paper was completed. I would like to thank Martin Hyland whose perceptive questions on a visit to Cambridge helped motivate part of this enquiry. The work in [J] for the case of Eilenberg-Moore algebras had a significant effect on both the exposition and content of the basic mathematical results. Thanks also to Ernie Manes for some useful comments and suggestions.

2 Lifting Theorems

We begin by considering monads (H, η, μ) and (K, ρ, ν) on categories $\mathbf{C}$ and $\mathbf{D}$ respectively. Let T be a functor $T : \mathbf{C} \to \mathbf{D}$. We are interested in determining when T can be extended to a functor between the corresponding Kleisli categories. We start with a definition making this notion precise. Let i_H denote the inclusion functor from $\mathbf{C}$ to $\mathbf{C_H}$.

Definition 2.1 A functor $\overline{T} : \mathbf{C_H} \to \mathbf{D_K}$ is a lifting of T if $\overline{T} \circ i_H = i_K \circ T$ or equivalently that the following diagram commutes.

$$\begin{tikzcd}
\mathbf{C_H} \arrow[r, "\overline{T}"] & \mathbf{D_K} \\
\mathbf{C} \arrow[u, "i_H"] \arrow[r, "T"] & \mathbf{C} \arrow[u, "i_K"']
\end{tikzcd}$$

We wish to specify when a given functor T has a lifting. The next result produces such conditions. For similar results in the algebra case see [A],[J],[Ma].

Theorem 2.2 For $\mathbf{C}$, $\mathbf{D}$, H, K, T as above, functors $\overline{T} : \mathbf{C_H} \to \mathbf{D_K}$ which are liftings are in 1-1 correspondence with natural transformations of the form $\lambda : TH \to KT$ that satisfy the following

1) $\lambda \circ T\eta = \rho_T$

2) $\nu_T \circ K\lambda \circ \lambda_H = \lambda \circ T\mu$.

Proof : Given $\lambda : TH \to KT$ and $f : A \to B$ an arrow in $\mathbf{C_H}$ (*i.e.* map $A \to HB$ in $\mathbf{C}$), $\overline{T}f$ is defined as $\lambda_B \circ Tf : TA \to KTB$. Some diagram chasing ensures that $\overline{T}$ is well defined and generates the desired commutative diagram. For example $\overline{T}(id_A) = \lambda_A \circ T\eta_A : TA \to KTA = \rho_{TA} = id_{TA}$ in $\mathbf{D_K}$.

Conversely suppose a lifting $\overline{T}$ of T exists. We denote the right adjoints to i_H and i_K by G_H, G_K respectively. First define the natural transformation $\underline{\lambda} : T \circ G_H \to G_K \circ \overline{T}$ as the transpose of $\overline{T}\epsilon : \overline{T}i_H G_H \to \overline{T}$ where $i_K \circ T \circ G_H = \overline{T} \circ i_H \circ G_H$. Composing by i_H gives the desired natural transformation $\lambda = \underline{\lambda} \circ i_H$. Once again some diagram chasing shows λ satisfies the required equations. □

The existence of a natural transformation λ for monads H and K satisfying the equations of 2.2 is not that unusual. In fact several well known examples such as tensorial strength and units of a monad are special cases of λ as the next few examples illustrate.

Example 2.3 Let category $\mathbf{C}$ be cartesian with monad H and endofunctor $T = \_ \times B$ for B an object in $\mathbf{C}$. The functor T has a lifting iff there exists a natural transformation $\lambda_{A,B} : HA \times B \to H(A \times B)$ satisfying

1) $\lambda_{A,B} \circ \eta_A \times B = \eta_{A \times B}$
2) $\mu_{A \times B} \circ H\lambda_{A,B} \circ \lambda_{HA,B} = \lambda_{A,B} \circ \mu_A \times B$.

These are precisely the equations corresponding to tensorial strength and the notion of strong monad [K]. Recall a monad H is strong if there exists a natural transformation $\lambda_{A,B}$ satisfying 1) and 2). See [Mu2] for details. It is an easy matter to make λ natural in both A and B by considering the monad $H \times H$ on $\mathbf{C} \times \mathbf{C}$, letting T be the functor $\_ \times \_$, and utilizing the unit of the monad. Thus tensorial strength is a special case of the existence of a natural transformation for a lifting.

Corollary 2.4 Tensorial strength exists for monad H on cartesian category $\mathbf{C}$ iff products in $\mathbf{C}$ lift to $\mathbf{C_H}$.
Proof : The result is immediate from Theorem 2.2 and Example 2.3. □

Example 2.5 Suppose $\mathbf{C}$ and $\mathbf{D}$ agree, $\mathbf{C}$ has the trivial identity monad and K becomes H. Then for a given endofunctor T of $\mathbf{C}$, the natural transformation $\eta_T : T \to HT$ satisfies the equations of Theorem 2.2 and so a lifting $\overline{T} : \mathbf{C} \to \mathbf{C_H}$ exists. Conversely if $\overline{T}$ exists then $\overline{T} = i_H \circ T$ and so λ must be η_T. In particular when T is the identity, λ is just the unit of the monad H, η, and $\overline{T} = i_H$.

Example 2.6 Suppose **C** and **D** agree, K is the trivial identity monad and T is the monad H itself on **C**. In this case H has a lifting and the corresponding natural transformation is just $\mu : \lambda : H^2 \to H$. The equations of Theorem 2.2 hold and reduce to the identities $\mu \circ H\eta = id_H$ and $\mu \circ \mu_H = \mu \circ H\mu$. The lifting $\overline{H}$ is the right adjoint to i_H, namely G_H.

Lemma 2.7 Lifting distributes over composition, *i.e.* $\overline{S} \circ \overline{T} = \overline{ST}$.
Proof : Let H, K, J be monads on **C**, **D**, **E** respectively with functors $T : \mathbf{C} \to \mathbf{D}$ and $S : \mathbf{D} \to \mathbf{E}$. Suppose that T and S have liftings $\overline{T}$, $\overline{S}$ respectively where $\lambda' : TH \to KT$ and $\lambda'' : SK \to JS$ are the natural transformations corresponding to $\overline{T}$ and $\overline{S}$ satisfying the equations of Theorem 2.2. There then exists a natural transformation $\lambda = \lambda''_T \circ S\lambda' : STH \to JST$. A little diagram chasing will readily show that λ satisfies the appropriate equations and thus produces a lifting $\overline{ST}$ of ST. However since $\overline{T}$ and $\overline{S}$ are liftings, the composition $\overline{S} \circ \overline{T}$ is also a lifting of ST. For $f : A \to HB$ an arrow in $\mathbf{C_H}$, $\overline{S} \circ \overline{T}(f) = \lambda''_T \circ S\lambda' \circ ST(f)$ and thus the lifting $\overline{S} \circ \overline{T}$ agrees with $\overline{ST}$. □

If we were not interested in providing a formula for λ in the above proof, one could easily prove the lemma by composing the two commutative squares. We also note that it need not be true that $\overline{id} = id$ as Example 2.5 illustrates. This point will be emphasized in the next section.

Corollary 2.8 Let monad (H, η, μ) be a monad on **C**. The comonad associated to H can be generated as a lifting.
Proof : Given monad (H, η, μ) we have the adjunction $i_H \dashv G_H$. The comonad formed by the adjunction is simply $H^* = i_H \circ G_H$. Let **C**, **D** and **E** agree, T be the monad functor H, $K = S$ be the identity functor on **C**, and $J = H$. By Example 2.6, λ' associated to T is just μ and the lifting of T is exactly G_H. By Example 2.5, λ'' associated to S is η and the lifting $\overline{S}$ is i_H. Thus by Lemma 2.7 the

natural transformation $\lambda = \eta_H \circ \mu$ satisfies the equations of Theorem 2.2 and corresponds to the lifting $\overline{H} = i_H \circ G_H = H^*$. □

We wish to examine how adjunctions on liftings relate to adjunctions on the original functors. The theorems that follow address this question.

Theorem 2.9 Let $T, \overline{T}, \lambda, H, K$ be as in Theorem 2.2. If K is a cartesian monad and **C** has equalizers of coreflexive pairs, then $\overline{T}$ has a right adjoint implies so does T.
Proof : Because K is cartesian, any object X, in **D** is an equalizer of a coreflexive pair of the form $KX \overset{f}{\underset{g}{\rightrightarrows}} KKX$. In fact the equalizer is just the unit of the monad, ρ_X. Since G_H and G_K are right adjoints and adjoints are unique up to iso, if R exists we must have $RG_K \cong G_H\overline{R}$. Thus we know what R is on cofree objects but by the above every object in **D** is an equalizer of cofree objects. Since R must preserve equalizers, we expect RX to be the equalizer of a pair of maps f, g , $G_H\overline{R} \overset{f}{\underset{g}{\rightrightarrows}} G_H\overline{R}$. We construct the maps f, g in **C** as follows. The map f is just $G_H\overline{R}(i_K\rho_X)$. For g, first take the composite $i_KG_K\epsilon_{i_KX} \circ i_K\lambda_{\overline{R}i_KX}$ where ϵ is the counit of the adjunction $\overline{T}, \overline{R}$ and λ refers to the natural transformation of Theorem 2.2. Taking the transpose once gives $i_HG_H\overline{R}i_KX \to \overline{R}i_KKX$ and a second time gives $G_H\overline{R}i_KX \to G_H\overline{R}i_KKX$. Some diagram chasing shows that the constructed maps f, g do the job. □

In [Mu1] it is shown that every ccc **C** with a *pmc* generates a *pccc* which is equivalent to its associated Kleisli category $\mathbf{C_H}$. The next result gives a partial converse to this result. An alternative approach can be found in [Mu3].

Corollary 2.10 Let **pC** be a *pccc* where **C** has equalizers of coreflexive pairs, then **C** is a ccc.
Proof : Since **pC** is a *pccc*, it has a *pmc* K which is a cartesian monad and for which **pC** is equivalent to $\mathbf{C_K}$(see [Mu3]). If T denotes the endofunctor $\_\times B : \mathbf{C} \to \mathbf{C}$, then there exists a lifting $\overline{T} : \mathbf{C} \to \mathbf{C_K}$ where the monad H is the identity and λ is just the

unit natural transformation ρ_T. Since **pC** is a *pccc*, $\overline{T}$ has a right adjoint $\overline{R}$ and thus by Theorem 2.9 so does T and we are done. $\square$

Example 2.11 Let **C** be a model of the computational lambda calculus, λ_c, in the sense of [Mo] where monad T is cartesian. If **C** has equalizers of coreflexive pairs then **C** is a ccc. The proof is essentially the same as in Corollary 2.10.

Example 2.12 Let **C** be either category **pDOM** or **pCPO**. It is well known that these categories are *pcccs*. Although neither **DOM** nor **CPO** is closed under equalizers of even coreflexive pairs, they do have equalizers for the coreflective pair f, g generated by Theorem 2.9. If K is the *pmc* lift monad associated to **pC** and T is the functor $\_\times B$, then in this case f and g are just $(\rho_X)_\perp^B$ and $(\rho_{X_\perp})^B$ respectively for X in **C**, where ρ is the unit of the lift monad $()_\perp$.

Theorem 2.13 Let **C**, **D**, $H, K, T, \overline{T}$ be as in Theorem 2.2. The natural transformation λ is an iso iff $G_K\overline{T} \cong TG_H$.
Proof : Suppose $\lambda : TH \to KT$ is an isomorphism. In the proof of Theorem 2.2 it was shown that $\lambda = \underline{\lambda} \circ i_H$. If λ is an iso it follows easily that $\underline{\lambda}$ is also iso and thus $G_K\overline{T} \cong TG_H$. Conversely suppose that we have an isomorphism $G_K\overline{T} \cong TG_H$. Since $\overline{T}$ is a lifting we have an isomorphism $TH = TG_Hi_H \cong G_K\overline{T}i_H = G_Ki_KT = KT$. It is easily checked that the isomorphism satisfies the conditions of Theorem 2.2 and thus must be λ. $\square$

Corollary 2.14 Let **C**, **D**, $H, K, T, \overline{T}$ be as in Theorem 2.2. If T is a full embedding and λ is an iso then the lifting $\overline{T}$ is also a full embedding.
Proof : Let f, g be arrows in $\mathbf{C_H}$ and suppose $\overline{T}(f) = \overline{T}(g)$. Then $\lambda \circ T(f) = \lambda \circ T(g)$. Since λ is mono, T(f) = T(g) and T an embedding implies that f = g and so $\overline{T}$ is faithful. Now suppose $h : \overline{T}(A) \to \overline{T}(B)$ is a map in $\mathbf{D_K}$, i.e. $h : \overline{T}(A) \to K\overline{T}(B)$ is a map in **D**. Since $\overline{T}$ is a lifting and T is full, there exists a map $k : A \to H(B)$ in **C** so that T(k) = $\lambda^{-1} \circ h$. But k is a map in $\mathbf{C_H}$ and so $\overline{T}(k) = \lambda \circ T(k) = h$. Thus $\overline{T}$ is also full and we are done. $\square$

Example 2.15 Let **C** have a *pmc* monad H. In [Mu3] it is shown that H must be the restriction of a *pmc* $\tilde{()}_p$ in the presheaf category $\hat{\mathbf{C}}$ over

C, *i.e.* there exists a natural transformation which is an isomorphism $YH \cong \tilde{()}_p Y$, where Y is the Yoneda embedding. What's more the isomorphism satisfies the equations of Theorem 2.2. By Theorem 2.13 then there exists a lifting $\overline{Y}$ of Y and by Corollary 2.14, $\overline{Y}$ is also an embedding. The following diagram then commutes where the vertical pairs form adjoint pairs.

$$\begin{tikzcd}
\mathbf{C_H} \arrow[r, "\overline{Y}"] \arrow[d, shift left, "G_H"] & \hat{\mathbf{C}}_{\tilde{()}_p} \arrow[d, shift left, "\tilde{()}_p"] \\
\mathbf{C} \arrow[r, "Y"] \arrow[u, shift left, "i"] & \hat{\mathbf{C}} \arrow[u, shift left, "i"]
\end{tikzcd}$$

The lifting of Y in the previous example is different from the extension construction $\hat{()}$, found in [Mo]. There $\hat{()}$ is computed using Kan extensions and is applied directly to monads, though the construction can be readily related to the present context as Example 2.17 shows. The results of the last example also can be utilized to describe how *pcccs* can be incorporated inside the setting of Kleisli categories of *pmc* monads on cartesian closed categories.

Corollary 2.16 Every *pccc* **pC** can be fully embedded inside the Kleisli category of a cartesian closed category.
Proof : Since **pC** is a *pccc*, **C** has a *pmc* monad H for which **pC** is equivalent to $\mathbf{C_H}$ (see[Mu3]). By the previous example H is the restriction of a *pmc* $\tilde{()}_p$ in $\hat{\mathbf{C}}$ and thus the Yoneda embedding has a lifting to **pC** which is again a full embedding. □

Example 2.17 The $\hat{()}$ construction on monads found in [Mo] fits nicely in the general context of the results above. For a given monad H on **C**, the existence of an extension $(\hat{H})$ simply provides a trivial natural transformation which is just an identity $\lambda : YH = (\hat{H})Y$. Likewise the corresponding equations trivialize. Although there is no mention of this in [Mo], by Theorem 2.2 there is also a lifting of Y, $\overline{Y} : \mathbf{C_H} \to \hat{\mathbf{C}}_{\hat{\mathbf{H}}}$ to the Kleisli categories. The other properties listed there, such as preservation of strength, follow immediately as a consequence of Corollary 2.14.

We include the next result for completeness. Although we do not use it now, the dual result plays an important role in comonadic computation. See Theorem 3.5 below.

Theorem 2.18 Let $\mathbf{C}$, $\mathbf{D}$, $H, K, T, \overline{T}$ be as in Theorem 2.13. If T has a left adjoint then so does $\overline{T}$.

Proof : The proof follows the approach in [J]. Suppose $L \dashv T$ exists. Since $G_K\overline{T} \cong TG_H$, if a left adjoint $\overline{L}$ to $\overline{T}$ exists it must satisfy $i_H \circ L \cong \overline{L} \circ i_K$. Thus $\overline{L}$ is a lifting of L and can be defined by specifying a natural transformation $\theta : LK \to HL$. Defining $\theta = c_{HL} \circ L\lambda^{-1}L \circ LKu$ where u and c denote the unit and the counit respectively of the adjunction $L \dashv T$, provides the necessary natural transformation. Some diagram chasing shows that θ satisfies the equations of Theorem 2.2. □

3 An Application using Duality

The past section dealt with lifting theorems for Kleisli categories of monads and various applications. In this section we examine an application that exploits the dual results to those presented in section 2, namely we consider how lifting theorems for comonads can be utilized to model intensional semantics in the sense of [BG].

In [BG], a categorical approach to intensional semantics is developed. If the extensional meaning of a program is represented by a map in category $\mathbf{C}$, then for a suitable choice of comonad H on $\mathbf{C}$, HA can be viewed as an object of computations over A, for any object A in $\mathbf{C}$. The intensional meaning of a program is then interpreted as a map from computations to values, *i.e.* a map in the Kleisli category of the comonad H. By defining a computational comonad in [BG], an extensional equivalence relation on algorithms is obtained thereby allowing reasoning at different levels of abstraction.

We consider comonads (H, ϵ, δ) and (K, α, β) on categories $\mathbf{C}$ and $\mathbf{D}$ respectively. T is a functor $T : \mathbf{C} \to \mathbf{D}$ and again we are interested in determining when T can be lifted to a functor $\overline{T} : \mathbf{C_H} \to \mathbf{D_K}$ between the corresponding Kleisli categories for the comonads. As before the lifting of T satisfies $\overline{T} \circ i_H = i_K \circ T$ where i_H is the usual functor $\mathbf{C} \to \mathbf{C_H}$ with adjoint G_H. Now the functor i_H is a right adjoint.

Theorem 3.1 For $\mathbf{C}$, $\mathbf{D}$, H, K, T as above, functors $\overline{T} : \mathbf{C_H} \to \mathbf{D_K}$ which are liftings are in 1-1 correspondence with natural transformations of the form $\sigma : KT \to TH$ that satisfy the following equations.

1) $T\epsilon \circ \sigma = \alpha_T$
2) $\sigma_H \circ K\sigma \circ \beta_T = T\delta \circ \sigma$.

Proof : This is just the dual of Theorem 2.2. $\square$

Example 3.2 Let (H, ϵ, δ) be a comonad on $\mathbf{C}$. In [BG] a comonad is called computational if there exists a natural transformation $\gamma : id \to H$ satisfying

1) $\epsilon \circ \gamma = id$
2) $\delta \circ \gamma = \gamma_H \circ \gamma$

It is then shown that a computational comonad produces functors alg and fun and an extensional equivalence of maps is achieved. The functors alg and fun are just special cases of the lifting construction. Specifically, let $\mathbf{C} = \mathbf{D}$, H be the identity comonad, and $K = H$ in the setup of Theorem 3.1. If T is the identity functor on $\mathbf{C}$ then by the dual to Example 2.5, $\overline{T} : \mathbf{C} \to \mathbf{C_H}$ is just i_H which is just alg and the natural transformation σ generated by the lifting is just the counit of the comonad H, namely ϵ. Reversing the direction of the lifting where T is still the identity, the existence of a lifting $\overline{T} : \mathbf{C_H} \to \mathbf{C}$ corresponds to the existence of a natural transformation $\sigma : KT \to TH$ satisfying the equations of theorem 3.1. In this case σ becomes $\gamma : id \to H$ and the equations reduce to those above defining a computational comonad. Further the lifting $\overline{T} : \mathbf{C_H} \to \mathbf{C}$ is exactly fun where being a lifting forces fun to satisfy $fun \circ alg = id_{\mathbf{C}}$ as required. Since alg and fun are both liftings it also follows immediately that $alg \circ fun = \overline{id}$ which is not $id_{\mathbf{C_H}}$ but rather only the identity up to the equivalence relation generated by fun. It should be remarked that the equivalence relation defined on fun in [BG] is a special case of a more general construction described in Example 3.4 below. So γ exists iff fun exists and is a lifting. Thus proposition 4.2 in [BG] should actually be an equivalence. For completeness we state the result formally.

Theorem 3.3 Let (H, ϵ, δ) be a comonad on **C**. H is computational iff the functor $fun : \mathbf{C_H} \to \mathbf{C}$ with the appropriate identities exists iff the lifting of the identity functor on **C**, $\overline{id} : \mathbf{C_H} \to \mathbf{C}$ exists. □

Example 3.4 Let $F : \mathbf{C} \to \mathbf{D}$ be a functor. We can always define an equivalence relation R on **C** as follows: for any two arrows f and g in **C**, f R g iff F(f) = F(g). It follows from functorality that R is a congruence relation(respects composition) and the quotient category **C/R** is nothing more than the category generated by the image of F. Since $E = alg \circ fun$ in Example 3.2 is a split idempotent factoring through **C**, it follows immediately that the quotient category of $\mathbf{C_H}$ via E is isomorphic to **C** and that $alg \circ fun$ is the identity up to the equivalence relation. The extensional collapse found in [BG] then follows directly from the above remarks.

If **C** has products then for comonad H on **C** it is easy to show that $\mathbf{C_H}$ also has products. In [BG] it is shown that the existence of such products implies the existence of a natural transformation satisfying certain equations. In the case of T the product functor, these equations coincide with those of Theorem 3.1. More however can be said as the converse is also true. The following corollary provides a different proof while showing that the conditions are in fact equivalent.

Corollary 3.5 Let **C** be a cartesian category with comonad H. Products in $\mathbf{C_H}$ which lift products from **C** correspond to the existence of natural transformations σ:H(_ × _)→ H _ × H_ satisfying the following equations

1) $\epsilon_A \times \epsilon_B \circ \sigma = \epsilon_{A \times B}$

2) $\delta_A \times \delta_B \circ \sigma = \sigma_{HA,HB} \circ H\sigma \circ \delta_{A \times B}$

Proof : Immediate from Theorem 3.1 where $K = H$ and $T = \_ \times \_$. □

We note that a third equation appears in [BG] which is present because of the assumption of a computational monad. The details are explained next.

Example 3.6 The third equation referred to above is $\sigma \circ \gamma_{\_\times\_} = \gamma \times \gamma$. It arises by changing the codomain of the lifting from $\mathbf{C_H}$

to **C**. Specifically the equation can be derived by utilizing the dual to lemma 2.7. Namely letting $T = \_ \times \_$ and S be the identity on **C**, we generate two corresponding natural transformations which are precisely σ and γ respectively. The natural transformation generated by the composition ST is, by the dual to Lemma 2.7, $\sigma \circ \gamma_T$ or $\sigma \circ \gamma_{\_\times\_}$ but is also equal to $T\gamma$ or $(\gamma \times \gamma)$ by the dual to Example 2.5.

Theorem 3.7 Let T, $\overline{T}, \lambda, H, K$ be as in Theorem 3.1. If K is a cocartesian comonad and **C** has coequalizers of reflexive pairs then $\overline{T}$ has a left adjoint implies so does T.
Proof : This is just the dual of Theorem 2.9. □

Theorem 3.8 Let **C**, **D**, $H, K, T, \overline{T}$ be as in Theorem 3.1. The natural transformation σ is an iso iff $G_K\overline{T} \cong TG_H$.
Proof : This is just the dual of Theorem 2.13. □

In [BG], the issue of exponentiation is raised. There the dual concern to that raised in section 2 emerges, namely given that **C** is a ccc, when is $\mathbf{C_H}$? A known sufficient condition is that T preserve products(see Corollary 3.10 below). In section 2 we were able to exploit the dual of theorem 3.7. Now we turn to the dual of theorem 2.18 which produces a general result which can be easily applied to the above remarks.

Theorem 3.9 Let **C**, **D**, $H, K, T, \overline{T}$ be as in Theorem 3.8. If T has a right adjoint then so does $\overline{T}$.
Proof : This is just the dual of Theorem 2.18. □

The last theorem now gives us an easy proof of the following well known result.

Corollary 3.10 Suppose **C** is a ccc and H is a comonad on **C** that preserves products. Then $\mathbf{C_H}$ is a ccc also.
Proof : Since **C** is cartesian, both $\overline{T}$ and σ exist by Theorem 3.3 (where $T = \_ \times \_$). Since H preserves products, σ is an isomorphism and also since **C** is a ccc, T has a right adjoint. By Theorem 3.9 $\overline{T}$ also has a right adjoint and we are done. □

Example 3.11 Corollary 3.10 gives sufficient conditions for $\mathbf{C_H}$ to be a ccc. As pointed out in [BG], the increasing paths comonad H on

Scott domains provides an example of corollary 3.10 since H preserves products. Since many interesting comonads do not preserve products, the weaker notion of a computational pairing is introduced in [BG]. This consists of a pair of natural transformations

$$\begin{aligned} &\mathit{split} : \mathrm{H}(\_ \times \_) \to \mathrm{H}(\_) \times \mathrm{H}(\_) \\ &\mathit{merge} : \mathrm{H}(\_) \times \mathrm{H}(\_) \to \mathrm{H}(\_ \times \_) \end{aligned}$$

and six identities. We now show how these ideas fit our general setup. Consider the setup of Corollary 3.5 so $K = H$ and $T = \_ \times \_$. The transformation *split* is just the usual natural transformation $\sigma : KT \to TH$ of Theorem 3.1 and *merge* is a transformation, $\lambda : TH \to KT$, in the reverse direction so that the diagrams generated by σ remain commutative with λ inserted. While *split* then is just the transformation guaranteed by the lifting of T, *merge* is an approximation of the natural transformation necessary to induce a lifting of T with respect to G_H and G_K . Since *merge* is not generally σ^{-1}, by Theorem 3.8 it does not satisfy $T \circ G_H = G_K \circ \overline{T}$. There is sufficient structure however so that for any map $f : HA \to B$ in $\mathbf{C_H}$ and for any T, $merge_B \circ TG_H(f) = G_K\overline{T}(f) \circ merge_A$ holds. Continuing in this way allows one to produce a weaker version of Theorem 3.9 for a weaker notion of right adjoint. In the case at hand one produces a weak form of exponentiation as described in [BG]. Of course when *merge* is the inverse of *split*, σ^{-1} is an iso and we have Theorem 3.9. More generally the existence of λ allows for the existence of a lifting of the right adjoint to T. This lifting however is not generally a right adjoint to $\overline{T}$.

As is correctly pointed out in [BG], the Kleisli category is independent of the choice of natural transformations *split* and *merge*. There is however far more variation present in our setup. Not only do the natural transformations change for different choices of comonad H , but they can also change for different liftings of a fixed H. Further one is not restricted to generating these transformations for a fixed functor T: $= \_ \times \_$ but rather for arbitrary functors $T : \mathbf{C} \to \mathbf{D}$. Many more applications of these results thus seem possible.

Generalizations of the results in this paper are certainly possible. For example one can combine the intensional and extensional approaches by considering the monadic and comonadic approach together. Work in that direction will appear elsewhere. We end this section with a different example of how the extensional semantics of section 2 and the intensional semantics of this section can be neatly combined. We consider comonad (H, ϵ, δ) and monad (K, ρ, ν) on categories $\mathbf{C}$ and $\mathbf{D}$ respectively. If T is a functor $T : \mathbf{C} \to \mathbf{D}$ we wish to determine when T can be lifted to a functor $\overline{T} : \mathbf{C_H} \to \mathbf{D_K}$ from the Kleisli category on comonad H to the Kleisli category on monad K. If f is an arrow $HA \to B$ in $\mathbf{C}$ then $\overline{T}(f)$ should be an arrow $TA \to KTB$ in $\mathbf{D}$. We have the following result.

Theorem 3.12 Let (H, ϵ, δ) be a comonad on $\mathbf{C}$. If H is a computational comonad then for any monad (K, ρ, ν) on any category $\mathbf{D}$, and any functor $T : \mathbf{C} \to \mathbf{D}$, a lifting $T^* : \mathbf{C_H} \to \mathbf{D_K}$ exists.
Proof : Suppose H is a computational comonad. By example 3.2, there exists a lifting $\overline{id} : \mathbf{C_H} \to \mathbf{C}$ which generates the natural transformation $\gamma : id \to H$. If $T : \mathbf{C} \to \mathbf{D}$ and monad (K, ρ, ν) are arbitrary then by an analogous argument to that found in Example 2.5, $\overline{T} = i_K \circ T$ exists, and $T^* = \overline{T} \circ \overline{id}$ defines a lifting where for $f : HA \to B$ in $\mathbf{C}$, $T^*(f) = \rho_{TB} T(f) T\gamma_A$. □

A converse to the above theorem is trivial by the discussion in example 3.2.

4 Conclusion

In this paper we have considered the general categorical question of when functorial processes may be lifted to corresponding Kleisli categories. A key theme is the recognition of the relationship between such extension results and the existence of natural transformations satisfying certain sets of equations. Particular attention was paid to finding conditions that ensured that adjoint pairs of functors could be lifted or inherited. This led naturally to various applications in both extensional and intensional semantics.

Much remains to be done. Special cases of the paper's results may be of interest. For example when categories $\mathbf{C}$ and $\mathbf{D}$ agree one can enumerate conditions when a particular endofunctor on $\mathbf{C}$ extends from a computational calculus generated by one monad to a computational calculus generated by another. This extension process need not be unique as more than one mediating natural transformation may be present. In light of the recent interest in the use of monads there should be many fruitful examples to explore, particularly for categories used to model semantics.

In a different direction the investigations in this paper have proved useful in formulating and describing results in sheaf semantics(see [Mu4]). In fact it was the analysis of certain technical conditions in this area that first motivated the questions raised in this paper. The methods have come into play by helping define and compare intrinsic orderings on objects in categories such as partial equivalence relations, building towards an axiomatic domain theory.

It is also hoped this paper will help contribute to our understanding of the algebraic relationships that exist between various semantic categories that implicitly or explicitly utilize monadic structure. Despite the huge volume of work to date in this area we still don't have a firm grasp of many of the fundamental algebraic mechanisms at play; mechanisms for example that determine which closure properties and evaluation strategies are definable or inherited when moving from one semantic setting to another.

5 References

[A] Applegate H., *Acyclic models and resolvent functors*, dissertation, Columbia University, 1965.

[BG] Brookes S., Geva S., *Computational comonads and intensional semantics*, **Applications of Category Theory**, LMSLNS 177, (1992) 1-44, Cambridge University Press.

[CP] Crole R. L., Pitts A. M., *New Foundations for FixPoint Computations*, **Proceedings of LICS**, University of Pennsylvania, 1990.

[F] Freyd P., *Algebraically Complete Categories*, preprint, 1991.

[FMRS] Freyd P., Mulry P., Rosolini G., Scott D., *Extensional PERs*, **Information and Computation**, 98 (1992), 211-227.

[J] Johnstone P. T., *Adjoint Lifting Theorems for Categories of Algebras*, **Bull. London Math. Soc.**, 7 (1975), 294-297.

[K] Kock A., *Strong functors and monoidal monads*, **Arch. Math.**, 23 (1972), 113-120.

[Ma] Manes E. G., *A triple miscellany: some aspects of the theory of algebras over a triple*, dissertation, Wesleyan University, 1967.

[Mo] Moggi E.,*Notions of Computations and Monads*, **Information and Computation**, 93 (1991), 55-92.

[Mu1] Mulry P. S., *Monads and Algebras in the Semantics of Partial Data Types*, **Theoretical Computer Science**, 99 (1992), 141-155.

[Mu2] Mulry P. S., *Strong Monads, Algebras and Fixed Points*, **Applications of Category Theory**, LMSLNS 177, (1992) 202-216, Cambridge University Press.

[Mu3] Mulry P. S., *Partial Map Classifiers and Partial Cartesian Closed Categories*, to appear.

[Mu4] Mulry P. S., *Partial Sheaf Semantics*, to appear.

Sequential Functions on Indexed Domains and Full Abstraction for a Sub-language of PCF

Stephen Brookes Shai Geva
brookes@cs.cmu.edu shai@cs.cmu.edu
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213

October 29, 1993

Abstract

We present a general semantic framework of sequential functions on domains equipped with a parameterized notion of incremental sequential computation. Under the simplifying assumption that computation over function spaces proceeds by successive application to constants, we construct a sequential semantic model for a non-trivial sub-language of PCF with a corresponding syntactic restriction — that variables of function type may only be applied to closed terms. We show that the model is fully abstract for the sub-language, with respect to the usual notion of program behavior.

1 Introduction

A semantics for a programming language is fully abstract with respect to a given notion of program behavior iff the semantics distinguishes between two terms exactly when there is a program context in which the terms induce different behavior. Intuitively, a fully abstract semantics is at precisely the right level of abstraction to support compositional reasoning about behavior. It has turned out to be surprisingly difficult to give natural (*i.e.*, language-independent) constructions of fully abstract semantic models for sequential languages such as PCF [Plo77, BCL85]. The constructions of fully abstract models for PCF given by Milner, Berry and Mulmuley [Mil77, Ber78, Mul87] are not natural. Yet there are natural fully abstract models for an extension of PCF with parallel facilities [Plo77] and, more recently, with control facilities [CF92, Cur92].

The first definitions of sequential functions, given by Milner [Mil77] and Vuillemin [Vui73], were limited to functions on products of flat domains. Sazonov's definition of sequential functions [Saz75] is also of limited scope. Kahn and Plotkin [KP78] introduced concrete data structures and concrete domains, and defined sequential functions between concrete domains. However, the sequential functions between two concrete domains do not form a concrete domain (under either the pointwise or stable orders). Berry introduced dI-domains, stable functions and the stable ordering [Ber78]; the stable functions between two dI-domains, ordered stably, form a dI-domain. However, the stable functions do not provide the desired notion of sequential functions, since some stable functions are not sequential. Berry and Curien [BC82, Cur86] defined sequential algorithms between concrete domains, and obtained a sequential intensional model from which one may recover the Kahn-Plotkin sequential functions by taking an extensional quotient. More recently, Bucciarelli and Ehrhard [BE91] introduced a notion of strongly stable functions between qualitative domains equipped with a coherence structure (QDC's), generalizing the Kahn-Plotkin definition. In earlier work [BG92] we defined sequential functions on Scott domains that generalized Kahn and Plotkin's sequential functions, and we obtained several closure results under the sequential function space.

We continue here the investigation of sequentiality. We present a framework of *indexed domains*, domains equipped with a parameterized notion of incremental sequential computation, formulated as an *index structure*. We give a general definition of *sequential functions between indexed domains*,

This research was supported in part by National Science Foundation grant CCR-9006064.

as continuous functions that, essentially, respect the index structure. We define an indexed domain product, and show closure of indexed domains under the sequential function space, for both the pointwise and the stable orderings (with different classes of underlying domains). Indexed domains are closely related to Bucciarelli and Ehrhard's domains with coherence structures; we discuss this relationship in the conclusion.

Our earlier definition of sequential functions [BG92] arises when all domains are equipped with a particular ***data-index structure***, which imposes a notion of incremental computation adequate for domains of data. This is the index structure which is (implicitly) present in Kahn and Plotkin's, Milner's and Vuillemin's definitions of sequentiality, as well as in Berry and Curien's sequential algorithms over concrete data structures and the language CDS0 [BC85, Cur86]; it is also used by Bucciarelli and Ehrhard for defining sequentiality at first-order. In PCF, however, computation over function spaces proceeds in an inherently different manner, and thus the use of data-indices is not always appropriate; a suitable higher-order notion of incremental computation is called for.

The framework of indexed domains and sequential functions is not adequate to provide a fully abstract model for PCF, since function application fails to be sequential according to our definitions. *Nevertheless, application of a sequential function to fixed arguments is a sequential function* in our sense. We introduce a new higher-order notion of *constant-applicative* sequentiality, in which computation over function spaces proceeds by successive application to constants. We also introduce a sub-language of PCF, which we call ca-PCF, obtained by imposing a corresponding syntactic constraint on uses of application: variables of function type may only be applied to closed terms. We show that a sequential model employing data sequentiality at ground types and the pointwise order and constant-applicative sequentiality at arrow types is fully abstract for ca-PCF, with respect to the usual notion of program behavior. We have not yet found a completely satisfactory higher-order notion of sequentiality, since the lack of sequentiality of application prevents us from obtaining a cartesian closed category.

2 Preliminaries

We assume conventional domain-theoretic definitions and notations. The original definitions of stability are due to Berry [Ber78], and Zhang [Zha91] gave a generalized topological characterization. We generalized Zhang's definitions to Scott domains and to the pointwise order in [BG92], where a full development may be found.

A *Scott domain* is a directed-complete, bounded-complete, ω-algebraic poset with a least element. We write $x \Uparrow y$ to indicate that x and y are bounded (consistent). We write $K(D)$ for the set of isolated (finite, compact) elements of D; when $X \subseteq D$ we also write $K(X)$ for $X \cap K(D)$. A *dI-domain* is a distributive Scott domain with property (I), *i.e.*, such that every isolated element dominates finitely many elements. A (non-empty) subset X of a poset is *filtered* iff every pair of elements of X has a lower bound in X. The *covering* relation is defined by setting $x \prec y$ iff $x < y$ and the set $\{z \mid x < z \;\&\; z < y\}$ is empty. We define the upper set of $x \in D$ by $\uparrow x = \{z \in D \mid x \leq z\}$. For $u \subseteq D$ let $\uparrow u = \bigcup\{\uparrow x \mid x \in u\}$. A set u is up-closed iff $u = \uparrow u$. Similarly the lower set of x is $\downarrow x$.

An *arithmetic domain* [GHK+80] is a Scott domain with the finite meet property (FM): the meet of each pair (or equivalently, every non-empty finite set) of isolated elements is itself isolated. Arithmetic domains are a proper intermediate class of domains between dI-domains and Scott domains.

In an algebraic poset, a subset $p \subseteq D$ is *Scott open* iff $p = \uparrow K(p)$, and it is *stable open* if, in addition, it is closed under bounded meets, *i.e.*, if $x_1, x_2 \in p$ and $x_1 \Uparrow x_2$ then $x_1 \wedge x_2 \in p$. Write Sc D for the set of Scott opens of D and St D for the set of stable opens of D. For every $x \in K(D)$, $\uparrow x$ is Scott open and stable open. The Scott opens and the stable opens of a domain D have T_0 separation, *i.e.*, for every $x, y \in D$, $x = y$ iff $\{p \in \text{Sc}\, D \mid x \in p\} = \{p \in \text{Sc}\, D \mid y \in p\}$, and likewise for stable opens.

Scott opens define the Scott topology. Stable opens do not form a true topology, but may be regarded as a generalized topology. Every stable open may be decomposed into a disjoint union of *lobes*, which are Scott open filters. In a dI-domain every lobe has a least element. Stable opens of a dI-domain are therefore upper sets of pairwise inconsistent sets of isolated elements, coinciding with Zhang's stable neighborhoods [Zha91].

A function $f : D \to D'$ is Scott continuous, or just *continuous*, iff $f^{-1}q \in \mathrm{Sc}\, D$ for every $q \in \mathrm{Sc}\, D'$. Equivalently, f is continuous iff it is monotone and preserves directed lubs. A function $f : D \to D'$ is stable continuous, or just *stable*, iff $f^{-1}q \in \mathrm{St}\, D$ for every $q \in \mathrm{St}\, D'$. Equivalently, f is stable iff f is continuous and preserves bounded meets, *i.e.*, if $x_1 \Uparrow x_2$ then $f(x_1 \wedge x_2) = fx_1 \wedge fx_2$.

For continuous functions $f, g : D \to D'$, we define the pointwise ordering by $f \leq g$ iff $fx \leq gx$ for every $x \in D$ or, equivalently, $f^{-1}q \subseteq g^{-1}q$ for every $q \in \mathrm{Sc}\, D'$. We write $\bigvee^{\mathrm{p}} F$ for the pointwise lub of a family F of functions, defined, if it exists, by $(\bigvee^{\mathrm{p}} F)x = \bigvee\{fx \mid f \in F\}$.

Scott domains and arithmetic domains are closed under the pointwise-ordered continuous function space. All existing lubs in the pointwise-ordered continuous function space are taken pointwise. Function application is continuous, and the category of Scott domains and continuous functions is cartesian closed, with a full sub-ccc of arithmetic domains and continuous functions.

For $x \in K(D)$ and $y \in K(D')$, define the *step function* $[x \Rightarrow y] : D \to D'$ by setting $[x \Rightarrow y]x' = y$ if $x' \in \uparrow x$, and $[x \Rightarrow y]x' = \bot$ otherwise. The notation $[x \Rightarrow y]$ will imply that x and y are isolated. The isolated elements of the pointwise-ordered continuous function space are those functions which are the pointwise lubs of finitely many step functions.

3 Sequentiality

3.1 Sequential Functions on Indexed Domains

In order to model sequential computation, we equip domains with a parameterized notion of *indices*, intended to formalize incremental steps of a computation. Let an *index function* for a domain D be a function $I : D \to \mathcal{P}(\mathrm{St}\, D)$ such that the following properties hold, for every $x \in D$:

- True increment: For every $r \in Ix$, $x \notin r$.
- Separation: If $x < y$ then there exists $r \in Ix$ such that $y \in r$.
- Upwards motion: If $x < y$, $r \in Ix$ and $y \notin r$ then $r \in Iy$.
- Finite origin: If $r \in I(\bigvee X)$ and X is a directed set then there exists $x_0 \in X$ such that $r \in Ix_0$. Equivalently, in an algebraic poset, if $r \in Ix$ then there exists some isolated $x_0 \leq x$ such that $r \in Ix_0$.
- Definiteness: For every $r \in Ix$, $r = \uparrow(\min r)$ for the set $\min r$ of minimal elements of r.

  (This is always the case for every stable open of a dI-domain.)

An *indexed domain* E is a pair $E = (D, I)$ of a domain D and an index function I for D. When convenient we blur the distinction between an indexed domain and its underlying domain.

For $x \in E$ and $s \subseteq \uparrow x$, we call $r \in Ix$ an index of s at x iff $s \subseteq r$. We write $I(x, s)$ for the set of indices of s at x, $I(x, s) = \{r \in Ix \mid s \subseteq r\}$.

Operationally, if the current approximation of a value v being computed is x then an index $r \in Ix$ is intended to represent a possible next step in the computation, resulting in an improved approximation by selecting among the alternatives that the index offers. A sequential computation over a domain may then be seen as a sequence of choices among alternatives posed by indices at an increasing sequence of approximations. The index function determines which sequences of approximations may be computable. It may help to think of v as an input value to a program, with the program improving its approximation x to v by a process of incremental approximation, until it has sufficient information to determine its output on input v. One may also think of a program computing a sequence of ascending approximations to some target output value.

A stable open r represents a choice between its lobes. Since an index $r \in Ix$ is definite, *i.e.*, $r = \uparrow(\min r)$, the choice is represented even more concretely as a choice between the elements of $\min r$, which may be seen as competing alternative approximations to the target value v. The increment in information will be to $x \vee y$, where y is the element of $\min r$ approximating v. Since r is stable open, its minimal elements are isolated and pairwise inconsistent, so that y will be unique, if it exists. The true increment property guarantees that $x \notin r$, and thus $x < x \vee y$. If $v \notin r$, then the computation step represented by r may be said to diverge; a program that attempts to take step r at x is undefined for input v, in the input scenario, or may not output v, in the output scenario. An index $r \in I(x,s)$ of s at x may be seen as an incremental step from current approximation x towards a choice represented by s, in that it guarantees non-divergence for $v \in \uparrow s$.

A subset $p \subseteq E$ is *sequential open* iff it is Scott open and, for every $x \in E$, either $x \in p$ or every finite $s \subseteq p \cap \uparrow x$ has some index at x, *i.e.*, $I(x,s) \neq \emptyset$. Write $\mathsf{Sq}\,E$ for the collection of sequential opens of E, ordered by set inclusion. A function $f : E \to E'$ is sequential iff $f^{-1}q \in \mathsf{Sq}\,E$ for every $q \in \mathsf{Sq}\,E'$. Let $E \to^{sq} E'$ be the sequential function space between E and E', ordered pointwise. Thanks to the generalized topological definition, it is trivial to check that the identity functions are sequential and that composition preserves sequentiality. so that indexed domains and sequential functions form a category (for any underlying class of domains).

3.2 Sequentiality in Terms of Critical Sets

By the separation property of indices, $I(x,s)$ is non-empty whenever $x < \wedge s$, so it is only interesting to ask if $I(x,s)$ is empty in the case where $x = \wedge s$. This gives rise to a definition of critical sets, which provide convenient alternative characterizations of sequentiality.

A *critical* set of an indexed domain $E = (D,I)$ is a non-empty finite subset $s \subseteq E$ that has no index at its meet, *i.e.*, such that $I(\wedge s, s) = \emptyset$. The fundamental properties of critical sets are developed in [BG92]. The following proposition summarizes the most important results.

Proposition 3.1

(1) Every finite set s with a least element, and, in particular, every singleton, is critical.

(2) A set p is sequential open iff it is Scott open and closed under critical meets. For every $x \in K(E)$, $\uparrow x$ is sequential open. Sequential opens have the T_0 separation property.

(3) A finite set s is critical iff every sequential open that contains it also contains its meet $\wedge s$.

(4) A function $f : E \to E'$ is sequential iff it is continuous and it preserves criticality and meets of critical sets, i.e., for every critical set s of E, $fs = \{fx \mid x \in s\}$ is critical, and $f(\wedge s) = \wedge(fs)$.

(5) Every finite bounded set is critical. Every sequential open is stable open. Every sequential function is stable.

3.3 Product of indexed domains

It seems reasonable to assume that an incremental step of a sequential computation in a product domain $D_1 \times D_2$ corresponds to an increment in one of the components, but not both. This leads us to define the indexed domain product of $E_1 = (D_1, I_1)$ and $E_2 = (D_2, I_2)$ to be the indexed domain $E_1 \times E_2 = (D_1 \times D_2, I_\times)$, where $D_1 \times D_2$ is the usual domain product, ordered componentwise, and $I_\times$ is defined by

$$\begin{aligned} I_\times(x,y) \quad = \quad & \{r \times (\uparrow z) \mid r \in I_1 x \;\&\; z \in K(\downarrow y)\} \cup \\ & \{(\uparrow z) \times r \mid r \in I_2 y \;\&\; z \in K(\downarrow x)\}. \end{aligned}$$

It is easy to check that $I_\times$ is an index function.

Proposition 3.2 *A finite set $s \subseteq E_1 \times E_2$ is critical iff both $\pi_1 s$ and $\pi_2 s$ are critical.*

An indexed domain product is, in fact, a categorical product in the category of indexed domains and sequential functions (for any underlying class of domains closed under product).

Proposition 3.3 *Indexed domain product is a categorical product.*

Proof: It is sufficient and easy to show that the projections $\pi_i : E_1 \times E_2 \rightarrow E_i$ are sequential, for $i = 1, 2$, and that for sequential functions $f_i : E \rightarrow E_i$, $i = 1, 2$, the mediating morphism $\lambda x \in E \,.\, (f_1 x, f_2 x)$ is a sequential function from E to $E_1 \times E_2$. ▪

A sequential function on an indexed domain product remains sequential when one of its arguments is fixed.

Proposition 3.4 *For every sequential function $f : E_1 \times E_2 \rightarrow E'$ and every $x \in E_1$, the function* curry $fx = \lambda y \in E_2 \,.\, f(x, y)$ *is a sequential function from E_2 to E'.*

3.4 Ordering the sequential function space

An adaptation of the development in [BG92] shows that arithmetic domains are closed under the pointwise-ordered sequential function space. In other words, if E and E' are indexed arithmetic domains then the sequential function space $E \rightarrow^{sq} E'$, equipped with the pointwise order, is an arithmetic domain. Property FM is essential for this. An even simpler development shows that dI-domains are closed in the same sense under the stably-ordered function space — this is an easy corollary of the downwards closure of sequential functions in the stably-ordered stable function space. The following proposition summarizes these results; proofs may be found in [BG92].

Proposition 3.5 *Arithmetic domains are closed under the pointwise-ordered sequential function space, regardless of the index structures used. Directed lubs and finite meets are taken pointwise, and the isolated elements are the sequential functions that are the pointwise lubs of finitely many step functions.*

3.5 Application is not sequential

Function application app : $(E \rightarrow^{sq} E', I) \times E \rightarrow E'$ is not sequential, no matter which index function I is used, and whether we employ the pointwise or stable orders. This also establishes that uncurrying does not preserve sequentiality, since function application is the uncurrying of an identity function. It is perhaps not surprising that uncurrying does not preserve sequentiality: the uncurried form of a function has a more complicated domain of definition, where more subtle interactions are possible that would prevent the uncurried form from being sequential.

The counter-example relies on the product structure and on the index domain axioms, and in particular on the criticality of a set with a least element, a corollary of the true increment property. Let Bool be the domain of booleans, with elements $\perp < \mathtt{T}, \mathtt{F}$. Consider the application function app : $(\mathtt{Bool}^3 \rightarrow^{sq} \mathtt{Bool}) \times \mathtt{Bool}^3 \rightarrow \mathtt{Bool}$, and the sets

$$\begin{aligned} s &= \{([(\mathtt{T},\mathtt{F},\perp) \Rightarrow \mathtt{T}] \vee [x \Rightarrow \mathtt{T}], x) \mid x \in t\} \\ t &= \{(\mathtt{T},\mathtt{F},\perp), (\perp,\mathtt{T},\mathtt{F}), (\mathtt{F},\perp\,\mathtt{T})\}. \end{aligned}$$

$\pi_1 s$ has least element $[(\mathtt{T},\mathtt{F},\perp) \Rightarrow \mathtt{T}]$ in both the pointwise and stable orderings on the function space, and $\pi_2 s = t$ is critical, since its projection on any of its three components has a least element $\perp$. Thus s is critical. But $\mathsf{app}(\wedge s) = [(\mathtt{T},\mathtt{F},\perp) \Rightarrow \mathtt{T}](\perp,\perp,\perp) = \perp \neq \mathtt{T} = \wedge\{\mathtt{T}\} = \wedge(\mathsf{app}\, s)$, so that app fails to preserve a critical meet, and is therefore not sequential.

This negative result implies that we cannot use the framework presented here — the category **IDOM** of indexed arithmetic domains and sequential functions — to give a sequential model for all of PCF, since application is definable in PCF (up to currying). Nevertheless, we will be able to give a sequential model for an appropriately restricted subset of PCF.

4 Interpreting types as indexed domains

We look now at ways of instantiating the index structure to obtain type interpretations in the category **IDOM**. We consider the simple type system generated by the grammar $\sigma ::= \rho \mid \sigma \rightarrow \sigma'$, where ρ ranges over a set of ground types.

We assume given a flat domain $\mathcal{A}[\![\rho]\!]$ for each ground type ρ. We need to choose an index function for each such $\mathcal{A}[\![\rho]\!]$ in order to interpret ρ as an indexed domain. We then intend to interpret an arrow type $\sigma \rightarrow \sigma'$ as the sequential function space between the indexed domains representing σ and σ', ordered pointwise, with a suitably chosen index structure on the function space. This raises the question of what kind of index function is appropriate for a sequential function space.

4.1 Data sequentiality

At first-order types like $\sigma \rightarrow \sigma'$, where σ and σ' are ground, the notion of sequential function defined in [BG92] is adequate. This notion of sequentiality, which we will call ***data sequentiality***, coincides with the Kahn-Plotkin sequential functions when σ and σ' are restricted to concrete domains [KP78, Cur86], and coincides with the Milner and Vuillemin notions of sequentiality when the types are restricted to products of flat domains [Mil77, Vui73].

Data sequentiality is characterized in terms of index functions as follows. Although we only need to use the definitions here when D is a flat domain (since ground types are flat), it is easy to give a more general definition for dI-domains.

For a dI-domain D we define the data-index function I^{d}_D at $x \in D$ by setting

$$I^{\mathrm{d}}_D x = \{r \in \mathsf{St}\, D \mid x \notin r \;\&\; \forall y \in \min r \,.\, (x \Uparrow y \Rightarrow x \prec x \vee y)\}\,.$$

The data-index function I^{d}_D is easily seen to be an index function for a dI-domain D.

This definition of data sequentiality requires *atomicity* of the increment represented by an index, so that successive approximations to an input will form a covering chain. If atomicity is not imposed, say, if we used $I_D x = \{r \in \mathsf{St}\, D \mid x \notin r\}$ then one could, for instance, check in a single step whether an input in $\mathtt{Bool}^3$ is in $\uparrow\{(\mathtt{T},\mathtt{F},\bot),(\bot,\mathtt{T},\mathtt{F}),(\mathtt{F},\bot,\mathtt{T})\}$. This would clearly not be appropriate for computation in a sequential language.

Data sequentiality interacts nicely with indexed domain product. By atomicity, progress cannot be made simultaneously in different components of a product, since $(x,y) \prec (x',y')$ iff either $x \prec x'$ and $y = y'$, or $x = x'$ and $y \prec y'$; this corresponds exactly to the reasoning behind the definition of product. Therefore, the index function for the product of data-indexed domains coincides with the data-index function for the product, *i.e.*,

$$(D_1 \times D_2, I^{\mathrm{d}}_{D_1 \times D_2}) = (D_1, I^{\mathrm{d}}_{D_1}) \times (D_2, I^{\mathrm{d}}_{D_2}).$$

In [BG92] we attempted to use data sequentiality uniformly for all domains, *i.e.*, to construct a sequential model in which each type is interpreted as a data-indexed domain. This corresponds to an operational assumption that incremental computation over a function space proceeds in the same way as incremental computation over data. This assumption is reasonable in some frameworks, such as concrete domains and sequential algorithms, and the language CDS0 [BC85, Cur86]. However, this operational assumption is not appropriate for PCF, where information about a functional argument is essentially incremented by *applying* it. We thus perceive the need to employ a different, higher-order, notion of sequentiality over the functional domains, that would correspond better to PCF's operational assumptions. (See [BG92] for further discussion.)

4.2 Constant-applicative sequentiality

In order to arrive at a higher-order notion of sequentiality more closely matching PCF's operational character, we analyze the way in which information about functional inputs is obtained in PCF. This is ultimately done by applying such an argument as a PCF variable, say $\mathtt{f}$, to an argument of appropriate type, say, a term M, with the result of the application $\mathtt{f}M$ conveying information about the input represented by $\mathtt{f}$. Call M the *prompter* of f.

For example, consider the following PCF term M_0:

$$\begin{array}{rcl} M_0 & = & \lambda\mathtt{f} : \mathtt{Bool} \rightarrow \mathtt{Bool} \rightarrow \mathtt{Bool}\,. \\ & & \quad \mathtt{if}(\mathtt{f}\,\mathtt{T}\,\Omega)\&(\mathtt{f}\,\mathtt{F}\,\mathtt{T})\&\neg(\mathtt{f}\,\mathtt{F}\,\mathtt{F})\,\mathtt{then}\,\mathtt{T}\,\mathtt{else}\,\Omega, \end{array}$$

where Ω is a divergent constant of type $\mathtt{Bool}$, and $\&$ is the PCF term for the left-strict-and function written in infix notation. When M_0 is applied to a term M of type $\mathtt{Bool} \rightarrow \mathtt{Bool} \rightarrow \mathtt{Bool}$, M_0 may be seen as successively increasing its information about its input. The result of the application is

T precisely when the sequence of approximations is

$$\bot,\ [(\mathtt{T},\bot)\Rightarrow \mathtt{T}],\ [(\mathtt{T},\bot)\Rightarrow \mathtt{T}]\vee[(\mathtt{F},\mathtt{T})\Rightarrow \mathtt{T}],\ [(\mathtt{T},\bot)\Rightarrow \mathtt{T}]\vee[(\mathtt{F},\mathtt{T})\Rightarrow \mathtt{T}]\vee[(\mathtt{F},\mathtt{F})\Rightarrow \mathtt{F}]$$

(up to currying). Each step of the computation corresponds to an application of $\mathtt{f}$ to some prompter. Divergence of any step would imply divergence of the entire computation. The term M_0 uses only *closed* prompters: each application of $\mathtt{f}$ is to "constant" arguments.

Consider next the prompters in the following PCF terms,

$$\begin{aligned} M_1 &= \lambda\mathtt{f}:\sigma\to\sigma'.\,\lambda\mathtt{x}:\sigma.\,\mathtt{fx},\\ M_2 &= \lambda\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool}\to\mathtt{Bool}.\\ &\quad\ \mathtt{if}(\mathtt{f}(\mathtt{f}\,\mathtt{T}\,\Omega)(\mathtt{f}\,\Omega\,\mathtt{T}))\&(\mathtt{f}\,\mathtt{T}\,\mathtt{F})\&(\mathtt{f}\,\mathtt{F}\,\mathtt{T})\&\neg(\mathtt{f}\,\mathtt{F}\,\mathtt{F})\,\mathtt{then}\,\mathtt{T}\,\mathtt{else}\,\Omega. \end{aligned}$$

In the first case, M_1 denotes the identity function on the type $\sigma\to\sigma'$, or, up to currying, the corresponding application function. It is not strict in the input $\mathtt{x}$, but it is strict in the input $\mathtt{f}$. The prompter $\mathtt{x}$ of $\mathtt{f}$ may be said to be *input-dependent*, in that it involves an input other than $\mathtt{f}$. In the second case, M_2 denotes the least functional that maps the left-strict-or function $\mathsf{lor} = [(\mathtt{T},\bot)\Rightarrow \mathtt{T}]\vee[(\mathtt{F},\mathtt{T})\Rightarrow \mathtt{T}]\vee[(\mathtt{F},\mathtt{F})\Rightarrow \mathtt{F}]$ and the right-strict-or function $\mathsf{ror} = [(\bot,\mathtt{T})\Rightarrow \mathtt{T}]\vee[(\mathtt{T},\mathtt{F})\Rightarrow \mathtt{T}]\vee[(\mathtt{F},\mathtt{F})\Rightarrow \mathtt{F}]$ to $\mathtt{T}$. It is defined using *imbrication* [BCL85, p. 129]; the prompter $\mathtt{f}\,\mathtt{T}\,\Omega$ may be said to be *self-dependent*, since it uses the input $\mathtt{f}$ about which information is being sought.

We are not yet able to give a satisfactory treatment of dependent prompters. Instead, in this paper we make the simplifying assumption that prompters must be constant, *i.e.*, independent of the input. On the syntactic side, we will impose a restriction on the use of application so that we need only consider PCF terms using closed prompters. The terms M_1 and M_2 are thus excluded from consideration. On the semantic side, we assume that a computation of a value f over a function space $E\to^{\mathrm{sq}}E'$ proceeds at each step by determining the result of applying f to a constant element in E. This gives rise to the notion of *constant-applicative sequentiality*.

Corresponding to a value $x\in K(E)$ and a "residual" index r' in E', we define a ca-index $[x\Rightarrow r']$ in the pointwise-ordered sequential function space $E\to^{\mathrm{sq}}E'$ between E and E' to be the stable open

$$[x\Rightarrow r'] = \uparrow\{[x\Rightarrow y]\mid y\in K(r')\} = \uparrow\{[x\Rightarrow y]\mid y\in \min r'\}$$

of $E\to^{\mathrm{sq}}E'$; and we define the ca-index function $I^{\mathrm{ca}}_{E,E'}$ on $E\to^{\mathrm{sq}}E'$ by:

$$I^{\mathrm{ca}}_{E,E'}f \quad=\quad \{[x\Rightarrow r']\mid x\in K(E)\ \&\ r'\in I'(fx)\}.$$

It is easy to check that $I^{\mathrm{ca}}_{E,E'}$ is an index function for $E\to^{\mathrm{sq}}E'$. From this point on, we will assume that the sequential function space is equipped with the ca-index function; the following results depend on this choice.

Proposition 4.1 *A finite set $s\subseteq E\to^{\mathrm{sq}}E'$ is critical iff for all $x\in K(E)$, $sx=\{fx\mid f\in s\}$ is critical.*

Proposition 4.2 *Currying preserves sequentiality, i.e., if $f:E_1\times E_2\to E'$ is a sequential function then $\mathrm{curry}\,f:E_1\to(E_1\to^{\mathrm{sq}}E')$ is a sequential function.*

Application of a fixed sequential function $f\in E\to^{\mathrm{sq}}E'$, *i.e.*, the function $\lambda z\in E\,.\,fz$, coincides with f and is therefore sequential. More importantly, application to a fixed argument is sequential.

Proposition 4.3 *For every $z\in E$, the function $\lambda f\in E\to^{\mathrm{sq}}E'\,.\,fz$ is sequential.*

Proof: If s is a critical set of $E\to^{\mathrm{sq}}E'$ then sz is critical, and $\wedge(sz)=(\wedge s)z$. ∎

4.3 Maximal uncurrying

As we have indicated, the meaning $[\![\sigma\to\sigma']\!]$ of an arrow type in the model will essentially be taken to be the sequential function space between $[\![\sigma]\!]$ and $[\![\sigma']\!]$. A further refinement is still needed. Type interpretations are usually defined in ccc's, where there is an isomorphism

$$[\sigma_1] \to ([\sigma_2] \to [\sigma']) \cong ([\sigma_1] \times [\sigma_2]) \to [\sigma']$$

via currying and uncurrying. In that case, it doesn't really matter which of the two is taken to define $[\sigma_1 \to (\sigma_2 \to \sigma')]$. This is not so in our case, since uncurrying does not preserve sequentiality.

The question now arises whether a function that is sequential in its curried form, but not in its uncurried form, should be included in the sequential model. For example consider the parallel-or function, $\mathtt{por} = [(\mathtt{T},\bot) \Rightarrow \mathtt{T}] \vee [(\bot,\mathtt{T}) \Rightarrow \mathtt{T}] \vee [(\mathtt{F},\mathtt{F}) \Rightarrow \mathtt{F}]$. Its curried form, $\mathtt{curry\,por} : \mathtt{Bool} \to (\mathtt{Bool} \to \mathtt{Bool})$, is sequential because of the trivial index structure of Bool. However, por itself, of type $\mathtt{Bool} \times \mathtt{Bool} \to \mathtt{Bool}$, is not sequential: $\mathtt{por}^{-1}\{\mathtt{T}\} = \uparrow\{(\mathtt{T},\bot),(\bot,\mathtt{T})\}$ is not sequential open. Moreover, parallel-or is not definable in PCF, and it is therefore desirable to exclude it from any sequential model. Thus, we will regard as "truly" sequential only those functions whose maximally uncurried form is sequential. To build a model including only such functions we will interpret arrow types in their maximally uncurried form.

4.4 The sequential type interpretation

We now define the sequential type interpretation $\mathcal{C}[-]$, mapping each type σ to an indexed domain $\mathcal{C}[\sigma]$:

- For a ground type ρ, $\mathcal{C}[\rho]$ is the flat domain $\mathcal{A}[\rho]$, equipped with the standard data-index structure.
- Each arrow type σ can be written uniquely in the form $\sigma_1 \to \cdots \to \sigma_n \to \rho$, where $n \geq 1$ and ρ is ground. We define $\mathcal{C}[\sigma]$ to be the sequential function space $\mathcal{C}[\sigma_1] \times \cdots \times \mathcal{C}[\sigma_n] \to^{sq} \mathcal{C}[\rho]$, ordered pointwise, with the standard constant-applicative index structure.

We assume that we have at least ground types Bool and Nat, corresponding to the the usual flat domains of truth values and natural numbers respectively.

5 A sub-language of PCF

5.1 The ca-PCF typing system and semantics

Raw (untyped) terms are built from a given set of constants, identifiers, application and abstraction in the usual way, as in PCF. We define axioms and inference rules for judgements of the form $\Gamma \vdash M : \sigma$, to be read as: the term M has type σ in type context Γ. A type context is a finite ordered list of identifier-type pairs, and we write $\Gamma, v : \sigma$ for the type context obtained by extending Γ with the binding $v : \sigma$. Identifiers may occur more than once in a type environment, and the rightmost occurrence always takes precedence. The essential restriction imposed by our typing system is that a variable of functional type may only be applied to closed terms. This captures the simplifying assumption that prompters cannot depend on the input. For convenience we also require that a variable of functional type be applied successively to as many arguments as needed to obtain a result of ground type; this restriction is less important.

The terms of ca-PCF are those terms M for which a judgement $\Gamma \vdash M : \sigma$ is derivable. We use L to range over terms, and K to range over closed terms. A term K is closed iff it has no free identifiers; equivalently, if $\vdash K : \sigma$ is derivable for some σ.

We define a semantic function $\mathcal{C}[-]$ for judgements $\Gamma \vdash M : \sigma$ by induction on the proof of the judgement. Throughout we assume that Γ has form $v_1 : \gamma_1, \ldots, v_m : \gamma_m$ and that σ is written in the form $\sigma_1 \to \cdots \to \sigma_n \to \rho$, where ρ is ground. The meaning of $\Gamma \vdash M : \sigma$ will be

$$\begin{aligned} \mathcal{C}[\Gamma \vdash M : \sigma] &\in \mathcal{C}[\gamma_1 \to \cdots \to \gamma_m \to \sigma] \\ &= \mathcal{C}[\gamma_1] \times \cdots \times \mathcal{C}[\gamma_m] \times \mathcal{C}[\sigma_1] \times \cdots \times \mathcal{C}[\sigma_n] \to^{sq} \mathcal{C}[\rho]. \end{aligned}$$

Note that the environment is "blended into" the semantic domains; this is necessary, since all functions in the model, including the meanings of terms, are to be fully uncurried.

We assume a semantic function $\mathcal{A}[-]$ for constants such that $\mathcal{A}[c] \in \mathcal{A}[\sigma]$ for each constant c of type σ. As in PCF we assume at least the following constants with their usual interpretations.

- A constant of type ρ for each element of each ground type ρ. In particular,
  - A numeral of type `Nat` for each natural number.
  - Constants `T` and `F` of type `Bool`, denoting the corresponding truth values.
  - For each ground type ρ a constant Ω^ρ of that type, denoting the least element of $\mathcal{A}[\![\rho]\!]$.
- For each ground type ρ, a constant $\mathtt{if}^\rho$ of type $\mathtt{Bool} \to \rho \to \rho \to \rho$ such that

$$\mathcal{A}[\![\mathtt{if}^\rho]\!] = \bigvee^{p} \{[(\mathtt{T}, x, \bot) \Rightarrow x], [(\mathtt{F}, \bot, x) \Rightarrow x] \mid x \in \mathcal{A}[\![\rho]\!]\} .$$

- Arithmetic constants: $(+1)$ and (-1) of type $\mathtt{Nat} \to \mathtt{Nat}$, $(=0)$ of type $\mathtt{Nat} \to \mathtt{Bool}$, denoting the successor and predecessor functions and the equal-to-zero predicate, respectively. As in PCF, $\mathcal{A}[\![(-1)]\!]0 = \bot$.
- Basic operations on ground types other than `Bool` and `Nat` are left unspecified. We will later assume existence of constants necessary for definability.

The following are the axioms and inference rules for the typing system, together with the definition of the semantic function $\mathcal{C}[\![-]\!]$.

- Constants:

$$\frac{}{\Gamma \vdash c : \sigma}\ \text{const}$$

$$\begin{aligned}\mathcal{C}[\![\Gamma \vdash c : \sigma]\!] \;=\; & \lambda(x_1 \in \mathcal{C}[\![\gamma_1]\!], \ldots, x_m \in \mathcal{C}[\![\gamma_m]\!], y_1 \in \mathcal{C}[\![\sigma_1]\!], \ldots, y_n \in \mathcal{C}[\![\sigma_n]\!]) . \\ & \mathcal{A}[\![c]\!](y_1, \ldots, y_n)\end{aligned}$$

for every constant c of type σ.

- Variables:

$$\frac{\vdash K_1 : \sigma_1 \quad \ldots \quad \vdash K_n : \sigma_n}{\Gamma \vdash vK_1 \ldots K_n : \rho}\ \text{ca-var}$$

$$\begin{aligned}\mathcal{C}[\![\Gamma \vdash vK_1 \ldots K_n : \rho]\!] \;=\; & \lambda(x_1 \in \mathcal{C}[\![\gamma_1]\!], \ldots, x_m \in \mathcal{C}[\![\gamma_m]\!]) . \\ & x_i(\mathcal{C}[\![\vdash K_1 : \sigma_1]\!], \ldots, \mathcal{C}[\![\vdash K_n : \sigma_n]\!])\end{aligned}$$

provided i is the rightmost position in Γ of an occurrence of v, and $\sigma = \gamma_i$.

For a variable of ground type, $n = 0$, this specializes to the familiar variable introduction rule:

$$\frac{}{\Gamma \vdash v : \rho}\ \text{var}_0$$

$$\mathcal{C}[\![\Gamma \vdash v : \rho]\!] = \lambda(x_1 \in \mathcal{C}[\![\gamma_1]\!], \ldots, x_m \in \mathcal{C}[\![\gamma_m]\!]) . x_i$$

provided i is the rightmost position in Γ of an occurrence of v, and $\rho = \gamma_i$.

- Application:

$$\frac{\Gamma \vdash L : \sigma_0 \to \sigma \quad \Gamma \vdash L_0 : \sigma_0}{\Gamma \vdash LL_0 : \sigma}\ \text{app}$$

$$\begin{aligned}\mathcal{C}[\![\Gamma \vdash LL_0 : \sigma]\!] \;=\; & \lambda(x_1 \in \mathcal{C}[\![\gamma_1]\!], \ldots, x_m \in \mathcal{C}[\![\gamma_m]\!], y_1 \in \mathcal{C}[\![\sigma_1]\!], \ldots, y_n \in \mathcal{C}[\![\sigma_n]\!]) . \\ & \mathcal{C}[\![\Gamma \vdash L : \sigma_0 \to \sigma]\!](x_1, \ldots, x_m, f(x_1, \ldots, x_m), y_1, \ldots, y_n)\end{aligned}$$

where $\sigma_0 = \sigma_0^1 \to \cdots \to \sigma_0^{n_0} \to \rho_0$ and

$$\begin{aligned} f &= \lambda(x_1 \in \mathcal{C}[\![\gamma_1]\!], \ldots, x_m \in \mathcal{C}[\![\gamma_m]\!]) \,. \\ &\quad \lambda(z_1 \in \mathcal{C}[\![\sigma_0^1]\!], \ldots, z_{n_0} \in \mathcal{C}[\![\sigma_0^{n_0}]\!]) \,.\, \mathcal{C}[\![\Gamma \vdash L_0 : \sigma_0]\!](x_1, \ldots, x_m, z_1, \ldots, z_{n_0}). \end{aligned}$$

- **Abstraction:**

$$\frac{\Gamma, v : \sigma \vdash L : \sigma'}{\Gamma \vdash (\lambda v : \sigma \,.\, L) : \sigma \to \sigma'}\ \text{abs}$$

$$\mathcal{C}[\![\Gamma \vdash (\lambda v : \sigma \,.\, L) : \sigma \to \sigma']\!] = \mathcal{C}[\![\Gamma, v : \sigma \vdash L : \sigma']\!]$$

Proposition 5.1 *Every term has a unique type: if $\Gamma \vdash L : \sigma$ and $\Gamma \vdash L : \sigma'$ are both derivable then $\sigma = \sigma'$.*

The semantic function $\mathcal{C}[\![-]\!]$ is well-defined, and for every derivable judgement $\Gamma \vdash L : \sigma$,

$\mathcal{C}[\![\Gamma \vdash L : \sigma]\!] \in \mathcal{C}[\![\gamma_1 \to \cdots \to \gamma_m \to \sigma]\!]$,

where $\Gamma = v_1 : \gamma_1, \ldots, v_m : \gamma_m$.

5.2 Definability of isolated elements

The link between the syntactic restrictions of ca-PCF and the semantic assumptions of the sequential model is formulated as a full abstraction result.

Proposition 5.2 *For every type σ and each isolated $x \in \mathcal{C}[\![\sigma]\!]$ there exists a closed term Def_x such that $\mathcal{C}[\![\vdash \mathrm{Def}_x : \sigma]\!] = x$.*

Moreover, for every ground type ρ', $k > 0$ and each finite sequence $X = x_1, \ldots, x_k$ of isolated elements of $\mathcal{C}[\![\sigma]\!]$, if $\uparrow X$ is an index at x then there is a closed term Sel_X such that

$$\begin{aligned} &\mathcal{C}[\![\vdash \mathrm{Sel}_X : \sigma \to (\rho')^k \to \rho']\!] = \\ &\quad \lambda(z \in \mathcal{C}[\![\sigma]\!], y_1 \in \mathcal{C}[\![\rho']\!], \ldots, y_k \in \mathcal{C}[\![\rho']\!]) \,.\, (\vee^{\mathrm{p}} \{[x_i \Rightarrow y_i] \mid i \le k\}) z. \end{aligned}$$

We call such a term a selector for X.

Proof: By type induction on σ.

If σ is a ground type we have already assumed the existence of the relevant defining constants.

We can choose for $\mathrm{Sel}_{\mathtt{T},\mathtt{F}} : \mathtt{Bool} \to \rho' \to \rho' \to \rho'$ the constant $\mathtt{if}^{\rho'}$. For other selectors over Bool use the obvious variations.

For $\mathrm{Sel}_{x_1,\ldots,x_k} : \mathtt{Nat} \to (\rho')^k \to \rho'$ take

```
Sel_{x_1,...,x_k}  =  λz : Nat . λy_1 : ρ' . ... λy_k : ρ' .
                        if z = 0 then $M_0$ else
                          if z = 1 then $M_1$ else
                          ...
                            if z = k' then $M_{k'}$ else
                              Ω
```

where $k' = \max\{x_1, \ldots, x_k\}$, and for $0 \le j \le k'$, $\mathtt{z} = j$ is short for $(=0)((-1)^j\,\mathtt{z})$, and $M_j = \mathtt{y}_i$ if there exists i such that $j = x_i$, and $M_j = \Omega$ otherwise[1].

For other ground types we need to assume the existence of appropriately interpreted constants to allow a similar definition of selector terms.

[1]This term tests $\mathtt{z}$ against the values $0, \ldots, k'$ in increasing order, to avoid attempting to subtract 1 from 0.

If σ is not ground, assume that $\sigma = \sigma_1 \to \ldots \to \sigma_n \to \rho$, and let f be an isolated sequential function in $\mathcal{C}[\![\sigma]\!]$. Since f is isolated, it is the lub of a finite set of step functions. Choose a minimal set F of step functions such that $f = \bigvee F$, say $F = \{[\bar{x}_i \Rightarrow y_i] \mid i \leq l\}$, where each $\bar{x}_i = (x_i^1, \ldots, x_i^n)$. By minimality of l, none of the y_i's is $\bot$. Continue now by induction on l.

If $l = 0$ then $f = \bot$, so we can let $\mathrm{Def}_f = \Omega^\sigma = \lambda \mathbf{v}_1 : \sigma_1 \ldots . \lambda \mathbf{v}_n : \sigma_n \,.\, \Omega^\rho$.

If $l = 1$ then $f = [(x_1^1, \ldots, x_1^n) \Rightarrow y_1]$. By the induction hypothesis there are closed terms Def_{y_1} and selectors $\mathrm{Sel}_{x_1^j}$ for each j. We can take

$$\mathrm{Def}_f = \lambda \mathbf{v}_1 : \sigma_1 \ldots . \lambda \mathbf{v}_n : \sigma_n \,.\, \mathrm{Sel}_{x_1^1}\, \mathbf{v}_1 (\ldots (\mathrm{Sel}_{x_1^n}\, \mathbf{v}_n (\mathrm{Def}_{y_1})) \ldots).$$

If $l > 1$, let $s = \{\bar{x}_i \mid i \leq l\}$. Clearly, $\uparrow s = f^{-1}(\mathcal{C}[\![\rho]\!] \setminus \{\bot\})$. By minimality of F, s has no least element, or else f would be a single step function, given that $\mathcal{C}[\![\rho]\!]$ is a flat domain. Therefore $\wedge s \notin \uparrow s$. By sequentiality of f, $\uparrow s$ is sequential open, so that s is not critical. It has an index at $\wedge s$ in the product $\mathcal{C}[\![\sigma_1]\!] \times \cdots \times \mathcal{C}[\![\sigma_n]\!]$, which is derived from an index in one of the components; assume without loss of generality that it is derived from an index in the m'th component, so that there is an $r \in I(\wedge(\pi_m\, s), \pi_m\, s)$. If we take a minimal r (with respect to number of lobes) it will have at most l lobes, by minimality, but at least 2 lobes, since $\wedge s \notin r = \uparrow(\min r)$, using definiteness. Let l' be the number of lobes of r, so that $r = \uparrow\{z_j \mid j \leq l'\}$. This now lets us split F into corresponding collections of step functions, each with less than l elements, that may be distinguished on the basis of r. More formally, for $j \leq l'$, let $f_j = \bigvee F_j$, where $F_j = \{[\bar{x}_i \Rightarrow y_i] \mid i \leq l \;\&\; z_j \leq x_i^m\}$. Since each f_j is the lub of less than l step functions, it is definable, by the induction hypothesis. We are now able to define f:

$$\mathrm{Def}_f = \lambda \mathbf{v}_1 : \sigma_1 \ldots . \lambda \mathbf{v}_n : \sigma_n \,.\, \mathrm{Sel}_{z_1, \ldots, z_{l'}}\, \mathbf{v}_m (\mathrm{Def}_{f_1}\, \mathbf{v}_1 \ldots \mathbf{v}_n) \ldots (\mathrm{Def}_{f_{l'}}\, \mathbf{v}_1 \ldots \mathbf{v}_n).$$

We now show definability of $\mathrm{Sel}_{f_1, \ldots, f_k}$ in the functional case. If $\uparrow\{f_i \mid i \leq k\}$ is an index at f in $\mathcal{C}[\![\sigma]\!]$ then there must exist $\bar{x}_0 = (x_0^1, \ldots, x_0^n)$ such that $f_i = [\bar{x}_0 \Rightarrow y_i]$, and $\{y_i \mid i \leq k\}$ is an index at $f\bar{x}_0$ in $\mathcal{C}[\![\rho]\!]$. We are therefore able to transform the selection problem in the function space into a selection problem in the ground case, which has already been solved. We thus obtain:

$$\begin{aligned} \mathrm{Sel}_{f_1, \ldots, f_k} \;=\;& \lambda \mathbf{f} : \sigma \,.\, \lambda \mathbf{v}_1 : \rho' \ldots . \lambda \mathbf{v}_k : \rho' \,. \\ & \mathrm{Sel}_{y_1, \ldots, y_k} (\mathbf{f}\; \mathrm{Def}_{x_0^1}, \ldots, \mathrm{Def}_{x_0^n}) \mathbf{v}_1 \ldots \mathbf{v}_k. \end{aligned}$$

Note that $\mathbf{f}$ is applied to closed arguments, so this is a valid term. ∎

The essential difference between this definability proof and Plotkin's proof for the parallel extension of PCF [Plo77, lemma 4.5] is in the synthesis of the defining term for arrow type with $l > 1$, in the above terminology. Plotkin's proof uses the parallel conditional facility to combine a defining term for the lub of l step functions with an additional step function to obtain a defining term for the lub of $l + 1$ step functions; we rely instead on the existence of an index that partitions the set of step functions into smaller sets.

Full abstraction — both inequational and equational — follows by standard arguments from the definability of all isolated elements [Mil77, Sto88].

Proposition 5.3 *The semantics $\mathcal{C}[\![-]\!]$ is inequationally fully abstract with respect to itself as a notion of program behavior. That is, for any pair of derivable judgements $\Gamma \vdash L : \sigma$ and $\Gamma \vdash L' : \sigma$,*

$$\mathcal{C}[\![\Gamma \vdash L : \sigma]\!] \leq \mathcal{C}[\![\Gamma \vdash L' : \sigma]\!]$$

iff, for every appropriate[2] program context $P[-]$ of type ρ,

$$\mathcal{C}[\![\vdash P[L] : \rho]\!] \leq \mathcal{C}[\![\vdash P[L'] : \rho]\!].$$

[2]Since we do not associate fixed types with variables we must assign to holes in program contexts a type context Γ which they provide, as well as the type of the term that they expect in the hole.

To link up this result with the standard notion of behavior for PCF programs, we verify that $\mathcal{C}[\![-]\!]$ agrees with the usual operational semantics for PCF, as presented in [Plo77].

Proposition 5.4 *The program behaviors induced by the semantics $\mathcal{C}[\![-]\!]$ and the operational semantics coincide. That is, for every closed term P of ground type ρ, $\mathcal{C}[\![\vdash P : \rho]\!] = x \neq \bot$ iff P evaluates to (the constant denoting) x, and $\mathcal{C}[\![\vdash P : \rho]\!] = \bot$ iff the evaluation of P diverges.*

In summary, the semantics $\mathcal{C}[\![-]\!]$ is fully abstract for ca-PCF with respect to the usual notion of program behavior.

5.3 Recursive definitions

Since the fixpoint operator is continuous but not sequential in our framework, we cannot simply add the usual fixpoint constants $\mathtt{Y}$ to the language ca-PCF. Nevertheless, any particular sequential function on an arithmetic domain D has a least fixed point in D. We may therefore add μ-abstraction to ca-PCF: for each ca-PCF term M of type $\tau \to \tau$ the term $\mu\mathtt{f} : \sigma\,.\,M\mathtt{f}$ of type τ is equivalent to $\mathtt{Y}M$. To permit non-trivial uses of recursion, such as

$$(\times 2) = \mu\mathtt{f} : \mathtt{Nat} \to \mathtt{Nat}\,.\,\lambda\mathtt{x} : \mathtt{Nat}\,.\,\mathtt{if}\,\mathtt{x} = 0\,\mathtt{then}\,0\,\mathtt{else}(\mathtt{f}(\mathtt{x}-1)+2),$$

in which the recursively defined variable $\mathtt{f}$ has an input-dependent prompter $\mathtt{x} - 1$, we then need to relax the term-forming syntactic constraints of ca-PCF to allow μ-bound variables to be applied to input arguments inside the body M. The meaning of every term is in the right semantic domain when supplied with appropriate values for its free μ-bound variables.

6 Conclusion

We have introduced a notion of indexed domain and shown that it permits a general definition of sequential function enjoying certain domain-theoretic properties. In particular, we obtain a class of indexed domains containing the flat domains, closed under product, and closed under the pointwise-ordered sequential function space. We have shown that a particular kind of index structure on function spaces gives rise to a fully abstract semantics for a non-trivial sub-language of PCF. Nevertheless, unrestricted application is not a sequential function in our model, and it remains to be seen if we can find a yet more sophisticated notion of index structure that would cope satisfactorily with full PCF. This would have to deal with the complications caused by imbrication and what we have called input- or self-dependent prompters. The generalized indices should, like the indices presented here, have a firm operational grounding, and they should carry information that can be used for showing definability of the sequential functions in the generalized framework.

There are interesting connections and significant differences with the work of Bucciarelli and Ehrhard [BE91]. The critical sets of an indexed domain always form a coherence structure in the sense of Bucciarelli and Ehrhard (and the sequential functions in our model correspond to their strongly stable functions). The converse is not true, because our requirements on index structures are stronger, so as to build in the ability to model incremental computation. Bucciarelli and Ehrhard also use data sequentiality at ground types, and essentially the same product. They obtained a cartesian closed category of strongly stable functions between qualitative domains equipped with coherence structure, using the stable ordering on function spaces; in particular, in their model application is sequential with respect to the *stable* ordering. However, the coherence structures that they use on function types do not correspond to index structures, and apparently do not convey enough operational information to model incremental sequential computation. Moreover, the *pointwise* ordering is of primary relevance for the PCF full abstraction problem, since it corresponds to the operational pre-order on terms of function type, and therefore we are more concerned to find a notion of sequential function space using the pointwise order.

References

[BC82] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.

[BC85] G. Berry and P.-L. Curien. Theory and practice of sequential algorithms: the kernel of the applicative language CDS0. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 2, pages 35–87. Cambridge University Press, 1985.

[BCL85] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 3, pages 89–132. Cambridge University Press, 1985.

[BE91] A. Bucciarelli and T. Ehrhard. Sequentiality and strong stability. In *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, July 1991.

[Ber78] G. Berry. Stable models of typed λ-calculi. In *Proc. 5^{th} Coll. on Automata, Languages and Programming*, number 62 in Lecture Notes in Computer Science, pages 72–89. Springer-Verlag, July 1978.

[BG92] S. Brookes and S. Geva. Stable and sequential functions on Scott domains. Technical Report CMU-CS-92-121, School of Computer Science, Carnegie Mellon University, June 1992.

[CF92] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 328–342. ACM Press, January 1992.

[Cur86] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science. Pitman, 1986. Second edition, expanded and updated, published by Birkhäuser, Boston, 1993.

[Cur92] P.-L. Curien. Observable algorithms on concrete data structures. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 432–443. IEEE Computer Society Press, June 1992.

[GHK+80] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices*. Springer Verlag, 1980.

[KP78] G. Kahn and G. D. Plotkin. Domaines concrets. Rapport 336, IRIA-LABORIA, 1978. English translation (with historical introduction by S. Brookes) to appear in *Theoretical Computer Science*, 1993.

[Mil77] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mul87] K. Mulmuley. *Full Abstraction and Semantic Equivalence*. MIT Press, 1987.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[Saz75] V. Yu. Sazonov. Sequentially and parallelly computable functionals. In *Proc. Symp. on Lambda-Calculus and Computer Science Theory*, number 37 in Lecture Notes in Computer Science. Springer-Verlag, 1975.

[Sto88] A. Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman, 1988.

[Vui73] J. Vuillemin. *Proof techniques for recursive programs*. PhD thesis, Stanford University, 1973.

[Zha91] G. Q. Zhang. *Logic of Domains*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1991.

Another approach to sequentiality: Kleene's unimonotone functions

Antonio Bucciarelli
LIENS-DMI, Ecole Normale Supérieure, 45 rue d'Ulm, Paris, France
buccia@dmi.ens.fr

Abstract

We show that Kleene's theory of unimonotone functions strictly relates to the theory of sequentiality originated by the full abstraction problem for PCF. Unimonotone functions are defined via a class of oracles, which turn out to be alternative descriptions of a subclass of Berry-Curien's sequential algorithms.

1 Introduction

In the late seventies, in order to define models of (simply typed) functional programming languages which were closer than Scott models to the operational semantics of such languages, the notions of *sequentiality* ([6, 4]) and *stability* ([1]) were introduced and studied. These works originated from the problem of *full abstraction*, raised in [12], which can be formulated as follows: find a model of PCF (a simply typed λ-calculus with recursion, taken in this framework as paradigm of functional languages) in which any (finite) element is the denotation of some term. Clearly this (still open) problem is strictly related to the characterization of the expressivity of PCF at higer types.

Quite in the same period S. C. Kleene, revisiting some of his earlier works on generalized recursion theory, attacked the problem of "generating a class of functions which shall coincide with all the partial functions which are "computable" or "effectively decidable", so that Church's 1936 thesis will apply with the higer types included" ([8] 1.2). His starting point was the definition of a list of schemata for the definition of partial recursive functionals. Computations in this framework are represented by trees: an expression E is defined (under a given assignment of values to free variables) if and only if the *principal branch* of the computation tree rooted in E ends with a value (a natural number).

This operational semantics allowed one to recover at higher types most of the basic results of classical recursion theory (enumeration theorem, substitution principle, recursion theorems) (see [8, 7]). The next point of Kleene's program ([9, 10]) consisted of providing a denotational semantics for his "type-j objects", i. e. in characterizing the class of functions and functionals definable by the schemata.

Thus Kleene and people interested in the problem of full abstraction for PCF began to work, independently, on two very related subjects: the definability problem for calculi based on recursive equations and λ-calculus.

People working on PCF aimed to define models of the full calculus by means of cartesian closed categories of domains and "sequential" morphisms, whereas Kleene focused his attention on finite types up to type three in a hierarchy starting from natural numbers (type 0).

As for the first approach, the notion of sequential function between concrete data structures ([6]) provided a complete characterization of PCF-definability at first order, but failed to give rise to cartesian closed categories. This notion suggested two main developments. The first one consisted in weakening sequentiality in a property extendable to higher orders, and this is what G. Berry [1] did introducing stable semantics. The second one, carried out by Berry and P.- L. Curien (see [2]) was to stick firmly to the notion of sequentiality, but the price to pay was the impossibility of keeping functions as morphisms; they were obliged to switch to sequential algorithms.

Kleene's approach to the problem of definability is based on the notion of *unimonotone* functions. The interesting fact is that the two clauses of the definition of unimonotonicity correspond to stability and sequentiality respectively.[1]

We establish the bridge between the theories of sequential algorithms and unimonotone functions. We want to stress that these theories are based on the same idea of computability at higher types, by showing that any unimonotone function is computed by some sequential algorithm. The converse does not hold, essentially because in order to get cartesian closedness, it is necessary to take into account sequential algorithms which do not compute any (monotone) function.

This is actually the main difference between the two approaches to the definability problem: in the Berry-Curien's model algorithms are first class objects, whereas Kleene uses algorithms (that he calls *oracles*) only to define the notion of unimonotonicity. The objects of his model are functions, obtained by an *extensional quotient* on oracles.

The same kind of quotient is used by Curien in [5, 4] to build the model of *extensional algorithms.* Hence Kleene's construction may be regarded ("a posteriori") as an attempt of collapsing in a single step the Berry-Curien's construction:

sequential functions $\rightarrow$ sequential algorithms $\rightarrow$ extensional algorithms

In this paper we do not tell the whole story, contenting ourselves with studying the relation between oracles and sequential algorithms. We show that oracles may be simulated by algorithms at any type and we give a simple full abstraction result of unimonotone functions at type 2.

[1]A puzzling point is that unimonotone functions are not required to be Scott-continuous.

In example 9 we show how the nesting of function calls in PCF (and in Kleene's schemata) makes the use of intensional descriptions of computations necessary to approach the problem of higher-order definability.

2 Concrete Data Structures (CDSs) and sequential algorithms

We first recall the basic definitions of CDSs as they can be found in [4].

Definition 1 *A CDS* $M = (C_M, V_M, E_M, \vdash_M)$ *is given by three sets* C_M*,* V_M *and* E_M *of cells, values and events such that*

$$E_M \subset C_M \times V_M \quad \text{and} \quad \forall c \in C_M \; \exists v \in V_M \quad (c, v) \in E_M$$

and a relation $\vdash_M$*, called an accessibility relation between finite parts of* E_M *and elements of* C_M*. A set* $\{e_1, \ldots, e_n\}$ *is an enabling of* c *if* $\{e_1, \ldots, e_n\} \vdash_M c$*.* C_M *and* V_M *are assumed countable.*

We will omitt the subscript M whenever possible. Given a CDS M, a *state* of M is subset x of E_M which is conflict-free (any cell is filled with at most one value) and safe (any filled cell is enabled):

Definition 2 *A state of* M *is a subset* x *of* E_M *such that*

1) $(c, v_1), (c, v_2) \in x \Rightarrow v_1 = v_2$

2) If $(c, v) \in x$*, then there exists a sequence of events* $e_0, \ldots, e_n = (c, v)$ *such that* $e_i = (c_i, v_i) \in x$ *and* $\{e_j \mid j < i\}$ *contains an enabling of* c_i *for all* $i \leq n$*.*

The set of states of a CDS M *ordered by inclusion is a partially ordered set denoted by* $D(M)$*.*

We define now the CDS's whose associated domains are (isomorphic to) the two points Siepinsky space ($\perp < \top$) and the flat domain of boolean values respectively.

example 1:

$$O = (\{*\}, \{T\}, \{(*, T)\}, \emptyset \vdash *)$$

$$D(O) = \begin{tikzcd} \{(*, T)\} \arrow[d, dash] \\ \emptyset \end{tikzcd}$$

■

example 2:

$$B = (\{*\}, \{true, false\}, \{(*, true), (*, false)\}, \emptyset \vdash *)$$

$$D(B) = \begin{tikzcd}
\{(*, true)\} & & \{(*, false)\} \\
& \emptyset \arrow[ul, no head] \arrow[ur, no head] &
\end{tikzcd}$$

■

Last we give useful notations:

Definition 3 *Let $x \in D(M)$ for a CDS M. A cell c is*

- *filled in x iff $\exists v \; (c, v) \in x$ ($F(x)$ will denote the set of filled cells)*
- *enabled in x iff x contains an enabling of c ($E(x)$ will denote the set of enabled cells)*
- *accessible from x iff it is enabled but not filled in x. ($A(x)$ will denote the set of accessible cells)*

In both the examples above the cell $*$ is accessible from the empty state. Cells having this property are called *initials*.

Definition 4 *If M, M' are CDSs, the CDS of* sequential algorithms *from M to M' is noted $[M \to M']$ and defined by*

- $$C_{[M \to M']} = D(M)_0 \times C_{M'}$$
  where $D(M)_0$ denotes the finite states of M.

- $$V_{[M \to M']} = \{valof\ c \mid c \in C_M\} \bigcup \{output\ v' \mid v' \in V_{M'}\}$$

- $$E_{[M \to M']} = \{((x, c'), valof\ c) \mid c \in A(x)\} \bigcup \{((x, c'), output\ v') \mid (c', v') \in E_{M'}\}$$

- $$((x, c'), valof\ c) \vdash (y, c') \;\; if \;\; \exists v \in V_M \;\; y = x \cup (c, v)$$
  $$((x_1, c_1'), output\ v_1'), ((x_2, c_2'), output\ v_2'), \ldots, ((x_n, c_n'), output\ v_n') \vdash (x, c')$$
  $$if\ x = \bigcup_{1 \le i \le n} x_i \ and\ (c_1', v_1'), \ldots, (c_n', v_n') \vdash c'$$

Given a sequential algorithm $a \in [M \to M']$, the function $f^a : D(M) \to D(M')$ associated to a is defined by $f^a(x) = \{(c', v') \mid \exists y \le x\ ((y, c'), \mathit{output}\ v') \in a\}$ (actually f^a is well defined only if M, M' are *stable* CDS's (see [4]); this will be always the case in what follows).

3 Unimonotone functions and Oracles

Let us begin by defining the *types* we are interested in: type 0 is the flat domain of natural numbers, and, for $i \le 2$, type $i+1$ is the set of *unimonotone* partial functions from type i to type 0, ordered extensionally (i.e. $f \le g$ if for all x $f(x) = n \to g(x) = n$). Unimonotone means MONOTONE with a UNique and Intrinsically determined basis.

Kleene gives the definition of unimonotone function in two steps: the first one (uniqueness of the basis) consists in requiring that, if $f(x)$ is defined, and hence $f(x) = n$ for some $n \in \omega$, then there exists $x' \le x$ such that $f(x') = n$ and for any $x'' \le x$, if $f(x'') = n$ then $x' \le x''$. Such x' is called the basis for x with respect to f. The Scott-continuous functions f of type i such that for any $x \in i-1$, if $f(x)$ is defined then there exists a basis for x with respect to f are exactly the stable functions. However in this framework Scott continuity is not required. The second step of the definition consists in requiring that the basis for x with respect to f, when x ranges in the domain of f (i.e. in type $i-1$ if f is of type i), be *intrinsically determined.* Actually a big part of Kleene's work is devoted to explaining what "intrinsic determination of bases" means, by the definition of a class of oracles which compute unimonotone functions. Our aim is to show that these oracles are a particular kind of sequential algorithms, and hence that unimonotone functions are sequential in the sense of Kahn-Plotkin [2].

Since any sequential function is stable, the first clause of the definition of unimonotone function (uniqueness of bases) is subsumed by the second one (intrinsical determination of bases). In what follows we focus on this second requirement: a function is unimonotone if it is computed by an oracle of the form that we are going to define.

From now on let f^i and O^i range over the classes of type i unimonotone functions and type i oracles respectively. An oracle O^i is described by her (we follow Kleene's indications about sex of oracles) behaviour when presented with an envelope containing a $i-1$ oracle (a type 0 oracle being simply an element of type 0).

[2]Actually, since sequential functions are continuous, only continuous and unimonotone functions are sequential.

3.1 Type 1 oracles

Type 1 oracles compute unimonotone functions from type 0 to type 0. When presented with an envelope containing an element of type 0 (the envelope being empty if this element is undefined), a type 1 oracle O^1 behaves in one of the following ways:

case 1.1 She does nothing. In this case O^1 computes the completely undefined function $\lambda x.\bot$. Such an oracle will be called the empty type 1 oracle and noted u^1.

case 1.2 Without opening the envelope, she prounounces that the result of the computation is the integer n. In this case O^1 computes the nonstrict constant function $\lambda x.n$.

case 1.3 She opens the envelope, so declaring that the function she computes is strict. If the envelope is empty (i.e. if we have presented her with the bottom element of type 0), she stands mute, if it contains an integer n, she may either stand mute or give an integer m as result, depending on n. In this case O^1 computes the function the graph of which is $\{(n_1, m_1), \ldots, (n_k, m_k), \ldots\}$, a pair (n_i, m_i) belonging to this graph if and only if O^1 gives m_i as result when presented with an envelope containing n_i.

Functions computed by type 1 oracles are clearly monotone, and it is easy to see that any monotone function from type 0 to type 0 is computed by some oracle. Actually any function other than $\lambda x.\bot$ is computed by exactly one oracle. The function $\lambda x.\bot$ is computed by u_1 (case 1.1) and by the oracle (operating under case 1.3) which opens envelopes but never gives a result.

3.2 Type 2 oracles

Type 2 oracles compute unimonotone functions from type 1 to type 0. When presented with an envelope containing a type 1 oracle O^1 which computes f^1, a type 2 oracle O^2 (computing f^2) behaves in one of the following ways:

case 2.1 She does nothing. In this case O^2 computes the completely undefined function $f^2 = \lambda f^1.\bot$. Such an oracle will be called the empty type 2 oracle and noted u^2.

case 2.2 Without opening the envelope, she pronounces that the result of the computation is the integer n. In this case O^2 computes the nonstrict constant function $f^2 = \lambda f^1.n$.

case 2.3 She opens the envelope, so revealing that she wants information about O^1 before deciding whether to give a result. To obtain such information, she begins to question O^1 by passing her an empty envelope ("there is nothing lost by our supposing that the α^2- oracle starts with the preliminary question "$\alpha^1(\alpha^0)$?" using an empty envelope"[9]. Actually, as we shall see, this is the only possible choice in the CDS framework). According to the behaviour of O^1, three cases are possible:

case 2.3.1 O^1 stands mute (she operates under case 1.1). In this case O^2 stands mute too (any type 2 oracle operating under case 2.3 defines a strict functional).

case 2.3.2 O^1 gives the result n without opening the envelope (she operates under case 1.2). Observing this, O^2 may either stand mute or give an integer m as result, depending on n. In any case O^2 cannot continue to question O^1, since she knows everything about f^1 (namely that $f^1 = \lambda x.n$).

case 2.3.3 O^1 opens the envelope (she operates under case 1.3). Observing this O^2 may either stand mute, or pose a first non-preliminary question $r_0 \in N$. Questioned with r_0, O^1 will either stand mute or give an integer result n_0. In the former case O^2 stands mute, in the latter she can either stand mute, thus deciding that the information so far recorded about O^1 (namely that O^1 opens envelopes and that she gives n_0 as result when presented with r_0) is sufficient to rule out that $f^2(f^1)$ be defined, or give a result $m \in N$ (declaring that $f^2(f^1) = m$), or query O^1 with another integer r_1. In general in this subcase 2.3.3, a series of questions (possibly extending into the transfinite) will be asked to O^1 by O^2 with distinct numbers

$$r_0, r_1, \ldots, r_k, \ldots$$

and will be answered by O^1 with numbers

$$n_0, n_1, \ldots, n_k, \ldots$$

where $n_i = f^1(r_i)$ and r_i is determined by O^2 from only the information that O^1 opens envelopes and $f^1(r_j) = n_j$ for $j < i$. This continues until either, for a given ordinal τ, O^1 does not answer to r_τ, which makes $f^2(f^1)$ undefined, or O^2 decides that the information so far collected about O^1 makes $f^2(f^1)$ undefined, or finally (only after at least one non-preliminary question has been posed) that it is sufficient to give m as result ($f^2(f^1) = m$).

We have to prove that any type 2 oracle computes a monotone function of type 2. Firstly we have to show that, when we apply an oracle O^2 to an oracle O^1 computing f^1 (i.e. when we present O^2 with an envelope containing O^1), the result of the computation (if any) depends only on f^1. This is trivially the case if $f^1 \neq \lambda x.\bot$, since in this case, as remarked in the section devoted to type 1 oracles, there is a unique O^1 which computes f^1. If $f^1 = \lambda x.\bot$, it is sufficient to remark that, if O^2 gives a result, she must operate under case 2.2, and hence the result does not depend at all on O^1.

Next we have to prove that the function computed by O^2 is monotone. Let f^1, g^1 be unimonotone type 1 functions computed by O^1 and P^1 respectively, and such that $f^1 \leq g^1$ (i.e. such that for any x of type 0, $f^1(x) = m \rightarrow g^1(x) = m$). We have to show that if O^2 gives a result when applied to O^1 than she gives the same result when applied to P^1. This holds trivially if O^2 operates under cases 2.1 or 2.2. Suppose that O^2 operates under case 2.3: if both O^1 and P^1 come under case 1.2, then $f^1 = g^1$; if both O^1 and P^1 come under case 1.3, then O^2 will ask O^1 the same questions $r_0, \ldots, r_k, \ldots$ and receive the same answers $n_0, \ldots, n_k, \ldots$ from O^1 as from P^1, and hence will give the same result. Moreover it is impossible that O^1 comes under case 1.2 and P^1 under case 1.3 (since $f^1 \leq g^1$) and that either O^1 or P^1 come under case 1.1, since we are supposing that O^2 gives a result when applied to O^1. The only case left is that O^1 comes under case 1.3 and P^1 under case 1.2, that is g^1 is the non-strict constant $\lambda x.m$ for some $m \in N$ and f^1 has graph $\{(n_1, m), (n_2, m), \ldots, (n_k, m), \ldots\}$ where $\{n_1, n_2, \ldots, n_k, \ldots\} \subseteq N$. Actually in this case O^2 could give value n on O^1 and value $l \neq n$ (or no value at all) on P^1, violating monotonicity. In this case the monotonicity is assured by stipulating that a type 2 oracle which gives result n in subcase 2.3.3 on the basis of knowing that $f^1(r_k) = m$ for all k less then a given ordinal τ, must give the same result in case 2.3.2 on the oracle computing the non-strict constant $\lambda x.m$. This assumption makes the function computed by a type 2 oracle monotone.

For any type 2 function f^2 other than a non-strict constant $f^2 = \lambda f^1.m$, there exist infinitely many type 2 oracles computing it. This is essentially due to the fact that when operating under case 2.3, an oracle O^2 may query arbitrary sequences of unuseful questions, a sequence being unuseful if, no matter what is answered by O^1, O^2 never gives a result.

In the following examples (borrowed from [9]) we show two functions from type 1 to type 0 which can not be computed by type 2 oracle. The first one does not respect the "unicity of bases" requirement, the second one has unique bases, but they are not intrinsically determined. The interesting fact is that these functions are the reformulation in the framework of Kleene's types of the parallel-or function and of Berry's example of a stable and non-sequential function [1].

example 3: Let f_i^1, $i = 1, 2$, be the type 1 functions defined by the following graphs:

$$f_1^1 = \{(0,0)\} \qquad f_2^1 = \{(1,1)\}$$

and let f^2 be such that $f^2(f^1) = 0$ if there exists $i \leq 2$ such that $f_i^1 \leq f^1$, and be undefined otherwise. The function $g^1 = \{(0,0),(1,1)\}$ has no basis with respect to f^2. An oracle for f^2 should operate under case 2.3, and, in subcase

2.3.3, she should pick up a $r_0 \in N$ such that, for $i \leq 2$, $f_i^1(r_0)$ be defined, but such a r_0 does not exist. ■

example 4: Let f_i^1, $i = 1, 2, 3$ be the type 1 functions defined by the following graphs:

$$\begin{aligned} f_1^1 &= \{ \qquad (1,0),(2,1)\} \\ f_2^1 &= \{(0,1), \qquad (2,0)\} \\ f_3^1 &= \{(0,0),(1,1) \qquad \} \end{aligned}$$

and let f^2 be such that $f^2(f^1) = 0$ if there exists $i \leq 3$ such that $f_i^1 \leq f^1$, and be undefined otherwise. An oracle for f^2 should operate under case 2.3, and, in subcase 2.3.3, she should pick up a $r_0 \in N$ such that, for $i \leq 3$, $f_i^1(r_0)$ be defined, but such a r_0 does not exist. ■

3.3 Tree-representation of type 2 oracles

Type 2 oracles may be represented by trees, as in [10]. A non-trivial oracle O^2 (i.e. an oracle operating under case 2.3), is represented by a tree of the following kind:

- Non-leaf nodes represent the queries of O^2 (the root containing the preliminary empty query).
- Arcs are labelled by answers provided by the argument oracle O^1.
- Leaves represent the result given by O^2 when presented with the O^1 (partially) described by the corresponding branch.

Such a tree describes only "useful" computations of O^2, i.e. computations at the end of which O^2 gives a result.

example 5: The following (linear) tree represents a type 2 oracle computing the (non-Scott-continuous) functional f^2 defined as follows:

$$f^2(f^1) = 0 \text{ if and only if } f^1 \text{ is the identity function on } N$$

$$\begin{tikzcd} \emptyset? \arrow[r, dash, "O^1 \text{ opens}"'] & 0? \arrow[r, dash, "0"'] & 1? \arrow[r, dash, "1"'] & 2? & \cdots & n? \arrow[r, dash, "n"'] & n+1? & \cdots & 0 \end{tikzcd}$$

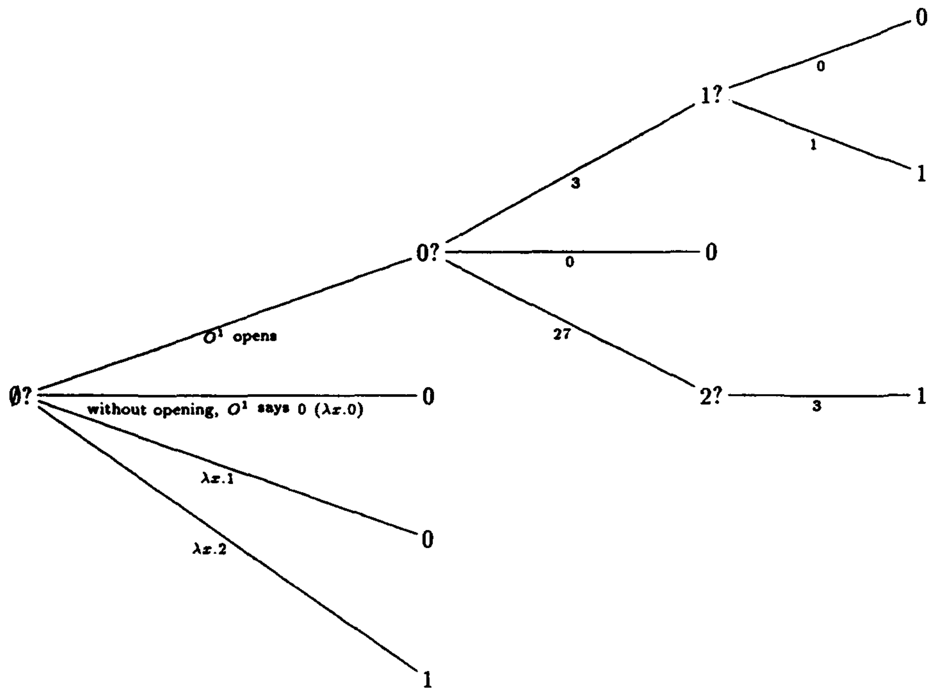
■

The functional f^2 of the example above is not Scott-continuous since the oracle computing it gets an infinite amount of information about her argument before giving a result; the following easy result relates continuity to finiteness of trees:

Proposition 1 *A unimonotone function f^2 computed by O^2 is Scott-continuous if and only if the tree representing O^2 has no infinite branch.*

example 6: The following tree represents an oracle computing the functional f^2 defined as follows:

$$f^2(f^1) = \begin{cases} 0 & \text{if } f^1 = \lambda x.1 \text{ or } \{(0,3),(1,0)\} \subseteq f^1 \text{ or } \{(0,0)\} \subseteq f^1 \\ 1 & \text{if } f^1 = \lambda x.2 \text{ or } \{(0,3),(1,1)\} \subseteq f^1 \text{ or } \{(0,27),(2,3)\} \subseteq f^1 \\ \text{undefined} & \text{otherwise} \end{cases}$$



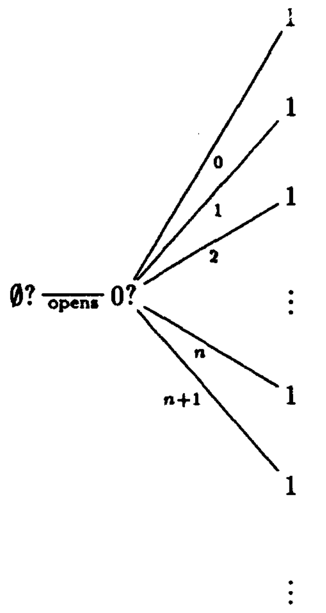
■

The extensionality constraint which assures the monotonicity of functions computed by type 2 oracles imposes a global condition on trees. In the example above, for instance, if we remove the branch labelled "without opening, O^1 says 0" we lose monotonicity (because of the branch labelled "O^1 opens" and "0"). Note that the condition $f^2(\lambda x.0) = 0$ is not explicitly stated in the definition of f^2, since it is subsumed by $f^2(\{(0,0)\}) = 0$, by monotonicity of f^2.

In the last example we show how branchings can be infinite:

example 7: The following tree represent an oracle computing the functional f^2 defined as follows:

$$f^2(f^1) = 1 \text{ if and only if } f^1(0) \text{ is defined}$$



$\blacksquare$

An interesting remark about type 2 oracles, in view of the comparison with sequential algorithms, is that the tree

$$\emptyset? \underset{O^1 \text{ opens}}{\overline{\hspace{8em}}} 0$$

which would represent an oracle that gives result 0 on the basis of the fact that his argument O^1 opens envelopes (i.e. that O^1 computes a strict function) is not allowed, since in case 2.3.3, if O^2 gives a result, she must do so after having asked at least one non-preliminary question.

4 Concrete data structures for Kleene's finite types

We define the concrete data structure we need to compare unimonotone functions and sequential algorithms: N^0 is the cds of natural number, and, for $j = 0, 1, 2$, N^{j+1} is the cds of sequential algorithms from N^j to N^0 (actually N^3 will not be trated in this section, see the last section).

$$N^0 = (\{*\}, \omega, \{(*, n) \mid n \in \omega\}, \{\vdash *\})$$

A state of N^0 is either the empty set or the singleton $\{(*, n)\}$ for some $n \in \omega$, simply noted by n.

The cells of the cds N^1 are elements of the cartesian product $D(N^0) \times C_{N^0}$, i.e.

$$C_{N^1} = \{(\emptyset, *)\} \bigcup \{(n, *) \mid n \in \omega\}$$

The values of N^1 are defined by

$$V_{N^1} = \{valof\ *\} \bigcup \{output\ n \mid n \in \omega\}$$

and its events by

$$E_{N^1} = \{(init_1, valof\ *)\} \bigcup \{(init_1, output\ n) \mid n \in \omega\} \bigcup \{((n,*), output\ m) \mid n, m \in \omega\}$$

The cell $(\emptyset, *)$ is initial (and we call it $init_1$), and any other cell is enabled by the event $(init_1, valof\ *)$, the enabling being of type "valof" (see definition 4). We remark that in the cds's N^j, j=1,2,3, enablings of type "output" cannot occur, since the unique cell of the target cds N^0 is initial. Hence nonempty and finite states of N_1 are either of the form

$$\{(init_1, output\ n)\}\ n \in \omega$$

(the function computed by this algorithm being the nonstrict constant $\lambda m.n$) or of the form

$$\{(init_1, valof\ *)((n_1,*), output\ m_1), \ldots, ((n_k.*), output\ m_k)\}$$

for $k \geq 0$, and for $1 \leq i < j \leq k\ n_i \neq n_j$ (the function computed by this algorithm being $\{(n_1, m_1), \ldots, (n_k, m_k)\}$).

In the latter case and for $k = 0$, we get the "purely intensional" algorithm $\{(init_1, valof\ *)\}$, which is extensionally equivalent to the empty algorithm , but plays a major role in the definition of type N^2. For making more readable the treatment of higher types, let us introduce some abbreviation for the finite states of N^1:
$\{(init_1, output\ n)\}$ will be noted $\lambda i.n$
$\{(init_1, valof\ *)((n_1,*), output\ m_1), \ldots, ((n_k,*), output\ m_k)\}$ for $k \geq 0$, will be noted $\{(n_1, m_1), \ldots, (n_k, m_k)\}$ (note that the algorithm $\{(n_1, m_1), \ldots, (n_k, m_k)\}$ for $k = 0$ ***is not*** the empty algorithm, since it contains the event $(init_1, valof\ *)$.

Let us now pass to N^2: its cells are obtained by coupling finite elements of N^1 with the unique cell of N^0:

$$C_{N^2} = \{(\emptyset, *)\} \bigcup \{(\lambda i.n, *)\} \bigcup \{(\{(n_1, m_1), \ldots, (n_k, m_k)\}, *)\}$$

for $n, k, n_i, m_i \in \omega$, $k \geq 0$ and $1 \leq i < j \leq k\ n_i \neq n_j$. As for N^1, the unique initial cell of N^2 is $(\emptyset, *)$, which we call $init_2$.

The values of N^2 are defined by:

$$V_{N^2} = \{valof\ c \mid c \in C_{N^1}\} \bigcup \{output\ n \mid n \in \omega\} =$$

$$= \{valof\ init_1\} \bigcup \{valof\ (n,*) \mid n \in \omega\} \bigcup \{output\ n \mid n \in \omega\}$$

Let us see which "valof events" are legal in N^2 (a "valof event" is one of the form $(c, \mathit{valof}\ c')$). Recall that in a functional cds an event $(xc', \mathit{valof}\ c)$ is legal if and only if $c \in A(x)$, hence the valof events in N^2 are the following:

$$\{(\mathit{init}_2, \mathit{valof}\ \mathit{init}_1)\} \bigcup \{((\{(n_1, m_1), \ldots, (n_k, m_k)\}, *), \mathit{valof}\,(n, *)) \mid \forall i \leq k\ n_i \neq n\}$$

On the other hand any couple (c, v) where $c \in C_{N^2}$ and v is an output value of N^2 (i.e. $v = \mathit{output}\ n$) is an output event of N^2 (this is always the case when the target cds is flat). Let us describe now the enabling relation in N^2: as already remarked init_2 is the unique initial cell; moreover

- $(\mathit{init}_2, \mathit{valof}\ \mathit{init}_1) \vdash (\lambda i.n, *)\ n \in \omega$
- $(\mathit{init}_2, \mathit{valof}\ \mathit{init}_1) \vdash ((\mathit{init}_1, \mathit{valof}\ *), *)$
- $((\{(n_1, m_1), \ldots, (n_k, m_k)\}, *), \mathit{valof}\,(n, *)) \vdash (\{(n_1, m_1), \ldots, (n_k, m_k), (n, m)\}, *)$ $m \in \omega$

We can describe a state A of N^2 by means of the step-by-step process which, starting from the empty state, leads to the construction of A. At each stage of this process cells enabled at the previous stage may be filled, and if they are filled by *valof* values, new cells are enabled, as described above. At the initial stage 0, the unique enabled cell is init_2.

stage 0 init_2 is enabled. It can be filled either by an output value *output* n (in this case no more cells are enabled and the functional defined by A is the constant $\lambda f.n$) or it is filled by *valof* init_1. In this latter case infinitely many cells are enabled, namely those of the form $(\lambda i.n, *)$ $n \in \omega$ and the cell $((\mathit{init}_1, \mathit{valof}\ *), *)$.

stage 1 Any cell $(\lambda i.n, *)$ may be filled by an output value *output* m, meaning that the functional defined by A gives m as result when applied to the nonstrict constant n. The cell $((\mathit{init}_1, \mathit{valof}\ *), *)$ may be filled either by an output value, meaning that the strictness of its argument is sufficient for A to give a result, or by a value *valof* $(n_0, *)$. In this latter case infinitely many cell are enabled, namely those of the form $(\{(n_0, m)\}, *)$ $m \in \omega$.

stage $i + 1$ $i > 0$ At stage i a (possibly empty) set of cells of the form

$$(\{(n_0, m_0), (n_1, m_1), \ldots, (n_{i-1}, m_{i-1})\}, *)$$

have been enabled. Any of these cells may be filled either by an *output* value, producing the event

$$((\{(n_0, m_0), (n_1, m_1), \ldots, (n_{i-1}, m_{i-1}\}, *), \mathit{output}\ k)$$

or by a value *valof* n_i such that $n_i \neq n_j$ for $j < i$, producing the event

$$((\{(n_0, m_0), (n_1, m_1), \ldots, (n_{i-1}, m_{i-1}\}, *), \mathit{valof}\ (n_i, *))$$

In this latter case the cells

$$(\{(n_0, m_0), \ldots, (n_{i-1}, m_{i-1}, (n_i, m)\}, *)$$

for $m \in \omega$ are enabled.

It is easy to see that

$$A = \bigcup_{i \in \omega}\{\text{ the events produced at stage } i\}$$

is a state of N^2, and that any state of N^2 may be constructed in this way.

5 From oracles to sequential algorithms

In this section we show that any (Scott-continuous and) unimonotone function is computed by some sequential algorithm. Since unimonotone functions are defined via oracles, it is sufficient to provide, for any given oracle, a sequential algorithm which simulates it. We begin by treating the rather simple case of type 1 oracles, proceeding by cases according to the definition of type 1 oracle given in section 3.1. For a given oracle O^1 we define a sequential algorithm $A^1 = Alg(O^1)$ which defines the same function as O^1 (we take for granted the obvious isomorphism between type 0 and N^0).

case 1.1 $O^1 = u^1$. In this case $Alg(O^1)$ is the empty algorithm.

case 1.2 O^1 is the oracle that, without opening envelopes, answers n. In this case $Alg(O^1) = \{(\mathit{init}_1, \mathit{output}\ n)\}$.

case 1.3 O^1 is the envelope-opening oracle that gives values $m_1, m_2, \ldots, m_k, \ldots$ on arguments $n_1, n_2, \ldots, n_k, \ldots$. In this case

$$Alg(O^1) = \{(\mathit{init}_1, \mathit{valof}\ *), ((n_1, *), \mathit{output}\ m_1), \ldots, ((n_k, *), \mathit{output}\ m_k), \ldots\}$$

It is clearly the case that O^1 and $Alg(O^1)$ define the same type 1 function. Actually *Alg* defines a bijection between type 1 oracles and algorithms.

We pass now to type 2: we should define, for a given type 2 oracle computing f^2, a sequential algorithm $Alg(O^2)$ such that, for any given type 1 oracle O^1 computing f^1, $Alg(O^2)(Alg(O^1)) = f^2(f^1))$. This cannot be done in general since, as showed in example 5, there exist oracles computing non-continuous functionals, and we know that sequential algorithms are continuous. So we consider only continuous oracles, i.e. oracles represented by trees with no infinite branch (proposition 1). Again we proceed by cases on the definition of type 2 oracles:

case 2.1 $O^1 = u^2$. In this case $Alg(O^2)$ is the empty algorithm.

case 2.2 O^2 is the oracle that, without opening envelopes, answers n. In this case $Alg(O^2) = \{(init_2, output\ n)\}$.

case 2.3 This is the principal case. We define, for any tree T representing a type 2 oracle O^2 (in the sense defined in the section devoted to type 2 oracles), an algorithm $Alg(O^2)$ which computes the same functional as O^2. This can be done by exploring T in a breadth-first manner: to each node q of T will correspond an event (c, v) of N^2, such that c describes the behaviour of the type 1 algorithm so far explored in the branch of q (remind that a cell c of N^2 is essentially a finite type 1 algorithm), and v describes the (valof or output) action performed by O^2 on such an algorithm, contained in q. Any event produced in this way will be enabled by the event corresponding to the predecessor of q in T, the event corresponding to the root of T ($\emptyset$?) being always $(init_2, valof\ init_1)$, enabled by the empty set.

Before giving the general definition of $Alg(O^2)$ in this last case, we show the sequential algorithms corresponding to the oracles of examples 6 and 7. We keep the tree-structure of oracles. Nodes contain events and arcs enablings (for typographical reasons the arcs are not explicitly given, but they can easily be reconstructed from the corresponding oracles). However it is worth noticing that the tree structure, essential in the tree description of oracles, is no more necessary for sequential algorithms, since all the information contained in the branch of the ancestors of any node q is supplied by the cell of the event corresponding to q.

We show how to produce the sequential algorithm $Alg(O^2)$ when O^2 is the oracle of the example 6, represented by the tree T, in a stepwise manner. At step i we produce an algorithm which simulates T up to level i (i.e. which simulate branches of depth less or equal than i). In this example we use the above introduced abbreviation for type 1 algorithms, namely

$$\{(init_1, valof\ *)((n_1, *), output\ m_1), \ldots, ((n_k, *), output\ m_k)\}$$

for $k \geq 0$, will be noted $\{(n_1, m_1), \ldots, (n_k, m_k)\}$. At step 0 we produce the (initial) event $(init_2, valof\ init_1)$ which correspond to the root of T. This event enables infinitely many cells, and in particular those we need for simulating depth-1 branches, as showed in the following algorithm (step 1):

$$\begin{array}{ll} & ((\{(init_1, valof\ *)\}, *), valof\ 0) \\ (init_2, valof\ init_1) & \\ & ((\{(init_1, output\ 0)\}, *), output\ 0) \\ & ((\{(init_1, output\ 1)\}, *), output\ 1) \\ & ((\{(init_1, output\ 2)\}, *), output\ 1) \end{array}$$

The event $((\{(init_1, valof *)\}, *), valof\, 0)$ enables the cells $(\{(0,3)\}, *), (\{(0,0)\}, *)$ and $(\{(0,27)\}, *)$ that we fill at step 2 in the following way:

$$((\{(0,3)\}, *), valof\, (1, *))$$

$$((\{(init_1, valof *)\}, *), valof\, 0) \quad ((\{(0,0)\}, *), output\; 0)$$

$$((\{(0,27)\}, *), valof\, (2, *))$$

$$(init_2, valof\; init_1)$$

$$((\{(init_1, output\; 0)\}, *), output\; 0)$$

$$((\{(init_1, output\; 1)\}, *), output\; 1)$$

$$((\{(init_1, output\; 2)\}, *), output\; 1)$$

The final step 3 produces $Alg(O^2)$ completely (for typographical reasons we omit this stage).

The following algorithm corresponds to the third (and final) step in the construction of $Alg(O^2)$, where O^2 is the oracle of example 7.

$$((\{(0,0)\}, *), output\; 1)$$

$$((\{(0,1)\}, *), output\; 1)$$

$$((\{(0,2)\}, *), output\; 1)$$

$$(init_2, valof\; init_1) \quad ((\{(init_1, valof *)\}, *), valof\, 0) \quad \vdots$$

$$((\{(0,n)\}, *), output\; 1)$$

$$((\{(0,n+1)\}, *), output\; 1)$$

$$\vdots$$

We can now give a procedure for constructing $Alg(O^2)$ for a type 2 oracle operating under case 2.3, described by a tree T: let $Alg^0(O^2)$ be the algorithm $\{(init_2, valof\; init_1)\}$ (step 0).

- *step 1*

  For any depth-1 branch of T of the form

  $$\emptyset? \xrightarrow[\text{without opening } O^2 \text{ says } m_i\ (\lambda x.m_i)]{} n_i$$

  we produce the event $((\{(init_1, output\; m_i)\}, *), output\; n_i)$.

If T contains the depth-1 branch:

$$\emptyset? \underset{O^2 \text{ opens}}{\rule{4cm}{0.4pt}} r_0?$$

(remark that T contains at most one branch of this kind, by unicity of the first non-preliminary question r_0 asked to envelopes-opening arguments), we produce the event $((\{(init_1, valof\ *)\}, *), valof\ r_0\})$. Let $Alg^1(O^2)$ be the algorithm

$$Alg^1(O^2) = Alg^0(O^2) \bigcup \{ \text{ all the events produced at step } 1\}$$

(remark that any cell filled in step 1 is enabled by $Alg^0(O^2)$)

- *step* $i+1$, $i > 0$

  For any depth-$i+1$ branch

$$\emptyset? \underset{O^1 \text{ opens}}{\rule{1.5cm}{0.4pt}} r_0? \underset{n_0}{\rule{1.5cm}{0.4pt}} r_1? \qquad \cdots \qquad r_{i-1}? \underset{n_{i-1}}{\rule{1.5cm}{0.4pt}} r_i?$$

  we produce the event

$$((\{(r_0, n_0), (r_1, n_1), \ldots, (r_{1-1}, n_{i-1})\}, *), valof\ (r_i, *))$$

  and for any depth$i+1$ branch of the form

$$\emptyset? \underset{O^1 \text{ opens}}{\rule{1.5cm}{0.4pt}} r_0? \underset{n_0}{\rule{1.5cm}{0.4pt}} r_1? \qquad \cdots \qquad r_{i-1}? \underset{n_{i-1}}{\rule{1.5cm}{0.4pt}} k$$

  we produce the event

$$((\{(r_0, n_0), (r_1, n_1), \ldots, (r_{1-1}, n_{i-1})\}, *), output\ k)$$

  Remark that any cell filled at this step is enabled by the event previously produced for the depth-i branch:

$$\emptyset? \underset{O^1 \text{ opens}}{\rule{1.5cm}{0.4pt}} r_0? \underset{n_0}{\rule{1.5cm}{0.4pt}} r_1? \qquad \cdots \qquad r_{i-1}?$$

  Let $Alg^{i+1}(O^2)$ be the algorithm

$$Alg^{i+1}(O^2) = Alg^i(O^2) \bigcup \{ \text{ all the events produced at step } i+1\}$$

Finally define

$$Alg(O^2) = \bigcup_{i \leq depth(T)} Alg^i(O^2)$$

Once again it is clear that if T has infinite branches, as in example 5, then $Alg(O^2)$ can not be defined (it could be defined if infinite cells were admitted in N^2)

Proposition 2 *If O^2 is a continuous type 2 oracle , then $Alg(O^2)$ is a sequential algorithm such that, for any type 1 oracle O^1,*

$$Alg(O^2)(Alg(O^1)) = O^2(O^1)$$

Proof: $Alg(O^2)$ is a sequential algorithm, i.e. a state of N^2, since at any step we fill only cells enabled in the previous step or initial cells (at step 0) and clearly any cell is filled with at most one value. Let us prove that if, for a given O^1, $O^2(O^1) = n$, then $Alg(O^2)(Alg(O^1)) = n$. If O^2 operates under cases 2.1 or 2.2, then this is trivially the case. Let O^2 operate under case 3.3 and T be the corresponding tree. In this case we reason by cases on the branch b of T that is followed in the computation $O^2(O^1)$. By hypothesis such a branch ends with "n". Two cases are possible for b:

- $$\emptyset? \xrightarrow[\text{without opening } O^1 \text{ says } m\ (\lambda x.m)]{} n$$

  In this case $Alg(O^2)$ contains the event

  $$((\{(init_1, output\ m)\}, *), output\ n)$$

  Moreover we know that O^1 is the oracle that, without opening envelopes, gives result m, hence

  $$Alg(O^1) = \{(init_1, output\ m)\}$$

  Hence we get $Alg(O^2)(Alg(O^1)) = n$

- $$\emptyset? \xrightarrow[O^1 \text{ opens}]{} r_0? \xrightarrow[n_0]{} r_1? \quad \cdots \quad r_{i-1}? \xrightarrow[n_{i-1}]{} n$$

  in this case $Alg(O^2)$ contains the event

  $$((\{(r_0, n_0), (r_1, n_1), \ldots, (r_{1-1}, n_{i-1})\}, *), output\ n)$$

  Moreover O^1 open envelopes and, for $j \leq i-1$, O^1 gives result n_j when presented with r_j. Hence we get

  $$\{(r_0, n_0), (r_1, n_1), \ldots, (r_{1-1}, n_{i-1})\} \subseteq Alg(O^1)$$

  and hence $Alg(O^2)(Alg(O^1)) = n$

Similarly one can prove that, if $Alg(O^2)(Alg(O^1)) = n$, than $O^2(O^1) = n$ ∎

We have seen that any (continuous) oracle may be simulated by an algorithm. The converse does not hold essentially because sequential algorithms may use *intensional* features of arguments in order to give a result. Consider for instance the following type 2 sequential algorithm

$$Strictness - tester = \{(init_2, valof\ init_1)((\{(init_1, valof\ *)\}, *), output\ 0)\}$$

The algorithm *Strictness* − *tester*, when applied to a type 1 algorithm A, gives value 0 if and only if the function computed by A is strict. Clearly *Strictness*−*tester* does not define a monotone functional, and hence it cannot be simulated by an oracle. Actually the tree representing an oracle simulating this algorithm should be

$$\emptyset? \xrightarrow[O^1 \text{ opens}]{\hspace{5em}} 0$$

Such a tree, as remarked at the end of the section devoted to oracles, is not legal since it violates the condition "at least one non preliminary question is asked" of case 2.3.3 of the type 2 oracles definition.

We end this section by showing that any finite type-2 unimonotone function (i.e. any type-2 unimonotone function computed by a finite tree) is PCF-definable. The argument is very simple, and we content ourselves with showing its application to the tree T of exemple 6: let f^2 be the functional computed by (the oracle associated to) T. We aim to define a PCF-term F defining f^2. It is clear that, for any type 1 function f^1, if $f^2(f^1)$ has to be defined then $f^1(0)$ has to be defined. We can hence safely construct a prefix of F of the form "λf^1 case $f^1(0)\ldots$ ". The only interesting case is $f^1(0) \in \{3, 0, 27\}$. If, for instance, $f^1(0) = 3$, we can safely branch on a " case $f^1(1)\ldots$", the only interesting case this time being $f^1(1) \in \{0, 1\}$. If, for instance, $f^1(1) = 0$, we end the branching operation by "... then 0". We simply follow the branches of T, constructing a case subterm for each branching. This is enough for simulating the subtree of T rooted in 0?. As for the branches labelled by " without opening, O^1 says i " ending in a leaf m_i, it is enough to add in each " case " branching the alternative " else if $f^1(\perp) = i \ldots$ then m_i".

Proposition 3 *Any finite type 2 unimonotone functional is PCF-definable.*

6 Type 3 oracles and algorithms

In [10] a complete description of type 3 oracles is given. Once again (continuous) type 3 oracles can be simulated by sequential algorithms. A first observation about the intensional behaviour of a type 3 oracle O^3 (i. e. about the interactions between this oracle and her type 2 argument O^2) is that in the principal case, the one in which both O^3 and O^2 are strict (open envelopes), O^3 cannot simply present O^2 with a first non-preliminary question O^1, as it was the case at lower types. Actually this simplistic approach had been followed by Kleene until a counter-example by D. Kierstead (reported in [9, page 27]) showed that it does not work in general. Kierstead's example involves non-continuous type 2 functions; we propose here an alternative example.

example 8: Consider the functions $f_1^1, f_2^1 : N \to N$ defined in example 3, and let F_1^1, F^1 be PCF-terms defining them (for instance, $F_1^1 = \lambda n$ if $n = 0$ then 0). Consider now the type 2 functionals f_1^2, f_2^2 defined as follows:

$$f_1^2 : \quad f_1^1 \mapsto 0$$

$$f_2^2 : \quad f_2^1 \mapsto 0$$

It is easy to see that f_1^2 and f_2^2 are actually unimonotone, and that they can be defined by terms F_i^1's as follows (we use a PCF-like syntax, but we could have used Kleene's schemata for recursive functionals as well):

$$F_1^2 = \lambda f^1 \text{ if } f^1(0) = 0 \text{ then } 0$$

$$F_2^2 = \lambda f^1 \text{ if } f^1(1) = 1 \text{ then } 0$$

Consider now the type 3 functional f^3 defined by:

$$f^3 : \quad f_i^2 \mapsto 0 \quad \text{for } 1 \leq i \leq 2$$

Such a functional is defined by the following term:

$$F = \lambda f^2 f^2(\lambda n \text{ if } \; n = 0 \; \text{ then } \; f^2(F_1^1) \text{ else } \; (\text{ if } \; n = 1 \; \text{ then } \; f^2(F_2^1))$$

and hence f^3 should be unimonotone. Suppose now that O^3 be an oracle for f^3: when presented with an envelopes-opening type 2 oracle O^2, O^3 cannot simply pass to O^2 a type 1 oracle O^1 and wait for an answer of O^2. O^3 must act in a more subtle way, taking into account the intensional behaviour of O^2. Let us see how an oracle O^3 computing f^3 has to behave when applied to O^2 computing f^2: having checked that O^2 works under case 2.3 (i.e. that f^2 is strict), O^3 questions O^2 with the type 1 oracle O^1 which opens envelopes but never gives a result (corresponding to the type 1 sequential algorithm $\{(\emptyset, *), \mathit{valof}\ *\}$). If $f^2 = f_1^2$, then O^2 will question O^1 with $\mathit{valof}\ (0, *)$, if $f^2 = f_2^2$, then O^2 will question O^1 with $\mathit{valof}\ (1, *)$. In the former case the following question of O^3 will be f_1^1 (i.e. O^3 will present O^2 with an oracle computing f_1^1) in the latter it will be f_2^1. In both cases O^3 uses her knowledge of the intensional behaviour of O^2 (i.e. of the question that O^2 has asked to O^1) for formulating the following question to O^2.

In the following diagram we show stage by stage the interaction between O^3 and O^2 (computing a type 2 function f^2). At even stages O^3 has the control, and she can either question O^2 with a type 1 oracle, or give a final answer. At odd stages O^2 can either question the O^1 she has been presented with, or give a final answer (answers are boxed in the diagram). We assume that at stage 0 O^3 has already learnt that O^2 behaves under case 2.3. We represent type 1 oracles by (corresponding) sequential algorithms. In this example O^2 computes f_1^2.

	O^3	O^2	What O^3 knows about O^2
			O^2 opens envelopes
0	*valof* $\{(\emptyset, *), valof\ *\}*$		
1		*valof* $(0, *)$	O^2 needs the value of O^1 on 0
2	*valof* $\{(0,0)\}*$		
3		$\boxed{output\ 0}$	$f^2 \geq f_1^2$
4	$\boxed{output\ 0}$		

The crucial stage is stage 1, in which O^3 learns that she can safely question O^2 with (an oracle for) f_1^1.

The interaction described by this diagram may be seen as an unfolding of the λ-term F. Actually, in F the formal parameter f^2 is applied to a type 1 function which is f_1^1 if $f^2 = f_1^2$, and f_2^1 if $f^2 = f_2^2$, getting in any case the right result. ■

Here is an alternative example:

example 9: Consider the functions $f_1^1, f_2^1, f_3^1 : N \rightarrow N$ defined in example 4, and let F_1^1, F_2^1, F_3^1 be PCF-terms defining them (for instance, $F_1^1 = \lambda n$ if $n = 1$ then 0 else if $n = 2$ then 1). Consider now the type 2 functionals f_1^2, f_2^2, f_3^2 defined as follows:

$$f_1^2 : \quad f_1^1 \mapsto 0 \quad f_2^1 \mapsto 1$$
$$f_2^2 : \quad f_2^1 \mapsto 0 \quad f_3^1 \mapsto 1$$
$$f_3^2 : \quad f_3^1 \mapsto 0 \quad f_1^1 \mapsto 1$$

It is easy to see that the f_i^2's are actually unimonotone, and that they can be defined by terms F_i^1's as follows

$$F_1^2 = \lambda f^1 \text{ case } \begin{array}{ll} f^1(2) = 1 & \text{if } f^1(1) = 0 \text{ then } 0 \\ f^1(2) = 0 & \text{if } f^1(0) = 1 \text{ then } 1 \end{array}$$

$$F_2^2 = \lambda f^1 \text{ case } \begin{array}{ll} f^1(0) = 1 & \text{if } f^1(2) = 0 \text{ then } 0 \\ f^1(0) = 0 & \text{if } f^1(1) = 1 \text{ then } 1 \end{array}$$

$$F_3^2 = \lambda f^1 \text{ case } \begin{array}{ll} f^1(1) = 1 & \text{if } f^1(0) = 0 \text{ then } 0 \\ f^1(1) = 0 & \text{if } f^1(2) = 1 \text{ then } 1 \end{array}$$

Consider now the type 3 functional f^3 defined by:

$$f^3 : \ f_i^2 \mapsto 0 \quad \text{for } 1 \leq i \leq 3$$

Such a functional is defined by the following term:

$$F = \lambda f^2 f^2(\lambda n \text{ case } \begin{array}{ll} n = 0 & f^2(F_3^1) \\ n = 1 & f^2(F_1^1) \\ n = 2 & f^2(F_2^1) \end{array})$$

and hence f^3 should be unimonotone. Suppose now that O^3 be an oracle for f^3: when presented with an envelopes-opening type 2 oracle O^2, O^3 cannot simply pass to O^2 a type 1 oracle O^1 and wait for an answer of O^2, since for any type 1 function f^1 there exists $i \leq 3$ such that $f_i^2(f^1)$ is not defined. ∎

We can now describe, following [10], the behaviour of a type 3 oracle O^3 presented with an envelope containing a type 2 oracle O^2. As for lower types, there are three cases:

case 3.1 O^3 stands mute. It computes the totally undefined functions $\lambda f^2 \perp$.

case 3.2 Without opening the envelope, O^3 gives result n. It computes the non-strict constant $\lambda f^2 n$.

case 3.3 O^3 opens the envelope, revealing that she will require some information about O^2. To obtain such information, she begins by questioning O^2 with the preliminary question $O^1 = u^1$. Three cases are possible, according to the behaviour of O^2:

case 3.3.1 O^2 does not open the envelope and stands mute. In this case O^3 stands mute too.

case 3.3.2 Without opening, O^2 gives result n. Depending on n, O^3 may either stand mute or give result m.

case 3.3.3 O^2 opens the envelope. Observing this O^3 may either stand mute or embark on a program of further systematic questioning of O^2: the goal of such questioning is to construct (a part of) the tree associated to O^2. To begin with, O^3 may choose the first non-preliminary question O^1 according to one of the following options:

option 1 O^1 is the non-strict constant n_0, for some $n \in \omega$. Questioned with O^1, O^2 will either stand mute (and the questioning *falters*) or give m_0 as result.

option 2 O^1 is the oracle which opens envelopes but never gives a result. Questioned with O^1, O^2 will either stand mute (and the questioning falters) or question O^1 with $r_0 \in \omega$.

Under either option, if the questioning has not faltered, O^3 records what she has thus far learned about O^2, namely:

option 1	option 2
$\emptyset ? \xrightarrow[\lambda x.n_0]{} m_0$	$\emptyset ? \xrightarrow[O^1 \text{opens}]{} r_0 ?$

Option 1 may be reused abitrarily many times, with different constants, whereas option 2 may be used at most one time. At any further stage of the questioning, O^3 may either use option 1 with a new constant or option 2 (if it has not been already used) or finally behave following the option we describe below.

option 3 O^3 picks an already explored branch of the form

$$\emptyset ? \xrightarrow[O^1 \text{opens}]{} r_0 ? \xrightarrow[n_0]{} r_1 ? \xrightarrow[n_1]{} \cdots \qquad r_k ? \xrightarrow[n_k]{} r_{k+1} ?$$

(where r_{k+1} is not necessarily a leaf) and answer the question r_{k+1} by n_{k+1} (if r_{k+1} is not a leaf, n_{k+1} has to be different from the answers previously provided) . That is O^3 questions O^2 with the type 1 oracle computing the function $\{(r_0, n_0), (r_1, n_1), \ldots, (r_{k+1}, n_{k+1})\}$. O^2 may either stand mute (and the questioning falters), or ask for r_{k+2} or finally give answer m. According [illegible]hich of the two last possibilities occurs the branch that h[illegible] choosed is completed, giving rise to

$$\emptyset ? \xrightarrow[O^1 \text{opens}]{} r_0 ? \xrightarrow[n_0]{} r_1 ? \xrightarrow[n_1]{} \cdots \qquad r_k ? \xrightarrow[n_k]{} r_{k+1} ? \xrightarrow[n_{k+2}]{} r_{k+2} ?$$

or to

$$\emptyset ? \xrightarrow[O^1 \text{opens}]{} r_0 ? \xrightarrow[n_0]{} r_1 ? \xrightarrow[n_1]{} \cdots \qquad r_k ? \xrightarrow[n_k]{} r_{k+1} ? \xrightarrow[n_{k+2}]{} m$$

A stage of the questioning is *final* if no branch of the tree constructed by O^3 is ended by a question $r?$. At any stage O^3 may decide to stop the questioning without giving any answer, or to pose a new question (following one of the described options) or finally (only if the stage is final) to give a global answer m on O^2.

This gives a complete description of type 3 oracles [3] but is far from assuring that these oracles actually compute type 3 monotone functions. The following requirement is needed: if O^2 and P^2 are oracles computing f^2 and g^2, and $f^2 \leq g^2$, then if $O^3(O^2) = n$, also $O^3(P^2) = n$. We can now see how a type 3 oracles O^3 may be simulated by a sequential algorithms $Alg(O^3)$. We follow the description above, and, in order to avoid the proliferation of brackets, we note cells of the form $(x, *)$ by $x*$ (x being a type 1 or 2 algorithm):

case 3.1 $Alg(O^3)$ is the empty algorithm.

case 3.2

$$Alg(O^3) = \{(\emptyset_2*,\ output\ n)\}$$

(we index empty algorithms by their type)

case 3.3

$$Alg(O^3) = \{\emptyset_2*,\ valof\ \emptyset_1*)\}$$

case 3.3.1 Nothing to say.

case 3.3.2

$$Alg(O^3) = \{(\emptyset_2*,\ valof\ \emptyset_1*), (\{(\emptyset_1*,\ output\ n)\}*,\ output\ m)\}$$

(the other case being trivial)

case 3.3.3

$$Alg(O^3) = \{(\emptyset_2*,\ valof\ \emptyset_1*), (\{(\emptyset_1*,\ valof\ \emptyset_0*)\}*,\ valof\ QUESTION)\}$$

The value of "$QUESTION$" depends on the option choosen by O^3, namely:

option 1

$$QUESTION = \{(\emptyset_0*,\ output\ n_0)\}*$$

In this case cells of the form

$$\{(\emptyset_1*,\ valof\ \emptyset_0*), (\{(\emptyset_0*,\ output\ n_0)\}*,\ output\ m_0)\}*$$

are enabled. If O^2 under option 1 gives answer m_0, this cell will be filled by a question arising from a next stage of the questioning.

option 2

$$QUESTION = \{(\emptyset_0*,\ valof\ *)\}*$$

In this case cells of the form

[3] Actually in Kleene's approach there exists a further option, related to the fact that the tree associated to a type 2 oracle may contain infinite branches. Since we are interested in unimonotone *and* continuous functions, we skip it.

$$\{(\emptyset_1*,\ \mathit{valof}\ \emptyset_0*),(\{(\mathit{vide}_0*,\ \mathit{valof}\ *)\}*,\ \mathit{valof}\ r_0*)\}*$$

are enabled. These cells may be filled by values

$$\mathit{valof}\ \{(\emptyset_0*,\ \mathit{valof}\ *)(r_0*,\ \mathit{output}\ n_0)\}*$$

and they will be actually filled and added to $Alg(O^3)$ if necessary, following the third option described below.

option 3 By induction on the number of stages so far performed, we get that the cell

$$C = \{(\emptyset_1*,\ \mathit{valof}\ \emptyset_0*),(\{(\emptyset_0*,\ \mathit{valof}\ *)\}*,\ \mathit{valof}\ r_0*),\ldots,$$

$$(\{(\emptyset_0*,\ \mathit{val}*),(r_0*,\ \mathit{output}\ n_0),\ldots,(r_k*,\ \mathit{output}\ n_k)\}*,\ \mathit{valof}\ r_{k+1}*\}*$$

is enabled. Option 3 is simulated by adding the event

$$(C,\ \mathit{valof}\ \{(\emptyset_0*,\ \mathit{val}*),(r_0*,\ \mathit{output}\ n_0),\ldots$$

$$\ldots,(r_k*,\ \mathit{output}\ n_k),(r_{k+1},\ \mathit{output}\ n_{k+1})\}*)$$

to $Alg(O^3)$.

When a final stage in which O^3 gives a global answer m on O^2 is reached, O^3 has explored a subtree $\overline{O^2}$ of O^2. Hence the cell $Alg(\overline{O^2})*$ is enabled by $Alg(O^3)$ constructed so far, and it can be filled by *output* m, completing the translation.

Proving that $Alg(O^3)$ computes the same type 3 function as O^3 is straightforward, using the same arguments as in proposition 2, but the complication of notations makes the proof longer and less understandable.

7 Conclusion

We have seen that any (continuous and) unimonotone function is computed by some sequential algorithm, and that the converse does not hold. It would be interesting to compare unimonotone functions and the *extensional* sequential algorithms defined in [5] (the algorithm $Strictness - tester$ at the end of section 5, that cannot be simulated by an oracle is not extensional).

It is natural to ask whether or not any finite unimonotone function is PCF-definable. By rearranging Curien's examples of sequential and non-definable functionals [4, page 269]) one can show that there exist type 3 continuous and unimonotone functionals which are non-definable. At type 2 the converse does hold, the λ-term defining a continuous and unimonotone type 2 functional being easily constructed from the finite tree associated to (an oracle for) f^2.

References

[1] G. Berry. *Modèles complètement adéquats et stables des lambda-calculs typés.* Thèse d'Etat, Université Paris 7, 1979.

[2] G. Berry and P.L. Curien. *Sequential algorithms on concrete data structures.* TCS 20, 1982.

[3] G. Berry, P.L. Curien, J.J. Levy. *Full abstraction for sequential languages: the state of the art.* Algebraic mehods in Semantics, Cambridge University Press 1985.

[4] P.L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical computer Science, Pitman (1986).

[5] P.L. Curien *Algorithmes séquentiels et extensionalité.* Rapport LITP 82-67, Université Paris 6-7, 1982.

[6] G. Kahn and G. Plotkin. *Domaines Concrets.* Rapport IRIA-LABORIA 336, 1978.

[7] D. P. Kierstead *A Semantics for Kleene's j-expressions* The Kleene Symposium (J. Barwise, H. J. Keisler and K. Kunen eds.). North-Holland 1980.

[8] S. C. Kleene *Recursive Functional and Quantifiers of Finite Types Revisited I* Generalized recursion theory II, Proceedings of the 1977 Oslo Symposium (J. E. Fenstad, R. O. Gandy and G. E. Sacks eds.). North-Holland 1978.

[9] S. C. Kleene *Recursive Functional and Quantifiers of Finite Types Revisited II* The Kleene Symposium (J. Barwise, H. J. Keisler and K. Kunen eds.). North-Holland 1980.

[10] S. C. Kleene *Recursive Functional and Quantifiers of Finite Types Revisited III* Patras Logic Symposium (G. Metakides ed.). North-Holland 1982.

[11] S. C. Kleene *Recursive Functional and Quantifiers of Finite Types Revisited IV* Proocedings of Symposia in Pure Mathematics, vol. 42, 1985. American Mathematical Society.

[12] G. Plotkin. *LCF considered as a programming language.* TCS 5, december 1977, p. 223-256.

Mechanizing Logical Relations

Allen Stoughton*
Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506, USA
E-mail: allen@cis.ksu.edu

Abstract. We give an algorithm for deciding whether there exists a definable element of a finite model of an applied typed lambda calculus that passes certain tests, in the special case when all the constants and test arguments are of order at most one. When there is such an element, the algorithm outputs a term that passes the tests; otherwise, the algorithm outputs a logical relation that demonstrates the nonexistence of such an element. Several example applications of the C implementation of this algorithm are considered.

1 Introduction

Given a model of an applied typed lambda calculus, it is natural to consider the problem of determining whether an element of that model is definable by a term, or, more generally, of determining whether there exists a definable element of the model that passes certain tests. One approach to settling such questions makes use of so-called "logical relations" [Plo80].

Building on recent work on logical relations by Sieber [Sie92], we give an algorithm for deciding whether there exists a definable element of a finite model that passes certain tests, in the special case when all the constants and test arguments are of order at most one. When there is such an element, the algorithm outputs a term that passes the tests; otherwise, the algorithm outputs a logical relation that demonstrates the nonexistence of such an element. Loader's recent proof of the undecidability of the lambda definability problem [Loa94] shows that the restriction to constants and test arguments of order at most one is necessary. (Specifically, Loader shows the undecidability of the problem of determining the definability of order-three elements of the full type hierarchy over a seven element set.)

The algorithm was first implemented in Standard ML and used to find an interesting non-definability proof (see Lemma 4.16 of [JS93]). An efficient implementation of the algorithm in ANSI C has now been written and applied to various definability problems, some examples of which are described below. A copy of this program, *lambda*, along with supporting documentation and a number of example lambda definability problems, can be obtained by anonymous ftp. Connect to `ftp.cis.ksu.edu`, login as `anonymous`, change directory to `pub/CIS/Stoughton/lambda`, retrieve the file `README`, and follow the instructions given in that file.

*The research reported here was partially supported by ESPRIT project CLICS-II and was performed while the author was on the faculty of the School of Cognitive and Computing Sciences of the University of Sussex.

2 The typed lambda calculus

This section consists of the mostly standard definitions concerning the syntax and semantics of the typed lambda calculus that will be required in the sequel. An introduction to the typed lambda calculus can be found, e.g., in [Mit90].

The set of *types* T is least such that

(i) $\iota \in T$,

(ii) $\sigma \to \tau \in T$ if $\sigma \in T$ and $\tau \in T$.

We let $\to$ associate to the right. The *order* $\operatorname{ord} \sigma \in \omega$ of a type $\sigma \in T$ is defined by $\operatorname{ord} \iota = 0$ and $\operatorname{ord}(\sigma \to \tau) =$ the maximum of $1 + \operatorname{ord} \sigma$ and $\operatorname{ord} \tau$. The *arity* $\operatorname{ar} \sigma \in \omega$ of a type σ is defined by $\operatorname{ar} \iota = 0$ and $\operatorname{ar}(\sigma \to \tau) = 1 + \operatorname{ar} \tau$. Thus, if $n > 0$ and $\sigma_i \in T$ for all $i \in n$, then $\operatorname{ord}(\sigma_0 \to \cdots \to \sigma_{n-1} \to \iota) = 1 +$ the maximum of $\{ \operatorname{ord} \sigma_i \mid i \in n \}$ and $\operatorname{ar}(\sigma_0 \to \cdots \to \sigma_{n-1} \to \iota) = n$.

Define σ^n, for $n \in \omega$, by: $\sigma^0 = \sigma$ and $\sigma^{n+1} = \sigma \to \sigma^n$. Thus, for all $n \in \omega$, $\operatorname{ar} \sigma^n = n + \operatorname{ar} \sigma$ and $\operatorname{ord} \sigma^n$ is $\operatorname{ord} \sigma$, if $n = 0$, and is $1 + \operatorname{ord} \sigma$, otherwise. It is easy to see that σ has order at most one just when it is of the form ι^n for some $n \in \omega$.

Many operations and concepts extend naturally from sets to T-indexed families of sets, in a pointwise manner. For example, given an ordinal α, an *α-ary relation* $R_{(-)}$ over a T-indexed family of sets $A_{(-)}$ is a T-indexed family of α-ary relations R_σ over A_σ. We will make use of this and other such extensions without explicit comment. We sometimes confuse a T-indexed family of sets A with $\bigcup_{\sigma \in T} A_\sigma$.

V is a T-indexed family of disjoint, denumerable sets of *variables*. A *family of constants* C is a T-indexed family of disjoint sets. We say that such a C is *finite* iff $\bigcup_{\sigma \in T} C_\sigma$ is finite, and that C is *infinite* otherwise. The *order* $\operatorname{ord} C \in \omega \cup \{\infty\}$ of C is the greatest element of $\{ \operatorname{ord} \sigma \mid \sigma \in T \text{ and } C_\sigma \neq \emptyset \}$ if it exists, and ∞ otherwise.

The family $\Lambda(C)$ of *typed λ-terms* over a family of constants C is least such that

(i) $c \in \Lambda(C)_\sigma$ if $c \in C_\sigma$,

(ii) $x \in \Lambda(C)_\sigma$ if $x \in V_\sigma$,

(iii) $M\,N \in \Lambda(C)_\tau$ if $M \in \Lambda(C)_{\sigma \to \tau}$ and $N \in \Lambda(C)_\sigma$,

(iv) $\lambda x.\, M \in \Lambda(C)_{\sigma \to \tau}$ if $x \in V_\sigma$ and $M \in \Lambda(C)_\tau$.

We call a term $M\,N$ an *application* and a term $\lambda x.\, M$ an *abstraction*. We let application associate to the left, and abbreviate $\lambda x_0. \cdots \lambda x_{n-1}. M$ to $\lambda x_0 \cdots x_{n-1}. M$. (When $n = 0$, $\lambda x_0 \cdots x_{n-1}. M = M$.) The set of *free variables* $\operatorname{fv} M \in \mathcal{P}(\bigcup_{\sigma \in T} V_\sigma)$ of a term $M \in \Lambda(C)$ is defined by $\operatorname{fv} c = \emptyset$, $\operatorname{fv} x = \{x\}$, $\operatorname{fv}(M\,N) = \operatorname{fv} M \cup \operatorname{fv} N$ and $\operatorname{fv}(\lambda x.\, M) = \operatorname{fv} M - \{x\}$. A term $M \in \Lambda(C)$ is *closed* iff $\operatorname{fv} M = \emptyset$, and *open* otherwise.

We write $\Gamma(C)$ for the family of *λ-free terms* over C: $\Gamma(C)_\sigma = \{ M \in \Lambda(C)_\sigma \mid M \text{ is } \lambda\text{-free} \}$. The *depth* $\operatorname{depth} M \in \omega$ of a λ-free term M is defined by $\operatorname{depth} c = \operatorname{depth} x = 0$ and $\operatorname{depth}(M\,N) =$ the maximum of $\operatorname{depth} M$ and $1 + \operatorname{depth} N$. The *size* $\operatorname{size} M \in \omega$ of a λ-free term M is defined by $\operatorname{size} c = \operatorname{size} x = 1$ and $\operatorname{size}(M\,N) = \operatorname{size} M + \operatorname{size} N$. Thus, if $n > 0$, $\sigma_i \in T$ for all $i \in n$, $M_i \in \Gamma(C)_{\sigma_i}$ for all $i \in n$ and d is a constant or variable of type $\sigma_0 \to \cdots \to \sigma_{n-1} \to \iota$, then $\mathrm{depth}(d\,M_0 \cdots M_{n-1}) = 1 +$ the maximum of $\{\,\mathrm{depth}\,M_i \mid i \in n\,\}$ and $\mathrm{size}(d\,M_0 \cdots M_{n-1}) = 1 + \mathrm{size}\,M_0 + \cdots + \mathrm{size}\,M_{n-1}$.

We write $f\,a$ for the application of a function f to an argument a, and let function application associate to the left. The set of all functions from a set A to a set B is denoted by $A \rightarrow B$, and $\rightarrow$ associates to the right.

A *type frame* A is a T-indexed set such that $A_\iota \neq \emptyset$ and $A_{\sigma\rightarrow\tau} \subseteq A_\sigma \rightarrow A_\tau$ for all $\sigma, \tau \in T$. We say that such an A is *finite* iff A_ι is finite, and that A is *infinite* otherwise. The set Env_A (or just Env) of *environments* over A consists of the set of all type-respecting functions from $\bigcup_{\sigma\in T} V_\sigma$ to $\bigcup_{\sigma\in T} A_\sigma$. If $\rho \in \mathrm{Env}$, $a \in A_\sigma$ and $x \in V_\sigma$, then $\rho[a/x] \in \mathrm{Env}$ is the environment that sends x to a, and sends all $y \not\equiv x$ to $\rho\,y$. We write Sem_A (or just Sem) for the T-indexed family of sets defined by $\mathrm{Sem}_\sigma = \mathrm{Env} \rightarrow A_\sigma$.

A $\Lambda(C)$-*model* $\mathcal{A}$ consists of a type frame A, together with an element $c_{\mathcal{A}} \in A_\sigma$ for each $c \in C_\sigma$, such that the following recursive definition of the meaning $[\![M]\!] \in \mathrm{Sem}_\sigma$ of a term $M \in \Lambda(C)_\sigma$ is well-defined:

$$\begin{aligned}
[\![c]\!]\rho &= c_{\mathcal{A}}\\
[\![x]\!]\rho &= \rho\,x\\
[\![M\,N]\!]\rho &= ([\![M]\!]\rho)([\![N]\!]\rho)\\
[\![\lambda x.\,M]\!]\,\rho\,a &= [\![M]\!]\rho[a/x].
\end{aligned}$$

When M is closed, we often write $[\![M]\!]$ for $[\![M]\!]\rho$, where $\rho \in \mathrm{Env}$ is arbitrary. An element $a \in A_\sigma$ is *definable* iff there exists a closed term $M \in \Lambda(C)_\sigma$ such that $a = [\![M]\!]$. We say that $\mathcal{A}$ is *finite* iff A is finite, and that $\mathcal{A}$ is *infinite* otherwise.

Our example model in the sequel will be the *monotone function model* of *Finitary PCF*: the restriction of PCF [Plo77] to the booleans. We write FPCF for the family of constants such that $\mathrm{FPCF}_\iota = \{\Omega, \mathrm{tt}, \mathrm{ff}\}$, $\mathrm{FPCF}_{\iota^3} = \mathrm{If}$, and $\mathrm{FPCF}_\sigma = \emptyset$ for all other $\sigma \in T$, and define a finite $\Lambda(\mathrm{FPCF})$-model $\mathcal{F}$ as follows. F_ι is the poset $\{\bot, \mathrm{tt}, \mathrm{ff}\}$, where $\bot$ is $\sqsubseteq$ the incomparable elements tt and ff, and $F_{\sigma\rightarrow\tau}$ is the set of all monotonic functions from F_σ to F_τ, ordered pointwise ($f \sqsubseteq g$ iff $f\,a \sqsubseteq g\,a$ for all a). We then set $\Omega_{\mathcal{F}} = \bot$, $\mathrm{tt}_{\mathcal{F}} = \mathrm{tt}$, $\mathrm{ff}_{\mathcal{F}} = \mathrm{ff}$ and define $\mathrm{If}_{\mathcal{F}}$ by

$$\mathrm{If}_{\mathcal{F}}\,x\,y\,z = \begin{cases} \bot & \text{if } x = \bot,\\ y & \text{if } x = \mathrm{tt},\\ z & \text{if } x = \mathrm{ff}. \end{cases}$$

One shows that the meaning function for $\mathcal{F}$ is well-defined by ordering $\mathrm{Env}_{\mathcal{F}}$ pointwise and showing by induction on M that $[\![M]\!]$ is both well-defined and monotonic.

3 Definability

We now consider the problem of determining whether an element of a $\Lambda(C)$-model is definable, or, more generally, of determining whether there exists a definable element of a $\Lambda(C)$-model that passes certain tests. For example, we can ask whether the "parallel or" operation of the Λ(FPCF)-model $\mathcal{F}$ is definable, i.e., whether there exists a closed term M of type ι^2 such that

$$\begin{array}{llll} [\![M]\!] & \mathrm{tt} & \perp & = \mathrm{tt} \\ [\![M]\!] & \perp & \mathrm{tt} & = \mathrm{tt} \\ [\![M]\!] & \mathrm{ff} & \mathrm{ff} & = \mathrm{ff}. \end{array}$$

One approach to settling such questions makes use of so-called "logical relations" [Plo80]. It is easier to say what logical relations are if we first extend function application from elements of type frames to tuples of elements of type frames, in a componentwise manner. Suppose A is a type frame, α is an ordinal and $\sigma, \tau \in T$. If $X = \langle x_\lambda \in A_{\sigma \to \tau} \mid \lambda \in \alpha \rangle$ and $Y = \langle y_\lambda \in A_\sigma \mid \lambda \in \alpha \rangle$, then we define the ***application*** XY of X to Y to be $\langle x_\lambda y_\lambda \in A_\tau \mid \lambda \in \alpha \rangle$, and let XY associate to the left. Given an $a \in A_\sigma$, we sometimes write a for $\langle a \mid \lambda \in \alpha \rangle \in A_\sigma^\alpha$.

An *α-ary logical relation* R over a type frame A is an α-ary relation over A such that $X \in R_{\sigma \to \tau}$ iff $XY \in R_\tau$ for all $Y \in R_\sigma$. We say that an α-tuple $X \in A_\sigma^\alpha$ *satisfies* such an R iff $X \in R_\sigma$. An *α-ary logical relation* R over a $\Lambda(C)$-model $\mathcal{A}$ is an α-ary logical relation over A such that $c_{\mathcal{A}}$ satisfies R for all $c \in C$.

The following theorem and its corollary show why logical relations are useful for showing non-definability results.

Theorem 3.1 (Plotkin) *If R is an α-ary logical relation over a $\Lambda(C)$-model $\mathcal{A}$, then $[\![M]\!]$ satisfies R for all closed $M \in \Lambda(C)$.*

Proof. An easy induction on $\Lambda(C)$ shows that, for all $M \in \Lambda(C)_\sigma$ and $\rho_\lambda \in \mathrm{Env}$ for all $\lambda \in \alpha$, if $\langle \rho_\lambda x \mid \lambda \in \alpha \rangle \in R_\tau$ for all $x \in \mathrm{fv}\, M \cap V_\tau$ and $\tau \in T$, then $\langle [\![M]\!]\rho_\lambda \mid \lambda \in \alpha \rangle \in R_\sigma$. The result then follows immediately. □

Corollary 3.2 *Let $\mathcal{A}$ be a $\Lambda(C)$-model, $\Delta_i \in A_{\sigma_i}^\alpha$ for all $i \in m$, $X \in A_\iota^\alpha$ and R be an α-ary logical relation over $\mathcal{A}$, for $m \in \omega$ and an ordinal α. If R is satisfied by Δ_i for all $i \in m$ but is not satisfied by X, then there is no definable $a \in A_{\sigma_0 \to \cdots \to \sigma_{m-1} \to \iota}$ such that $a\,\Delta_0 \cdots \Delta_{m-1} = X$.*

Proof. Immediate from Theorem 3.1. □

We can, e.g., use Corollary 3.2 to prove Plotkin's result [Plo77] that parallel or is not definable in Finitary PCF. (Although the following proof is due to Plotkin, he never published it. It was recently rediscovered by Sieber [Sie92].) Define *argument tuples* $\Delta_i \in F_\iota^3$ for all $i \in 2$ and a *result tuple* $X \in F_\iota^3$ by taking $\Delta_0 = \langle \mathrm{tt}, \perp, \mathrm{ff} \rangle$ (the first argument column of the display at the beginning of this section), $\Delta_1 = \langle \perp, \mathrm{tt}, \mathrm{ff} \rangle$ (the second argument column of that display) and $X = \langle \mathrm{tt}, \mathrm{tt}, \mathrm{ff} \rangle$ (the result column of that display). Let R be the ternary logical relation over F such that $\langle x, y, z \rangle \in R_\iota$ iff $x = y = z$ or one of x or y is $\perp$. It is easy to show that R is satisfied by the interpretations of Ω, tt, ff and If. But R

is satisfied by Δ_0 and Δ_1 but not by X, allowing us to conclude that there is no definable $f \in F_{\iota^3}$ such that $f\,\Delta_0\,\Delta_1 = X$.

Loader's recent proof of the undecidability of the lambda definability problem [Loa94] shows that Corollary 3.2 fails to provide a complete method for showing non-definability (and thus definability) results. However, a slight generalization of Theorem 4.1 of [Sie92] shows that it does provide a complete method in the special case where the orders of C and the σ_i's are at most one (cf., Theorem 1 of [Plo80] and Theorem 5 of [JT93]).

Definition 3.3 Suppose $\mathcal{A}$ is a $\Lambda(C)$-model and $\Delta = \langle \Delta_i \in A^{\alpha}_{\sigma_i} \mid i \in m \rangle$, for $m \in \omega$ and an ordinal α and where C and the σ_i's have order at most one. Then, $R(\Delta)$ is the α-ary logical relation over A such that $X \in R(\Delta)_\iota$ iff $a\,\Delta_0 \cdots \Delta_{m-1} = X$ for some definable $a \in A_{\sigma_0 \to \cdots \to \sigma_{m-1} \to \iota}$.

Lemma 3.4 (Sieber) *Suppose $\mathcal{A}$ is a $\Lambda(C)$-model and $\Delta = \langle \Delta_i \in A^{\alpha}_{\sigma_i} \mid i \in m \rangle$, for $m \in \omega$ and an ordinal α and where C and the σ_i's have order at most one. Then, $R(\Delta)$ is an α-ary logical relation over $\mathcal{A}$ that is satisfied by Δ_i for all $i \in m$.*

Proof. Suppose that $c \in C_{\iota^n}$. If $Y_0, \ldots, Y_{n-1} \in R(\Delta)_\iota$, then there are closed terms $M_0, \ldots, M_{n-1}$ of type $\sigma_0 \to \cdots \to \sigma_{m-1} \to \iota$ such that $[\![M_j]\!]\,\Delta_0 \cdots \Delta_{m-1} = Y_j$ for all $j \in n$. Then, the term

$$M = \lambda x_0 \cdots x_{m-1}.\, c\,(M_0\, x_0 \cdots x_{m-1}) \cdots (M_{n-1}\, x_0 \cdots x_{m-1})$$

of type $\sigma_0 \to \cdots \to \sigma_{m-1} \to \iota$ is such that

$$[\![M]\!]\,\Delta_0 \cdots \Delta_{m-1} = c_{\mathcal{A}}\, Y_0 \cdots Y_{n-1},$$

showing that $c_{\mathcal{A}}\, Y_0 \cdots Y_{n-1} \in R(\Delta)_\iota$. Thus $c_{\mathcal{A}}$ satisfies $R(\Delta)$. The proof that Δ_i satisfies $R(\Delta)$ for all $i \in m$ is almost identical (x_i is used in the term M instead of c). □

Theorem 3.5 (Sieber) *Suppose $\mathcal{A}$ is a $\Lambda(C)$-model, $\Delta = \langle \Delta_i \in A^{\alpha}_{\sigma_i} \mid i \in m \rangle$ and $X \in A^{\alpha}_{\iota}$, for $m \in \omega$ and an ordinal α and where C and the σ_i's have order at most one. Then, $a\,\Delta_0 \cdots \Delta_{m-1} = X$ for some definable $a \in A_{\sigma_0 \to \cdots \to \sigma_{m-1} \to \iota}$ iff every α-ary logical relation over $\mathcal{A}$ that is satisfied by Δ_i for all $i \in m$ is also satisfied by X.*

Proof. Immediate from Corollary 3.2 and Lemma 3.4. □

Although Theorem 3.5 gives a characterization of $R(\Delta)_\iota$, the fact that this characterization involves the universal quantification over all α-ary logical relations over $\mathcal{A}$ that are satisfied by the Δ_i limits its practical utility. It turns out, however, that we can give a much more direct characterization of $R(\Delta)_\iota$.

Definition 3.6 Suppose $\mathcal{A}$ is a $\Lambda(C)$-model and $\Delta = \langle \Delta_i \in A^{\alpha}_{\sigma_i} \mid i \in m \rangle$, for $m \in \omega$ and an ordinal α and where C and the σ_i's have order at most one. Then, $L(\Delta)$ is the α-ary logical relation over $\mathcal{A}$ such that $L(\Delta)_\iota$ is the least α-ary relation over A_ι that is closed under $c_{\mathcal{A}}$, for all $c \in C$, and Δ_i, for all $i \in m$, where the $c_{\mathcal{A}}$'s and Δ_i's are viewed as operations over A^{α}_{ι} in the obvious way.

Lemma 3.7 *Suppose $\mathcal{A}$ is a $\Lambda(C)$-model and $\Delta = \langle \Delta_i \in A^{\alpha}_{\sigma_i} \mid i \in m \rangle$, for $m \in \omega$ and an ordinal α and where C and the σ_i's have order at most one. Let $x_i \in V_{\sigma_i}$ for all $i \in m$ be distinct variables.*

(i) *Suppose that $c \in C_{\iota^n}$, $Y_0, \ldots, Y_{n-1} \in A^{\alpha}_{\iota}$ and $M_0, \ldots, M_{n-1} \in \Gamma(C)_{\iota}$. If $\mathrm{fv}\, M_j \subseteq \{x_0, \ldots, x_{m-1}\}$ and*

$$[\![\lambda x_0 \cdots x_{m-1}.\, M_j]\!]\, \Delta_0 \cdots \Delta_{m-1} = Y_j,$$

for all $j \in n$, then the λ-free term $M = c\, M_0 \cdots M_{n-1}$ of type ι is such that $\mathrm{fv}\, M \subseteq \{x_0, \ldots, x_{m-1}\}$ and

$$[\![\lambda x_0 \cdots x_{m-1}.\, M]\!]\, \Delta_0 \cdots \Delta_{m-1} = c_{\mathcal{A}}\, Y_0 \cdots Y_{n-1}.$$

(ii) *Suppose that $i \in m$, $Y_0, \ldots, Y_{\mathrm{ar}\,\sigma_i - 1} \in A^{\alpha}_{\iota}$ and $M_0, \ldots, M_{\mathrm{ar}\,\sigma_i - 1} \in \Gamma(C)_{\iota}$. If $\mathrm{fv}\, M_j \subseteq \{x_0, \ldots, x_{m-1}\}$ and*

$$[\![\lambda x_0 \cdots x_{m-1}.\, M_j]\!]\, \Delta_0 \cdots \Delta_{m-1} = Y_j,$$

for all $j \in \mathrm{ar}\,\sigma_i$, then the λ-free term $M = x_i\, M_0 \cdots M_{\mathrm{ar}\,\sigma_i - 1}$ of type ι is such that $\mathrm{fv}\, M \subseteq \{x_0, \ldots, x_{m-1}\}$ and

$$[\![\lambda x_0 \cdots x_{m-1}.\, M]\!]\, \Delta_0 \cdots \Delta_{m-1} = \Delta_i\, Y_0 \cdots Y_{\mathrm{ar}\,\sigma_i - 1}.$$

(iii) *For all $X \in L(\Delta)_{\iota}$, there is an $M \in \Gamma(C)_{\iota}$ such that $\mathrm{fv}\, M \subseteq \{x_0, \ldots, x_{m-1}\}$ and*

$$[\![\lambda x_0 \cdots x_{m-1}.\, M]\!]\, \Delta_0 \cdots \Delta_{m-1} = X.$$

Proof. (i) and (ii) are immediate, and (iii) follows from (i) and (ii) by induction on $L(\Delta)_{\iota}$. □

Lemma 3.8 *Suppose $\mathcal{A}$ is a $\Lambda(C)$-model and $\Delta = \langle \Delta_i \in A^{\alpha}_{\sigma_i} \mid i \in m \rangle$, for $m \in \omega$ and an ordinal α and where C and the σ_i's have order at most one. Then, $L(\Delta) = R(\Delta)$.*

Proof. $L(\Delta)$ is clearly an α-ary logical relation over $\mathcal{A}$ that is satisfied by Δ_i for all $i \in m$, and $L(\Delta)_{\iota} \subseteq R(\Delta)_{\iota}$ follows from Lemma 3.7 (iii). For the opposite inclusion, if $X \notin L(\Delta)_{\iota}$, then there is no definable $a \in A$ such that $a\, \Delta_0 \cdots \Delta_{m-1} = X$, by Corollary 3.2, and thus $X \notin R(\Delta)_{\iota}$. □

Theorem 3.9 *Suppose $\mathcal{A}$ is a $\Lambda(C)$-model, $\Delta = \langle \Delta_i \in A^{\alpha}_{\sigma_i} \mid i \in m \rangle$ and $X \in A^{\alpha}_{\iota}$, for $m \in \omega$ and an ordinal α and where C and the σ_i's have order at most one. Then, $a\, \Delta_0 \cdots \Delta_{m-1} = X$ for some definable $a \in A_{\sigma_0 \to \cdots \to \sigma_{m-1} \to \iota}$ iff $X \in L(\Delta)_{\iota}$.*

Proof. Immediate from Lemma 3.8. □

Theorem 3.9 and Lemma 3.7 suggest the following algorithm schema.

Algorithm Schema 3.10 *Inputs.* A finite family of constants C of order at most one, $m, \alpha \in \omega$, types $\sigma_0, \ldots, \sigma_{m-1}$ of order at most one, a finite, nonempty set A_ι, $c_A \in A_{\iota^n}$ for each $c \in C_{\iota^n}$, $\Delta = \langle \Delta_i \in A^\alpha_{\sigma_i} \mid i \in m \rangle$ and $X \in A^\alpha_\iota$, where we extend A_ι to a type frame A by taking $A_{\sigma \to \tau}$ to be the set of all functions from A_σ to A_τ for all $\sigma, \tau \in T$.

Initialization. Pick distinct variables $x_i \in V_{\sigma_i}$ for all $i \in m$. Initialize the *stage* $k \in \omega$ to 0. Let $Z \subseteq U$ be

$$\{\langle c_A, c\rangle \mid c \in C_\iota\} \cup \{\langle \Delta_i, x_i\rangle \mid i \in m \text{ and } \sigma_i = \iota\},$$

where U is set of all pairs $\langle Y, M\rangle$ such that $Y \in A^\alpha_\iota$, $M \in \Gamma(C)_\iota$ and $\operatorname{fv} M \subseteq \{x_0, \ldots, x_{m-1}\}$. Initialize the *state* $S \subseteq U$ to a subset of Z that is a function with domain $\operatorname{dom} Z$. (The particular subset chosen is left unspecified, as is the method used to compute that subset; it need not involve the construction of Z.) If $\langle X, M\rangle \in S$ for some term M, then terminate with k and the term $\lambda x_0 \cdots x_{m-1}. M$.

Loop. Let $Z = Z_1 \cup Z_2$, where $Z_1 \subseteq U$ is the set of all

$$\langle c_A Y_0 \cdots Y_{n-1}, c M_0 \cdots M_{n-1}\rangle$$

such that $c \in C_{\iota^n}$, $n > 0$ and $\langle Y_j, M_j\rangle \in S$ for all $j \in n$, and $Z_2 \subseteq U$ is the set of all

$$\langle \Delta_i Y_0 \cdots Y_{\operatorname{ar}\sigma_i - 1}, x_i M_0 \cdots M_{\operatorname{ar}\sigma_i - 1}\rangle$$

such that $i \in m$, $\operatorname{ar}\sigma_i > 0$ and $\langle Y_j, M_j\rangle \in S$ for all $j \in \operatorname{ar}\sigma_i$. Pick a subset S' of Z such that S' is a function with domain $\operatorname{dom} Z - \operatorname{dom} S$ and (†) $\langle Y, M\rangle \in S'$ implies that $\operatorname{size} M \leq \operatorname{size} N$ for all N that are paired with Y in Z. (The particular subset chosen is left unspecified, as is the method used to compute that subset; it need not involve the construction of Z, Z_1 or Z_2.) If $S' = \emptyset$, then terminate with k and $\operatorname{dom} S$. Otherwise, increment k by one and add the elements of S' to S. (‡) If $\langle X, M\rangle \in S$ for some term M, then terminate with k and the term $\lambda x_0 \cdots x_{m-1}. M$. Otherwise, repeat.

An *instance* of Algorithm Schema 3.10 is an algorithm formed from the schema by specifying the details that were left open. Condition (†) is included since experience suggests that this will ensure that instances of the schema will generate good quality terms. Theorem 3.11 doesn't depend upon (†) being included, however.

Theorem 3.11 *If we supply the required inputs to an instance of Algorithm Schema 3.10, then one of the following statements holds.*

(i) *The algorithm terminates with a stage l and a closed term of the form $\lambda x_0 \cdots x_{m-1}.M$, for distinct variables $x_i \in V_{\sigma_i}$ and a λ-free term M of type ι and depth l. Let $\mathcal{B}$ be any $\Lambda(C)$-model such that $B_\iota = A_\iota$, $c_{\mathcal{B}} = c_A$ for all $c \in C$, and $\Delta_i \in B^\alpha_{\sigma_i}$ for all $i \in m$. Then, $[\![\lambda x_0 \cdots x_{m-1}. M]\!]\, \Delta_0 \cdots \Delta_{m-1} = X$. Furthermore, if $N \in \Gamma(C)_\iota$ is such that $\operatorname{fv} N \subseteq \{x_0, \ldots, x_{m-1}\}$ and $[\![\lambda x_0 \cdots x_{m-1}. N]\!]\, \Delta_0 \cdots \Delta_{m-1} = X$, then* $\operatorname{depth} M \leq \operatorname{depth} N$.

(ii) *The algorithm terminates with a stage l and an α-ary relation Q over A_ι such that $X \notin Q$. If $\mathcal{B}$ is a $\Lambda(C)$-model with the above properties, then $Q = L(\Delta)_\iota$, so that there is no definable $b \in B$ such that $b\,\Delta_0 \cdots \Delta_{m-1} = X$.*

Proof. Let S_0 be the initial value of S, and S_l, for $l \geq 1$, be S's value when point (‡) is reached for the lth time (at which point k's value will be l; S_l is undefined if the algorithm terminates before (‡) has been executed l times). Then, the following properties hold (for (d)–(f), $\mathcal{B}$ is a $\Lambda(C)$-model with the properties specified in the theorem's statement):

(a) If S_l is defined, then S_l is a function.

(b) If S_{l+1} is defined, then it is a proper superset of S_l.

(c) If S_l is defined, $\langle Y, M\rangle \in S_l$ and either $l = 0$ or $Y \notin \operatorname{dom} S_{l-1}$, then depth $M = l$.

(d) If S_l is defined, then $\operatorname{dom} S_l \subseteq L(\Delta)_\iota$.

(e) If S_l is defined and $\langle Y, M\rangle \in S_l$, then $[\![\lambda x_0 \cdots x_{m-1}.\, M]\!]\,\Delta_0 \cdots \Delta_{m-1} = Y$.

(f) If S_l is defined, $M \in \Gamma(C)_\iota$, fv $M \subseteq \{x_0, \ldots, x_{m-1}\}$ and depth $M = l$, then $[\![\lambda x_0 \cdots x_{m-1}.\, M]\!]\,\Delta_0 \cdots \Delta_{m-1} \in \operatorname{dom} S_l$.

The proofs of properties (a), (d) and (e) are by induction on l, and Lemma 3.7 (i) and (ii) are used in (e)'s proof. The proof of (b) is obvious.

For (c), we use a course of values induction on l. We consider the case where M has the form $c\,M_0 \cdots M_{n-1}$ (the case where M has the form $x_i\,M_0 \cdots M_{\mathrm{ar}\,\sigma_i - 1}$ is similar). If $l = 0$, then $n = 0$, and thus depth $M =$ depth $c = 0$. So, suppose that $l > 0$, so that $Y \notin \operatorname{dom} S_{l-1}$. Then, $n > 0$ and there are $Y_j \in A^\alpha_\iota$ for all $j \in n$ such that $\langle Y_j, M_j\rangle \in S_{l-1}$ for all $j \in n$ and $Y = c_A\,Y_0 \cdots Y_{n-1}$. Let the stages $p_j < l$ for all $j \in n$ be such that $Y_j \in \operatorname{dom} S_{p_j}$ and either $p_j = 0$ or $Y_j \notin \operatorname{dom} S_{p_j - 1}$. Then, depth $M_j = p_j$ for all $j \in n$, by the inductive hypotheses for the p_j's, so that depth $M \leq l$. But, there must be a $j \in n$ such that $p_j = l - 1$, since otherwise $Y \in \operatorname{dom} S_{l-1}$. Thus, depth $M = l$.

The proof of (f) also proceeds by course of values induction on l, and, again, we consider the case where M has the form $c\,M_0 \cdots M_{n-1}$. If $l = 0$, then $n = 0$, and thus $[\![\lambda x_0 \cdots x_{m-1}.\, M]\!]\,\Delta_0 \cdots \Delta_{m-1} = c_A \in \operatorname{dom} S_l$. So, suppose that $l > 0$, so that $n > 0$. Let $p_j < l$ and $Y_j \in A^\alpha_\iota$, for all $j \in n$, be depth M_j and $[\![\lambda x_0 \cdots x_{m-1}.\, M_j]\!]\,\Delta_0 \cdots \Delta_{m-1}$, respectively. Then, by the inductive hypotheses for the p_j's, we have that $Y_j \in \operatorname{dom} S_{p_j}$ for all $j \in n$, so that $c_A\,Y_0 \cdots Y_{n-1} \in \operatorname{dom} S_l$. But $[\![\lambda x_0 \cdots x_{m-1}.\, M]\!]\,\Delta_0 \cdots \Delta_{m-1} = c_A\,Y_0 \cdots Y_{n-1}$, by Lemma 3.7 (i), and thus we are done.

From (a) and (b) and the fact that there are only finitely many α-tuples over A_ι, we can conclude that there is a largest l such that S_l is defined.

Suppose $\langle X, M\rangle \in S_l$ for some M, so that either $l = 0$ or $X \notin \operatorname{dom} S_{l-1}$ (otherwise, S_l would be undefined). Then, the algorithm terminates with a stage of l and the closed term $\lambda x_0 \cdots x_{m-1}.M$, and depth $M = l$ follows by (c). Let $\mathcal{B}$ be a $\Lambda(C)$-model satisfying the specified conditions. Then, $[\![\lambda x_0 \cdots x_{m-1}.\, M]\!]\,\Delta_0 \cdots \Delta_{m-1} = X$ by (e). Furthermore, if $N \in \Gamma(C)_\iota$ is such that fv $N \subseteq \{x_0, \ldots, x_{m-1}\}$ and $[\![\lambda x_0 \cdots x_{m-1}.\, N]\!]\,\Delta_0 \cdots \Delta_{m-1} = X$, then

Figure 1: Lambda definability problems

problem	$\rightarrow$	*iota_sect funs_sect cons_sect tests_sect*
iota_sect	$\rightarrow$	**iota** *Elem* { *Elem* }
funs_sect	$\rightarrow$	**functions** { *fun* }
fun	$\rightarrow$	*Fun clause* { *clause* }
clause	$\rightarrow$	*pat* { *pat* } = *result*
pat	$\rightarrow$	*Elem* | *Var* | _
result	$\rightarrow$	*Elem* | *Var*
cons_sect	$\rightarrow$	**constants** { *con* }
con	$\rightarrow$	*Elem* | *Fun*
tests_sect	$\rightarrow$	**tests** *test* { *test* }
test	$\rightarrow$	{ *test_arg* } = *test_result*
test_arg	$\rightarrow$	*Elem* | *Fun*
test_result	$\rightarrow$	*Elem*

depth $M \leq$ depth N, since otherwise (f) would imply that $X \in \operatorname{dom} S_{l'}$ for some $l' < l$.

Otherwise, $X \notin \operatorname{dom} S_l$, and thus the algorithm terminates with a stage of l and $\operatorname{dom} S_l$. Let $\mathcal{B}$ be a $\Lambda(C)$-model satisfying the specified conditions. By (d) and the fact that S_{l+1} is undefined, we have that $\operatorname{dom} S_l = L(\Delta)_\iota$. Thus, there is no definable $b \in B$ such that $b\,\Delta_0 \cdots \Delta_{m-1} = X$, by Theorem 3.9. □

Although instances of Algorithm Schema 3.10 always produce terms of minimal depth, they often fail to produce terms of minimal size. In fact, it is not hard to find an example of a pair of terms with identical depth and meaning, where the first term is produced by a schema instance and the second has strictly smaller size than the first (see the lambda definability problem **size.lam** that is included with *lambda*'s distribution).

4 Implementation

In this section, we describe an implementation, *lambda*, of an instance of Algorithm Schema 3.10, and give several examples of its use. *Lambda* doesn't carry out the algorithm's steps itself. Instead, it takes in a lambda definability problem, representing the algorithm's input data, and generates a C program that solves this problem, producing the algorithm's output.

The grammar in Figure 1 describes the syntax of *lambda definability problems*. In this grammar, curly brackets are used to denote repetition (zero or more occurrences of the phrases they surround). An element name, *Elem*, consists of a single upper case letter or digit. A function name, *Fun*, consists of an upper case letter, followed by one or more letters or digits. A variable name, *Var*, consists of a lower case letter, followed by zero or more lower case letters or digits. As usual, white space characters and comments (which begin with # and continue until end of line) separate tokens but are otherwise ignored.

A lambda definability problem has four sections. The *iota section* lists the elements of the set A_ι—the elements that exist at type ι.

The *functions section* defines zero or more first-order functions, using ML-style pattern matching. Each function definition consists of the function's name followed by a sequence of clauses, each of which must have the same number of patterns in its left hand side. A given variable may not appear twice in the left hand side of the same clause, and, if the right hand side of a clause is a variable, then that variable must appear in the left hand side of that clause.

Suppose that the body of a given function definition has the form

$$\begin{array}{ccccc} p_0^0 & \cdots & p_{m-1}^0 & = & r^0 \\ & & & \vdots & \\ p_0^{n-1} & \cdots & p_{m-1}^{n-1} & = & r^{n-1}. \end{array}$$

A clause j *matches* a sequence of argument elements $a_0, \ldots, a_{m-1}$ iff, for all $i \in m$, the pattern p_i^j is the *wildcard* _ or is a variable or is equal to a_i. The function definition must be completely specified in the sense that it has at least one clause that matches any given sequence of arguments. Furthermore, each of its clauses must be non-redundant in the sense that the clause matches some sequence of elements that isn't matched by any preceding clause in the definition. The function defined by the function definition is the element of A_{ι^m} that sends a sequence of arguments $a_0, \ldots, a_{m-1}$ to r^j, if clause j is the first clause that matches the argument sequence and r^j is an element, and sends the argument sequence to a_i, if clause j is the first clause that matches the argument sequence, r^j is a variable and $p_i^j = r^j$.

The *constants section* specifies the family of constants C, and thus the functions c_A for $c \in C$.

Finally, the *tests section* must have the form

$$\begin{array}{ccccc} \Delta_0^0 & \cdots & \Delta_{m-1}^0 & = & X^0 \\ & & & \vdots & \\ \Delta_0^{\alpha-1} & \cdots & \Delta_{m-1}^{\alpha-1} & = & X^{\alpha-1}. \end{array}$$

It implicitly specifies the natural numbers m and α, the types $\sigma_0, \ldots, \sigma_{m-1}$, the argument tuples $\Delta_0 \in A_{\sigma_0}^\alpha, \ldots, \Delta_{m-1} \in A_{\sigma_{m-1}}^\alpha$ and the result tuple $X \in A_\iota^\alpha$. The number of tests, α, is required to be non-zero, since otherwise a method of explicitly specifying the types σ_i would have to be devised.

Lambda is written in ANSI C, with the exception of its lexical analyzer and parser, which are written in *lex* and *yacc* source, respectively. It uses one UNIX System V system call. The C programs that it generates also conform to the ANSI standard; they use several UNIX System V system calls in order to implement checkpointing. The programs generated by *lambda* make no use of dynamic storage allocation (except during their initialization phases).

A program generated by *lambda* codes tuples of elements as integers, and represents the algorithm's state as an array indexed by those codes. An element of this array records (among other things) whether the tuple coded by its index has been found. If it has, the way in which it was constructed from previously produced tuples is also recorded; implicit in this information is a term that computes the tuple from the argument tuples. When a new tuple is found during a given stage of the closure process, its element of the array is updated to record this fact, but new tuples are distinguished from existing tuples until the stage's end. New tuples are produced by n nested for loops over the tuple codes, where n is the greatest number of arguments that any constant or argument tuple expects. When a given new tuple can be formed in multiple ways, the first way found whose implicit term has minimal size is selected.

Figure 2 contains our first example lambda definability problem (in the left column), along with its solution (in the right column). The comment indicates that this problem is contained in the file `por1.lam` that is included as part of *lambda*'s distribution. We think of `B`, `T` and `F` as standing for the elements $\bot$, tt and ff, respectively, of the monotone function model $\mathcal{F}$ of Finitary PCF. The occurrence of `B` in the constants section stands for the constant Ω of Finitary PCF, which is interpreted as $\bot$ in $\mathcal{F}$. The problem is to determine whether parallel or is definable in models of Finitary PCF that consist of $\{\bot, \mathrm{tt}, \mathrm{ff}\}$ at type ι and in which the constants are interpreted in the same way as in $\mathcal{F}$ (it will either be definable in all or no models of this sort).

Applying *lambda* to `por1.lam` generates a C program that carries out the algorithm's closure process, producing the relation listed in the figure. The stage of one indicates that it took only one stage of this process for the relation to stabilize, and it is easy to see that this relation is the one used to show the non-definability of parallel or in the preceding section. (A triple $\langle x, y, z\rangle$ is in the relation iff $x = y = z$ or $x = \bot$ or $y = \bot$.) Note that the result triple $\langle \mathrm{tt}, \mathrm{tt}, \mathrm{ff}\rangle$ is in the complement of the relation.

Figure 3 shows that parallel or remains non-definable when parallel convergence is added to Finitary PCF (the original proof of this result can be found in [Abr90]). This time the C program produced by *lambda* was run in verbose mode, with the consequence that the elements of the resulting relation are labeled with the stages at which they were found. The relation contains two more triples than does the relation of Figure 2: $\langle \mathrm{tt}, \mathrm{tt}, \bot\rangle$ (found at stage 2) and $\langle \mathrm{ff}, \mathrm{ff}, \bot\rangle$ (found at stage 3).

Figure 4 shows how a non-definability result from Proposition 4.4.2 of [Cur93] can be proved using logical relations. The resulting relation consists of those triples $\langle x, y, z\rangle$ such that $x = y = z$ or one of x, y or z is $\bot$. Oddly, it can be formed by adding two triples to the relation of Figure 3.

Figure 5 shows how the Berry-Plotkin function (cf., Exercise 4.1.18.2 of [Cur93]) can be used to separate Curien's A_1, A_2 and A_3. This time, the program produced by *lambda* was run in both ordinary (middle column) and

Figure 2: Non-definability of parallel or

```
$ por1.lam

iota

  B T F

functions

  If B _ _ = B
     T x _ = x
     F _ y = y

constants

  B T F If

tests

  T B = T
  B T = T
  F F = F
```

```
por1

Stage: 1

Relation (17 elements):

<B B B>
<B B T>
<B B F>
<B T B>
<B T T>
<B T F>
<B F B>
<B F T>
<B F F>
<T B B>
<T B T>
<T B F>
<T T T>
<F B B>
<F B T>
<F B F>
<F F F>

Relation complement (10 elements):

<T T B>
<T T F>
<T F B>
<T F T>
<T F F>
<F T B>
<F T T>
<F T F>
<F F B>
<F F T>
```

Figure 3: Parallel or is not definable using parallel convergence

```
# por2.lam

iota

  B T F

functions

  If B _ _ = B
     T x _ = x
     F _ y = y

  PConv B B = B
        _ _ = T

constants

  B T F If PConv

tests

  T B = T
  B T = T
  F F = F
```

```
por2

Stage: 3

Relation (19 elements):

<B B B>  0
<B B T>  1
<B B F>  1
<B T B>  1
<B T T>  1
<B T F>  0
<B F B>  1
<B F T>  1
<B F F>  1
<T B B>  1
<T B T>  1
<T B F>  0
<T T B>  2
<T T T>  0
<F B B>  1
<F B T>  1
<F B F>  1
<F F B>  3
<F F F>  0

Relation complement (8 elements):

<T T F>
<T F B>
<T F T>
<T F F>
<F T B>
<F T T>
<F T F>
<F F T>
```

Figure 4: The impossibility of separating Curien's A_1, A_2 and A_3.

```
# curien1.lam

iota

  B T F

functions

  If B _ _ = B
     T x _ = x
     F _ y = y

  A1 T F _ = T
     F _ T = F
     _ _ _ = B

  A2 _ T F = T
     T F _ = F
     _ _ _ = B

  A3 F _ T = T
     _ T F = F
     _ _ _ = B

constants

  B T F If

tests

  A1 = T
  A2 = F
  A3 = F
```

```
curien1

Stage: 2

Relation (21 elements):

<B B B>
<B B T>
<B B F>
<B T B>
<B T T>
<B T F>
<B F B>
<B F T>
<B F F>
<T B B>
<T B T>
<T B F>
<T T B>
<T T T>
<T F B>
<F B B>
<F B T>
<F B F>
<F T B>
<F F B>
<F F F>

Relation complement (6 elements):

<T T F>
<T F T>
<T F F>
<F T T>
<F T F>
<F F T>
```

Figure 5: The Berry-Plotkin function can be used to separate A_1, A_2 and A_3.

```
# curien3.lam

iota

  B T F

functions

  If B _ _ = B
     T x _ = x
     F _ y = y

  A1 T F _ = T
     F _ T = F
     _ _ _ = B

  A2 _ T F = T
     T F _ = F
     _ _ _ = B

  A3 F _ T = T
     _ T F = F
     _ _ _ = B

  BP T F _ = F
     _ T F = T
     F _ T = F
     _ _ _ = B

constants

  B T F If BP

tests

  A1 = T
  A2 = F
  A3 = F
```

```
curien3

Stage: 2

Term:

lambda x0.
    BP
        x0
            B
            T
            F
        x0
            T
            F
            B
        x0
            F
            B
            T
```

```
curien3

Stage: 2

Term:

lambda x0.
    BP     <T F F>
        x0    <B T F>
            B     <B B B>
            T     <T T T>
            F     <F F F>
        x0    <T F B>
            T     <T T T>
            F     <F F F>
            B     <B B B>
        x0    <F B T>
            F     <F F F>
            B     <B B B>
            T     <T T T>
```

verbose (right column) modes. The output indicates that the term

$$H = \lambda x_0.\, \mathrm{BP}\,(x_0\, \Omega\, \mathrm{tt}\, \mathrm{ff})\,(x_0\, \mathrm{tt}\, \mathrm{ff}\, \Omega)\,(x_0\, \mathrm{ff}\, \Omega\, \mathrm{tt})$$

(the $\perp$'s have been replaced by Ω's) was found after two stages of the closure process. The verbose version of the program's output shows that the result triple $\langle \mathrm{tt}, \mathrm{ff}, \mathrm{ff}\rangle$ became paired with the body of H at stage 2 of the closure process since

$$\langle \mathrm{tt}, \mathrm{ff}, \mathrm{ff}\rangle = \mathrm{BP}\,\langle \perp, \mathrm{tt}, \mathrm{ff}\rangle\,\langle \mathrm{tt}, \mathrm{ff}, \perp\rangle\,\langle \mathrm{ff}, \perp, \mathrm{tt}\rangle$$

and the triples $\langle \perp, \mathrm{tt}, \mathrm{ff}\rangle$, $\langle \mathrm{tt}, \mathrm{ff}, \perp\rangle$ and $\langle \mathrm{ff}, \perp, \mathrm{tt}\rangle$ were paired with the terms $x_0\, \Omega\, \mathrm{tt}\, \mathrm{ff}$, $x_0\, \mathrm{tt}\, \mathrm{ff}\, \Omega$ and $x_0\, \mathrm{ff}\, \Omega\, \mathrm{tt}$, respectively, at stage 1. Similarly, the triple $\langle \perp, \mathrm{tt}, \mathrm{ff}\rangle$ is paired with the term $x_0\, \Omega\, \mathrm{tt}\, \mathrm{ff}$ at stage 1 since

$$\langle \perp, \mathrm{tt}, \mathrm{ff}\rangle = \langle A_1, A_2, A_3\rangle\,\langle \perp, \perp, \perp\rangle\,\langle \mathrm{tt}, \mathrm{tt}, \mathrm{tt}\rangle\,\langle \mathrm{ff}, \mathrm{ff}, \mathrm{ff}\rangle$$

and the constantly $\perp$, tt and ff triples were paired with the terms Ω, tt and ff at stage 0.

As a final example, we consider the problem of determining whether there is a definable element of type $\iota^3 \to \iota$ of the monotone function model $\mathcal{F}$ of Finitary PCF that sends an argument x to tt, if $x \sqsupseteq A_i$ for some i, and sends x to $\perp$, otherwise. Since there are many elements of F_{ι^3} that don't dominate any of the A_i's, *lambda* can't be used in a purely mechanical way to solve this problem.

One can, however, use *lambda* to solve lambda definability problems that specify that certain hand-picked functions must be sent to $\perp$. A bit of experimentation (see `curien4.lam` and `curien5.lam` in *lambda*'s distribution) lead to the problem of Figure 6, which specifies that parallel or, parallel and, and their "negations" should be sent to $\perp$. Running the program generated from this problem by *lambda* takes a considerable amount of time (about eight hours of cpu time on a Sun 690MP) and produces the term $\lambda x_0.\, G$, where

$$\begin{aligned}
G &= x_0\, L\, M\, N \\
L &= \mathrm{If}\, Z\, (\mathrm{If}\, Y\, \Omega\, \mathrm{ff})\, (\mathrm{If}\, X\, \mathrm{tt}\, \Omega) \\
M &= \mathrm{If}\, X\, (\mathrm{If}\, Z\, \Omega\, \mathrm{ff})\, (\mathrm{If}\, Y\, \mathrm{tt}\, \Omega) \\
N &= \mathrm{If}\, Y\, (\mathrm{If}\, X\, \Omega\, \mathrm{ff})\, (\mathrm{If}\, Z\, \mathrm{tt}\, \Omega) \\
X &= x_0\, \mathrm{tt}\, \mathrm{ff}\, \Omega \\
Y &= x_0\, \Omega\, \mathrm{tt}\, \mathrm{ff} \\
Z &= x_0\, \mathrm{ff}\, \Omega\, \mathrm{tt}.
\end{aligned}$$

By considering the possible values of X, Y and Z, it is straightforward to show that G produces tt iff x_0 dominates one of the A_i's or is the constantly tt function. Furthermore, the term $\lambda x_0.\, H$, where

$$H = x_0\, M\, N\, L,$$

is produced as the solution of the variation of this problem (called `curien7.lam` in *lambda*'s distribution) that specifies that the A_i's should be sent to ff rather

Figure 6: Synthesis of a term sending the A's to tt and parallel or, parallel and, and their negations to $\perp$.

```
# curien6.lam

iota

  B T F

functions

  If B _ _ = B
     T x _ = x
     F _ y = y

  A1 T F _ = T
     F _ T = F
     _ _ _ = B

  A2 _ T F = T
     T F _ = F
     _ _ _ = B

  A3 F _ T = T
     _ T F = F
     _ _ _ = B

  POr F F F = F
      T _ _ = T
      _ T _ = T
      _ _ T = T
      _ _ _ = B

  NPOr F F F = T
       T _ _ = F
       _ T _ = F
       _ _ T = F
       _ _ _ = B

  PAnd T T T = T
       F _ _ = F
       _ F _ = F
       _ _ F = F
       _ _ _ = B

  NPAnd T T T = F
        F _ _ = T
        _ F _ = T
        _ _ F = T
        _ _ _ = B

constants

  B T F If

tests

  A1    = T
  A2    = T
  A3    = T
  POr   = B
  NPOr  = B
  PAnd  = B
  NPAnd = B
```

than to tt, and H produces ff iff x_0 dominates one of the A_i's or is the constantly ff function. Thus, it is easy to see that the term

$$Q = \lambda x_0.\,\mathrm{If}\, G\, (\mathrm{If}\, H\, \Omega\, \mathrm{tt})\, \Omega$$

solves the problem of sending an argument to tt, when it dominates one of the A_i's, and sending the argument to $\perp$, otherwise.

Interestingly, I wasn't able to generate such a term as the solution to a single lambda definability problem. One obstacle to my doing so was the necessity of employing at most seven tests, since it would take weeks rather than hours to solve a problem with eight tests. In any event, there is no chance of producing Q itself in such a way, since its body has depth six and there is another known solution to the problem whose body has depth five.

Acknowledgments

It is a pleasure to acknowledge many fruitful discussions with Achim Jung. Conversations with Antonio Bucciarelli, Shai Geva, Alan Jeffrey, Ralph Loader and Edmund Robinson were also helpful.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. L. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[Cur93] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhäuser, 1993.

[JS93] A. Jung and A. Stoughton. Studying the fully abstract model of PCF within its continuous function model. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 1993.

[JT93] A. Jung and J. Tiuryn. A new characterization of lambda definability. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 245–257. Springer-Verlag, 1993.

[Loa94] R. Loader. The undecidability of λ-definability. In M. Zeleny, editor, *The Church Festschrift*. CSLI/University of Chicago Press, 1994.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 367–458. Elsevier Science Publishers, 1990.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.

[Plo80] G. D. Plotkin. Lambda-definability in the full type hierarchy. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–374. Academic Press, 1980.

[Sie92] K. Sieber. Reasoning about sequential functions via logical relations. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *LMS Lecture Note Series*, pages 258–269. Cambridge University Press, 1992.

Some Quasi-Varieties of Iteration Theories

Stephen L. Bloom[*]
Stevens Institute of Technology
Department of Computer Science
Hoboken, NJ 07030
bloom@gauss.stevens-tech.edu

Zoltán Ésik[†]
A. József University
Department of Computer Science
Szeged, Hungary
h754esi@ella.hu

Abstract

All known structures involving a constructively obtainable fixed point (or iteration) operation satisfy the equational laws defining iteration theories. Hence, there seems to be a general equational theory of iteration. This paper provides evidence that there is no general *implicational* theory of iteration. In particular, the quasi-variety generated by the continuous ordered theories, in which fixed point equations have least solutions, is incomparable with the quasi-variety generated by the pointed iterative theories, in which fixed point equations have unique solutions.

1 Introduction

Iteration theories were introduced in 1980 by Bloom, Elgot and Wright, and independently by Z. Ésik, in order to formalize the equational properties of the stepwise behavior of flowchart algorithms and to provide a calculus for solving systems of fixed point equations. Iteration theories, which are (Lawvere) algebraic theories enriched by a fixed point operation, have basic operations which, in the flowchart setting, denote composition, a case statement and a looping or iteration operation. It now appears that the equational laws of iteration theories are quite comprehensive. It has been shown that in all structures that have been used as semantic models, the equational properties of the fixed point operation are captured by the axioms describing iteration theories. These structures include

- the (equivalence classes) of the flowchart schemes themselves
- ω-continuous algebras
- theories of partial functions

[*]Partially supported by a joint grant from the NSF and the Hungarian Academy of Science
[†]Partially supported by a grant from the National Foundation for Scientific Research of Hungary and a joint grant from the NSF and the Hungarian Academy of Science

- finitary and infinitary regular languages
- trees and synchronization trees
- the continuous functors involved in the specification of circular data types

and others.

Thus, the notion of iteration theory appears to be a unifying concept in many areas of theoretical computer science. We think it is important therefore to investigate various aspects of this notion. Equational axioms for iteration theories were given in [13, 14, 21, 22, 9, 16]. All of these sets of axioms involve a complicated equation scheme that we call the ***commutative identity***. For example, in [13], other than the commutative identity, there are three equational schemes: the left and right zero identities and the pairing identity (see below).

Most of the known examples of iteration theories which are closely related to natural models of computation satisfy a simple implication scheme, the ***functorial dagger implication***, which is much easier to establish than the commutative identity and which in fact implies the commutative identity. The quasi-variety FD of structures which are models of the functorial dagger implication, the zero identities, and the pairing identity, has the property that the least equational class containing FD is the class of all iteration theories. This fact is closely related to the fact, recently discovered independently by K.B. Arkhangelskii and P.V. Gorshkov [1], D. Kozen [18] and D. Krob [19] that the regular sets have simple finite implicational axiomatizations, although they have no finite equational axiomatization.

One might ask whether there is a general implicational theory of iteration, as general as the equational axioms determining the variety of iteration theories. In order to answer this question, we investigated the implicational theories of a number of quasi-varieties which are subclasses of the class of all iteration theories. Many of these quasi-varieties are of interest in themselves. Further, each has the property that the least variety it generates is either the variety of all iteration theories or the variety of all iteration theories with a unique morphism $1 \longrightarrow 0$. As is shown below, apparently there is no general implicational theory applicable to all of our examples. In particular, the quasi-variety Ω in which systems of fixed point equations have **least solutions**, and the quasi-variety PI in which (nontrivial) systems of fixed point equations have **unique solutions**, have incomparable implicational theories.

2 Preliminaries

In this section, we give the precise definitions needed to understand the later results. Familiarity with [7] or [8] would be helpful. We will use the following notation. For $n \geq 0$, the set $[n]$ is

$$[n] = \{1, 2, \ldots, n\}.$$

In any category, the composite of morphisms $f : X \to Y$ and $g : Y \to Z$ is written $f \cdot g : X \to Z$.

We prefer the following definition of an algebraic theory.

Definition 2.1 *An* **algebraic theory** *is a category T whose objects are the nonnegative integers n, $n \geq 0$. For each $n \geq 0$, there are n* distinguished morphisms

$$i_n : 1 \longrightarrow n$$

with the following coproduct property. For any family of morphisms $f_i : 1 \longrightarrow p$, for $i \in [n]$, there is a unique morphism $f : n \longrightarrow p$ such that

$$i_n \cdot f \quad = \quad f_i, \tag{1}$$

for each $i \in [n]$. A **morphism of algebraic theories** *$\varphi : T \longrightarrow T'$ is a functor which preserves objects and distinguished morphisms, i.e., $n\varphi = n$ and $i_n\varphi = i_n$, for all $n \geq 0$ and all $i \in [n]$.*

The morphism f determined by (1) is called the *source tupling* of the morphisms f_i, and is written

$$f \quad = \quad \langle f_1, \ldots, f_n \rangle.$$

In the case that $n = 0$, the condition (1) amounts to the requirement that there is a unique morphism $0_p : 0 \longrightarrow p$, for each p. When $n = 1$, we always assume that $f_1 = \langle f_1 \rangle$. For any $n \geq 0$, the identity morphism $n \longrightarrow n$ will be denoted using boldface by $\mathbf{1}_n$. Note that $\mathbf{1}_n = \langle 1_n, 2_n, \ldots, n_n \rangle$.

Suppose that T is an algebraic theory. For each (set theoretic) function $f : [n] \longrightarrow [p]$, there is a "base morphism" $f' : n \longrightarrow p$ defined as the source tupling of the distinguished morphisms $(if)_p : 1 \longrightarrow p$, $i \in [n]$. When T is nontrivial, i.e., when there are at least 2 morphisms $1 \longrightarrow 2$ in T, the map from functions to base morphisms is injective. We will usually identify base morphisms $n \longrightarrow p$ with functions $[n] \longrightarrow [p]$. A base morphism is called surjective or a permutation, etc., when the corresponding function has that property.

The coproduct properties of theories imply that for any pair of morphisms $f : n \longrightarrow p$ and $g : m \longrightarrow p$ in T, there is a unique morphism $\langle f, g \rangle : n+m \longrightarrow p$ such that $\kappa \cdot \langle f, g \rangle = f$ and $\lambda \cdot \langle f, g \rangle = g$, where $\kappa : n \longrightarrow n+m$ and $\lambda : m \longrightarrow n+m$ are base morphisms corresponding to the inclusion and translated inclusion functions. The morphism $\langle f, g \rangle : n+m \longrightarrow p$ is called the **source pairing** of f, g.

For any pair of morphisms $f : n \longrightarrow p$ and $g : m \longrightarrow q$, we write $f \oplus g$ for the morphism $\langle f \cdot \kappa, g \cdot \lambda \rangle : n+m \longrightarrow p+q$,, where now $\kappa : p \longrightarrow p+q$ and $\lambda : q \longrightarrow p+q$. The morphism $f \oplus g$ is called the **separated sum** of f, g.

Definition 2.2 *A* **preiteration theory** *T is an algebraic theory enriched by an iteration operation $f \mapsto f^\dagger$, where $f : n \longrightarrow n+p$ and $f^\dagger : n \longrightarrow p$. A* **morphism of preiteration theories** *$\varphi : T \longrightarrow T'$ is a theory morphism which preserves the iteration operation, i.e., $f^\dagger\varphi = (f\varphi)^\dagger$, for all $f : n \longrightarrow n+p$ in T.*

The operation † need not satisfy any properties.

Definition 2.3 *An* **iteration theory** *is a preiteration theory in which the iteration operation satisfies the following four identities: (see [13])*

- LEFT ZERO IDENTITY

$$(0_n \oplus f)^\dagger = f,$$

all $f : n \longrightarrow p$

- RIGHT ZERO IDENTITY

$$(f \oplus 0_q)^\dagger = f^\dagger \oplus 0_q,$$

all $f : n \longrightarrow n + p$.

- PAIRING IDENTITY

$$\langle f, g\rangle^\dagger = \langle f^\dagger \cdot \langle h^\dagger, \mathbf{1}_p\rangle,\ h^\dagger\rangle$$

all $f : n \longrightarrow n + m + p,\ g : m \longrightarrow n + m + p$, *where*

$$h := g \cdot \langle f^\dagger,\ \mathbf{1}_{m+p}\rangle.$$

- COMMUTATIVE IDENTITY

$$\langle \mathbf{1}_m \cdot \rho \cdot f \cdot (\rho_1 \oplus \mathbf{1}_p), \ldots, m_m \cdot \rho \cdot f \cdot (\rho_m \oplus \mathbf{1}_p)\rangle^\dagger = \rho \cdot (f \cdot (\rho \oplus \mathbf{1}_p))^\dagger,$$

all $f : n \longrightarrow m + p$, *surjective base* $\rho : m \longrightarrow n$, *and base* $\rho_i : m \longrightarrow m,\ i \in [m]$, *such that* $\rho_i \cdot \rho = \rho$.

The above four identities imply the following two:

- FIXED POINT IDENTITY

$$f^\dagger = f \cdot \langle f^\dagger, \mathbf{1}_p\rangle,$$

all $f : n \longrightarrow n + p$.

- PERMUTATION IDENTITY

$$(\pi \cdot f \cdot (\pi^{-1} \oplus \mathbf{1}_p))^\dagger = \pi \cdot f^\dagger,$$

for all $f : n \longrightarrow n + p$ and all base permutations $\pi : n \longrightarrow n$.

The commutative identity is the axiom which is most difficult to verify (and to understand!). By replacing it with certain implications, we will obtain some quasi-varieties which generate the class of all iteration theories.

First, we give a name to a simpler group of identities.

Definition 2.4 *A* **Conway theory** *is a preiteration theory which satisfies the zero identities, the pairing identity and the permutation identity.*

The term Conway theory is due to the fact that in matrix preiteration theories, an equivalent set of identities is given by the familiar star sum and product identities which were studied by Conway [10]. See also [22, 5, 6].

$$\begin{aligned}(a+b)^* &= (a^*b)^*a^* \\ (ab)^* &= 1 + a(ba)^*b\end{aligned}$$

In any Conway theory we define $\perp := \mathbf{1}_1{}^\dagger : 1 \longrightarrow 0$ and $\perp_{np} := \langle \perp \cdot 0_p, \ldots, \perp \cdot 0_p \rangle$. It follows from the Conway theory axioms that

$$\perp_{np} \ = \ (\mathbf{1}_n \oplus 0_p)^\dagger,$$

for all $n, p \geq 0$.

Now we describe two implication schemes.

- Functorial dagger implication

$$f \cdot (\rho \oplus \mathbf{1}_p) = \rho \cdot g \ \Rightarrow \ f^\dagger = \rho \cdot g^\dagger,$$

for all $f : n \longrightarrow n+p$, $g : m \longrightarrow m+p$ and surjective base $\rho : n \longrightarrow m$. (When the implication holds for all morphisms ρ in some class $\mathcal{C}$, we say that the theory satisfies the **functorial dagger** for $\mathcal{C}$.)

- The GA-implication

$$f^{\dagger\dagger} = g^{\dagger\dagger} \ \Rightarrow \ (g \cdot \langle f^\dagger, \mathbf{1}_{1+p} \rangle)^\dagger = f^{\dagger\dagger},$$

for all $f, g : 1 \longrightarrow 2+p$.

Note that both the functorial dagger implication and the GA-implication in fact consist of infinitely many implications. The GA-implication was introduced in the setting of matrix theories in [1] to give a set of implicational axioms for the regular sets.

It is easy to verify the following fact.

Proposition 2.5 [13] *If T is a preiteration theory which satisfies the functorial dagger implication, then T satisfies the commutative identity. Hence any Conway theory which satisfies the functorial dagger implication is an iteration theory.* □

The next proposition was proved in [8].

Proposition 2.6 *If T is a Conway theory which satisfies the GA-implication, then T also satisfies the functorial dagger implication.* □

The first class of iteration theories we describe is a class of ordered theories.

Definition 2.7 • *An* **ordered algebraic theory** T *is an algebraic theory such that for each pair n, p of nonnegative integers, the set $T(n,p)$ is equipped with a partial order. The order on $T(n,p)$ will be written $f \leq g : n \longrightarrow p$. The theory operations respect the ordering: if $f_1 \leq f_2 : n \longrightarrow p$ and $g_1 \leq g_2 : p \longrightarrow q$ then*

$$f_1 \cdot g_1 \leq f_2 \cdot g_2.$$

Further, if $f_i \leq g_i : 1 \longrightarrow p$, for each $i \in [n]$, then

$$\langle f_1, \ldots, f_n \rangle \leq \langle g_1, \ldots, g_n \rangle.$$

- *A* **pointed ordered theory** *is an ordered theory which is pointed; i.e., there is a distinguished morphism $\perp : 1 \longrightarrow 0$; as usual, we define $\perp_{1p}$ as $\perp \cdot 0_p$, for all $p \geq 0$, and $\perp_{np}$ as $\langle \perp_{1p}, \ldots, \perp_{1p} \rangle$, for $n \neq 1$. Furthermore, the morphisms $\perp_{np}$ are the* least *elements in $T(n,p)$. Note that composition in pointed theories is* **left strict***:*

$$\perp_{np} \cdot f = \perp_{nq},$$

for all $f : p \longrightarrow q$.

- *A pointed ordered theory T is* ω**-continuous** *if each hom-set $T(n,p)$ is an ω-complete poset and if composition is also ω-continuous:*

$$\begin{aligned} (\sup_n f_n) \cdot g &= \sup_n f_n \cdot g \\ f \cdot (\sup_n g_n) &= \sup_n f \cdot g_n, \end{aligned}$$

for ω-chains $(f_n), (g_n)$, where $f_n : m \longrightarrow p$ and $g_n : p \longrightarrow q$, $n \geq 0$, and for $f : m \longrightarrow p$, $g : p \longrightarrow q$.

(The importance of certain kinds of ordered theories for semantics, in particular the ω-continuous theories, was emphasized by the ADJ group (J. Goguen, J. Thatcher, E.G. Wagner and J. Wright) in a number of papers [24, 17, 23].) In [8] it is shown that each ω-continuous theory is an iteration theory, where for $f : n \longrightarrow n + p$,

$$f^\dagger := \sup_k f^k \cdot \langle \perp_{np}, \mathbf{1}_p \rangle.$$

The powers f^k of f are defined as follows:

$$f^0 := \mathbf{1}_n \oplus 0_p \qquad (2)$$

$$f^{k+1} := f \cdot \langle f^k, 0_n \oplus \mathbf{1}_p \rangle. \qquad (3)$$

Thus, in ω-continuous theories, $f^\dagger$ is the least solution to the iteration equation for f:

$$\xi = f \cdot \langle \xi, \mathbf{1}_p \rangle.$$

Now we recall a class of theories first introduced by Elgot [11].

Definition 2.8 *An* **ideal theory** *is an algebraic theory T with the property that each morphism $1 \longrightarrow p$ in T is either a distinguished morphism i_p or is* ideal. *An* **ideal morphism** *$f : 1 \longrightarrow p$ is a morphism with the property that for each $g : p \longrightarrow q$, the composite $f \cdot g : 1 \longrightarrow q$ is not distinguished. An* **iterative theory** *is an ideal theory such that for each ideal morphism $f : 1 \longrightarrow 1 + p$, there is a unique morphism $f^\dagger : 1 \longrightarrow p$ such that*

$$f^\dagger \;=\; f \cdot \langle f^\dagger, 1_p \rangle.$$

Proposition 2.9 [3] *If $t_0 : 1 \longrightarrow 0$ is any morphism in an iterative theory T, there is a unique extension of the operation † from the ideal morphisms to all morphisms such that T becomes an iteration theory satisfying $\mathbf{1}_1{}^\dagger = t_0$. The resulting iteration theory is denoted $(T, \mathbf{1}_1{}^\dagger = t_0)$.* □

An iteration theory $(T, \mathbf{1}_1{}^\dagger = t_0)$, where T is an iterative theory, is called a **pointed iterative theory**. In [8], it is shown that any pointed iterative theory satisfies the GA-implication.

Tree theories are examples of pointed iterative theories as well as ω-continuous theories.

Example 2.10 Theories of trees. Let Σ be a *signature*, i.e., $\Sigma = \bigcup_{n \geq 0} \Sigma_n$ is the union of the pairwise disjoint sets Σ_n. Suppose the set $V = \{x_1, x_2, \ldots\}$ is disjoint from Σ. In the theory $\Sigma\,\mathbf{TR}$, a morphism $1 \rightarrow p$ is a Σ-tree $t : 1 \rightarrow p$. (A Σ-tree $t : 1 \longrightarrow p$ is a partial function whose domain is a nonempty prefix closed subset of the set $[\omega]^*$ of finite sequences of positive integers. The target of t is the set $\Sigma \cup \{x_1, \ldots, x_p\}$. Further, if $u \in \operatorname{dom} t$ and $u\,t \in \Sigma_n$ then $ui \in \operatorname{dom} t$ iff $i \in [n]$; also, if $u\,t \in \Sigma_0 \cup \{x_1, \ldots, x_p\}$, then u is a leaf; i.e., ui is not in $\operatorname{dom} t$, for any $i > 0$. See [12] for a thorough study of the algebraic theory of trees.) We identify the varible x_i with the partial function defined only on the empty word λ with value x_i. Similarly, we identify $\sigma \in \Sigma_n$ with the partial function defined on λ and the length one sequences $1, \ldots, n$ as follows: $\lambda\,\sigma := \sigma$; $i\,\sigma := x_i$, $i \in [n]$.

If $n \neq 1$, a morphism $n \rightarrow p$ in $\Sigma\,\mathbf{TR}$ is an n-tuple $(t_1, \ldots, t_n)$ of morphisms $1 \rightarrow p$. The **composite** of $t : 1 \rightarrow p$ with $s = (s_1, \ldots, s_p) : p \rightarrow q$ is the tree obtained by attaching the tree s_i to each leaf of t labeled x_i, $i \in [p]$. When $n \neq 1$, the composite of $t = (t_1, \ldots, t_n) : n \rightarrow p$ with $s : p \rightarrow q$ is defined as

$$t \cdot s \;:=\; (t_1 \cdot s, \ldots, t_n \cdot s).$$

The distinguished morphism i_n is the tree $x_i : 1 \rightarrow n$, for each $i \in [n]$.

Note that if $f : 1 \longrightarrow 1 + p$ is any tree other than 1_{1+p}, there is a unique tree $f^\dagger : 1 \longrightarrow p$ such that

$$f^\dagger \;=\; f \cdot \langle f^\dagger, 1_p \rangle.$$

Thus, $\Sigma\,\mathbf{TR}$ is an iterative theory.

If $\perp$ is a letter not in the set Σ, let $\Sigma_\perp$ denote the signature obtained by adding $\perp$ to Σ_0. The pointed iterative theory

$$(\Sigma_\perp\,\mathbf{TR},\ \mathbf{1}_1{}^\dagger = \perp)$$

is an ω-continuous ordered theory, where the ordering can be described as follows: $f \leq g$ if g can be obtained from f by replacing some occurrences of $\perp$ by other trees. In fact, $(\Sigma_\perp\,\mathbf{TR},\ \mathbf{1}_1{}^\dagger = \perp)$ is the free ω-continuous theory on Σ (see [24, 17]).

A tree theory is a pointed iterative theory

$$(\Sigma_\perp \mathbf{TR}, 1_1{}^\dagger = \perp),$$

where the "point" $\perp$ is the new atomic letter of rank 0. This theory is usually written $\Sigma_\perp \mathbf{TR}$, for short.

We will occasionally make use of the subiteration theory $\Sigma\mathbf{tr}$ of $\Sigma_\perp \mathbf{TR}$, which consists of the **regular** trees. Recall that a tree $t : 1 \longrightarrow p$ is regular if it has a finite number of subtrees. When $t : n \longrightarrow p$, for some $n \neq 1$, t is regular if the components $i_n \cdot t$ are regular, for all $i \in [n]$. According to the definition in [3, 4] that $\Sigma\mathbf{tr}$ can be characterized as the iteration theory freely generated by Σ.

We mention two other important classes of ω-continuous theories.

Example 2.11 The theory $\mathbf{Rel}_A$ has as morphisms $n \longrightarrow p$ all relations

$$A \times [n] \quad \longrightarrow \quad A \times [p].$$

Composition in the theory is composition of relations. Identifying A with $A \times [1]$, the distinguished morphism $i_n : 1 \longrightarrow n$, is the function, considered as a relation,

$$\begin{array}{ccc} A & \longrightarrow & A \times [n] \\ a & \mapsto & (a, i). \end{array}$$

The tupling $f := \langle f_1, \ldots, f_n \rangle$ of the morphisms $f_i : 1 \longrightarrow p$ is the relation

$$\begin{array}{ccc} A \times [n] & \longrightarrow & A \times [p] \\ (a, i)\ f\ (a', j) & \Leftrightarrow & a\ f_i\ (a', j). \end{array}$$

With the standard ordering of relations, for any relation $f : A \times [n] \longrightarrow A \times [n + p]$, there is a least relation $f^\dagger : A \times [n] \longrightarrow A \times [p]$ which satisfies

$$f^\dagger \quad = \quad f \cdot \langle f^\dagger, 1_p \rangle.$$

The theory $\mathbf{Rel}_A$ is also an ω-continuous theory. The morphism $\perp_{np}$ in $\mathbf{Rel}_A$ is the empty relation $n \longrightarrow p$.

The theory $\mathbf{Pfn}_A$ is the subtheory of $\mathbf{Rel}_A$ whose morphisms $n \longrightarrow p$ are the partial functions $A \times [n] \longrightarrow A \times [p]$. The distinguished morphisms and the theory and iteration operations are the same as those in $\mathbf{Rel}_A$.

The notion of an *iteration term* is defined in the expected way as a formal expression which denotes a morphism in a (pre)iteration theory. These terms are constructed from an infinite set of variables for morphisms $n \longrightarrow p$, for each $n, p \geq 0$, constants for each of the distinguished morphisms, and the operation symbols for composition, tupling and dagger. (Source pairing and separated sum are understood as abbreviations.)

For our purposes, an *implication* or *quasi-identity* is an expression of the form

$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n \quad \Rightarrow \quad s_{n+1} = t_{n+1},$$

where $n \geq 0$ and s_i, t_i are iteration terms. Note that when $n = 0$, an implication is an equation. Each instance of the functorial dagger or GA-implication is an implication. We understand an implication to be **satisfied** by, or **true** or **valid** in a preiteration theory T if the implication is true for any interpretation of the variables as morphisms in T with the appropriate source and target.

If K is any class of preiteration theories, let *Imp*(K) denote the collection of all implications valid in each theory in K. For any set I of implications, let *Mod*(I) denote all preiteration theories in which each implication in I is true.

Suppose that K is a class of iteration theories.

Definition 2.12 *The* **quasi-variety generated by** K, *in symbols* $Qv(K)$, *is the class* *Mod*(*Imp*(K)), *the class of all iteration theories which satisfy all implications valid in all theories in* K. K *is a* **quasi-variety** *if* $K = Qv(K)$.

Note that if $K \subseteq K'$, then $Qv(K) \subseteq Qv(K')$. If I is some set of implications, then the class of all models of I is a quasi-variety.

3 Some Quasi-Varieties of Iteration Theories

Aside from *IT*, the variety of all iteration theories, we will be considering the following quasi-varieties.

- V_t, the quasi-variety generated by the class of tree theories.
- V_y, the quasi-variety generated by the class of theories

$$(\Sigma\,\mathbf{TR},\ \mathbf{1}_1{}^{\dagger} = t_0).$$

- *PI*, the quasi-variety generated by all pointed iterative theories.
- *PFN*, the quasi-variety generated by the theories $\mathbf{Pfn}_A$.
- *REL*, the quasi-variety generated by the theories $\mathbf{Rel}_A$.
- Ω, the quasi-variety generated by the class of all ω-continuous theories.
- Ω_0, the quasi-variety generated by all ω-continuous theories with a unique morphism $1 \longrightarrow 0$.
- *MAT*, the quasi-variety generated by all matrix iteration theories [8, 5].
- *GA*, the collection of Conway theories which satisfy the GA-implication.
- *FD*, the collection of Conway theories which satisfy the functorial dagger implication.
- Manes [20] has called a morphism $h : n \longrightarrow p$ in a preiteration theory *pure* if $h \cdot \mathbf{1}_p{}^{\dagger} = \mathbf{1}_n{}^{\dagger}$. Note that any base morphism is pure. Let FD_p denote the collection

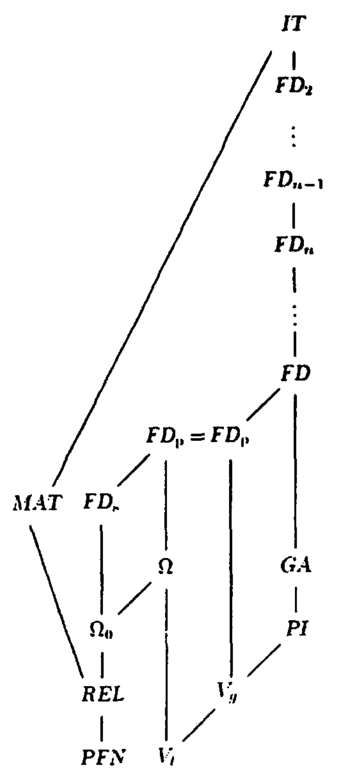


Figure 1: The Quasi-Variety Poset

of Conway theories which satisfy the functorial dagger implication for the class of all pure morphisms, namely, the implication

$$h \cdot 1_p{}^\dagger = 1_n{}^\dagger \ \wedge \ f \cdot (h \oplus 1_p) = h \cdot g \quad \Rightarrow \quad f^\dagger = h \cdot g^\dagger,$$

for all $f : n \longrightarrow n+p$, $g : m \longrightarrow m+p$, and $h : n \longrightarrow m$.

- Lastly, we let FD_κ denote the quasi-variety of all Conway theories which satisfy the functorial dagger implication for the class of all morphisms.

4 The Results

We will prove that if the quasi-varieties are ordered by set inclusion, they form a poset whose structure is indicated in Figure 1.

Each of the inclusions is strict. If there is no chain of inclusions from one quasi-variety to another, the two are incomparable with respect to inclusion. Thus, we have a complete description of the poset of these classes.

Further, the variety of iteration theories generated by V_t, and all of the quasi-varieties above it, is IT. The variety generated by PFN, REL, Ω_0, FD_κ, and MAT is the variety of all iteration theories with a unique morphism $1 \longrightarrow 0$.

In addition, we will show that there is an infinite chain of quasi-varieties

$$FD \subset \ldots \subset FD_3 \subset FD_2 \subset FD_1 = IT$$

between FD and IT. FD_n is the quasi-variety of all iteration theories which satisfy the functorial dagger implication for the class of base morphisms $k \longrightarrow 1$, for $1 \leq k \leq n$.

5 Inclusions

It is clear that the following inclusions hold:

$$\begin{array}{lllllll} V_t & \subseteq & V_g & \subseteq & PI & & \\ FD_\kappa & \subseteq & FD_0 & \subseteq & FD & & \\ PFN & \subseteq & REL & \subseteq & \Omega_0 & \subseteq & \Omega. \end{array}$$

and each class is contained in the class IT of all iteration theories. The inclusions

$$\begin{array}{lllll} PI & \subseteq & GA & \subseteq & FD \\ \Omega_0 & \subseteq & FD_\kappa & & \\ REL & \subseteq & MAT & & \end{array}$$

are proved in [8], and the inclusion

$$V_t \subseteq \Omega$$

is known from [24]. We will show now that each of the inclusions indicated in Figure 1, page 11, is proper, and that two quasi-varieties are incomparable unless there is a chain of inclusions from one to the other. It was proved in [15] that $FD \neq IT$. We will give two new independent arguments for this fact below.

The organization of the argument follows the shape of Figure 1. We proceed first up the right side, and continue to the left.

5.1 $V_t \subset V_g$

We need show only that $V_t \neq V_g$. Consider the following implication.

$$f \cdot g = \mathbf{1}_1{}^\dagger \quad \Rightarrow \quad f^\dagger = \mathbf{1}_1{}^\dagger, \quad \text{all } f : 1 \longrightarrow 1,\ g : 1 \longrightarrow 0. \tag{4}$$

This implication is valid in all tree theories, since in the tree theory

$$(\Sigma_\perp \mathbf{TR},\ \mathbf{1}_1{}^\dagger = \perp),$$

the tree $\mathbf{1}_1{}^\dagger$ is the atomic tree $\perp$. Thus, if $f \cdot g = \perp$, then either $f = \mathbf{1}_1$ (and $g = \perp$) or $f = \perp \cdot 0_1$. However, if Σ contains a letter $f \in \Sigma_1$ and a letter g in Σ_0, the implication fails in the theory

$$(\Sigma_\perp \mathbf{TR},\ \mathbf{1}_1{}^\dagger = f \cdot g).$$

5.2 $V_g \subset PI$

The inclusion $V_g \subseteq PI$ holds since every theory $(\Sigma\mathbf{TR}, \mathbf{1}_1{}^\dagger = t_0)$ is a pointed iterative theory. Also the following implication is valid in these theories:

$$f \cdot f = g \cdot g \quad \Rightarrow \quad f = g, \quad f, g : 1 \longrightarrow 1. \tag{5}$$

But we show that (5) is not valid in an iterative theory $\mathbf{TTF}_X$ of timed terminal functions on X [11]. Indeed, writing $\mathbf{N}$ for the set of nonnegative integers, let $X = \{a, b\}$ and let $f, g : X \times \mathbf{N} \longrightarrow X \times \mathbf{N}$ be the timed terminal functions defined by

$$\begin{aligned} (a,n)f &:= (b, n+1) \\ (b,n)f &:= (a, n+1) \\ (x,n)g &:= (x, n+1), \quad x \in \{a, b\}. \end{aligned}$$

Then $(x,n)f^2 = (x,n)g^2 = (x, n+2)$, but $f \neq g$.

5.3 $PI \subset GA$

It was shown in [8] that $PI \subseteq GA$. In order to show the inclusion is strict, we will show that the following implication is valid in PI.

$$p \cdot q = \mathbf{1}_1 \quad \Rightarrow \quad \langle p,\ p \cdot \pi \rangle \cdot \langle p,\ p \cdot \pi \rangle = \mathbf{1}_2, \tag{6}$$

where $p : 1 \longrightarrow 2$, $q : 2 \longrightarrow 1$ and where $\pi := \langle 0_1 \oplus \mathbf{1}_1, \mathbf{1}_1 \oplus 0_1 \rangle$ is the nontrivial base permutation $2 \longrightarrow 2$. Indeed, in any ideal theory, if $p : 1 \longrightarrow 2$ and if $p \cdot q = \mathbf{1}_1$ for some $q : 2 \longrightarrow 1$, then either $p = \mathbf{1}_1 \oplus 0_1$ or $p = 0_1 \oplus \mathbf{1}_1$. Hence $f := \langle p,\ p \cdot \pi \rangle$ is either $\mathbf{1}_2$ or π. In either case $f \cdot f = \mathbf{1}_2$.

If S is any semiring, $\mathbf{Mat}_S$ is the theory whose morphisms $n \longrightarrow p$ are the n by p matrices with entries in S; matrix multiplication is the theory composition. For other details, see [5, 8]. When S is the semiring of regular subsets of A^*, we denote the corresponding matrix theory by $\mathbf{Reg}_A$. Now suppose that $T = \mathbf{Reg}_A$, and that p and q are the following matrices:

$$\begin{aligned} p &:= [1\ x] \\ q &:= \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \end{aligned}$$

where x is some nonempty regular subset of A^*. Then $p \cdot q = \mathbf{1}_1$. However if $f = \langle p,\ p \cdot \pi \rangle$, then

$$\begin{aligned} f &= \begin{bmatrix} 1 & x \\ x & 1 \end{bmatrix} \\ f \cdot f &= \begin{bmatrix} 1 + x^2 & x \\ x & 1 + x^2 \end{bmatrix} \\ &\neq \mathbf{1}_2. \end{aligned}$$

It is known from [1] that $\mathbf{Reg}_A$ satisfies the GA-implication. Hence, $\mathbf{Reg}_A$ is in $GA - PI$.

Remark 5.1 We note here that the collection of all pointed iterative theories does not form a quasi-variety since this collection is not closed under binary products. For the same reason, the collection of tree theories or theories $(\Sigma_\perp\mathbf{TR}, \mathbf{1}_1{}^\dagger = t_0)$ is not a quasi-variety.

5.4 $GA \subset FD$

It was shown in [8] that $GA \subseteq FD$. In order to show that the inclusion is strict, we apply the following extension of the Zero Congruence Lemma [8].

Lemma 5.2 *If θ is a zero congruence on the free iteration theory $\Sigma\mathbf{tr}$, then the theory $\Sigma\mathbf{tr}/\theta$ satisfies the functorial dagger implication.*

We give a proof of this fact, together with a concrete description of θ, in the Appendix.

Now define Σ as the signature having only two symbols f, g of rank 2. Let θ be the zero congruence generated by

$$f^{\dagger\dagger} \;\; \theta \;\; g^{\dagger\dagger}.$$

In the Appendix (Theorem 8.2) it is shown that two regular trees $1 \longrightarrow p$ are related by θ if one can be obtained from the other by replacing some subtrees of the form $f^{\dagger\dagger}$ by $g^{\dagger\dagger}$, and some subtrees $g^{\dagger\dagger}$ by $f^{\dagger\dagger}$. The theory $\Sigma\mathbf{tr}/\theta$ satisfies the functorial dagger implication, by the lemma, but does not satisfy the GA-implication. Indeed, if

$$h \;\; := \;\; g \cdot \langle f^\dagger, \mathbf{1}_1 \rangle,$$

then it is not the case that $h^\dagger \; \theta \; f^{\dagger\dagger}$, since $h^\dagger$ has no subtrees of the form $f^{\dagger\dagger}$ or $g^{\dagger\dagger}$.

5.5 $V_g \subset FD_p$

First, we prove the following lemma.

Lemma 5.3 *Suppose that*

$$T := (\Sigma\mathbf{TR}, \mathbf{1}_1{}^\dagger = t_0).$$

Then $T \in FD_p$.

We write $\perp$ for $\mathbf{1}_1{}^\dagger$, as usual, rather than t_0. In [4] it was shown that for any $f : n \longrightarrow n+p$ in T, $f^\dagger$ is given by a metric limit:

$$f^\dagger \;\; = \;\; \lim_{k \longrightarrow \infty} f^k \cdot (\perp_{n0} \oplus \mathbf{1}_p),$$

where f^k was defined in (2) and (3) above. Now, suppose that $h : n \longrightarrow m$ and that

$$f \cdot (h \oplus \mathbf{1}_p) \;\; = \;\; h \cdot g,$$

where $g : m \longrightarrow m + p$. Then it follows that for each $k \geq 1$,

$$f^k \cdot (h \oplus \mathbf{1}_p) = h \cdot g^k.$$

Hence, if h is pure,

$$\begin{aligned} f^k \cdot (\perp_{n0} \oplus \mathbf{1}_p) &= f^k \cdot (h \cdot \perp_{m0} \oplus \mathbf{1}_p) \\ &= f^k \cdot (h \oplus \mathbf{1}_p) \cdot (\perp_{m0} \oplus \mathbf{1}_p) \\ &= h \cdot g^k \cdot (\perp_{m0} \oplus \mathbf{1}_p). \end{aligned}$$

The result follows from the fact that

$$\begin{aligned} \lim_k h \cdot g^k \cdot (\perp_{m0} \oplus \mathbf{1}_p) &= h \cdot (\lim_k g^k \cdot (\perp_{m0} \oplus \mathbf{1}_p)) \\ &= h \cdot g^\dagger. \quad \square \end{aligned}$$

Corollary 5.4 $V_g \subseteq FD_p$. $\square$

To show that the inclusion $V_g \subseteq FD_p$ is proper, note that the theory $\mathbf{TTF}_X$ of timed terminal functions in section 5.2 is in PFN as well as in PI, and hence in FD_p. It was shown in that section that $\mathbf{TTF}_X$ is not in V_g.

5.6 $FD_p \subset FD$

We now find an iteration theory which satisfies the functorial dagger implication for all base morphisms but not for all pure morphisms. Let Σ be a signature which has just two symbols σ, τ in Σ_1, and which is empty otherwise. We define an iteration theory congruence $\sim$ on $\Sigma_\perp \mathbf{TR}$ as follows. For any trees $f, g : 1 \longrightarrow p$, $f \sim g$ iff

- both trees have a leaf labeled by some variable x_i, $i \in [p]$, and the set of all labels of all of the vertices of f are the same as those of g, or
- neither has leaf labeled by a variable, and both symbols σ, τ occur **infinitely often** in both f and g, or
- neither tree has a leaf labeled by a variable, and neither tree has both symbols σ, τ occurring infinitely often as vertex labels.

Of course, if $f, g : n \longrightarrow p$, where $n > 1$, then $f \sim g$ iff $i_n \cdot f \sim i_n \cdot g$, for all $i \in [n]$. Let T denote the quotient theory $\Sigma_\perp \mathbf{TR}/\sim$. Then the morphisms $1 \longrightarrow 0$ in T are the two congruence classes

$$[\perp] \quad \text{and} \quad [(\sigma \cdot \tau)^\dagger].$$

When $p \geq 1$, the morphisms $1 \longrightarrow p$ in T are the $4p + 2$ equivalence classes

$$[\perp \cdot 0_p],\ [(\sigma \cdot \tau)^\dagger \cdot 0_p],\ [i_p],\ [\sigma \cdot i_p],\ [\tau \cdot i_p],\ [\sigma \cdot \tau \cdot i_p]$$

for $i \in [p]$. It is easy to check that T satisfies the functorial dagger implication for any base ρ with target 1. It follows from [16], that T satisfies the functorial dagger implication for all base morphisms. Since

$$[\sigma] \cdot [\perp] = [\perp],$$

$[\sigma] : 1 \longrightarrow 1$ is pure. Also,

$$[\sigma \cdot \tau] \cdot [\sigma] = [\sigma] \cdot [\tau].$$

But $[\tau^\dagger] = [\perp]$, so that

$$\begin{aligned} [(\sigma \cdot \tau)^\dagger] &\neq [\perp] \\ &= [\sigma] \cdot [\tau^\dagger]. \end{aligned}$$

Hence T is in $FD - FD_p$.

5.7 $V_t \subset \Omega$

Again, we need show only that $V_t \neq \Omega$. Consider the implication

$$f^\dagger = 1_1{}^\dagger \quad \Rightarrow \quad f \cdot f = f, \quad \text{all } f : 1 \longrightarrow 1. \tag{7}$$

This implication is clearly valid in all tree theories, since in $\Sigma_\perp \mathbf{TR}$, if $f : 1 \longrightarrow 1$ and $f^\dagger = \perp$, then either $f = 1_1$ or $f = \perp \cdot 0_1$.

We show that if A is a set with at least 2 elements, the implication does not hold in $\mathbf{Pfn}_A$. Indeed, let f be a nontrivial permutation of A of order two. Then, since there is a unique morphism $1 \longrightarrow 0$ in $\mathbf{Pfn}_A$, $f^\dagger = 1_1{}^\dagger$. But $f \cdot f \neq f$.

5.8 $\Omega \subset FD_p$

Using an argument just like that given above for Lemma 5.3, replacing $\lim_k$ with $\sup_k$, it can be shown that $\Omega \subseteq FD_p$.

Now we show that the two quasi-varieties are distinct. The implication

$$f \cdot g = \perp \quad \Rightarrow \quad f^\dagger = \perp, \quad \text{all } f : 1 \longrightarrow 1,\ g : 1 \longrightarrow 0, \tag{8}$$

holds in Ω. Indeed, in any ω-continuous theory, if $f \cdot g = \perp$,

$$\begin{aligned} \perp &\leq f \cdot \perp \\ &\leq f \cdot g \\ &= \perp. \end{aligned}$$

Hence $f \cdot \perp = \perp$, which in turn implies $f^n \cdot \perp = \perp$, all $n \geq 1$. Thus $f^\dagger = \sup_n f^n \cdot \perp = \perp$.

However, the implication (8) fails in the theory

$$T \;:=\; (\Sigma\mathbf{TR},\ 1_1{}^\dagger = t_0)$$

when Σ_1 contains the letter σ say, and Σ_0 contains the letter δ and $t_0 := \sigma \cdot \delta$. But $T \in FD_p$, by Corollary 5.4. Thus $T \in FD_p - \Omega$.

5.9 $\Omega_0 \subset \Omega$

This follows immediately from the fact that there are theories in Ω with more than one morphism $1 \longrightarrow 0$, e.g. tree theories.

5.10 $FD_s \subset FD_p$

Since it is clear that $FD_s \subseteq FD_p$, we show that there is a theory in $FD_p - FD_s$. Indeed, choose any theory T in V_g with at least two morphisms $1 \longrightarrow 0$. Then T cannot be in FD_s, since any such theory has a unique morphism $1 \longrightarrow 0$ (for a proof, see [8], f mple.)

5.11 $PFN \subset REL$

Since each partial function $A \times [n] \longrightarrow A \times [p]$ is a relation, $PFN \subseteq REL$. We show the inclusion is strict.

For any morphism $p : 1 \longrightarrow 2$ in a preiteration theory, define the two morphisms p_T, p_F : $1 \longrightarrow 1$ as follows:

$$\begin{aligned} p_T &:= p \cdot (1_1 \oplus \perp) \\ p_F &:= p \cdot (\perp \oplus 1_1). \end{aligned}$$

The following implication is valid in PFN:

$$p \cdot \langle 1_1, 1_1 \rangle = 1_1 \quad \Rightarrow \quad p_T \cdot p_F = \perp_{1,1}. \tag{9}$$

Indeed, if $p : A \longrightarrow A \times [2]$ is a partial function, then if $p \cdot \langle 1_1, 1_1 \rangle = 1_1$, then for each $a \in A$, either $ap = (a, 1)$ or $ap = (a, 2)$. If $ap = (a, 1)$ then ap_F is undefined, ap_T is defined, and $ap_T = a$. If $ap = (a, 2)$, then ap_T is undefined and $ap_F = a$. In either case, $ap_T \cdot p_F$ is undefined. Thus, $p_T \cdot p_F = \perp_{1,1}$.

To see that this implication (9) is not valid in REL, let $A := \{x\}$ and suppose that $p : 1 \longrightarrow 2$ is the relation satisfying $xp = \{(x, 1), (x, 2)\}$. Then $p \cdot \langle 1_1, 1_1 \rangle = 1_1 = p_T \cdot p_F$.

5.12 $REL \subset \Omega_0$

Note that the following implication is valid in REL.

$$\begin{aligned} & p \cdot \langle 1_1, 1_1 \rangle = 1_1 \;\wedge\; p_T \cdot p_F = \perp_{1,1} \\ & \quad \Rightarrow \;\; p \cdot (p \oplus p) = p \cdot (1_1 \oplus 0_2 \oplus 1_1), \end{aligned} \tag{10}$$

where $p : 1 \longrightarrow 2$. The morphisms $p_T, p_F : 1 \longrightarrow 1$ are defined above in section 5.11. Indeed, if the equation $p \cdot \langle 1_1, 1_1 \rangle = 1_1$ holds in $\mathbf{Rel}_A$, then for each $a \in A$, $a\, p\, (a, 1)$ or $a\, p\, (a, 2)$, or both. The second equation guarantees that at most one of these conditions can hold, so that in fact, p is a total function $A \longrightarrow A \times [2]$ which takes $a \in A$ to either $(a,1)$ or $(a,2)$. It follows that if the hypotheses of (10) hold in $\mathbf{Rel}_A$, so does the conclusion.

But let D be the three element chain $\text{BOT} < a < \text{TOP}$. Consider the least subtheory T of $\mathbf{Pow}_D$, the theory of all functions on D, containing the constant function BOT and the meet function $x \wedge y$. A morphism $f : 1 \longrightarrow p$ in T is either the constant function $D^p \longrightarrow D$ with value BOT or $f(x_1, \ldots, x_p) = \bigwedge\{x_i : i \in I\}$, for some nonempty $I \subseteq [p]$. Thus, the unique morphism $\perp : 1 \longrightarrow 0$ in T has the value BOT. It follows that T is a subiteration theory of the theory of all continuous functions $D^p \longrightarrow D^n$. Hence T is in Ω, and

$$f^\dagger \;:=\; \bigvee_n f^n \cdot (\perp \oplus \mathbf{1}_p),$$

for $f : 1 \longrightarrow 1 + p$. But T does not satisfy the implication (10). Indeed, let $p(x_1, x_2) = x_1 \wedge x_2$. Then

$$\begin{aligned} p(x,x) &= x \\ p_T \cdot p_F(x) &= x \wedge \text{BOT} \\ &= \perp_{1,1}. \end{aligned}$$

However,

$$\begin{aligned} p \cdot (p \oplus p) &= p(p(x,y), p(,u,v)) \\ &= x \wedge y \wedge u \wedge v \\ &\neq p \cdot (\mathbf{1}_1 \oplus 0_2 \oplus \mathbf{1}_1) \\ &= p(x,v) \;=\; x \wedge v. \end{aligned}$$

5.13 $\Omega_0 \subset FD_s$

In each theory in Ω_0, every morphism is pure. Since $\Omega \subseteq FD_p$, it follows that $\Omega_0 \subseteq FD_s$.

We will use the following implication, which holds in Ω_0:

$$f \cdot \langle \perp_{1,1}, \mathbf{1}_1 \rangle = \mathbf{1}_1 \wedge f \cdot \langle \mathbf{1}_1, \mathbf{1}_1 \rangle = \mathbf{1}_1 \quad \Rightarrow \quad f^\dagger = \mathbf{1}_1,$$

where $f : 1 \longrightarrow 2$. Indeed, note that if the hypotheses of the implication hold for f, then $f^n \cdot \langle \perp_{1,1}, \mathbf{1}_1 \rangle = \mathbf{1}_1$, for all $n \geq 1$.

Now consider the three element idempotent star semiring $S_0 = \{0, 1, 1^*\}$, with the star defined by

$$x^* \;:=\; \begin{cases} 1 & x = 0 \\ 1^* & \text{otherwise.} \end{cases}$$

This semiring is ω-complete, when any infinite sum with infinitely many nonzero summands is defined to be 1^*. It follows that the matrix theory $\mathbf{Mat}_{S_0}$ is an iteration theory (see [8]). The iteration operation applied to any n by $n + p$ matrix $f = [a \quad b]$ (where a is n by n and b is n by p), yields

$$f^\dagger \;:=\; a^* b,$$

where $a^* = \sum_{n=0}^{\infty} a^n$.

Now if $T = \mathbf{Mat}_S$ and if S is *any* ω-complete semiring, then T is in FD_* [5, 8]. But in $\mathbf{Mat}_{S_0}$, let $f := [1 \quad 1]$. Then

$$\begin{aligned} f \cdot \langle \bot_{1,1}, \mathbf{1}_1 \rangle &= [1 \quad 1]\begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= f \cdot \langle \mathbf{1}_1, \mathbf{1}_1 \rangle \\ &= [1 \quad 1]\begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= 1 \end{aligned}$$

but $f^\dagger = 1^*$. Hence $T \in FD_* - \Omega_0$.

5.14 $REL \subset MAT$

It is known from [8] that $REL \subseteq MAT$. The theory $T = \mathbf{Mat}_{S_0}$ in the previous section does not belong to Ω_0, and hence does not belong to REL.

5.15 $MAT \subset IT$

This follows immediately from the fact that all theories in MAT have a unique morphism $1 \longrightarrow 0$.

6 Incomparable Quasi-Varieties

It is obvious that $V_t - FD_*$ and $V_t - MAT$ are nonempty, since all theories in FD_* and MAT have a unique morphism $1 \longrightarrow 0$. Thus, by inspecting Figure 1, page 11, it will be seen that all of the incomparability results will follow once we can prove that each of the following classes is nonempty:

$$PFN - GA, \quad PI - FD_p, \quad FD_* - \Omega, \quad \Omega_0 - MAT, \quad MAT - FD_2. \tag{11}$$

Indeed, for example, if $PFN \subseteq X$ and $Y \subseteq GA$, for some quasi-varieties X, Y, then it follows from the fact that $PFN - GA$ is nonempty that $X - Y \neq \emptyset$.

We now proceed to prove the statements in (11).

6.1 $PFN - GA \neq \emptyset$

Let $A := \{a, b\}$ and let $f, g : A \longrightarrow A \times [3]$ be the following (total) functions.

$$af \ := \ (b, 1)$$

$$\begin{aligned} bf &:= (b,3) \\ ag &:= (b,3) \\ bg &:= (a,2) \end{aligned}$$

Then

$$af^{\dagger\dagger} = bf^{\dagger\dagger} = ag^{\dagger\dagger} = bg^{\dagger\dagger} = (b,1).$$

Hence $f^{\dagger\dagger} = g^{\dagger\dagger}$ is the total function $A \longrightarrow A$ with value b. Now if the GA-implication were true in $\mathbf{Pfn}_A$,

$$f^{\dagger\dagger} \;=\; h^{\dagger},$$

where $h := f \cdot \langle g^{\dagger}, \mathbf{1}_2 \rangle$. But note that

$$\begin{aligned} ah &= bg^{\dagger} \\ &= (a,1), \end{aligned}$$

so that $h^{\dagger}$ is not defined on a.

6.2 $PI - FD_p \neq \emptyset$

Let Σ consist of two letters $\{g,h\}$ of rank 1, and let T be the quotient theory of the free iteration theory $\Sigma\mathbf{tr}$ with respect to the smallest iteration theory congruence θ such that

$$g \cdot h \equiv g \pmod{\theta}, \quad g \cdot \perp \equiv \perp \pmod{\theta}, \quad g^{\dagger} \equiv \perp \pmod{\theta}.$$

The morphisms $1 \longrightarrow 1$ in T are the congruence classes of the following trees:

- $h^p \cdot g^q \cdot h^{\dagger} \cdot 0_1, \quad q \neq 0$
- $h^{\dagger} \cdot 0_1$
- $h^p \cdot \perp \cdot 0_1$
- $h^p \cdot g^q$.

The theory T is ideal, and for any ideal $r : 1 \longrightarrow 1$, the fixed point equation $x = r \cdot x$ has a unique solution. Indeed, consider the equation $x = h^n \cdot g^m \cdot x$, where $x : 1 \longrightarrow 0$. If $m \neq 0$, the unique solution is $h^n \cdot \perp$. When $m = 0$ but $n \neq 0$, the unique solution is $h^{\dagger}$. Thus T is in PI.

But T is not in FD_p. Indeed,

$$\mathbf{1}_1 \cdot g \;\equiv\; g \cdot h \pmod{\theta},$$

and g is pure, but

$$\mathbf{1}_1{}^{\dagger} \;=\; \perp \;\not\equiv\; g \cdot h^{\dagger} \pmod{\theta}.$$

6.3 $FD_s - \Omega \neq \emptyset$

This fact follows immediately from the fact that $FD_s - \Omega_0 \neq \emptyset$, proved above in Section 5.13.

6.4 $\Omega_0 - MAT \neq \emptyset$

The example in Section 5.13 shows that there is some theory in $MAT - \Omega_0$. We now show that there is a theory in $\Omega_0 - MAT$.

Let T be the least subiteration theory of the iteration theory of all order preserving functions on the 3 element chain $\text{BOT} < a < \text{TOP}$ containing the constant function BOT and the function

$$f(x,y) \quad := \quad \begin{cases} y & \text{if } x = \text{BOT} \\ \text{BOT} & \text{if } y = \text{BOT} \\ \text{TOP} & \text{otherwise.} \end{cases}$$

We note that

$$\begin{aligned} f^{\dagger}(x) &= f(x,x) \\ &= \begin{cases} \text{BOT} & \text{if } x = \text{BOT} \\ \text{TOP} & \text{otherwise} \end{cases} \\ f^{\dagger\dagger} &= \text{BOT}. \end{aligned}$$

Since T has a *unique morphism* $1 \longrightarrow 0$, T is in Ω_0.

The following implication is valid in MAT. For all $f, g : 1 \longrightarrow 2$, if

$$\begin{aligned} f \cdot \langle \mathbf{1}_1, \bot_{1,1} \rangle &= g \cdot \langle \mathbf{1}_1, \bot_{1,1} \rangle \quad \text{and} \\ f \cdot \langle \bot_{1,1}, \mathbf{1}_1 \rangle &= g \cdot \langle \bot_{1,1}, \mathbf{1}_1 \rangle \end{aligned}$$

then

$$f = g.$$

Indeed, write $f = [a,b]$, $g = [c,d]$. Then the first equation says $a = c$ and the second says $b = d$.

Now in the above theory T, let $g = 0_1 \oplus \mathbf{1}_1$; i.e., $g(x,y) := y$. Then

$$\begin{aligned} f \cdot \langle \bot_{1,1}, \mathbf{1}_1 \rangle &= f(\text{BOT}, x) \\ &= x \\ &= g(\text{BOT}, x); \\ f \cdot \langle \mathbf{1}_1, \bot_{1,1} \rangle &= f(x, \text{BOT}) \\ &= \text{BOT} \\ &= g(x, \text{BOT}). \end{aligned}$$

But $f \neq g$, since, e.g., $f(a,a) = \text{TOP}$ and $g(a,a) = a$.

6.5 $MAT - FD_2 \neq \emptyset$

We will construct a matrix theory T over a semiring which is a quotient of the semiring of regular subsets of Σ^*, where $\Sigma = \{a, b, c, d\}$. The congruence is the least *-semiring congruence which identifies the sets $\{a, b\}$ and $\{c, d\}$.

Recall, that for any language $L \subseteq \Sigma^*$, the set of **factors** of L is defined as follows.

$$fac\, L \ := \ \{v \in \Sigma^* : uvw \in L, \text{ some } u, v \in \Sigma^*\}.$$

An **n-state nondeterministic finite automaton** (nfa) over Σ is a triple $M = (\alpha, A, \gamma)$, where A is an $n \times n$ matrix whose entries are subsets of Σ, and where $\alpha : 1 \longrightarrow n$ and $\beta : n \longrightarrow 1$ are 0-1 matrices. The **behavior** of (α, A, γ) is the regular language

$$|M| \ := \ \alpha A^* \gamma.$$

The **states** of M are the integers in $[n]$; there is an edge (i, j) with source i and target j in M if $A_{i,j}$ is nonempty, in which case a label of such an edge is any letter in the set $A_{i,j}$. (Equivalently, one might say there is one edge from i to j for each letter in $A_{i,j}$.) For states j, j', a **path** from j to j' is a sequence of states $j = i_0, i_1, \ldots, i_m = j'$, such that (i_k, i_{k+1}) is an edge, for each $k < m$; a **label** of such a path is any word $x_1 \ldots x_m \in \Sigma^m$ such that $x_k \in A_{i_{k-1}, i_k}$, for $0 < k \leq m$. The **initial states** of M are those states i such that $\alpha_{1,i} = 1$; the **final states** are those states j such that $\gamma_{j,1} = 1$. The **accessible states** are those on paths whose source is an initial state; the **coaccessible states** are those on paths whose target is a final state.

It may easily be shown that the behavior of (α, A, γ) is the set of all words which label paths from an initial state to a final state, since $A^* = \bigcup_{k \geq 0} A^k$.

Thus, if L is the behavior of some nfa M, $v \in fac\, L$ iff v is a label of some path in M whose source is an accessible state and whose target is a coaccessible state.

For a positive integer K, say that a language is **K-bounded** if for all words u, and all nonnegative integers m,

$$(u(a+b))^m \subseteq fac\, L \ \Rightarrow \ m < K, \quad \text{and} \tag{12}$$

$$(u(c+d))^m \subseteq fac\, L \ \Rightarrow \ m < K. \tag{13}$$

L is **bounded** if L is K-bounded, for some integer K. Note that for any L, u, n, m, if

$$(u(c+d))^{n+m} \subseteq fac\, L$$

then

$$(u(c+d))^n \subseteq fac\, L.$$

Example 6.1 The language

$$(a + b + c + d)^*$$

is not bounded, and

$$(a + bd^*c)^*(1 + bd^*)$$

is 2-bounded. □

Let $M = (\alpha, A, \gamma)$ and $M' = (\alpha, B, \gamma)$ be two n-state finite automata, with the same initial and final states. We write $M \sim M'$ if there is an edge (i,j) of M with $A_{i,j} = \{a,b\} \cup Z$ and $B_{i,j} = \{c,d\}$ or if $A_{i,j} = \{c,d\} \cup Z$ and $B_{i,j} = \{a,b\}$. (Here, in the first case Z is some subset of $\{c,d\}$ and in the second, Z is a subset of $\{a,b\}$.) Thus, $M \sim M'$ if it is possible to obtain M' by changing the set of labels of one edge of M by replacing $\{a,b\}$ by $\{c,d\}$ or vice versa. Note that the set of accessible or coaccessible states in M are exactly those in M'.

Theorem 6.2 *Suppose that $M = (\alpha, A, \gamma)$ and $M' = (\alpha, B, \gamma)$ are automata with $M \sim M'$. If $L = |M|$ is K-bounded, then $R = |M'|$ is $2K$-bounded.*

Proof. Let $e = (i,j)$ be the edge in M whose label is changed in order to obtain M'. Suppose that L is K-bounded, and in order to produce a contradiction, suppose that the implication

$$\forall u \forall m \, [(u(c+d))^m \subseteq \mathit{fac}\, R \;\; \Rightarrow \;\; m < 2K].$$

is false. Then there is a word u and some integer $m \geq 2K$ such that

$$(u(c+d))^m \;\; \subseteq \;\; \mathit{fac}\, R.$$

As noted above, it then follows that

$$\begin{aligned} (u(c+d))^{2K} \;\; &\subseteq \;\; \mathit{fac}\, R \quad \text{and} \\ (u(c+d))^{K} \;\; &\subseteq \;\; \mathit{fac}\, R. \end{aligned}$$

Since L is K-bounded, there is some word w in $(u(c+d))^K$ with the property that there is no path labeled w in M whose source is an accessible state and whose target is coaccessible. But there is such a path in M', so this path must use the edge e whose label was changed. Similarly, there is a path in M' whose source is accessible, whose target is coaccessible, and whose label is ww, since $ww \in (u(c+d))^{2K}$. Thus, this second path must use the edge e twice, showing that the edge e lies on a cycle all of whose vertices are both accessible and coaccessible. If $v : j \longrightarrow i$ is a label of the rest of the cycle, then v is a label of a path $j \longrightarrow i$ in both M and M', and if $a, b \in A_{i,j}$, then $(v(a+b))^m \subseteq \mathit{fac}\, L$, for all m, contradicting the hypothesis. (Similarly if $c, d \in A_{i,j}$, then $(v(c+d))^m \subseteq \mathit{fac}\, L$, all m.)

The same argument shows that the implication

$$\forall u \forall m \, [(u(a+b))^m \subseteq \mathit{fac}\, R \;\; \Rightarrow \;\; m < 2K]$$

holds as well. Thus, R is $2K$-bounded. □

For languages L, L', say $L \sim L'$ if there are automata M, M' with $M \sim M'$, where L is the behavior of M and L' is the behavior of M'.

Definition 6.3 *For languages L, R, say $L \approx R$ if either $L = R$ or there is a finite sequence $L_1, L_2, \ldots, L_n$ of languages such that $L = L_1$, $R = L_n$ and $L_i \sim L_{i+1}$, for $i < n$.*

We omit the proof of the following theorem, which makes use of some of the constructions in Chapter 9, Section 4 of [8].

Theorem 6.4 *The relation $\approx$ is the least congruence relation on the *-semiring of regular subsets of Σ^* such that $\{a,b\} \approx \{c,d\}$*

Corollary 6.5 *Suppose that $L \approx R$ and L is bounded. Then R is bounded.*

Proof. This follows immediately from Theorem 6.2. □

Now let $T = \mathbf{Mat}_S$, where S is the quotient of the regular subsets of Σ^* by $\approx$. Define $A : 2 \longrightarrow 2$ as the matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

Then, if $\rho : 2 \longrightarrow 1$ is the base morphism, we have

$$A \cdot \rho \quad = \quad \rho \cdot [\{a,b,c,d\}]$$

in T, since $\{a,b\} \approx \{c,d\} \approx \{a,b,c,d\}$. If T satisfies the functorial dagger implication for the base morphism with source 2, then

$$A^* \cdot \rho \quad = \quad \rho \cdot \{a,b,c,d\}^*.$$

But

$$A^* \quad = \quad \begin{bmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{bmatrix}.$$

Letting L denote the sum of the first two entries of A^*, we see that L is bounded. But $\{a,b,c,d\}^*$ is not (see Example 6.1). Hence $A^* \cdot \rho \neq \rho \cdot \{a,b,c,d\}^*$ in T. Thus $T \in MAT - FD_2$. □

Note that T is an example of an iteration theory not satisfying the functorial dagger implication.

7 A Chain of Quasi-Varieties

In this section we prove that there is an infinite number of quasi-varieties of iteration theories between FD and IT. It was shown in [13] that if a Conway theory T has a functorial dagger for the base morphism $n \longrightarrow 1$, where $n \geq 1$ is any integer, then T has a functorial dagger for all (surjective) base morphisms $n \longrightarrow m$. The following proposition can be proved in straightforward way.

Proposition 7.1 *If a Conway theory has a functorial dagger for the base morphism $n \longrightarrow 1$, where $n \geq 1$ is a given integer, then it has a functorial dagger for all base morphisms $m \longrightarrow 1$, $m \in [n]$.* □

Let $n \geq 1$ be an integer. Recall that FD_n denotes the quasi-variety of iteration theories satisfying the quasi-identity

$$f \cdot (\rho \oplus \mathbf{1}_p) = \rho \cdot g \quad \Rightarrow \quad f^\dagger = \rho \cdot g^\dagger, \tag{14}$$

where $f : n \longrightarrow n+p$, $g : 1 \longrightarrow 1+p$, and where ρ denotes the base morphism $n \longrightarrow 1$. Thus $FD_1 = IT$ and FD is the intersection of the quasi-varieties FD_n. By Proposition 7.1, we have

$$FD_n \subseteq FD_{n-1},$$

for all $n \geq 2$. Below we show that each inclusion is proper.

We assume that $n \geq 2$ is a fixed integer. We will be considering trees in $\Sigma_\perp\mathbf{TR}$, where $\Sigma_n = \{\sigma_1, \ldots, \sigma_n\}$, and $\Sigma_k = \emptyset$, for $k \neq n$. We let $X_p := \{x_1, \ldots, x_p\}$ be the set of the first p variables.

Definition 7.2 *Suppose $f : 1 \longrightarrow p$. We say that f is* **perfect** *if $f \neq \perp \cdot 0_p$ and the following conditions hold for all $u \in [n]^*$.*

- *If $ui \in \operatorname{dom} f$, for some $i \in [n]$, then $f(ui) = \sigma_i$ or $f(ui) \in X_p$.*
- *If $f(u1), \ldots, f(un)$ are all in X_p, then not all values are the same variable.*

A tree $n \longrightarrow p$ is perfect if each $i_n \cdot f$ is perfect, for all $i \in [n]$.

Thus there are n perfect trees $1 \longrightarrow 0$.

Definition 7.3 *Suppose that $f, g : 1 \longrightarrow p$. We define $f \sim g$ iff $f = g$ or neither f nor g is perfect. When $f, g : n \longrightarrow p$, for some $n \neq 1$, then $f \sim g$ iff $i_n \cdot f \sim i_n \cdot g$, for all $i \in [n]$.*

Thus all non-perfect scalar trees $1 \longrightarrow p$ are identified. We list some elementary consequences of the definition.

1. If a tree $t : 1 \longrightarrow p$ has a non-perfect subtree, then t is not perfect.
2. If f is not perfect then $f \cdot g$ is not perfect.
3. If f is not perfect then $f^\dagger$ is not perfect.
4. If f is perfect, x_i occurs in f, and if the i-th component of g is not perfect or has root labeled σ_j for some $j \neq i$, then $f \cdot g$ is not perfect.

Proposition 7.4 *The relation $\sim$ is an iteration theory congruence on trees.*

Proof. This follows from the above facts. □

Note that each perfect tree $1 \longrightarrow p$ forms a singleton $\sim$-congruence class. For the rest of this section we let $T := \Sigma_\perp\mathbf{TR}/\sim$.

Proposition 7.5 *The theory T is contained in $FD_{n-1} - FD_n$.*

Proof. Suppose that

$$f \cdot (\rho \oplus 1_p) \sim \rho \cdot g,$$

where $f : k \longrightarrow k + p$, $g : 1 \longrightarrow 1 + p$ are trees, and where ρ is the base morphism $k \longrightarrow 1$. Suppose that $1 \leq k < n$. We will show that $f^\dagger \sim \rho \cdot g^\dagger$.

Case 1: g is perfect. Then f is perfect and

$$f \cdot (\rho \oplus 1_p) = \rho \cdot g.$$

Thus,

$$f^\dagger = \rho \cdot g^\dagger,$$

since $\Sigma_\perp \mathbf{TR}$ has a functorial dagger.

Case 2: g is not perfect. Then for each $i \in [k]$, either $i_k \cdot f$ is not perfect or $i_k \cdot f$ is perfect but has a subtree of the form

$$\sigma(z_1, \ldots, z_n),$$

where $\sigma \in \Sigma$ and the z_j's are variables in X_k. If $i_k \cdot f$ is not perfect, then

$$i_k \cdot f^\dagger = i_k \cdot f \cdot \langle f^\dagger, 1_p \rangle$$

is not perfect. If $i_k \cdot f$ has a subtree of the form $\sigma(z_1, \ldots, z_n)$, then, since $k < n$, at least two of the z_j's must be the same. Suppose that $z_1 = z_2$, say. But then

$$i_k \cdot f^\dagger = i_k \cdot f \cdot \langle f^\dagger, 1_p \rangle = i_k \cdot f \cdot \langle f \cdot \langle f^\dagger, 1_p \rangle, 1_p \rangle$$

has a subtree of the form

$$\sigma(t, t, t_3, \ldots, t_n),$$

where none of the trees $t, t_3, \ldots, t_n$ is a variable. Again, it follows that $i_k \cdot f^\dagger$ is not perfect.

Thus T is in FD_{n-1}. To prove that T is not in FD_n, consider the tree

$$f := \langle \sigma_1, \ldots, \sigma_n \rangle : n \longrightarrow n.$$

If ρ denotes the base morphism $n \longrightarrow 1$ then

$$f \cdot \rho \sim \rho \cdot \sigma_1 \cdot \rho,$$

but

$$1_n \cdot f^\dagger \not\sim (\sigma_1 \cdot \rho)^\dagger,$$

since the tree $1_n \cdot f^\dagger$ is perfect and $(\sigma_1 \cdot \rho)^\dagger$ is not. □

Any theory in $FD_{n-1} - FD_n$ is another example of an iteration theory which does not satisfy the functorial dagger implication. Thus, we have infinitely many examples of such theories.

8 Appendix

Suppose that T is any theory. A **zero congruence** Θ on T is a theory congruence which is generated by an equivalence relation θ_0 on the morphisms with target 0; i.e., Θ is the smallest congruence on T such that $\theta_0 \subseteq \Theta$. Zero congruences were first considered in [2], where it was shown that the smallest zero congruence on any iteration theory is in fact an iteration theory congruence. In [8], this result was extended to show that in any preiteration theory satisfying the parameter identity, namely

$$(f \cdot (\mathbf{1}_n \oplus g))^\dagger = f^\dagger \cdot g, \quad \text{all } f : n \longrightarrow n+p,\ g : p \longrightarrow q,$$

any zero congruence preserves the dagger operation.

We prove here the following theorem.

Theorem 8.1 *Suppose that Θ is a zero congruence on a free iteration theory $\Sigma\mathbf{tr}$ generated by an equivalence relation θ_0 on the morphisms with target* 0. *Then for any $f, g : n \longrightarrow p$ in $\Sigma\mathbf{tr}$, $f\ \Theta\ g$ iff for some $F : n \longrightarrow p+k$, and some $\alpha, \beta : k \longrightarrow 0$,*

$$\begin{aligned} f &= F \cdot (\mathbf{1}_p \oplus \alpha) \\ g &= F \cdot (\mathbf{1}_p \oplus \beta) \quad \textit{and} \\ \alpha &\equiv \beta \pmod{\theta_0} \end{aligned}$$

Corollary 8.2 *With the notation of Theorem 8.1, let T denote the quotient theory $\Sigma\mathbf{tr}/\Theta$. Then the functorial dagger implication holds in T.*

The following proposition, the **Zero Congruence Lemma**, is known from [2]. See also [8] for further refinements.

Proposition 8.3 *For any theory T, the least theory congruence Θ containing θ_0 is the transitive closure of the following relation $\sim$: for $f, g : n \longrightarrow p$, $f \sim g$ iff for some $F : n \longrightarrow p+k$, and some $\alpha, \beta : k \longrightarrow 0$,*

$$\begin{aligned} f &= F \cdot (\mathbf{1}_p \oplus \alpha) \\ g &= F \cdot (\mathbf{1}_p \oplus \beta) \quad \textit{and} \\ \alpha &\equiv \beta \pmod{\theta_0}. \quad \square \end{aligned}$$

We will show that in $\Sigma\mathbf{tr}$, the relation $f \sim g$ just defined coincides with the relation Θ.

We make use of the following lemma, proved in the next section.

Lemma 8.4 *Suppose that $F \cdot (\mathbf{1}_p \oplus \beta) = G \cdot (\mathbf{1}_p \oplus \gamma)$ for some $F : n \longrightarrow p+k$, $G : n \longrightarrow p+k'$, $\beta : k \longrightarrow 0$, $\gamma : k' \longrightarrow 0$ in $\Sigma\mathbf{tr}$. Then for some integer $m \geq 0$, there are trees $H : n \longrightarrow p+m$ and $Q : m \longrightarrow k$, $Q' : m \longrightarrow k'$ in $\Sigma\mathbf{tr}$ such that*

$$\begin{aligned} F &= H \cdot (\mathbf{1}_p \oplus Q) \\ G &= H \cdot (\mathbf{1}_p \oplus Q') \\ Q \cdot \beta &= Q' \cdot \gamma. \end{aligned}$$

Corollary 8.5 *The relations $\sim$ of Proposition 8.3 and Θ are the same.*

Proof. We need show only that $\sim$ is transitive, since Θ is the transitive closure of $\sim$. Assume that $f \sim g$, and $g \sim h$, where $f, g, h : n \longrightarrow p$ in $\Sigma\mathbf{tr}$. Then

$$\begin{aligned} f &= F \cdot (\mathbf{1}_p \oplus \alpha) \\ g &= F \cdot (\mathbf{1}_p \oplus \beta), \end{aligned}$$

for some $F : n \longrightarrow p + k$, some $\alpha, \beta : k \longrightarrow 0$ with $\alpha\ \theta_0\ \beta$. Also,

$$\begin{aligned} g &= G \cdot (\mathbf{1}_p \oplus \gamma) \\ h &= G \cdot (\mathbf{1}_p \oplus \delta), \end{aligned}$$

for some $G : n \longrightarrow p + k'$, some $\gamma, \delta : k' \longrightarrow 0$ with $\gamma\ \theta_0\ \delta$.

Since $g = F \cdot (\mathbf{1}_p \oplus \beta) = G \cdot (\mathbf{1}_p \oplus \gamma)$, then, by the lemma, for some trees H, Q, Q' in $\Sigma\mathbf{tr}$, $F = H \cdot (\mathbf{1}_p \oplus Q)$, $G = H \cdot (\mathbf{1}_p \oplus Q')$ and $Q \cdot \beta = Q' \cdot \gamma$. But then

$$\begin{aligned} f &= H \cdot (\mathbf{1}_p \oplus Q) \cdot (\mathbf{1}_p \oplus \alpha) \\ &= H \cdot (\mathbf{1}_p \oplus (Q \cdot \alpha)) \\ h &= H \cdot (\mathbf{1}_p \oplus Q') \cdot (\mathbf{1}_p \oplus \delta) \\ &= H \cdot (\mathbf{1}_p \oplus (Q' \cdot \delta)) \end{aligned}$$

Also,

$$Q \cdot \alpha \equiv Q \cdot \beta = Q' \cdot \gamma \equiv Q' \cdot \delta \quad (\text{mod } \theta_0).$$

Thus, $Q \cdot \alpha \equiv Q' \cdot \delta \quad (\text{mod } \theta_0)$, showing $f \sim h$. □

Proof of Corollary 8.2. It is enough to prove that the implication

$$f \cdot (\rho \oplus \mathbf{1}_p) \sim \rho \cdot g \quad \Rightarrow \quad f^\dagger \sim \rho \cdot g^\dagger$$

holds, when $f : n \longrightarrow n + p$, $g : 1 \longrightarrow 1 + p$ in $\Sigma\mathbf{tr}$ and when $\rho : n \longrightarrow 1$ is the unique base morphism. Thus, suppose that

$$f \cdot (\rho \oplus \mathbf{1}_p) \quad \sim \quad \rho \cdot g.$$

By definition, then, there is some $F : n \longrightarrow 1 + p + k$, some $\alpha, \beta : k \longrightarrow 0$ in $\Sigma\mathbf{tr}$ with

$$\begin{aligned} f \cdot (\rho \oplus \mathbf{1}_p) &= F \cdot (\mathbf{1}_{1+p} \oplus \alpha) \\ \rho \cdot g &= F \cdot (\mathbf{1}_{1+p} \oplus \beta), \quad \text{and} \\ \alpha &\equiv \beta \quad (\text{mod } \theta_0). \end{aligned}$$

Write $F = \langle F_1, \ldots, F_n \rangle$. Since $\rho \cdot g = F \cdot (\mathbf{1}_{1+p} \oplus \beta)$, it follows that

$$g \quad = \quad F_i \cdot (\mathbf{1}_{1+p} \oplus \beta),$$

for each $i \in [n]$. We will define the tree $G_i : 1 \longrightarrow n + p + k$ for each $i \in [n]$, such that if

$$G \quad := \quad \langle G_1, \ldots, G_n \rangle : n \longrightarrow n + p + k$$

then

$$\begin{aligned} f &= G \cdot (\mathbf{1}_{n+p} \oplus \alpha) \\ F &= G \cdot (\rho \oplus \mathbf{1}_{p+k}). \end{aligned}$$

Indeed, for each $i \in [n]$, let G_i be obtained from F_i by relabeling any leaf vertex u of F_i labeled x_1 by $u(i_n \cdot f)$ and by relabeling any leaf v of F_i labeled x_{1+j}, $j \in [p+k]$, by x_{n+j}. (Necessarily $u(i_n \cdot f) = x_j$, for some $j \in [n]$.)

Note the following facts.

$$\begin{aligned} f \cdot (\rho \oplus \mathbf{1}_p) &= G \cdot (\rho \oplus \mathbf{1}_p \oplus \alpha) \\ \rho \cdot g &= G \cdot (\rho \oplus \mathbf{1}_p \oplus \beta). \end{aligned}$$

We will prove that

$$G^{\dagger} \cdot (\mathbf{1}_p \oplus \beta) = \rho \cdot g^{\dagger}. \qquad (15)$$

Indeed,

$$G \cdot (\mathbf{1}_{n+p} \oplus \beta) \cdot (\rho \oplus \mathbf{1}_p) = \rho \cdot g,$$

and since the functorial dagger implication is valid in $\Sigma\mathbf{tr}$, it follows that (15) holds. But then, by the parameter identity,

$$\begin{aligned} f^{\dagger} &= G^{\dagger} \cdot (\mathbf{1}_p \oplus \alpha) \\ &\sim G^{\dagger} \cdot (\mathbf{1}_p \oplus \beta) \\ &= \rho \cdot g^{\dagger}. \quad \square \end{aligned}$$

One application of these results was given in Section 5.4 above. Assume that $f, g \in \Sigma_2$, and let Θ be the zero congruence on $\Sigma\mathbf{tr}$ generated by the least equivalence such that $f^{\dagger\dagger} \sim g^{\dagger\dagger}$. Since both $f^{\dagger\dagger}$ and $g^{\dagger\dagger}$ are morphisms $1 \longrightarrow 0$, the theory $\Sigma\mathbf{tr}/\Theta$ satisfies the functorial dagger implication, by Corollary 8.2. Note that the tree $f^{\dagger\dagger}$ is the complete binary tree with each vertex labeled f. If $h : 1 \longrightarrow 1$ is the tree

$$h := g \cdot \langle f^{\dagger}, \mathbf{1}_1 \rangle,$$

then it is not the case that

$$h^{\dagger} \equiv f^{\dagger\dagger} \pmod{\Theta},$$

by Theorem 8.1, since $h^{\dagger}$ has no subtree equal to either $f^{\dagger\dagger}$ or to $g^{\dagger\dagger}$. Thus, the GA-implication fails in T.

It remains to prove Lemma 8.4.

8.1 Proof of the Lemma

We call a vertex u of f a **zero vertex** if the subtree f_u of f rooted at u is a zero subtree, i.e., f_u is $t \cdot 0_p$, for some $t : 1 \longrightarrow 0$ in $\Sigma\mathbf{tr}$. A tree with no zero vertices is *coaccessible*. A **minimal zero vertex** of f is a zero vertex of f which has no proper prefix which is also a zero vertex. We let $MZ(f)$ denote the set of minimal zero vertices of f.

Suppose that $f = F \cdot (\mathbf{1}_p \oplus \alpha) : 1 \longrightarrow p$, where $F : 1 \longrightarrow p + k$ and $\alpha : k \longrightarrow 0$.

1. If f has no zero vertices, then F factors through p, i.e., $F = F' \oplus 0_k$, for some $F' : 1 \longrightarrow p$, and F' is coaccessible. If the empty word is a zero vertex, then F factors outside of p, i.e., $F = 0_p \oplus F'$, for some $F' : 1 \longrightarrow k$.
2. If u is a zero vertex of f and $uv \in \operatorname{dom} f$, then uv is also a zero vertex of f.
3. If u is a minimal zero vertex of f, then $u \in \operatorname{dom} F$. Indeed, otherwise $u = vv'$, where $vF = x_{p+j}$ and $v' \in \operatorname{dom} \alpha_j$. But then v is also a zero vertex of f, showing u is not minimal.
4. Since f is regular, the collection of trees $\{f_u : 1 \longrightarrow 0 \mid u \in MZ(f)\}$ is finite.

Now assume that $f = F \cdot (\mathbf{1}_p \oplus \beta) = G \cdot (\mathbf{1}_p \oplus \gamma)$. Let X be the (regular) set of all minimal zero vertices u of f such that F_u or G_u contains a leaf labeled by some variable (which is necessarily x_{p+j}, for some $j > 0$.) Note that if $u \in X$, then $u \in \operatorname{dom} F \cap \operatorname{dom} G$, and if v is any leaf of F_u, then the label of v is x_{p+j} for some $j > 0$. Indeed, otherwise, u would not be a zero vertex of f. Similarly, if v is a leaf of G_u labeled by a variable, its label is x_{p+j} for some $j > 0$.

We let H be f "cut off" at X. Assume there are m trees $\delta_1, \ldots, \delta_m$ of the form $f_u : u \in X$.

Definition 8.6 *The tree $H : 1 \longrightarrow p + m$ is defined as follows. The domain of H is the regular set consisting of X together with the set of all vertices of f having no prefix in X. For $u \in \operatorname{dom} H$,*

$$uH \quad := \quad \begin{cases} uf & \textit{if } u \textit{ has no prefix in } X \\ x_{p+i} & \textit{if } u \in X \textit{ and } f_u = \delta_i \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

Remark 8.7 The tree H is regular. If the set X is empty, then $m = 0$.

Now we define the trees $Q : m \longrightarrow k$, $Q' : m \longrightarrow k'$. Suppose that $u \in X$ and $f_u = \delta_i$. Then $u \in \operatorname{dom} F \cap \operatorname{dom} G$, by item 3. above.

Definition 8.8

$$vQ_i \quad := \quad \begin{cases} vF_u & \textit{if } vF_u \textit{ is not a variable} \\ x_j & \textit{if } vF_u = x_{p+j}. \end{cases}$$

$$vQ'_i \quad := \quad \begin{cases} vG_u & \textit{if } vG_u \textit{ is not a variable} \\ x_j & \textit{if } vG_u = x_{p+j}. \end{cases}$$

Then, by construction, for $u \in X$ and $f_u = \delta_i$,

$$f_u = Q_i \cdot \beta = Q'_i \cdot \gamma.$$

It follows that

$$Q \cdot \beta = Q' \cdot \gamma.$$

Further,

$$\begin{aligned} F &= H \cdot (\mathbf{1}_p \oplus Q) \\ G &= H \cdot (\mathbf{1}_p \oplus Q'). \end{aligned}$$

When $n > 1$ and $f = \langle f_1, \ldots, f_n \rangle : n \longrightarrow p$ can be written in two ways as

$$\langle F_1, \ldots, F_n \rangle \cdot (\mathbf{1}_p \oplus \beta) = \langle G_1, \ldots, G_n \rangle \cdot (\mathbf{1}_p \oplus \gamma),$$

we use the same procedure; we now let X_i be the set of minimal zero vertices u in the tree f_i, such that $(F_i)_u$ or $(G_i)_u$ contains a leaf labeled x_{p+j}. Let m be the number of all subtrees $(f_i)_u, u \in X_i$, $i \in [n]$. Define the domain of the tree $H_i : 1 \longrightarrow p + m$, $i \in [n]$ as the set of vertices in X_i together with those vertices in the domain of f_i having no prefix in X_i; the values of H_i are as above. The trees Q, Q' are defined exactly as before. We omit the remaining details. □

8.2 A Generalization

The proof of Theorem 8.2 suggests that the following notion may be of interest. Suppose that T is any theory.

Definition 8.9 T *has the* **lifting property** *if for any morphisms* $f : n \longrightarrow n + p$, $F : n \longrightarrow 1 + p + k$ *and* $\alpha : k \longrightarrow 0$ *in* T, *if*

$$f \cdot (\rho \oplus \mathbf{1}_p) = F \cdot (\mathbf{1}_{1+p} \oplus \alpha),$$

where $\rho : n \longrightarrow 1$ *is the unique base morphism, then there is some* $G : n \longrightarrow n + p + k$ *such that*

$$\begin{aligned} f &= G \cdot (\mathbf{1}_{n+p} \oplus \alpha) \\ F &= G \cdot (\rho \oplus \mathbf{1}_{p+k}). \end{aligned}$$

It was shown above that $\Sigma\mathbf{tr}$ has the lifting property. We state without proof the following facts.

Proposition 8.10 *Each matricial theory* [11, 8] *has the lifting property, as does each theory* $\mathbf{Pfn}_A$ *and* $\mathbf{Rel}_A$. □

Proposition 8.11 *Suppose that* T *is an iteration theory which has the lifting property. Then if* T *satisfies the functorial dagger implication, so does* T/θ, *where* θ *is any zero congruence on* T. □

9 Acknowledgment

The authors wish to thank L. Bernátsky for his very careful reading of a draft of this paper.

References

[1] K.B. Arkhangelskii and P.V. Gorshkov. Implicational axioms for the algebra of regular languages (in Russian). *Doklady Akad. Nauk. USSR. ser A.*, 10:67–69, 1987.

[2] S. L. Bloom and R. Tindell. A note on zero congruences. *Acta Cybernetica*, 8:1–4, 1987.

[3] S.L. Bloom, C.C. Elgot, and J.B. Wright. Solutions of the iteration equation and extensions of the scalar iteration operation. *SIAM Journal of Computing*, 9:26–45, 1980.

[4] S.L. Bloom, C.C. Elgot, and J.B. Wright. Vector iteration in pointed iterative theories. *SIAM Journal of Computing*, 9:525–540, 1980.

[5] S.L. Bloom and Z. Ésik. Matrix and matricial iteration theories, Part I. *Journal of Computer and System Sciences*. 46:381–408, 1993.

[6] S.L. Bloom and Z. Ésik. Matrix and matricial iteration theories, Part II. *Journal of Computer and System Sciences*. 46:409-439, 1993.

[7] S.L. Bloom and Z. Ésik. Varieties of iteration theories. *SIAM Journal of Computing*, 17:939–966, 1988.

[8] S.L. Bloom and Z. Ésik. *Iteration theories: the equational logic of iterative processes.* EATCS Monographs on Theoretical Computer Science. Springer–Verlag, 1993.

[9] V.E. Cazanescu and Gh. Stefanescu. Towards a new algebraic foundation of flowchart scheme theory. *Fundamenta Informaticae*, 13:171–210, 1990.

[10] J.C. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, 1971.

[11] C.C. Elgot. Monadic computation and iterative algebraic theories. In J.C. Shepherdson, editor, *Logic Colloquium 1973. Studies in Logic*, volume 80. North Holland, Amsterdam, 1975.

[12] C.C. Elgot, S.L. Bloom, and R. Tindell. On the algebraic structure of rooted trees. *Journal of Computer and System Sciences*, 16:362–399, 1978.

[13] Z. Ésik. Identities in iterative and rational theories. *Computational Linguistics and Computer Languages*, 14:183–207, 1980.

[14] Z. Ésik. On generalized iterative algebraic theories. *Computational Linguistics and Computer Languages*, 15:95–110, 1982.

[15] Z. Ésik. Independence of the equational axioms of iteration theories. *Journal of Computer and System Sciences*, 36:66–76, 1988.

[16] Z. Ésik. A note on the axiomatization of iteration theories. *Acta Cybernetica*, 9:375–384, 1990.

[17] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24:68–95, 1977.

[18] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 214–225, 1991.

[19] D. Krob. Complete systems of B-rational identities. *Theoretical Computer Science*, 89:207–343, 1991.

[20] E.G. Manes. *Predicate Transformer Semantics*. Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1992.

[21] Gh. Stefanescu. On flowchart theories: Part I. The deterministic case. *Journal of Computer and System Sciences*, 35:163–191, 1987.

[22] Gh. Stefanescu. On flowchart theories: Part II. The nondeterministic case. *Theoretical Computer Science*, 52:307–340, 1987.

[23] E. Wagner, J. Thatcher, and J. Wright. Programming languages as mathematical objects. In *Mathematical Foundations of Computer Science*, volume 64 of *Lecture Notes in Computer Science*, pages 84–101. Springer–Verlag, 1978.

[24] J.B. Wright, J.W. Thatcher, J. Goguen, and E.G. Wagner. Rational algebraic theories and fixed–point solutions. In *Proceedings 17th IEEE Symposium on Foundations of Computing*, pages 147–158, Houston, Texas, 1976.

Probabilistic Power Domains, Information Systems, and Locales

Reinhold Heckmann[*]

FB 14 - Informatik, Prof. Wilhelm
Universität des Saarlandes, Postfach 151150
D-66041 Saarbrücken, Germany
e-mail: heckmann@cs.uni-sb.de

Abstract

The probabilistic power domain construction of Jones and Plotkin [6, 7] is defined by a construction on dcpo's. We present alternative definitions in terms of information systems à la Vickers [12], and in terms of locales. On continuous domains, all three definitions coincide.

1 Introduction

To model probabilistic and randomized algorithms in the semantic framework of dcpo's and Scott continuous functions, Jones and Plotkin introduce in [6, 7] the *probabilistic power domain construction* $\mathcal{P}_D$. It forms a computational monad in the sense of [8] in the category of dcpo's and continuous functions and various of its subcategories of 'domains'. Every probabilistic powerdomain $\mathcal{P}_D X$ is equipped with a family of binary operations $+_p$ indexed by a real number p between 0 and 1 such that $A +_p B$ denotes the result of choosing A with probability p and B with probability $1 - p$.

Other applications of $\mathcal{P}_D$ were found in [1]. The probabilistic powerdomain of the upper power space [10] of a second countable locally compact Hausdorff space X can be used for an effective treatment of probability measures on X, and thus for the study of coloured fractals on X, where the colour is modeled by a probability distribution. For these applications, a description of $\mathcal{P}_D$ in terms of information systems would be useful.

[*]This paper was written while the author was visiting Imperial College of Science, Technology and Medicine in London, England. This visit was made possible by a grant of the Deutsche Forschungsgemeinschaft.

In [6, 7], it is shown that $\mathcal{P}_D$ preserves (ω-)continuity of dcpo's. On the other hand, it does not preserve algebraicity: even the power domain of the one-point domain is not algebraic. Thus, $\mathcal{P}_D$ cannot be described by the more conventional information systems, which are only suited for certain classes of algebraic domains, as for instance the information systems of [9] for the class of bounded complete algebraic domains (Scott Domains). The information systems of [3, 4] present bounded complete continuous domains, but this is still not sufficient, since $\mathcal{P}_D$ does not preserve bounded completeness as shown in [6, Section 4.5].

In [12], Vickers introduced a kind of information systems (*infosyses*) suitable to cover all continuous domains. In the paper at hand, we show that $\mathcal{P}_D$ can be described in terms of these infosyses. This was already conjectured by Vickers at the end of his paper. We also looked for a localic description of the power construction. Starting from the frame of opens of a continuous base domain, we show how the frame of opens of the power domain may be constructed in terms of generators and relations.

The paper is organized as follows: in Section 2, we sketch the theoretical background and introduce the probabilistic power domain construction $\mathcal{P}_D$. It can be applied to all topological systems producing a dcpo. In Section 3, we introduce the probabilistic power locale construction $\mathcal{P}_L$. It can be applied to all topological systems producing a locale. We show that for all topological systems, power domain and power locale have the same points.

In Section 4, we show how to construct a probabilistic power infosys $\mathcal{P}_I D$ for every infosys D. In Section 5, we prove that for continuous domains X, the power infosys of an infosys for X has the same opens as the power locale of X. The proof uses a hard lemma, which is postponed to Section 6. The two properties that power domain and power locale have the same points, and power infosys and power locale have the same opens imply that all three of $\mathcal{P}_D$, $\mathcal{P}_I$, and $\mathcal{P}_L$ agree on continuous domains.

2 The Background

In this section, we introduce the background of the theory in this paper. We assume the reader to be familiar with the basic notions of dcpo, continuous dcpo, continuous function between dcpo's, and the frames and locales of [5] or [11]. We denote the least element of a frame by 0 or F standing for 'false', and the greatest element by T for 'true', never 1. We use 'domain' as synonym for 'dcpo', in particular when speaking of continuous dcpo's.

In Subsection 2.1, we recall the topological systems of [11]. In Subsection 2.2, we review the definition and properties of infosyses from [12]. Then we present an infosys for the unit interval of the real line in Subsection 2.3. In Subsection 2.4, we introduce the construction $\mathcal{P}_D$ of [6, 7].

2.1 Topological Systems

In [11], Vickers introduced ***topological systems*** as a common generalization of topological spaces and locales. In analogy to his abbreviation 'infosys' for 'information system', we shall abbreviate 'topological system' by 'topsys'. A ***topsys*** is a pair $X = (\mathrm{pt}\, X, \Omega X)$ of a set of 'points' $\mathrm{pt}\, X$ and a frame of 'opens' ΩX together with a relation '$\models$' between points and opens, which respects the frame operations: for every index set I, $x \models \bigvee_{i\in I} u_i$ iff there is i in I with $x \models u_i$; and for every finite index set I, $x \models \bigwedge_{i\in I} u_i$ iff for all i in I, $x \models u_i$.

A topological space X induces a topsys, where $\mathrm{pt}\, X$ is the underlying set of X, ΩX is the frame of open sets of X, and $x \models u$ iff $x \in u$. This in particular applies to dcpo's with their Scott topology.

A locale X is defined by a frame ΩX. It becomes a topsys by taking $\mathrm{pt}\, X$ as the set of frame homomorphisms from ΩX to $2 = \{\mathsf{F}, \mathsf{T}\}$, with $x \models u$ iff $xu = \mathsf{T}$.

A continuous function $f : X \to Y$ between topsyses has two components, $\mathrm{pt}\, f : \mathrm{pt}\, X \to \mathrm{pt}\, Y$ and $\Omega f : \Omega Y \to \Omega X$, where Ωf is a frame homomorphism and $\mathrm{pt}\, fx \models v$ iff $x \models \Omega f v$ for all x in $\mathrm{pt}\, X$ and v in ΩY. In case of topological spaces, these are the usual continuous functions, and f is determined by $\mathrm{pt}\, f$. In case of locales, these are frame homomorphisms in the opposite direction, and f is determined by Ωf.

2.2 Vickers Information Systems (Infosyses)

In this subsection, we briefly review the theory of infosyses as given in [12].

An ***infosys*** is a pair $(D, <)$ of a set of *tokens* D and a binary relation '$<$' on D which is ***transitive*** — $a < b$ and $b < c$ implies $a < c$ — and *interpolative* — if $a < c$, then there is a token b with $a < b$ and $b < c$. In contrast to the more conventional preorders, reflexivity is not required.

We need several notions and notations: for a set $A \subseteq D$, $\Uparrow A = \{b \in D \mid \exists a \in A : a < b\}$ is the upper set of A. We use the abbreviation $\Uparrow a$ for $\Uparrow\{a\}$.[1] The operator '$\Uparrow$' is monotonic — $A \subseteq B$ implies $\Uparrow A \subseteq \Uparrow B$ — and idempotent — $\Uparrow(\Uparrow A) = \Uparrow A$. In posets, there is an additional property $A \subseteq \Uparrow A$, which is a consequence of reflexivity and thus not true for general infosyses.

The set of upper bounds of a token set A is $\mathrm{ub}\, A = \{b \in D \mid \forall a \in A : a < b\}$.

A ***point*** of an infosys is a subset x of D with properties analogous to those of ideals: if $a < b$ and $b \in x$, then $a \in x$; and for $a_1, \ldots, a_n$ in x, there is b in x with $a_1, \ldots, a_n < b$. For the second condition, cases $n = 0$ (x is not empty) and $n = 2$ are enough. The points of an infosys D ordered by inclusion form a continuous dcpo $\mathrm{pt}\, D$. This is the continuous domain represented by the infosys D.

[1] Vickers has no such abbreviations.

Conversely, for every continuous dcpo X, there is an infosys I with $\mathrm{pt}\, I \cong X$, namely $I = (X, \ll)$, where '$\ll$' is the way-below relation on X. In general, there are many other infosyses I with $\mathrm{pt}\, I \cong X$.

If the dcpo X is even algebraic, then there is an infosys I with reflexive order, i.e., a poset, with $\mathrm{pt}\, I \cong X$, namely the basis of X.

An *open* of an infosys is a subset u of D with $\uparrow u = u$. Ordered by inclusion, the opens of D form a frame ΩD. The joins of this frame are simply given by union, whereas the meets are *not* given by intersection: $u \wedge v$ is $\uparrow(u \cap v)$. The mapping $\omega : \Omega D \to 2^{\mathrm{pt}\, D}$ with $\omega u = \{x \in \mathrm{pt}\, D \mid x \cap u \neq \emptyset\}$ provides an isomorphism between the frame ΩD and the frame $\Omega(\mathrm{pt}\, D)$ of Scott open sets of the continuous domain $\mathrm{pt}\, D$. Thus, the topsys induced by the dcpo $\mathrm{pt}\, D$ is isomorphic to $(\mathrm{pt}\, D, \Omega D)$ with $p \models u$ iff $p \cap u \neq \emptyset$. We call the latter topsys $[D]$. Two infosyses D and E are *equivalent* iff $[D] \cong [E]$. Equivalent infosyses may look quite different, for instance one may be finite and the other one infinite.

The frame ΩD of an infosys D can be presented by generators and relations: Generators are $\sharp a$ for a in D, and the relations are $\bigwedge_{a \in S} \sharp a = \bigvee_{b \in \mathrm{ub}\, S} \sharp b$ for every finite subset S of D. This one relation can be equivalently replaced by three:

(1) Monotonicity: If $a < b$, then $\sharp a \geq \sharp b$;

(2) All tokens: $\bigvee_{a \in D} \sharp a = \top$;

(3) Meets: $\sharp a \wedge \sharp b \leq \bigvee_{c > a,b} \sharp c$.

Here, (1) corresponds to '$\geq$' in the single relation scheme, (2) to '$\leq$' with empty S, and (3) to '$\leq$' with $S = \{a, b\}$.

2.3 The Unit Interval

The unit interval of the real line plays a major role in the theory of the probabilistic power construction. In this paper, we denote the usual order on real numbers by '$\sqsubseteq$' instead of '$\leq$', and accordingly its strict variant ($x \sqsubseteq y$ and $x \neq y$) by '$\sqsubset$' instead of '$<$'. This is done to avoid confusion with the infosys order introduced below.

By these conventions, the unit interval is $\mathbf{I} = \{r \in \mathbf{R} \mid 0 \sqsubseteq r \sqsubseteq 1\}$ ordered by '$\sqsubseteq$'. It forms a continuous domain, which can be described by many different information systems. As pointed out above, one of these is $(\mathbf{I}, <)$, where '$<$' is the way-below relation of $\mathbf{I}$, namely $x < y$ iff $x \sqsubset y$ or $x = y = 0$. Another one with less tokens is $\mathbf{Q}_0^1 = \{q \in \mathbf{Q} \mid 0 \sqsubseteq q \sqsubset 1\}$ with the same order '$<$'.

In the sequel, we list some arithmetic properties of this non-standard order. All these properties are thought of to be quantified over the non-negative reals ($r \sqsupseteq 0$). In contrast to '$\sqsubset$', $a < b$ does not imply $a + c < b + c$ (take $a = b = 0$ and $c \neq 0$). On the other hand, $a_1 < b_1$ and ... and $a_n < b_n$ implies $\sum_{i=0}^{n} a_i < \sum_{i=0}^{n} b_i$ even in the case $n = 0$; and $a < b$ implies $a \cdot c < b \cdot c$ even if $c = 0$. Furthermore, if $a > b$, then there is some rational q with $a > q > b$, and there is some rational $r < 1$ with $r \cdot a > b$. The latter can be extended to any finite number of relations: if $a_1 > b_1$ and ... and $a_n > b_n$, then there is one rational $r < 1$ with $r \cdot a_i > b_i$ for all i.

The frame $\Omega \mathbf{Q}_0^1 \cong \Omega \mathbf{I}$ can be presented by generators $\sharp q$ for q in $\mathbf{Q}_0^1$ with the three relations at the end of Subsection 2.2. By special properties of $\mathbf{Q}_0^1$, namely linearity and existence of joins of tokens, these relations can be simplified to

(1) Rational zero: $\sharp 0 = \top$;

(2) Rational continuity: $\bigvee_{r>q} \sharp r = \sharp q$.

The case $q = 0$ of rational continuity is redundant, since it follows from rational zero and $0 > 0$.

2.4 Probabilistic Power Domains

For given dcpo X, the *probabilistic power domain* $\mathcal{P}_D X$ is defined in [6, 7] as $\mathcal{P}_D X = [\Omega X \overset{mod}{\to} \mathbf{I}]$, the dcpo of *continuous evaluations* from ΩX to $\mathbf{I}$ with the pointwise order. A function μ from a frame to $\mathbf{I}$ is an evaluation if it satisfies the zero law $\mu(0) = 0$ and the modular law $\mu(a \vee b) + \mu(a \wedge b) = \mu a + \mu b$.

Since the definition of $\mathcal{P}_D X$ only refers to ΩX, it can be applied to any topsys X producing a dcpo and thus a topsys. Similarly, the probabilistic power locale construction $\mathcal{P}_L$ soon to be introduced only relies on ΩX and thus can be applied to any topsys. We shall prove (Theorem 3.2):

(1) For every topsys X, pt $\mathcal{P}_D X$ and pt $\mathcal{P}_L X$ are isomorphic posets (in fact dcpo's).

In Section 4, we shall introduce the probabilistic power infosys construction $\mathcal{P}_I$ mapping infosyses to infosyses. We shall prove (Theorem 5.2):

(2) For every infosys D, $\Omega \mathcal{P}_I D$ and $\Omega \mathcal{P}_L [D]$ are isomorphic frames.

Since the topsyses described by infosyses are continuous domains and thus localic (sober), (2) suffices to conclude:

(3) For every infosys D, $[\mathcal{P}_I D]$ and $\mathcal{P}_L [D]$ are isomorphic topsyses.

First, we can conclude from this that $\mathcal{P}_I$ preserves equivalence of infosyses, which would not be obvious a priori. Second, we can conclude that for every continuous domain X, $\mathcal{P}_L X$ is a continuous domain again, and thus a dcpo. The topsyses induced by dcpo's are spatial, whence (1) suffices to conclude

(4) For every continuous domain X, $\mathcal{P}_D X$ and $\mathcal{P}_L X$ are isomorphic topsyses (in fact continuous domains).

This is our main result: on continuous domains, $\mathcal{P}_D$, $\mathcal{P}_L$, and $\mathcal{P}_I$ coincide.

3 A Probabilistic Power Locale

In this section, we introduce the probabilistic power locale construction $\mathcal{P}_L$ and show that its power locales have the same points as Jones's power domains.

3.1 The Generators

In this subsection, we motivate our choice for the generators of $\Omega\mathcal{P}_L X$. Let us first analyze the probabilistic power domain construction $\mathcal{P}_D X = [\Omega X \overset{mod}{\longrightarrow} \mathbf{I}]$ for dcpo's X. It can be dismantled into two consecutive steps: first mapping the dcpo X into its frame of opens ΩX, and then applying a construction $D : F \mapsto [F \overset{mod}{\longrightarrow} \mathbf{I}]$ mapping frames into dcpo's. In investigating D, we may assume that it is applied to arbitrary frames, not just those frames arising from the Scott topology of a dcpo.

Analogously, we shall define a construction L mapping frames into locales, and then let $\mathcal{P}_L X = L(\Omega X)$. The locale $L(F)$ to a frame F should be close to the dcpo $D(F) = [F \overset{mod}{\longrightarrow} \mathbf{I}]$. Ideally, $\Omega L(F)$ would be the frame of Scott open sets of $D(F)$, but this frame is difficult to axiomatize. Instead, we shall consider the pointwise topology on the function space $[F \to \mathbf{I}]$. Its subbasic opens are $F(u, v)$ for members u of F and opens v in $\Omega\mathbf{I}$, where $F(u,v) = \{\mu : F \to \mathbf{I} \mid \mu u \in v\}$. Instead of all opens in $\Omega\mathbf{I}$, it suffices to consider the basic opens $\Uparrow q$ for q in $\mathbf{Q}_0^1$. Thus, the frame $\Omega L(F)$ will have generators $\langle u, q\rangle$ with u in F and q in $\mathbf{Q}_0^1$, where the intended meaning of $\langle u, q\rangle$ is $\{\mu \mid \mu u > q\}$.

Now, our goal is to find suitable relations on these generators such that the posets $\operatorname{pt} D(F)$ and $\operatorname{pt} L(F)$ become isomorphic. Indeed, we shall prove a little bit more:

Theorem 3.1 For every frame F, there is a continuous function $f : D(F) \to L(F)$, whose points part $\operatorname{pt} f : \operatorname{pt} D(F) \to \operatorname{pt} L(F)$ is an isomorphism.

We shall prove this theorem by first comparing the dcpo D_0 of *all* functions from F to $\mathbf{I}$ with a certain locale L_0, and then gradually introducing more restrictions on the functions of D_0 as well as more relations on the frame of L_0.

3.2 Arbitrary Functions

For our fixed frame F, we define D_0 to be the dcpo of *all* (not even necessarily monotonic) functions from F to $\mathbf{I}$. For the corresponding locale L_0, we present the frame ΩL_0 by the generators $\langle u, q\rangle$ for u in F and q in $\mathbf{Q}_0^1$ with two relations which are directly motivated by the two relations on $\Omega\mathbf{Q}_0^1$ of Subsection 2.3:

- Rational zero: $\langle u, 0\rangle = \top$ for all u in F.
- Rational continuity: $\bigvee_{r>q}\langle u, r\rangle = \langle u, q\rangle$ for all u in F and q in $\mathbf{Q}_0^1$.

The second relation implies that $\langle u, r\rangle \leq \langle u, q\rangle$ whenever $r > q$. Notice that '>' here is meant to refer to (the dual of) the infosys relation of Subsection 2.3, which differs from the usual meaning of '<' by the additional relationship $0 < 0$.

Now, we define a continuous function $f : D_0 \rightarrow L_0$ by defining $\Omega f : \Omega L_0 \rightarrow \Omega D_0$ with

$$\Omega f\langle u,q\rangle \;=\; \{\mu : F \rightarrow \mathbf{I} \mid \mu u > q\}$$

and $\mathrm{pt}\, f : \mathrm{pt}\, D_0 \rightarrow \mathrm{pt}\, L_0 = \mathit{Frame}(\Omega L_0, 2)$ with

$$\mathrm{pt}\, f \mu\,\langle u,q\rangle \;=\; [\mu u > q]$$

where $[\mu u > q] = \mathsf{T}$ if $\mu u > q$, and $= \mathsf{F}$ otherwise.

To show that this makes sense, we have to check several things. First $\{\mu \mid \mu u > q\}$ is indeed Scott open in D_0.

Second, $\mathrm{pt}\, f$ and Ωf fit together, i.e., for all μ in $\mathrm{pt}\, D_0$ and U in ΩL_0, $\mu \models \Omega f U$ iff $\mathrm{pt}\, f \mu \models U$. It suffices to show this for generators, and $\mu \models \Omega f\langle u,q\rangle$ iff $\mu u > q$ iff $\mathrm{pt}\, f\mu \models \langle u,q\rangle$ holds indeed.

Third, Ωf and $\mathrm{pt}\, f\mu$ preserve the two relations: for rational zero, $\mu u > 0$ holds for every μ in D_0; for rational continuity, note that $\mu u > q$ iff there is a rational r with $\mu u > r > q$.

In the sequel, we show that $\mathrm{pt}\, f : \mathrm{pt}\, D_0 \rightarrow \mathrm{pt}\, L_0$ is an isomorphism. The proof becomes simpler if $\mathbf{I}$ is considered as the set of ideals $\mathrm{pt}\, \mathbf{Q}_0^1$ of the infosys $\mathbf{Q}_0^1$. The dcpo's $\mathbf{I}$ and $\mathrm{pt}\, \mathbf{Q}_0^1$ are isomorphic, and directed join in $\mathbf{I}$ corresponds to directed union in $\mathrm{pt}\, \mathbf{Q}_0^1$. The definition of $\mathrm{pt}\, f$ then becomes

$$\mathrm{pt}\, f\mu\,\langle u,q\rangle \;=\; [q \in \mu u].$$

We claim that the inverse of $\mathrm{pt}\, f$ is $\gamma : \mathrm{pt}\, L_0 \rightarrow \mathrm{pt}\, D_0$ with

$$\gamma p u = \{q \in \mathbf{Q}_0^1 \mid p\langle u,q\rangle = \mathsf{T}\}$$

for p in $\mathrm{pt}\, L_0$ and u in F. If r in $\gamma p u$ and $r > q$, then q is in $\gamma p u$ since $\langle u,r\rangle \leq \langle u,q\rangle$. If q is in $\gamma p u$, then there is $r > q$ with r in $\gamma p u$ by rational continuity. 0 is always in $\gamma p u$ by rational zero. These facts suffice to show that $\gamma p u$ is a point (a directed lower set) because $\mathbf{Q}_0^1$ is linearly ordered. Thus, $\gamma p u$ is in $\mathrm{pt}\, \mathbf{Q}_0^1$ as required.

Next, we show that γ is the inverse of $\mathrm{pt}\, f$. One direction is easy:

$$q \in \gamma(\mathrm{pt}\, f\mu)u \;\text{ iff }\; \mathrm{pt}\, f\mu\,\langle u,q\rangle = \mathsf{T} \;\text{ iff }\; q \in \mu u.$$

Conversely, we claim $\mathrm{pt}\, f(\gamma p) = p$ for all p in $\mathrm{pt}\, L_0$. It suffices to show that both sides coincide on generators. The statement $\mathrm{pt}\, f(\gamma p)\langle u,q\rangle = \mathsf{T}$ is by definition of $\mathrm{pt}\, f$ equivalent to $q \in \gamma p u$, which in turn is equivalent to $p\langle u,q\rangle = \mathsf{T}$.

3.3 Restricted Functions

We carry on by comparing various subdcpo's of D_0 defined by restrictions on the functions with sublocales of L_0 defined by additional relations. This neither affects the definitions of Ωf, $\mathrm{pt}\, f$, and γ, nor the proof that $\mathrm{pt}\, f$ and γ are inverse to each other. We only have to show in any case that Ωf and $\mathrm{pt}\, f\mu$ preserve the additional relations and that γp satisfies the restrictions. For all but modularity, these proofs are quite obvious and omitted.

Monotonicity: Restriction: If $u \le v$, then $\mu u \le \mu v$.
Relations: If $u \le v$, then $\langle u, q\rangle \le \langle v, q\rangle$.

Continuity: Restriction: If $\mathcal{U} \subseteq F$ is directed, then $\mu(\bigvee \mathcal{U}) = \bigvee_{u\in\mathcal{U}} \mu u$.
Relations: If $\mathcal{U} \subseteq F$ is directed, then $\langle \bigvee \mathcal{U}, q\rangle = \bigvee_{u\in\mathcal{U}} \langle u, q\rangle$.

Zero law: Restriction: $\mu(0) = 0$.
Relations: If $q \ne 0$, then $\langle 0, q\rangle = \mathsf{F}$.
By the rational zero relation, $\langle 0, 0\rangle = \mathsf{T}$ holds. All these relations look quite contradictory in presence of $\bigvee_{r>0}\langle 0, r\rangle = \langle 0, 0\rangle$, but remember $0 > 0$.

Modularity: Restriction: For all u and v, $\mu u + \mu v = \mu(u \vee v) + \mu(u \wedge v)$.
To find the corresponding relations, we have to check when $\mu u + \mu v > q$ holds. It is the case iff there are rational numbers r and s with $\mu u > r$, $\mu v > s$, and $r + s = q$. This suggests the following
Relations: For all rational numbers q with $0 \sqsubseteq q \sqsubset 2$,

$$\bigvee\{\langle u, r\rangle \wedge \langle v, s\rangle \mid r, s \in \mathbf{Q}_0^1,\ r + s = q\} =$$
$$\bigvee\{\langle u \vee v, r\rangle \wedge \langle u \wedge v, s\rangle \mid r, s \in \mathbf{Q}_0^1,\ r + s = q\}.$$

Ωf preserves the relations by the reasoning above. To show that γp is modular, we perform the following computation, where q always ranges over the rationals with $0 \sqsubseteq q \sqsubset 2$:

$$\begin{array}{lll} p(\bigvee\{\langle u, r\rangle \wedge \langle v, s\rangle \mid r, s \in \mathbf{Q}_0^1,\ r + s = q\}) = \mathsf{T} & & \\ \text{iff}\ \ \bigvee\{p\langle u, r\rangle \wedge p\langle v, s\rangle \mid r, s \in \mathbf{Q}_0^1,\ r + s = q\} = \mathsf{T} & & (p \text{ is homomorphism}) \\ \text{iff}\ \ \exists r, s \in \mathbf{Q}_0^1 : r + s = q,\ p\langle u, r\rangle = \mathsf{T},\ p\langle v, s\rangle = \mathsf{T} & & (p \text{ maps to } \{\mathsf{F}, \mathsf{T}\}) \\ \text{iff}\ \ \exists r, s \in \mathbf{Q}_0^1 : r + s = q,\ \gamma p u > r,\ \gamma p v > s & & \\ \text{iff}\ \ \gamma p u + \gamma p v > q. & & \end{array}$$

An analogous computation may be performed for $s = \gamma p(u \vee v) + \gamma p(u \wedge v)$, and the relations assure that $s > q$ iff $\gamma p u + \gamma p v > q$, whence $s = \gamma p u + \gamma p v$.

This completes the proof of Theorem 3.1. Summarizing, we obtain:

Theorem 3.2 For every topsys X, let $\mathcal{P}_D X$ be the dcpo $[\Omega X \xrightarrow{mod} \mathbf{I}]$, and $\mathcal{P}_L X$ be the locale, whose frame of opens $\Omega\mathcal{P}_L X$ is presented by:

Generators: $\langle u, q\rangle$ with u in ΩX and q in $\mathbf{Q}_0^1$,

Relations: Rational zero: $\langle u, 0\rangle = \mathsf{T}$ for all u in ΩX;

Rational continuity: $\bigvee_{r>q}\langle u, r\rangle = \langle u, q\rangle$ for all u in ΩX and q in $\mathbf{Q}_0^1$;

Continuity: If $\mathcal{U} \subseteq \Omega X$ is directed, then $\langle \bigvee \mathcal{U}, q\rangle = \bigvee_{u\in\mathcal{U}}\langle u, q\rangle$;

Zero law: If $q \ne 0$, then $\langle 0, q\rangle = 0$;

Modularity: For all rational q with $0 \sqsubseteq q \sqsubset 2$, and all u, v in ΩX,

$$\bigvee\{\langle u, r\rangle \wedge \langle v, s\rangle \mid r, s \in \mathbb{Q}_0^1,\ r + s = q\} =$$
$$\bigvee\{\langle u \vee v, r\rangle \wedge \langle u \wedge v, s\rangle \mid r, s \in \mathbb{Q}_0^1,\ r + s = q\}.$$

Then there is a continuous function $f : \mathcal{P}_D X \rightarrow \mathcal{P}_L X$ defined by

$$\Omega f\langle u, q\rangle = \{\mu \in [\Omega X \overset{mod}{\rightarrow} \mathbf{I}] \mid \mu u > q\}$$

such that $\mathrm{pt}\, f : \mathrm{pt}\, \mathcal{P}_D X \rightarrow \mathrm{pt}\, \mathcal{P}_L X$ is a poset isomorphism. □

4 The Power Construction on Infosyses

In this section, we define the probabilistic construction $\mathcal{P}_I$ on infosyses. Subsection 4.1 deals with the tokens of the power infosyses, and Subsection 4.2 with their order.

4.1 The Tokens of the Power Infosys

In this subsection, we define the tokens of the power infosys $\mathcal{P}_I D$ of some given infosys D. We then investigate two different ways to interpret such power tokens as functions from the frame ΩD to $\mathbf{I}$. These functions will be used in the next subsection to define the order on the power tokens.

The power tokens may be thought of as formal convex combinations $\sum_{i=1}^{n} r_i \cdot a_i$ with a_i in D and r_i in $\mathbb{Q}_0^1$ such that $\sum_{i=1}^{n} r_i < 1$. This is formalized in the following definition, which was already conjectured in [12].

Definition 4.1 The tokens of $\mathcal{P}_I D$ are finite bags A of pairs (r, a) from $\mathbb{Q}_0^1 \times D$ with $\sum\{\!|r \mid (r, a) \in A|\!\} < 1$.

Bags or ***multi-sets*** are similar to sets, but elements may occur more than once in them. Notationally, we use $\{\!|.|\!\}$ for bags in contrast to $\{.\}$ for sets, but keep on using '$\emptyset$', '$\in$', and '$\subseteq$' as for sets. The bag union of two bags is denoted by '+', e.g., $\{\!|1|\!\} + \{\!|1, 2|\!\} = \{\!|1, 1, 2|\!\}$. Bags will be manipulated by means of *bag abstractions*, whose meaning is analogous to that of set abstractions, but keeping different occurrences of the same element apart. Thus, for instance $\sum\{\!|r \mid (r, a) \in \{\!|(0.2, b), (0.2, b), (0.2, c)|\!\}|\!\} = \sum\{\!|0.2, 0.2, 0.2|\!\} = 0.6$.

The power infosys of a finite infosys need not be finite because of the rational numbers involved. On the other hand, countability of infosyses is preserved by the power infosys operation.

In the sequel, we need some notions and notations concerning power tokens.

Definition 4.2 For power tokens A in $\mathcal{P}_I D$, let $\mathrm{set}\, A = \{a \mid (r, a) \in A\}$ be the *set* of D-tokens occurring in A, and $\mathrm{num}\, A = \{\!|r \mid (r, a) \in A|\!\}$ be the *bag* of numbers in A.

For power tokens A in $\mathcal{P}_I D$ and opens u in ΩD, let $A_p u = \sum\{\!|r \mid (r,a) \in A,\ a \in u|\!\}$ and $A_o u = \sum\{\!|r \mid (r,a) \in A,\ \Uparrow a \subseteq u|\!\}$.

We use the indices p and o to indicate that in the first case, the token a is considered as a *point* of the open u, whereas in the latter case, it codes for the *open* $\Uparrow a$. If the infosys D is reflexive, i.e., in the algebraic case, the conditions $a \in u$ and $\Uparrow a \subseteq u$ are equivalent, whence $A_p = A_o$ for all power tokens A. This is not valid in the general case.

In the sequel, we look at the elementary properties of the two functional interpretations $A_p, A_o : \Omega D \to \mathbf{Q}_0^1 \subseteq \mathbf{I}$.

Proposition 4.3 For all A in $\mathcal{P}_I D$:

(1) For all u in ΩD: $0 < A_p u \sqsubseteq A_o u \sqsubseteq \sum \mathsf{num}\, A < 1$.

(2) A_p is continuous, and A_o is monotonic.

(3) A_p satisfies the zero law: $A_p \emptyset = 0$.

(4) For all u in ΩD: $\emptyset_p u = \emptyset_o u = 0$.

Unfortunately, neither A_p nor A_u can be expected to be modular. Since joins in ΩD are unions, '$a \in u \vee v$ iff $a \in u$ or $a \in v$' holds, but since meets are in general not intersections, '$a \in u \wedge v$ iff $a \in u$ and $a \in v$' does not hold in general. Dually, '$\Uparrow a \subseteq u \wedge v$ iff $\Uparrow a \subseteq u$ and $\Uparrow a \subseteq v$' holds by the very nature of meets, but since $\Uparrow a$ is not a single token, '$\Uparrow a \subseteq u \vee v$ iff $\Uparrow a \subseteq u$ or $\Uparrow a \subseteq v$' does not hold in general. Fortunately, A_p and A_o satisfy some kind of modularity in co-operation.

Proposition 4.4
For all A in $\mathcal{P}_I D$ and u, v in ΩD, $A_p u + A_p v \sqsubseteq A_p(u \vee v) + A_o(u \wedge v) \sqsubseteq A_o u + A_o v$.

Proof: For a in D and u in ΩD, let $[a \in u]$ be 1 or 0 depending on if a is in u or not. Then $A_p u = \sum\{\!|r \cdot [a \in u] \mid (r,a) \in A|\!\}$, and $A_o u = \sum\{\!|r \cdot [\Uparrow a \subseteq u] \mid (r,a) \in A|\!\}$ with an analogous notation. Thus, the statement of the proposition can be derived from the more basic statement

$$[a \in u] + [a \in v] \sqsubseteq [a \in u \vee v] + [\Uparrow a \subseteq u \wedge v] \sqsubseteq [\Uparrow a \subseteq u] + [\Uparrow a \subseteq v],$$

which can be shown by case analysis. □

4.2 The Order of the Power Infosys

So far, we only defined the tokens of $\mathcal{P}_I D$. For the order, we shall establish several equivalent definitions in this subsection.

Definition 4.5 For two functions f and g from ΩD to $\mathbf{I}$, we define $f < g$ iff for all u in ΩD, $fu < gu$ holds. For two tokens A and B in $\mathcal{P}_I D$, we define $A < B$ iff $A_o < B_p$ holds.

So far, the token order is not quite effective because of the potentially infinite quantification over all opens. We shall soon derive effective characterizations.

The definition above has the virtue that transitivity can be shown easily: from $A < B$ and $B < C$, $A_o u < B_p u \sqsubseteq B_o u < C_p u$ for all opens u, whence $A < C$. Also, $\emptyset$ is readily seen to be the least power token: $\emptyset_o u = 0 < B_p u$ holds for all opens u. For interpolation however, some hard work is needed. Before doing it, we derive two effective characterizations.

Proposition 4.6 For a token A in $\mathcal{P}_I D$ and a monotonic function $\mu : \Omega D \rightarrow \mathbf{I}$, the following three statements are equivalent:

(1) $A_o < \mu$, i.e., for all u in ΩD, $A_o u < \mu u$ holds.

(2) $A_o u < \mu u$ holds for all u in $\mathcal{U}(A) = \{\uparrow \mathsf{set}\, S \mid S \subseteq A\}$.

(3) For all subbags S of A, $\sum \mathsf{num}\, S < \mu(\uparrow \mathsf{set}\, S)$ holds.

In (3), one may also quantify 'for all non-empty subbags'.

Notice how the universal quantification of (1) is reduced to the finite quantifications in (2) and (3).

Proof:

(1) $\Rightarrow$ (2) : Trivial; $\mathcal{U}(A)$ is a subset of ΩD.

(2) $\Rightarrow$ (3) : Let $u = \uparrow \mathsf{set}\, S$, which is in $\mathcal{U}(A)$. If (r, a) is in S, then a is in $\mathsf{set}\, S$, whence $\sharp a \subseteq u$. Thus,

$$\textstyle\sum \mathsf{num}\, S \;=\; \sum\{\!|r \mid (r, a) \in S|\!\} \;\sqsubseteq\; \sum\{\!|r \mid (r, a) \in A,\, \sharp a \subseteq u|\!\} \;=\; A_o u \;<\; \mu u.$$

(3) $\Rightarrow$ (1) : For given u, let $S = \{\!|(r, a) \in A \mid \sharp a \subseteq u|\!\}$. Then $\uparrow \mathsf{set}\, S \subseteq u$, whence with monotonicity of μ, $A_o u = \sum \mathsf{num}\, S < \mu(\uparrow \mathsf{set}\, S) \sqsubseteq \mu u$.

The empty subbag need not be included in (3), since $\sum \mathsf{num}\, \emptyset = 0 < s$ always holds.□

The proposition applies in particular to the case $\mu = B_p$ needed for $A < B$:

Corollary 4.7

For two tokens A and B in $\mathcal{P}_I D$, $B > A$ iff for all $S \subseteq A$, $B_p(\uparrow \mathsf{set}\, S) > \sum \mathsf{num}\, S$.

The corollary shows that the order on $\mathcal{P}_I D$ is decidable, if the order on D is decidable: $A < B$ can be checked by performing a finite number of comparisons of rational numbers

$\sum \mathsf{num}\, S = \sum\{\!|r \mid (r, a) \in S|\!\}$ and

$B_p(\uparrow \mathsf{set}\, S) = \sum\{\!|s \mid (s, b) \in B,\, \exists (r, a) \in S : b > a|\!\}$,

which can be computed with a finite number of comparisons in D and additions of rationals.

The intuitions of Vickers in [12] suggest a quite different definition of the order on the tokens. A token A is below a token B iff every element (r, a) can be split into some (r_i, a) with $\sum r_i = r$, then grown by enlarging both the rational number and the ground token, then recombined again to obtain some (not necessarily all) of the elements of B. A formalization of these intuitions is provided by Lemma 4.13 of [6], which handles the more concrete problem of comparing convex combinations of point evaluations. This leads to the following theorem:

Theorem 4.8 Let $A = \{\!|(r_1, a_1), \ldots, (r_n, a_n)|\!\}$ and $B = \{\!|(s_1, b_1), \ldots, (s_m, b_m)|\!\}$ be two power tokens with a fixed order to enumerate their elements, and let $I = \{1, \ldots, n\}$ and $J = \{1, \ldots, m\}$. Then the following are equivalent:

(1) $A < B$, i.e., for all opens u, $A_o u < B_p u$;

(2) there are numbers t_{ij} in I for every i in I and j in J, such that $t_{ij} \neq 0$ implies $a_i < b_j$, $\sum_{j \in J} t_{ij} = r_i$ for every i in I, and $\sum_{i \in I} t_{ij} < s_j$ for every j in J.

(3) as (2), but the t_{ij} are in Q_0^1, and $\sum_{j \in J} t_{ij} > r_i$.

Proof:

(1) $\Rightarrow$ (2) : This is essentially the proof of the Splitting Lemma 4.10 of [6] or Lemma 9.2 of [7]. It applies the Max-Flow Min-Cut Theorem 5.1 of [2]. In our case, it is applied to a graph with nodes $\perp$ (source), $1, \ldots, n, 1', \ldots, m'$, and $\top$ (sink), and edges from $\perp$ to i with capacities r_i, from i to j' with capacities 1 if $a_i < b_j$ and 0 otherwise, and from j' to $\top$ with capacities $\rho \cdot s_j$, where ρ is a previously chosen rational number with $\sum \mathsf{num}\, S < \rho \cdot B_p(\uparrow \mathsf{set}\, S)$ for all subbags S of A. The remainder of the proof is in analogy to [6, 7] and thus omitted.

(2) $\Rightarrow$ (3) : Choose a (rational) $0 \neq \rho < 1$ with still $\sum_{i \in I} t_{ij} < \rho \cdot s_j$ for all j in J. For every i in I and j in J, choose a rational t'_{ij} with $t_{ij} < t'_{ij} < t_{ij} \cdot \rho^{-1}$. The numbers t'_{ij} do the job for (3).

(3) $\Rightarrow$ (1) : We use Cor. 4.7 to prove $A < B$. Let S be a subbag of A, I' a corresponding subset of I, and $J' = \{j \in J \mid b_j \in \uparrow \mathsf{set}\, S\}$. We start with

$$\sum \mathsf{num}\, S = \sum_{i \in I'} r_i < \sum_{i \in I'} \sum_{j \in J} t_{ij}$$

We only need to sum those t_{ij} with $t_{ij} \neq 0$. For these, $a_i < b_j$ holds. Thus it suffices to consider those j with $b_j \in \uparrow \mathsf{set}\, S$, and we may continue

$$\cdots \sqsubseteq \sum_{i \in I'} \sum_{j \in J'} t_{ij} \sqsubseteq \sum_{j \in J'} \sum_{i \in I} t_{ij} < \sum_{j \in J'} s_j = B_p(\uparrow \mathsf{set}\, S)$$

□

The last statement of Theorem 4.8 enables us to prove interpolation. Let A and B as in the theorem with $A < B$, and let t_{ij} be the numbers of its last statement. For every i in I and j in J with $a_i < b_j$, we choose a ground token c_{ij} with $a_i < c_{ij} < b_j$, and construct the bag C of pairs (t_{ij}, c_{ij}). Then $A < C < B$ holds, as is easily verified using parts (2) or (3) of Theorem 4.8.

5 Comparing Power Infosys and Power Locale

In this section, we show — as announced in the beginning — that the frames of opens of the power infosys and the power locale are isomorphic.

5.1 The Two Frames

In this subsection, we present the two frames to be compared. Let D be some fixed infosys. The frame F_1 is $\Omega\mathcal{P}_I D$, which can be presented as shown in Section 2.2. The frame F_2 is $\Omega\mathcal{P}_L[D]$. It is defined in terms of $\Omega[D]$, which is isomorphic to ΩD. Thus, we arrive at the following presentations:

- Generators for F_1: $\sharp A$ for A in $\mathcal{P}_I D$.
- Relations:
  (1) Monotonicity: If $A < B$, then $\sharp A \geq \sharp B$;
  (2) All tokens: $\bigvee_{A \in \mathcal{P}_I D} \sharp A = \top$;
  (3) Meets: $\sharp A \wedge \sharp B \leq \bigvee_{C > A,B} \sharp C$.
  With monotonicity, one obtains '=' in (3). There is also the special case $A = B$ of this, which yields
  (4) $\sharp A = \bigvee_{C > A} \sharp C$.
- Generators for F_2: $\langle u, q\rangle$ with u in ΩD and q in $\mathbb{Q}_0^1$,
- Relations:
  (1) $\langle u, 0\rangle = \top$;
  (2) $\bigvee_{r > q} \langle u, r\rangle = \langle u, q\rangle$;
  (3) If $\mathcal{U} \subseteq \Omega X$ is directed, then $\langle \bigvee \mathcal{U}, q\rangle = \bigvee_{u \in \mathcal{U}} \langle u, q\rangle$;
  (4) If $q \neq 0$, then $\langle 0, q\rangle = 0$;
  (5) For all rational q with $0 \sqsubseteq q \sqsubset 2$, $\bigvee\{\langle u, r\rangle \wedge \langle v, s\rangle \mid r, s \in \mathbb{Q}_0^1, r + s = q\} = \bigvee\{\langle u \vee v, r\rangle \wedge \langle u \wedge v, s\rangle \mid r, s \in \mathbb{Q}_0^1, r + s = q\}$.

In case of F_2, relation (1) makes the cases $q = 0$ of (2), (3), and (5) redundant; we may assume $q \neq 0$ in (2) through (5).

5.2 A Homomorphism from F_2 to F_1

In view of the 'intended meaning' of the generators $\langle u, q\rangle$, we define $\varphi : F_2 \to F_1$ by

$$\varphi\langle u, q\rangle = \bigvee\{\sharp A \mid A_p u > q\}$$

We have to show that this definition preserves the relations of F_2.

(1) $\varphi\langle u, 0\rangle = \bigvee\{\sharp A \mid A_p u > 0\} = \bigvee\{\sharp A \mid A \in \mathcal{P}_I D\} = \top$ by relation (2) of F_1.

(2) $\bigvee_{r>q} \varphi\langle u, r\rangle = \bigvee_{r>q} \bigvee\{\sharp A \mid A_p u > r\} = \bigvee\{\sharp A \mid \exists r > q : A_p u > r\}$. By transitivity and interpolation in $\mathbb{Q}_0^1$, the latter equals $\bigvee\{\sharp A \mid A_p u > q\} = \varphi\langle u, q\rangle$.

(3) Here, we need continuity of A_p (Prop. 4.3 (2)). Let $(u_i)_{i\in I}$ be a directed family in ΩD.

$$\begin{aligned}\varphi\langle \bigvee_{i\in I} u_i, q\rangle &= \bigvee\{\sharp A \mid A_p(\bigvee_{i\in I} u_i) > q\}\\ &= \bigvee\{\sharp A \mid \exists i \in I : A_p u_i > q\}\\ &= \bigvee_{i\in I} \bigvee\{\sharp A \mid A_p u_i > q\}\\ &= \bigvee_{i\in I} \varphi\langle u_i, q\rangle\end{aligned}$$

(4) By Prop. 4.3 (3), $A_p 0 = 0$ holds. Thus, $\varphi\langle 0, q\rangle = \bigvee\{\sharp A \mid A_p 0 > q\}$ is an empty join if $q \neq 0$.

(5) For modularity, we compute

$$\begin{aligned}\bigvee_{r+s=q} \varphi\langle u, r\rangle \wedge \varphi\langle v, s\rangle &= \bigvee_{r+s=q}(\bigvee\{\sharp A \mid A_p u > r\}) \wedge (\bigvee\{\sharp B \mid B_p v > s\})\\ &= \bigvee\{\sharp A \wedge \sharp B \mid r+s=q,\ A_p u > r,\ B_p v > s\}\end{aligned}$$

With the '=' version of relation (3) of F_1, we obtain

$$\bigvee\{\sharp C \mid C > A, B,\ A_p u + B_p v > q\}.$$

Applying relation (4) of F_1 yields

$$\bigvee\{\sharp D \mid D > C > A, B,\ A_p u + B_p v > q\}.$$

In this situation, $A_p u + B_p v > q$ implies $C_p u + C_p v > q$. Conversely, if $D > C$ and $C_p u + C_p v > q$, then we can interpolate a new C' between D and C and let $A = B = C$. Thus, the join above equals

$$x = \bigvee\{\sharp D \mid D > C,\ C_p u + C_p v > q\}.$$

Analogously, the other side of the modularity relation becomes

$$y = \bigvee\{\sharp D \mid D > C,\ C_p(u \vee v) + C_p(u \wedge v) > q\}.$$

We have to show $x = y$. For this, we use Prop. 4.4. It states

$$C_p u + C_p v \sqsubseteq C_p(u \vee v) + C_o(u \wedge v) \sqsubseteq C_o u + C_o v.$$

For $x \leq y$, we interpolate $D > C$ to $D > C' > C$. Then $C'_p > C_o \sqsupseteq C_p$, whence

$$C'_p(u \vee v) + C'_p(u \wedge v) \sqsupseteq C_p(u \vee v) + C_o(u \wedge v) \sqsupseteq C_p u + C_p v > q.$$

The other relation $x \geq y$ is shown analogously.

Now, we have shown that φ preserves all relations of F_2. Thus, it extends to a frame homomorphism $\varphi : F_2 \to F_1$.

5.3 A Homomorphism from F_1 to F_2

To establish a frame homomorphism $\psi : F_1 \to F_2$, we have to specify $\psi(\sharp A)$ for A in $\mathcal{P}_f D$. Here, we refer to Prop. 4.6, which says $A_o < \mu$ iff for all subbags S of A, $\sum \mathsf{num}\, S < \mu(\uparrow \mathsf{set}\, S)$. This motivates the following definition:

$$\psi(\sharp A) = \bigwedge_{S \subseteq A} \gamma S \quad \text{where} \quad \gamma S = \langle \uparrow \mathsf{set}\, S, \textstyle\sum \mathsf{num}\, S\rangle,$$

which involves a finite meet by finiteness of A. We have to prove that ψ preserves the relations of F_1.

For monotonicity, we have to show that $A < B$ implies $\psi(\sharp A) \geq \psi(\sharp B)$, or $\bigwedge_{S \subseteq A} \gamma S \geq \bigwedge_{T \subseteq B} \gamma T$. To prove this, it is sufficient to show that for each $S \subseteq A$, there is $T \subseteq B$ with $\gamma S \geq \gamma T$. For $S \subseteq A$, let $T = \{\!|(r, b) \in B \mid b \in \uparrow\mathsf{set}\, S|\!\}$. Then $\mathsf{set}\, T \subseteq \uparrow\mathsf{set}\, S$, whence $\uparrow\mathsf{set}\, T \leq \uparrow\mathsf{set}\, S$. Moreover, $\sum \mathsf{num}\, T = B_p(\uparrow\mathsf{set}\, S) > \sum \mathsf{num}\, S$ holds by $B > A$ and Cor. 4.7. Since the generators of F_2 are monotonic in the first and anti-monotonic in the second argument, these facts imply $\gamma T \leq \gamma S$.

For the all-tokens relation, we have to show $\bigvee\{\psi(\sharp A) \mid A \in \mathcal{P}_I D\} = \mathsf{T}$. This equality holds, since $\psi(\sharp\emptyset) = \gamma\emptyset = \langle\emptyset, 0\rangle = \mathsf{T}$ by relation (1) of F_2.

Finally, for the meets relation, we have to show $\psi(\sharp A) \wedge \psi(\sharp B) \leq \bigvee_{C > A,B} \psi(\sharp C)$. This turns out to be difficult and is postponed until later. For the moment, we assume that it has been shown, so that $\psi : F_1 \to F_2$ is a well defined frame homomorphism.

5.4 The Homomorphisms are Inverse

Now, we show that the two frame homomorphisms $\varphi : F_2 \to F_1$ and $\psi : F_1 \to F_2$ are inverse to each other. It suffices to apply φ and ψ to generators.

First, we show $\varphi(\psi(\sharp A)) = \sharp A$ for all A in $\mathcal{P}_I D$.

$$\begin{aligned} \varphi(\psi(\sharp A)) &= \bigwedge_{S \subseteq A} \varphi\langle\uparrow\mathsf{set}\, S, \textstyle\sum \mathsf{num}\, S\rangle \\ &= \bigwedge_{S \subseteq A} \bigvee\{\sharp B \mid B_p(\uparrow\mathsf{set}\, S) > \textstyle\sum \mathsf{num}\, S\} \end{aligned}$$

We have to show that this equals $\sharp A$. For this, we do not use the presentation of F_1 by generators and relations, but the concrete form of F_1 as consisting of the 'open' subsets of $\mathcal{P}_I D$, with join being union, meet being intersection followed by '$\uparrow$', and $\sharp A = \{B \mid B > A\}$.

First, let D be in the meet above. Then $D > C$ for some C such that for all $S \subseteq A$, $C > B^S$ for some B^S with $B^S_p(\uparrow\mathsf{set}\, S) > \sum \mathsf{num}\, S$. By $C > B^S$, for all $S \subseteq A$, $C_p(\uparrow\mathsf{set}\, S) > \sum \mathsf{num}\, S$ holds. By Cor. 4.7, this just means $C > A$. Thus, $D > C > A$, whence D is in $\sharp A$.

Conversely, let D be in $\sharp A$. By interpolation, let $D > C > B > A$. By Cor. 4.7, $B > A$ means $B_p(\mathsf{set}\, S) > \sum \mathsf{num}\, S$ for all $S \subseteq A$. By $C > B$, C is in $\bigvee\{\sharp B \mid B_p(\uparrow\mathsf{set}\, S) > \sum \mathsf{num}\, S\}$ for all $S \subseteq A$. Thus, C is in $\bigcap_{S \subseteq A} \cdots$, and because of $D > C$, D is in $\bigwedge_{S \subseteq A} \cdots$.

Next, we compute $\psi(\varphi\langle u, q\rangle) = \psi(\bigvee\{\sharp A \mid A_p u > q\}) = \bigvee\{\bigwedge_{S \subseteq A} \gamma S \mid A_p u > q\}$.

First, we show that this is below $\langle u, q\rangle$. For this, it suffices to show that for every A with $A_p u > q$, there is some $S \subseteq A$ with $\gamma S \leq \langle u, q\rangle$. Given A with $A_p u > q$, let $S = \{\!|(r, a) \in A \mid a \in u|\!\}$. Then $\mathsf{set}\, S \subseteq u$, whence $\uparrow\mathsf{set}\, S \leq u$; and $\sum \mathsf{num}\, S = A_p u > q$. By monotonicity of the generators of F_2 in the first argument and anti-monotonicity in the second, $\gamma S = \langle\uparrow\mathsf{set}\, S, \sum \mathsf{num}\, S\rangle \leq \langle u, q\rangle$ holds.

The final thing to show is $\psi(\varphi(u,q)) \geq \langle u,q\rangle$. Again, this turns out to be difficult, and we postpone it until later. Except for the two postponed statements, the proof of $F_1 \cong F_2$ is now completed.

5.5 The Missing Relations

Let us now analyze the two relations, whose proofs were postponed. The meet relation is

$$\psi(\Uparrow A) \wedge \psi(\Uparrow B) \leq \bigvee_{C>A,B} \psi(\Uparrow C)$$

or

$$\bigwedge_{R \subseteq A} \gamma R \wedge \bigwedge_{S \subseteq B} \gamma S \leq \bigvee\{\bigwedge_{T \subseteq C} \gamma T \mid C > A, B\}.$$

By Cor. 4.7, the condition $C > A$ may be replaced by $C_p(\uparrow\mathsf{set}\, R) > \sum \mathsf{num}\, R$ for all $R \subseteq A$, and analogously for $C > B$.

After renaming the bound variables, the missing part of the inverse relations is

$$\langle u,q\rangle \leq \bigvee\{\bigwedge_{T \subseteq C} \gamma T \mid C_p u > q\}.$$

Written this way, the two postponed relations reveal a common structure. On the left, there is a finite meet of generators $\langle u_i, q_i\rangle$, and the join on the right is quantified over those C with $C_p u_i > q_i$ for all i. Thus, both relations may be derived from a more general *critical lemma*:

Lemma 5.1 For every infosys D, for every finite index set I and all families $(u_i)_{i\in I}$ in ΩD and $(q_i)_{i\in I}$ in $\mathbf{Q}_0^1$,

$$\bigwedge_{i\in I}\langle u_i, q_i\rangle \leq \bigvee\{\bigwedge_{S \subseteq C}\langle \uparrow\mathsf{set}\, S, \textstyle\sum \mathsf{num}\, S\rangle \mid C \in \mathcal{P}_I D,\ C_p u_i > q_i\ \forall i \in I\}$$

holds in $F_2 = \Omega\mathcal{P}_L[D]$.

Because of the complexity of our proof of the critical lemma, we devote the whole next section to it. Summarizing the results of this section, we have shown — modulo a proof of the critical lemma — the following theorem:

Theorem 5.2 For every infosys D, the frames $\Omega\mathcal{P}_I D$ and $\Omega\mathcal{P}_L[D]$ are isomorphic.

6 Proof of the Critical Lemma

In this section, the critical lemma will be proved. The very basic idea of the proof is taken from the proof of Lemma 5.3 in [6] or Lemma 8.3 in [7], which state a vaguely similar property involving points and concrete open sets. The added difficulty in our proof is due to the fact that we have to work pointless, because the spatiality of the locale of F_2 is a priori unknown. (A posteriori, it follows from Theorem 5.2.)

6.1 Outline of the Proof

The above mentioned proof of Jones employs so-called *crescents*, which are set differences of two concrete open sets. To mimic this, we set up a new frame F_3 in Subsection 6.3 with generators $\langle u, v, q\rangle$, whose intended spatial meaning is the set of all μ with $\mu(u \setminus v) > q$, or $\mu u - \mu(u \wedge v) > q$. With an appropriate choice of relations for F_3, the obvious assignment $\langle u, q\rangle \mapsto \langle u, 0, q\rangle$ becomes a frame homomorphism $\alpha : F_2 \to F_3$.

Let the critical lemma be $L \leq R$ in F_2. The added possibilities of F_3 allow proving $\alpha L \leq \alpha R$ in F_3 in vague analogy to Jones's proof of the Lemmas mentioned above (Subsection 6.4). After having managed this, we only have to show that α is an order embedding; then $\alpha L \leq \alpha R$ in F_3 implies $L \leq R$ in F_2.

By the Corollary in paragraph II, 2.6 of [5, page 53], F_2 can be embedded by a frame homomorphism η into a complete Boolean algebra G. Given η and G, we are able to define a (non-trivial) frame homomorphism $\beta : F_3 \to G$ with $\beta \circ \alpha = \eta$ (Subsection 6.6). Then α is an embedding, since η is an embedding.

The remaining subsections 6.2 and 6.5 introduce auxiliary notation to master the complexities of the proofs.

6.2 Real Valued Functions I

Before we present F_3 and prove the α-image of the critical lemma in F_3, we introduce some pieces of auxiliary notation, which allows for replacing complex join-and-meet expressions by simple arithmetically looking expressions with familiar laws.

Let F be an arbitrary frame and X the locale with $\Omega X = F$. Then continuous functions $f : X \to \mathbf{R}_0^\infty$ with values in the positive reals (including 0 and ∞) correspond to frame homomorphisms $\Omega f : \Omega\mathbf{R}_0^\infty \to F$. Analogously to the infosys $\mathbf{Q}_0^1$ for $\mathbf{I}$, an infosys for $\mathbf{R}_0^\infty$ is given by the positive rationals $\mathbf{Q}_0^+$ (including 0, but not ∞), with order '$<$' which is the usual order '$\sqsubset$' plus the one additional relationship $0 < 0$. Like $\Omega\mathbf{Q}_0^1$, the frame $\Omega\mathbf{R}_0^\infty \cong \Omega\mathbf{Q}_0^+$ can be presented by generators $\sharp q$ for q in $\mathbf{Q}_0^+$ and the two relations of rational zero and rational continuity. Thus, frame homomorphisms from $\Omega\mathbf{R}_0^\infty$ to F correspond to functions defined on the generators $\sharp q$ satisfying the two relations.

To obtain a real notational benefit, we now forget about X, write f for Ωf, and q for $\sharp q$. Thus, we define:

Definition 6.1 For every frame F, let $\mathcal{R}F$ be the set of all functions f from $\mathbf{Q}_0^+$ to F satisfying the two properties

(1) $f0 = \mathsf{T}$;

(2) $\bigvee_{r>q} fr = fq$ (and thus $r > q$ implies $fr \leq fq$).

A function f is *bounded* iff there is q with $fq = \mathsf{F}$.

We now define several operations of $\mathcal{R}F$. For every operation, satisfaction of the two conditions for $\mathcal{R}F$ has to be shown. We shall omit most of these proofs.

The members of $\mathcal{R}F$ are ordered pointwise: $f \le g$ iff for all q, $fq \le gq$ holds. Obviously, if $f \le g$ and g is bounded, then f is bounded.
Non-empty joins may be defined pointwise: $(\bigvee_{i\in I} f_i)q = \bigvee_{i\in I}(f_i q)$.
The empty join 0 or F is given by $0(0) = \mathsf{T}$ and $0(q) = \mathsf{F}$ for $q \ne 0$. It is not given pointwise, since $0(0)$ is T instead of F.
Finite meets (including the empty meet T) are given pointwise: $(\bigwedge_{i\in I} f_i)q = \bigwedge_{i\in I}(f_i q)$. With these joins and meets, $\mathcal{R}F$ becomes a frame.

We define a special member 1 of $\mathcal{R}F$ by $1(q) = \mathsf{T}$ for $q \sqsubset 1$ and F otherwise. In spatial intuitions, this corresponds to the function with constant real value 1. Thus, functions from the locale of F to I exactly correspond to those f in $\mathcal{R}F$ with $f \le 1$, or equivalently $f1 = \mathsf{F}$.

For two reals x and y, $x + y > q$ holds iff $x > r$ and $y > s$ for some rationals r and s with $r + s = q$. Thus, we define

$$(f+g)q \;=\; \bigvee_{r,s:\,r+s=q} fr \wedge gs.$$

(This is the same kind of expression as used in the modularity relation.) It is easy to show that ($\mathcal{R}F$, $+$, 0) forms a commutative monoid.
Because they are given pointwise, addition preserves non-empty joins: $f + \bigvee_{i\in I} g_i = \bigvee_{i\in I}(f + g_i)$. In particular, it is continuous and thus monotonic. By monotonicity, $f = f + 0 \le f + g$ always holds, whence $f \vee g \le f + g$. Addition does not preserve the empty join, since $f + 0 = f$ holds instead of $f + 0 = 0$.
Because of their pointwise nature, finite joins and non-empty meets of bounded functions are bounded. Also the sum of two bounded functions is bounded: if $fr_0 = \mathsf{F}$ and $gs_0 = \mathsf{F}$, then $(f+g)(r_0 + s_0) = \mathsf{F}$, since $r + s = r_0 + s_0$ implies $r \sqsupseteq r_0$ or $s \sqsupseteq s_0$.

The benefit of the new notations becomes obvious when we consider the frame F_2. For every u in ΩD, the assignment $q \mapsto \langle u, q\rangle$ for $q \sqsubset 1$ and $q \mapsto \mathsf{F}$ for $q \sqsupseteq 1$ becomes a function in $\mathcal{R}F_2$ by the relations of rational zero and rational continuity. We call this function $[u]$. Abstracting out the q's in the generators, we reach at a formal 'presentation' of $\mathcal{R}F_2$ with 'generators' $[u]$ for u in ΩD and 'relations'

- Bounded by one: $[u] \le 1$;
- Continuity: for directed families $(u_i)_{i\in I}$, $[\bigvee_{i\in I} u_i] = \bigvee_{i\in I}[u_i]$;
- Zero law: $[0] = 0$;
- Modularity: $[u \vee v] + [u \wedge v] = [u] + [v]$.

We do not claim that this 'presentation' presents anything directly, although it could be probably achieved by some more work, but we consider it as notational shorthand for our presentation of F_2. Intuitively and informally, $[u]$ can be best understood as the size or area of the 'region' u; the relations above then get a quite appealing interpretation.

6.3 The Frame F_3

We now present the frame F_3 by giving a 'presentation' of $\mathcal{R}F_3$: it has generators $[u,v]$ for every u, v in ΩD, whose intuitive meaning is the size or area of the 'region' $u \setminus v$, and relations

(1) Bounded by one: $[u,v] \leq 1$;

(RC) Restricted Continuity: for directed families $(u_i)_{i\in I}$ and v with $v \leq u_i$ for all i, $[\bigvee_{i\in I} u_i, v] = \bigvee_{i\in I}[u_i, v]$;

(0) Zero law: $[0,0] = 0$;

($\wedge$) Meet law: $[u,v] = [u, u \wedge v]$;

($\vee$) Join law: $[u,v] = [u \vee v, v]$;

(S) Split law: if $u \geq v \geq w$, then $[u,w] = [u,v] + [v,w]$.

Again, this should not be understood as an actual presentation of anything, but as shorthand for a presentation of F_3, which results by adding q's. Thus, the generators for F_3 are $\langle u,v,q\rangle = [u,v](q)$, and the split law for instance becomes: if $u \geq v \geq w$, then for all q, $\langle u,w,q\rangle = \bigvee_{r,s:\,r+s=q}\langle u,v,r\rangle \wedge \langle v,w,s\rangle$.

A frame homomorphism $\alpha : F_2 \to F_3$ is specified by $\alpha\langle u,q\rangle = \langle u,0,q\rangle$, or in shorthand $\alpha[u] = [u,0]$. (Actually, it should be $\alpha \circ [u] = [u,0]$, but we drop '$\circ$'.) Preservation of the relations of F_2 is immediate except for modularity. In the sequel, we shall take the relations for $\mathcal{R}F_3$ as axioms and derive a host of conclusions, including modularity, needed for the proof of the α-image of the critical lemma.

(C) Full Continuity: for directed families $(u_i)_{i\in I}$ and arbitrary v, $[\bigvee_{i\in I} u_i, v] = \bigvee_{i\in I}[u_i, v]$.

Proof: Applying the join law on both sides yields $[\bigvee_{i\in I} u_i \vee v, v] = \bigvee_{i\in I}[u_i \vee v, v]$, which holds by restricted continuity (RC). □

(M1) Monotonicity: if $u \leq u'$, then $[u,v] \leq [u',v]$.

Proof: Directly from full continuity (C), or from the split law like (M2) below. □

(M2) Anti-Monotonicity: if $v \leq v'$, then $[u,v] \geq [u,v']$.

Proof: $[u,v] \overset{\wedge}{=} [u, u \wedge v] \overset{S}{=} [u, u \wedge v'] + [u \wedge v', u \wedge v] \geq [u, u \wedge v'] \overset{\wedge}{=} [u, v']$. □

($\vee'$) Extended join law: if $v' \leq v$, then $[u \vee v', v] = [u,v]$.

Proof: $[u,v] \overset{M1}{\leq} [u \vee v', v] \overset{M1}{\leq} [u \vee v, v] \overset{\vee}{=} [u,v]$. □

($\wedge'$) Extended meet law: if $u' \geq u$, then $[u, u' \wedge v] = [u,v]$.

Proof: $[u,v] \overset{M2}{\leq} [u, u' \wedge v] \overset{M2}{\leq} [u, u \wedge v] \overset{\wedge}{=} [u,v]$. □

(E) Extinction: if $u \leq v$, then $[u,v] = 0$.

Proof: $[u,v] = [0 \vee u, v] \overset{\vee'}{=} [0,v] \overset{\wedge}{=} [0,0] \overset{0}{=} 0$. □

The next property is the α-image of modularity.

(Mod) $[u \vee v, 0] + [u \wedge v, 0] = [u, 0] + [v, 0]$.

Proof:

$$\begin{aligned} [u \vee v, 0] + [u \wedge v, 0] &\overset{S}{=} [u \vee v, v] + [v, 0] + [u \wedge v, 0] \\ &\overset{\vee,\wedge}{=} [u, u \wedge v] + [u \wedge v, 0] + [v, 0] \\ &\overset{S}{=} [u, 0] + [v, 0] \qquad \square \end{aligned}$$

The following properties are auxiliary lemmas needed in the proof of the α-image of the critical lemma. The first statement shows how to partition $[u, v]$ into parts inside and outside some w.

(1) $[u, v] = [u \wedge w, v] + [u, v \vee w]$.

Proof: By ($\vee$) and split, $[u, v] = [u \vee v, v] = [u \vee v, (u \wedge w) \vee v] + [(u \wedge w) \vee v, v]$. By ($\vee$), the second summand equals $[u \wedge w, v]$. By ($\vee'$), the first becomes $[u, (u \wedge w) \vee v]$. By two applications of ($\wedge$), this equals $[u, w \vee v]$. $\square$

Next, we generalize (1) to a finite number of w's.

(2) *If W is a finite set of opens, then* $[u, v] = \sum_{T \subseteq W} [u \wedge \bigwedge T, v \vee \bigvee (W \setminus T)]$.

Proof: The proof is performed by induction on $|W|$. For $W = \emptyset$, the sum just equals $[u, v]$. For $W \neq \emptyset$, let w be a member of W and $W' = W \setminus \{w\}$. By (1), $[u, v]$ is $[u \wedge w, v] + [u, v \vee w]$. By induction hypothesis, we obtain

$$[u \wedge w, v] = \sum_{T \subseteq W'} [u \wedge w \wedge \bigwedge T, v \vee \bigvee (W' \setminus T)]$$

Replacing T by $T \cup \{w\}$, this becomes

$$\sum_{T : w \in T \subseteq W} [u \wedge \bigwedge T, v \vee \bigvee (W \setminus T)]$$

The other summand is treated similarly:

$$\begin{aligned} [u, v \vee w] &= \textstyle\sum_{T \subseteq W'} [u \wedge \bigwedge T, v \vee w \vee \bigvee (W' \setminus T)] \\ &= \textstyle\sum_{T : w \notin T \subseteq W} [u \wedge \bigwedge T, v \vee \bigvee (W \setminus T)] \qquad \square \end{aligned}$$

For the actual proof of the critical lemma, we need a slight variant of (2):

(3) *Let U be a finite set of opens. For all u in U:* $[u, 0] = \sum_{T : u \in T \subseteq U} [\bigwedge T, \bigvee (U \setminus T)]$.

Proof: Apply (2) to $[u, 0]$ with $W = U \setminus \{u\}$. $\square$

The next statement shows how to transform a join into a sum.

(4) $[u \vee u', v] = [u, v] + [u', u \vee v]$.

Proof: Applying (1) with $w = u$, we obtain $[u \vee u', v] = [(u \vee u') \wedge u, v] + [u \vee u', v \vee u]$. The first summand is $[u, v]$, and with ($\vee'$), the second becomes $[u', u \vee v]$. $\square$

The next step is the generalization of (4) to arbitrary finite joins.

(5) $[\bigvee_{i=1}^{n} u_i, v] = \sum_{i=1}^{n} [u_i, v \vee \bigvee_{j=1}^{i-1} u_j]$

Proof: For $n = 0$, the equation is $[0, v] = 0$, which is true by (E). For $n > 0$, we compute:

$$\begin{aligned}&\textstyle\sum_{i=1}^{n-1}[u_i, v \vee \bigvee_{j=1}^{i-1} u_j] + [u_n, v \vee \bigvee_{j=1}^{n-1} u_j]\\ &\overset{IH}{=} [\textstyle\bigvee_{i=1}^{n-1} u_i, v] + [u_n, v \vee \bigvee_{j=1}^{n-1} u_j]\\ &\overset{4}{=} [\textstyle\bigvee_{i=1}^{n} u_i, v] \qquad \Box\end{aligned}$$

The next statement is another form of modularity.

(7) **If $u_1 \geq v_1$ and $u_2 \geq v_2$, then $[u_1, v_1] + [u_2, v_2] = [u_1 \vee u_2, v_1 \vee v_2] + [u_1 \wedge u_2, v_1 \wedge v_2]$.**

Proof: By (1) with $w = v_2$, $[u_1, v_1] = [u_1 \wedge v_2, v_1] + [u_1, v_1 \vee v_2]$.
By (1) with $w = u_1$, $[u_2, v_2] = [u_1 \wedge u_2, v_2] + [u_2, u_1 \vee v_2]$.
We show that the right hand side can be partitioned into the same four summands.
By (4), $[u_1 \vee u_2, v_1 \vee v_2] = [u_1, v_1 \vee v_2] + [u_2, u_1 \vee v_1 \vee v_2]$ holds. By $u_1 \geq v_1$, the second summand simplifies to $[u_2, u_1 \vee v_2]$.
By (1) with $w = v_2$, $[u_1 \wedge u_2, v_1 \wedge v_2] = [u_1 \wedge u_2 \wedge v_2, v_1 \wedge v_2] + [u_1 \wedge u_2, (v_1 \wedge v_2) \vee v_2]$.
By $u_2 \geq v_2$, the first summand simplifies to $[u_1 \wedge v_2, v_1 \wedge v_2]$, which by ($\wedge'$) equals $[u_1 \wedge v_2, v_1]$. $\Box$

Our next goal is to show that the sum of the areas of a finite number of disjoint regions is bounded by the area of a region that covers them all. To formalize disjointness in our framework, consider proper sets. The intersection of $A \setminus B$ and $A' \setminus B'$ is $(A \cap A') \setminus (B \cup B')$; it is empty iff $A \cap A' \subseteq B \cup B'$.

(8) Let $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ be opens with $u_i \wedge u_j \leq v_i \vee v_j$ for $i \neq j$ (disjointness condition). Then $\sum_{i=1}^{n}[u_i, v_i] \leq [\bigvee_{i=1}^{n} u_i, 0]$.

Proof: Applying ($\wedge$), we may replace v_i by $u_i \wedge v_i$. The disjointness condition then still holds. Thus, we may assume without restriction $u_i \geq v_i$ for all i.

Before proving the statement by induction on n, we consider the effect of one application of (7) from left to right to two summands of $\sum[u_i, v_i]$. By simple computations, it can be checked that such a rewriting step keeps the number of summands and the join $\bigvee u_i$ invariant, and preserves the conditions $u_i \geq v_i$ and the disjointness condition.

For $n = 0$, $0 \leq [0, 0]$ holds. For the case of $n + 1$, consider $[u_0, v_0] + [u_1, v_1] + \cdots + [u_n, v_n]$. We rewrite this expression n times by (7), going from left to right:

$$\begin{aligned}&[u_0, v_0] + [u_1, v_1] + \cdots + [u_n, v_n]\\ &= [u_0 \wedge u_1, v_0 \wedge v_1] + [u_0 \vee u_1, v_0 \vee v_1] + [u_2, v_2] + \cdots + [u_n, v_n]\\ &= [u_0 \wedge u_1, v_0 \wedge v_1] + [(u_0 \vee u_1) \wedge u_2, (v_0 \vee v_1) \wedge v_2]\\ &\qquad + [u_0 \vee u_1 \vee u_2, v_0 \vee v_1 \vee v_2] + [u_3, v_3] + \cdots + [u_n, v_n]\\ &= \cdots\end{aligned}$$

The final outcome is $\sum_{i=1}^{n}[u_i', v_i'] + [u, v]$, where $u_i' = (\bigvee_{j=0}^{i-1} u_j) \wedge u_i$, $v_i' = (\bigvee_{j=0}^{i-1} v_j) \wedge v_i$, $u = \bigvee_{i=0}^{n} u_i$, and $v = \bigvee_{i=0}^{n} v_i$. As indicated above, the rewriting steps preserve the disjointness condition. Thus, the induction hypothesis may be used to show that the sum is bounded by $[u,v] + [\bigvee_{i=1}^{n} u'_i, 0]$. Using the disjointness condition, we may compute:

$$u'_i \;=\; \bigvee_{j=0}^{i-1}(u_j \wedge u_i) \;\le\; \bigvee_{j=0}^{i-1}(v_j \vee v_i) \;\le\; v$$

Thus,

$$[u,v] + [\bigvee_{i=1}^{n} u'_i, 0] \overset{M1}{\le} [u,v] + [v,0] \overset{S}{=} [u,0]$$

which is the required result. □

Next, we proof that summands as occurring in (3) satisfy the disjointness condition of (8).

(9) Let U be a finite set of opens, and let T vary over the subsets of U. The families $u^T = \bigwedge T$ and $v^T = \bigvee(U \setminus T)$ satisfy the disjointness condition of (8).

Proof: If $T \neq T'$, then without restriction, there is u in T with u not in T'. Then $u^T \wedge u^{T'} \le \bigwedge T \le u \le \bigvee(U \setminus T') \le v^T \vee v^{T'}$. □

For the last auxiliary statement, opens have to be considered as token sets, to which set difference and subset relation can be applied.

(10) $[u,v] = \bigvee_{F \subseteq_{fin} u \setminus v}[\uparrow F, v]$; this join is directed.

Proof: The join is directed by monotonicity of '$\uparrow$'. Thus, continuity (C) can be applied, and we have to show $[u,v] = [\bigvee_{F \subseteq_{fin} u \setminus v} \uparrow F, v]$. The relation '$\ge$' directly follows from monotonicity (M1). For '$\le$', we show $u \le v \vee \bigvee_{F \subseteq_{fin} u \setminus v} \uparrow F$, and then apply (M1) and ($\vee$). Let a be in u. If a is in v, we are done. Otherwise, let $a > b \in u$. Token b is not in v, since a is not in v. Thus, a is in $\uparrow\{b\}$ where $\{b\} \subseteq u \setminus v$. □

For the application of the statements above, it is useful to note how statements involving sums may be turned into statements involving meets. If $f_1, \ldots, f_n$ are functions in $\mathcal{R}F$ for some frame F and we know $\sum_{i=1}^{n} f_i \le g$, then $\bigwedge_{i=1}^{n} f_i q_i \le g(\sum_{i=1}^{n} q_i)$ follows, since the meet on the left is just one join component of the expanded form of $(\sum_i f_i)(\sum_i q_i)$.

6.4 The Critical Lemma in the Frame F_3

With the auxiliary statements collected in the previous subsection, we are now able to prove the α-image of the critical lemma. The statement to be shown is:

$$\bigwedge_{i \in I}\langle u_i, 0, q_i\rangle \le \bigvee\{\bigwedge_{S \subseteq A}\langle \uparrow \mathsf{set}\, S, 0, \sum \mathsf{num}\, S\rangle \mid A \in \mathcal{P}_I D,\ A_p u_i > q_i\ \forall i \in I\}$$

for finite families $u_1, \ldots, u_n$ and $q_1, \ldots, q_n$.

For $n = 0$, the statement holds, since $A = \emptyset$ is then involved in the join on the right, and $\bigwedge_{S \subseteq \emptyset}\langle \uparrow \mathsf{set}\, S, 0, \sum \mathsf{num}\, S\rangle = \top$.

For $n \neq 0$, our strategy is to break up the left hand side into a huge join, and then show that every component of this join is below the right hand side. Without restriction, we may assume that the u_i are pairwise different; for the left, this is since $\langle u, 0, q\rangle \wedge \langle u, 0, q'\rangle = \langle u, 0, q \sqcup q'\rangle$, and on the right, since $A_p u > q$ and $A_p u > q'$ iff $A_p u > q \sqcup q'$. Thus, we may assume in the sequel that the index set I coincides with $\{u_1, \ldots, u_n\}$.

By rational continuity, $\langle u_i, 0, q_i\rangle$ is join of $\langle u_i, 0, q_i'\rangle$ with $q_i' > q_i$. Applying (3) to $\langle u_i, 0, q_i'\rangle$, we see that this generator is a join of components $\bigwedge_{T: i\in T\subseteq I}\langle u^T, v^T, q_i^T\rangle$, where $u^T = \bigwedge_{j\in T} u_j$ and $v^T = \bigvee_{j\in I\setminus T} u_j$, and the numbers $(q_i^T)_{T: i\in T\subseteq I}$ are some numbers with $\sum_{T: i\in T\subseteq I} q_i^T = q_i' > q_i$.

By the step just performed, the left hand side is a meet of joins of meets. Applying distributivity, it can be rewritten into a join of more meets. For a fixed $T \subseteq I$, the meet of all $\langle u^T, v^T, q_i^T\rangle$ with i in T can be contracted into $\langle u^T, v^T, q^T\rangle$, where $q^T = max_{i\in T}\, q_i^T$. Thus, we reach at a join of components $\bigwedge_{T: \emptyset\neq T\subseteq I}\langle u^T, v^T, q^T\rangle$ with u^T and v^T as above, and some numbers q^T such that for all i in I, $\sum_{T: i\in T\subseteq I} q^T > q_i$. If we index something over T in the sequel, this should be always understood as indexed over $\{T \mid \emptyset \neq T \subseteq I\}$.

There are two kinds of components $\bigwedge_T\langle u^T, v^T, q^T\rangle$: those with $q := \sum_T q^T < 1$ and those with $q \geq 1$. We treat the latter first. The component $\bigwedge_T\langle u^T, v^T, q^T\rangle$ is just one join component of $(\sum_T[u^T, v^T])(q)$. By (9), the opens u^T and v^T satisfy the disjointness condition of (8), whence $(\sum_T[u^T, v^T])(q)$ is below $[\bigvee_T u^T, 0](q)$. If $q \geq 1$, the latter is F by the 'below 1' condition of $\mathcal{R}F_3$. Thus, components $\bigwedge_T\langle u^T, v^T, q^T\rangle$ with $\sum_T q^T \geq 1$ equal F and deserve no further attention.

Using (10), the remaining components $\bigwedge_T\langle u^T, v^T, q^T\rangle$ can be written as a join of $\bigwedge_T\langle \uparrow F^T, v^T, q^T\rangle$, where F^T is some finite subset of $u^T\setminus v^T$. Let $F^T = \{a_1^T, \ldots, a_{m^T}^T\}$. Using (5), we can write $\bigwedge_T\langle \uparrow F^T, v^T, q^T\rangle$ as a join of components

$$\bigwedge_T \bigwedge_{j=1}^{m^T} \langle \sharp a_j^T, \bigvee_{k=1}^{j-1} \sharp a_k^T \vee v^T, r_j^T\rangle \quad \text{for some numbers } r_j^T \text{ with } \textstyle\sum_{j=1}^{m^T} r_j^T = q^T.$$

For every such component z, we shall now construct a power token A with two properties: it is in the set on the right of the critical lemma, and the join component z is below $\bigwedge_{S\subseteq A}\langle \uparrow\mathsf{set}\, S, 0, \sum \mathsf{num}\, S\rangle$. We define $A = \sum_T \{\!|(r_1^T, a_1^T), \ldots, (r_{m^T}^T, a_{m^T}^T)|\!\}$. This is a legal power token since $\sum_T \sum_j r_j^T = \sum_T q^T < 1$.

First, we verify $A_p u_i > q_i$ for every i in I. We may compute $A_p u_i = \sum\{\!|r_j^T \mid a_j^T \in u_i|\!\}$. We know $a_j^T \in F^T \subseteq u^T = \bigwedge_{l\in T} u_l \leq u_i$ if i is in T. Thus, i in T implies a_j^T in u_i, whence $A_p u_i \geq \sum\{\!|r_j^T \mid i \in T|\!\} = \sum_{T\ni i} q^T > q_i$.

Second, we have to show

$$\bigwedge_T \bigwedge_{j=1}^{m^T} \langle \sharp a_j^T, \bigvee_{k=1}^{j-1} \sharp a_k^T \vee v^T, r_j^T\rangle \leq \bigwedge_{S\subseteq A} \langle \uparrow\mathsf{set}\, S, 0, \textstyle\sum \mathsf{num}\, S\rangle$$

Let S be a fixed subbag of A. For every T, let S^T be the part of S consisting of pairs (r_j^T, a_j^T) only. Then we obtain with $(M2)$:

$$\bigwedge_T \bigwedge_{j=1}^{m^T} \langle \sharp a_j^T, \bigvee_{k=1}^{j-1} \sharp a_k^T \vee v^T, r_j^T \rangle \leq \bigwedge_T \bigwedge_{j \in S^T} \langle \sharp a_j^T, \bigvee_{k \in S^T, k<j} \sharp a_k^T \vee v^T, r_j^T \rangle$$

By (5), the latter is below $\bigwedge_T \langle \uparrow \mathsf{set}\, S^T, v^T, \sum \mathsf{num}\, S^T \rangle$. It remains to show that this is below $\langle \uparrow \mathsf{set}\, S, 0, \sum \mathsf{num}\, S \rangle$. This is true by (8), since the opens $\uparrow \mathsf{set}\, S^T \subseteq u^T$ and v^T satisfy the disjointness condition by (9), and $\bigvee_T \uparrow \mathsf{set}\, S^T = \uparrow \mathsf{set}\, S$ and $\sum_T \sum \mathsf{num}\, S^T = \sum \mathsf{num}\, S$ hold.

This concludes the proof of the critical lemma in F_3.

6.5 Real Valued Functions II

As announced in Subsection 6.1, we want to construct a frame homomorphism $\beta : F_3 \to G$ for complete Boolean algebras with $\eta : F_2 \to G$ such that $\beta \circ \alpha = \eta$. Before we do so, we investigate which additional properties the 'real valued functions' $\mathcal{R}G$ have if G is not just a frame, but a complete Boolean algebra. We shall show that the cBa structure allows defining a partial operation of subtraction on $\mathcal{R}G$.

Given two positive reals x and y with $x \sqsupseteq y$ and a positive rational q (all possibly being 0), when is $x - y > q$? If $q = 0$, the answer is always, because our order even satisfies $0 < 0$. For $q \neq 0$, $x - y > q$ iff there is $r > q$ with $x > r \sqsupseteq q + y$. With $s = r - q$, *this is equivalent to: there is* $s \neq 0$ *with* $x > q + s$ *and* $y \sqsubseteq s$, i.e., not $y > s$. In this existential statement, the case $s = 0$ may be included without harm since 'not $y > 0$' is always false.

This derivation motivates the following definition of subtraction in $\mathcal{R}G$ for a complete Boolean algebra G:

Definition 6.2 For f, g in $\mathcal{R}G$ with $f \geq g$, let

$$(f - g)(q) = \begin{cases} \top & \text{if } q = 0 \\ \bigvee_{s \in \mathbb{Q}_0^+} f(q+s) \wedge \neg g s & \text{if } q \neq 0 \end{cases}$$

The first thing to verify is that $f - g$ is a legal member of $\mathcal{R}G$. The condition $(f - g)(0) = \top$ is part of the definition, and $\bigvee_{q'>q}(f - g)(q') = (f - g)(q)$ holds for $q \neq 0$, since the argument q occurs positively in the definition of $f - g$.

In the sequel, we state and prove some properties of subtraction.

S0: '$-$' preserves non-empty joins in its first argument (but not in the second); thus, in particular it is continuous there.

Proof: Non-empty joins are given pointwise, and f occurs positively in the definition. □

S1: For all f, $f - 0 = f$.

Proof: For $q \neq 0$, $(f-0)(q) = \bigvee_s f(q+s) \wedge \neg 0(s) = \bigvee_{s\neq 0} f(q+s) = fq$ by rational continuity. □

S2: For all f, $f - f = 0$.

Proof: For $q \neq 0$, $(f - f)(q) = \bigvee_s f(q+s) \wedge \neg fs$. By $q + s \sqsupseteq s$, this is below $\bigvee_s fs \wedge \neg fs = \mathsf{F}$. □

The next statement prepares the following mixed associativity law.

S3: If $g \geq h$ and h is bounded, then $(f+g) - h \geq f$.

Proof: For $q \neq 0$, we obtain

$$((f+g)-h)(q) = \bigvee_{s,q_1,q_2:\, q_1+q_2=q+s} fq_1 \wedge gq_2 \wedge \neg hs$$

We have to show that this is above $fq = \bigvee_{r>q} fr$. For fixed $r > q$, consider fr. Fixing $q_1 = r$, we obtain

$$((f+g)-h)(q) \geq fr \wedge \bigvee_{s,q_2:\, r+q_2=q+s} gq_2 \wedge \neg hs$$

We show that the big join in this expression equals T. With $d = r - q$, it becomes $\bigvee_{q_2} gq_2 \wedge \neg h(q_2 + d)$. Using $g \geq h$ and restricting to multiples of d, the latter join is above $\bigvee_{n\in\mathbf{N}_0} h(n \cdot d) \wedge \neg h((n+1)\cdot d)$. Since h is bounded and $d \sqsupset 0$, there is some m with $h((m+1)\cdot d) = \mathsf{F}$. Thus, the join equals $(\mathsf{T} \wedge \neg hd) \vee (hd \wedge \neg h(2d)) \vee \cdots \vee (h(m\cdot d) \wedge \neg \mathsf{F})$, which can be contracted to T. □

The next statement is the important mixed associativity law.

S4: If $g \geq h$ and h is bounded, then $f + (g - h) = (f+g) - h$.

Proof: For $q \neq 0$, we have $(f + (g-h))(q) = \bigvee_{q_1+q_2=q} fq_1 \wedge (g-h)(q_2)$. Here, we must single out the case $q_2 = 0$, i.e., $q_1 = q$. It yields a single join component fq. Thus,

$$Lq = fq \vee \bigvee_{q_1,q_2,s:\, q_2\neq 0,\, q_1+q_2=q} fq_1 \wedge g(q_2+s) \wedge \neg hs$$

The right hand side yields

$$Rq = \bigvee_{s,q_1',q_2':\, q_1'+q_2'=q+s} fq_1' \wedge gq_2' \wedge \neg hs$$

For $Lq \leq Rq$, $fq \leq Rq$ holds by (S3). The remaining part of Lq is below Rq, as can be seen by letting $q_1' = q_1$ and $q_2' = q_2 + s$. For $Rq \leq Lq$, we have to differentiate two cases. The R-components with $q_2' \sqsupset s$ are below Lq, as can be seen by letting $q_1 = q_1'$ and $q_2 = q_2' - s$. If $q_2' \sqsubseteq s$, then $q_1' \sqsupseteq q$, whence $fq_1' \wedge gq_2' \wedge \neg hs \leq fq_1' \leq fq \leq Lq$. □

The next two statements show that subtraction inverts addition.

S5: If g is bounded, then $(f+g) - g = f$.

Proof: By (S4), $(f+g) - g = f + (g-g)$, which is f by (S2). □

S6: If $f \geq g$ and g is bounded, then $(f-g)+g = f$, and thus $f \geq f - g$.

Proof: By (S4) and commutativity of addition, $(f-g)+g = (f+g)-g$, which equals f by (S5). □

S7: If $f \geq g \geq h$ and g is bounded, then $(f-g)+(g-h) = f-h$.

Proof: Since g is bounded, h is bounded, too, and by (S6), $g-h$ is also bounded. By (S4), $(f-g)+(g-h) = ((f-g)+g)-h$, which by (S6) equals $f-h$. □

S8: Let $f \geq f'$, $g \geq g'$, and f', g' be bounded. Then $f-f' = g-g'$ iff $f+g' = f'+g$.

Proof: Add / subtract f' and g' and apply (S4), (S5), and (S6). □

6.6 A Frame Homomorphism from F_3 to G

Now we come to the final step of the proof of the critical lemma: given a frame embedding η from F_2 to a complete Boolean algebra G, define a frame homomorphism $\beta : F_3 \to G$ with $\beta \circ \alpha = \eta$. According to the spatial intuition that $\langle u, v, q\rangle$ be the set of evaluations μ with $\mu(u \setminus v) > q$, or $\mu u - \mu(u \wedge v) > q$, we define $\beta\langle u, v, q\rangle = (\eta \circ [u] - \eta \circ [u \wedge v])q$ using the difference operator of the previous subsection.

In the sequel, we shall not explicitly write down η, and abstract out the q-argument. Thus, we obtain the concise definition $\beta[u,v] = [v]-[u \wedge v]$. We have to show $\beta \circ \alpha = \eta$ and the preservation of the relations of F_3 by β. In doing so, we may use all properties of subtraction, since all the functions $[u]$ are bounded: $[u](1) = \mathsf{F}$.

For the composition, we compute $\beta(\alpha[u]) = \beta[u,0] = [u] - [u \wedge 0]$. By the zero law of F_2, $[u \wedge 0] = [0] = 0$, and by (S1), $[u] - 0 = [u]$ holds. Since this actually stands for $\eta \circ [u]$, we have shown $\beta \circ \alpha = \eta$.

For restricted continuity, let $(u_i)_{i \in I}$ be a directed family of opens, and v an open with $u_i \geq v$ for all i. We have to show $\beta[\bigvee_{i \in I} u_i, v] = \bigvee_{i \in I} \beta[u_i, v]$, or

$$[\bigvee_{i \in I} u_i] - [\bigvee_{i \in I} u_i \wedge v] = \bigvee_{i \in I}([u_i] - [u_i \wedge v]).$$

By the condition $u_i \geq v$, this simplifies to $[\bigvee_{i \in I} u_i] - [v] = \bigvee_{i \in I}([u_i] - [v])$, which is a valid statement by the continuity relation in F_2 and the continuity of subtraction in its first argument.

The zero relation $[0,0] = 0$ becomes $[0] - [0 \wedge 0] = 0$, which is true since $[0] = 0$ by the zero relation of F_2, and $0 - 0 = 0$ by (S1) or (S2).

The meet relation $[u, v] = [u, u \wedge v]$ becomes $[u] - [u \wedge v] = [u] - [u \wedge (u \wedge v)]$, which is obviously true.

The join relation $[u, v] = [u \vee v, v]$ translates into $[v] - [u \wedge v] = [u \vee v] - [v]$. By (S8), this is equivalent to $[u] + [v] = [u \vee v] + [u \wedge v]$, which is just the modularity relation of F_2.

Finally, the split relation states $[u, w] = [u, v] + [v, w]$ if $u \geq v \geq w$. By β, this becomes $[u] - [w] = ([u] - [v]) + ([v] - [w])$ — a valid statement by (S7).

7 Conclusion and Future Work

In [6, 7], a probabilistic power construction $\mathcal{P}_D$ is defined for dcpo's. In this paper, we define a construction $\mathcal{P}_I$ for infosyses in the sense of Vickers [12], and a construction $\mathcal{P}_L$ for locales, and prove that $\mathcal{P}_D$, $\mathcal{P}_I$, and $\mathcal{P}_L$ are equivalent when restricted to continuous domains. In particular, the infosys construction is effective: given a countable infosys D with decidable order, the power infosys $\mathcal{P}_I D$ is again countable with decidable order.

The dcpo construction $\mathcal{P}_D$ is part of a monad on dcpo's, as shown in [6]. One might also wish to make $\mathcal{P}_I$ and $\mathcal{P}_L$ into monads with operations equivalent to those of Jones. This is a non-trivial task, since Jones's definition of the multiplication of the monad $\mathcal{P}_D$ involves Lebesgue integration (of Scott continuous functions w.r.t. continuous evaluations). For $\mathcal{P}_L$, we were already successful in defining the monad operations, but this a topic for a different paper.

Another interesting problem is to work out the theory of '$\mathbf{R}_0^\infty$-frames' that may hide behind the auxiliary notations of Subsection 6.2. With a proper axiomatization of $\mathbf{R}_0^\infty$-frames or of $\mathbf{I}$-frames, one would obtain a theory of locales that does not rely on the Sierpinski space $\mathbf{2}$ as usual, but on the positive reals $\mathbf{R}_0^\infty$ or the unit interval $\mathbf{I}$.

Acknowledgements

I thank all my colleagues in the Theory and Formal Methods Section of Imperial College for their stimulating environment. Samson Abramsky and Steven Vickers proposed to further investigate the probabilistic power domain construction. Steven Vickers let me know about the problem to find an infosys and a localic description for it. Later on, he helped me in many discussions with valuable hints and encouraged me to proceed when I was stuck. Finally, he carefully read drafts of this paper.

References

[1] A. Edalat. Dynamical systems, measures and fractals via domain theory. In G.L. Burn, S.J. Gay, and M.D. Ryan, editors, *Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*, Workshops in Computing. Springer-Verlag, 1993.

[2] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[3] R. Hoofman. Continuous information systems. Technical Report RUU-CS-90-25, Rijksuniversiteit Utrecht, July 1990. To appear in Information and Computation.

[4] R. Hoofman. *Non-Stable Models of Linear Logic*. PhD thesis, Rijksuniversiteit Utrecht, 1992.

[5] P. Johnstone. *Stone Spaces*. Cambridge University Press, 1982.

[6] C.J. Jones. *Probabilistic Non-Determinism*. PhD thesis, University of Edinburgh, 1990.

[7] C.J. Jones and G.D. Plotkin. A probabilistic powerdomain of evaluations. In *LICS '89*, pages 186–195. IEEE Computer Society Press, 1989.

[8] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[9] D. Scott. Domains for denotational semantics. In *Proc. 9th Coll. on Automata, Languages, and Programming*, pages 577–613. *Lecture Notes in Comp. Sc. 140*, Springer-Verlag, 1982.

[10] M.B. Smyth. Power domains and predicate transformers: A topological view. In J. Diaz, editor, *ICALP '83*, pages 662–676. *Lecture Notes in Comp. Sc. 154*, Springer-Verlag, 1983.

[11] S. Vickers. *Topology via Logic*, volume 5 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[12] S. Vickers. Information systems for continuous posets. *Theoretical Computer Science*, 116(2):201–229, 1993.

Linear Domains and Linear Maps

Michael Huth
Department of Computing
Imperial College
London SW7 2BZ, England
mrah@doc.ic.ac.uk

Abstract. We study the symmetric monoidal closed category LIN of *linear domains*. Its objects are inverse limits of finite, bounded complete posets with respect to projection-embedding pairs preserving all suprema. The full reflective subcategory LL of *linear lattices* is a denotational model of linear logic; the negation is $A \mapsto A^{op}$ and $!(A)$ is the lattice of all Scott-closed sets of A. The Scott-continuous function space $[A \rightarrow B]$ models intuitionistic implication. Prime-algebraic lattices are linear and $\wp$ equals $\otimes$ for these lattices; in general, $\wp \neq \otimes$ in LL. Distributive, linear domains are exactly the prime-algebraic ones.

1 Introduction

One of the most frequently used cartesian closed categories in Denotational Semantics is that of Scott-domains and Scott-continuous maps [22]. If A and B are Scott-domains, then the exponential object $[A \rightarrow B]$ is the Scott-domain of all Scott-continuous functions $f\colon A \rightarrow B$, ordered pointwise. The product $A\&B$ is the order-theoretic product of A and B and *curry* and *apply* are defined as in the cartesian closed category SET of sets and (total) functions.

In [8], we find a finer universe of types which can be used to build up a cartesian closed category: the symmetric monoidal closed category of *coherence spaces* with stable maps in the stable order is a denotational semantics of linear logic equipped with *linear types*. Such types are finer than exponentiation and product in the sense that they decompose the exponential object into $!(A)\multimap B$. Intuitively, $f \in A\multimap B$ is the denotation of a proof that 'A implies B' such that the proof uses the hypothesis A only once. The construct $!(A)$ allows us to use A finitely many times, so $g \in !(A)\multimap B$ is the denotation of a proof of the intuitionistic implication 'A implies B'.

The goal of this paper is to specify a symmetric monoidal closed category of Scott-domains LIN with an internal hom $\multimap$ and a modality $!()$ such that $!(A)\multimap B$ is isomorphic to the function space $[A \rightarrow B]$ of all Scott-continuous maps, ordered pointwise. We will also obtain the other linear types: a *dualizing object* $\perp$, a contravariant functor $()^{\perp}$ on LIN modeling *negation*, a tensor product $\otimes$ modeling *parallel conjunction* and its De Morgan dual $\wp$; further, we will have the De Morgan dual $?()$ of $!()$ and the additive operations $\oplus$ and $\&$.

Intuitively, the objects of such a category should encompass all finite Scott-domains and this category should be closed under inverse limits of projection-embedding pairs.

As a methodological strategy, we will take on this task in the wider universe of ***bounded complete*** domains [14]. For these objects, we will specify the required morphisms and type constructors and we are 'only' left with the problem of finding a universe of Scott-domains in which algebraicity is preserved by all linear types; note that a bounded complete domain is a Scott-domain iff it is algebraic [15].

Assuming that we have already constructed this category of bounded complete domains, there are at least three conceptual questions to ask:

- Is algebraicity preserved under all the linear type constructors,
- and if not, are there at least subcategories of Scott-domains which are closed under all linear types and
- do we have maximal such categories?

In [14], a similar project has been successfully completed in the ***distributive*** setting. There, we studied the category BC of bounded complete domains with maps preserving all suprema. The full subcategory of *prime-algebraic* domains [27, 29] PRIME $\subseteq$ BC is such that

- every object in PRIME is distributive and algebraic,
- PRIME is closed under the linear types in [14] and
- if C is a full subcategory of BC closed under the negation $()^{\perp}$ such that every object in C is distributive and algebraic, then C is a full subcategory of PRIME [14, Theorem 3.5].

The full subcategory PAL $\subseteq$ PRIME of *prime-algebraic lattices* is a degenerate model of linear logic as we have $\otimes \cong \wp$ in PAL [14, Proposition 5.9]. Thus, we could add two further specifications for the category LIN.

- Every prime-algebraic domain should be linear and $\otimes$ and $\wp$ should be different type operations in LIN.

The fundamental questions in the 'design' of such a category are

- what is the Scott-domain $\perp$ and
- how can we characterize the Scott-domain $A \multimap B$ in terms of set-theoretic functions $f: A \to B$?

These questions will be dealt with first. By definition, $\perp$ will have to be a Scott-domain. If $\perp = \{*\}$ is a singleton domain, then $A^{\perp} \cong A \multimap \perp$ could only be $\perp$ again if the category LL is concrete, i.e. if morphisms in LL(A, B) correspond to set-theoretic functions $f: A \to B$. But then $A^{\perp\perp} \cong \perp^{\perp} \cong \perp$ demonstrates the impossibility of having $()^{\perp}$ as an involution on Scott-domains other than $\perp$. If $\perp$ has at least two elements, it is easily seen that the cardinality of $[[A \to \perp] \to \perp]$ exceeds that of A for all non-trivial, finite Scott-domains A. So while $\perp := \{0 < 1\}$ seems to be the only reasonable candidate for a dualizing object in LIN, the cardinality problem persists since $\perp$ has more than one element.

The mismatch in size stems from allowing *all* Scott-continuous functions $f \in [A \to \bot]$. Note that $[A \to \bot]$ is nothing but an isomorphic copy of the lattice of ***Scott-open sets*** $\sigma(A)$ [7, 15], and there are simply more Scott-open sets of A than there are points $a \in A$. What we need is that every set $f^{-1}(0)$, $f \in A \multimap \bot$, is Scott-closed and corresponds to a point in A. To show $f^{-1}(0) = \downarrow(a)$ for some $a \in A$, we need that $f^{-1}(0)$ is bounded in A and that then $f^{-1}(0)$ has a maximal element. For a Scott-domain with top, this is guaranteed if $f\colon A \to \bot$ preserves all suprema.

Alternatively, we are lead to the same choice of morphisms if we consider the desired isomorphism $[A \to B] \cong !(A) \multimap B$ for Scott-domains A and B assuming that B has a top. If $\wp_l(A)$ denotes the *lower power domain* of A [12] (which has a topological representation as the lattice of all ***non-empty*** Scott-closed subsets of A, ordered by inclusion), then $\eta_A := \lambda a.\downarrow_A(a)\colon A \to \wp_l(A)$ is Scott-continuous and is universal in the following sense: for all $f \in [A \to B]$, there exists a unique $\bar{f}\colon \wp_l(A) \to B$ preserving all *non-empty* suprema such that $\bar{f} \circ \eta_A = f$. If $!(A)$ denotes the lattice of *all* Scott-closed subsets of A, ordered by inclusion, then there exists a unique $\hat{f}\colon !(A) \to B$ preserving all suprema with $\hat{f} \circ up_{\wp_l(A)} \circ \eta_A = f$, for $\hat{f}$ has to map $\emptyset$ to 0_B and behaves like $\bar{f}$ otherwise; $up_{\wp_l(A)}\colon \wp_l(A) \to !(A)$ is the natural inclusion.

This discussion not only strengthens the justification for the choice of $A \multimap B$ as the space of functions preserving all suprema, ordered pointwise, it also suggests the mathematical nature of $!(A)$ as the *lifted lower power domain* of A.

We briefly review the linear types for bounded complete domains as presented in [14].

Definition 1 *A set-theoretic function $f\colon A \to B$ between bounded complete domains A and B preserves all suprema iff for all $X \subseteq A$ bounded in A, the set $f(X) \subseteq B$ is bounded in B and $f(\sqcup_A X) = \sqcup_B f(X)$. Let $A \multimap B$ denote the poset of all maps $f\colon A \to B$ preserving all suprema, ordered in the* pointwise order*:*

$$f \sqsubseteq g \text{ iff } f(a) \sqsubseteq g(a) \text{ for all } a \in A. \tag{1}$$

Define

$$A^\perp := A \multimap \bot. \tag{2}$$

Let BC be the category with all bounded complete domains as objects and maps preserving all suprema as morphisms. Let SCOTT denote the full subcategory of BC which has all Scott-domains as objects. Let SUP be the full subcategory of BC which has as objects all complete (sup)lattices. □

Let us point out that $\multimap$ and $[\,\to\,]$ are well-defined operations on ob(BC).

Lemma 1 *Let A and B be objects in BC. Then $A \multimap B$ and $[A \to B]$ are objects in BC, and the supremum operation in $A \multimap B$ and $[A \to B]$ is the pointwise one. In particular, the inclusion map $A \multimap B \hookrightarrow [A \to B]$ preserves all suprema.* □

The tensor product $\otimes$ is uniquely determined up to isomorphism if

$$A \otimes B \multimap C \cong A \multimap (B \multimap C) \tag{3}$$

is a natural isomorphism in BC; this is a consequence of basic category theory [16]. We want to motivate the tensor product by a universal property which appeals to our thinking in terms of functional programming. For objects A, B and C in BC, we can consider $A \multimap (B \multimap C)$ as a *subset* of $(C^B)^A$ as BC is a concrete [2] category. The category SET of sets and set-theoretic functions is cartesian closed [16] and the functions

$$\begin{aligned} \textit{curry}: C^{A \times B} \rightarrow (C^B)^A, \ \textit{curry} := \lambda f.\lambda a.\lambda b.f\langle a, b\rangle \\ \textit{uncurry}: (C^B)^A \rightarrow C^{A \times B}, \ \textit{uncurry} := \lambda g.\lambda\langle a, b\rangle.g(a)(b) \end{aligned} \tag{4}$$

are mutually inverse bijections. This provides us with the concept of *bilinearity* if we characterize the set $\textit{uncurry}(A \multimap (B \multimap C))$ in $C^{A \times B}$.

Definition 2 *For objects A, B and C in BC, a set-theoretic function f of type $f: A \times B \rightarrow C$ is* bilinear *iff*

$$\begin{aligned} \forall a \in A : \ \lambda b.f\langle a, b\rangle : B \rightarrow C \textit{ is linear} \\ \forall b \in B : \ \lambda a.f\langle a, b\rangle : A \rightarrow C \textit{ is linear.} \end{aligned} \tag{5}$$

We denote by $Bil(A \times B, C)$ the domain of all bilinear functions $f: A \times B \rightarrow C$ in the pointwise *order.* □

Note that $Bil(A \times B, C)$ is indeed an object in BC and that a bilinear map $f: A \times B \rightarrow C$ need not be a morphism in BC, nor is a map $g \in A \times B \multimap C$ bilinear in general [14]. If we restrict the maps *curry* and *uncurry* to $Bil(A \times B, C)$ and $A \multimap (B \multimap C)$, we get a natural order-isomorphism between $Bil(A \times B, C)$ and $A \multimap (B \multimap C)$ [14, Lemma 2.6]. Therefore, we obtain the natural isomorphism $A \otimes B \multimap C \cong A \multimap (B \multimap C)$ by showing

$$Bil(A \times B, C) \cong A \otimes B \multimap C. \tag{6}$$

For that, it is sufficient to have a domain $A \otimes B$ in BC and a bilinear map $\otimes: A \times B \rightarrow A \otimes B$ which is universal among all bilinear maps of type $f: A \times B \rightarrow C$: for all such f, there exists a unique map $\bar{f}: A \otimes B \rightarrow C$ preserving all suprema such that $\bar{f} \circ \otimes = f$. The isomorphism is then verified by sending f to $\bar{f}$ [14, Theorem 2.9]. This situation is quite common in a category with *universal bimorphisms* [3].

To construct the domain $A \otimes B$ and the bilinear map $\otimes: A \times B \rightarrow A \otimes B$, let A^+ be the domain $A \setminus \{0_A\}$ for an object A in BC. Note that A^+ is not an object in BC in general.

For a poset P, let $L_b(P)$ be the domain of all *lower* sets $L \subseteq P$ such that L is ***bounded*** in P. Then,

$$A * B := L_b(A^{+} \times B^{+}) \tag{7}$$

can be shown to be an object in BC [14, Lemma 2.8]. But $A * B$ is not a tensor product since

$$\langle a, b\rangle \mapsto {\downarrow}_{A^{+}}(a) \times {\downarrow}_{B^{+}}(b) \colon A \times B \to A * B \tag{8}$$

is not a bilinear map. Therefore, we have to consider $A \otimes B$, the domain of all $T \subseteq A * B$ satisfying the following condition:

$$\forall \emptyset \neq X \times Y \subseteq T \text{ bounded in } A * B : \ \langle \sqcup X, \sqcup Y\rangle \in T. \tag{9}$$

The domain $A \otimes B$ is an object in BC [14, Lemma 2.8] and the map $\otimes \colon A \times B \to A \otimes B$ defined by

$$\lambda\langle a, b\rangle.\ \mathit{if}\ \ (a = 0_A\ \ \mathit{or}\ \ b = 0_B)\ \ \mathit{then}\ \ \emptyset\ \ \mathit{else}\ \ {\downarrow}(\langle a, b\rangle) \tag{10}$$

is a universal bilinear map with $A \times B$ as a source [14, Theorem 2.9].

The unary operation $()^{\perp}$ is not quite an involution on objects in BC. This is the very reason why we had to construct $\otimes$ *explicitly*; otherwise, $A \otimes B \cong (A \multimap B^{\perp})^{\perp}$ would follow from the general theory of $*$-autonomous categories [4] and could be viewed as a *definition* of $\otimes$.

The forgetful functor SUP $\to$ BC has a left adjoint where $\lambda a.\lambda f.f(a) \colon A \to A^{\perp\perp}$ is the front adjunction. In particular, we have $A^{\perp} \cong A^{\perp\perp\perp}$ for all objects in BC [14, Lemma 4.3]. In [14, Lemma 4.3], we also find:

Remark 1 *For every object A in SUP, we have $A^{\perp} \cong (A)^{op}$. For a domain B in BC, we have $B^{\perp\perp} \cong B$ iff B is an object in SUP.* □

We let $A + B$ denote the *coalesced sum* [12, 20] of A and B which is a categorical coproduct in BC [14, Lemma 4.5]. Since SUP is a full reflective subcategory of BC, the operation $A \oplus B := (A + B)^{\perp\perp}$ is a categorical coproduct in SUP.

The category BC is seen to be symmetric monoidal closed and SUP is a model of linear logic [14]. Of course, our enterprise would be trivial if SCOTT were indeed closed under the type constructors of linear logic in BC.

Proposition 1 *SCOTT is not closed under the negation $()^{\perp}$ in BC.* □

Since SCOTT cannot be a category we are looking for, we should formalize what our desired category should satisfy.

Definition 3 *A full subcategory C of SCOTT* has linear types *iff*

- *C is closed under $()^{\perp}, \otimes, !()$ and $\oplus$ and*
- *C is closed under inverse limits of projection-embedding in BC.* □

We can focus on just the operations $()^{\perp}, \otimes, !()$ and $\oplus$, for we can define the remaining linear types in terms of these—at least for those objects satisfying $A \cong A^{\perp\perp}$, i.e. complete lattices.

The last condition imposed on a category of Scott-domains with linear types ensures that we can build sufficiently many objects. Intuitively, such a category should be 'determined' by all its objects of finite size.

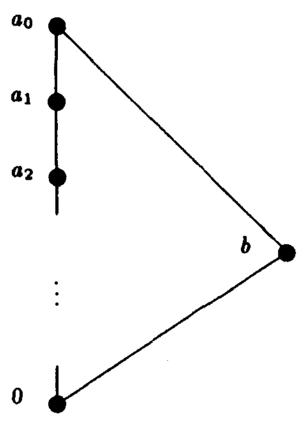


Fig. 1. A Scott-domain A_1 such that $A_1{}^{\perp} \cong (A_1)^{op}$ is not a Scott-domain

2 Linear Domains

Recall that a bounded complete domain A is algebraic iff it can be written as the inverse limit of finite, bounded complete posets under *Scott-continuous* projection-embedding pairs [15, 19]. This has also a well-known internal description [10, 15, 19] in terms of Scott-continuous *idempotent deflations* on A.

Proposition 2 *For a bounded complete domain A, the following are equivalent:*

1. *A is algebraic.*
2. *There exists a directed set $\mathcal{D}$ in $[A \to A]$ such that for all $d \in \mathcal{D}$, we have $dd = d$, $im(d)$ is finite and $\sqcup\mathcal{D} = id_A$.*

□

If we now view the equivalence in Proposition 2 as a *definition* of algebraicity in the category BC, we have seen in Figure 1 that this definition does not respect the operation $()^{\perp}$ on BC. A cheap escape route might be to consider ***bialgebraic*** lattices [7], i.e. lattices A such that A and $(A)^{op}$ are algebraic. The class of bialgebraic lattices is by definition closed under $()^{\perp}$ as then $A^{\perp} \cong (A)^{op}$; but one can raise a fatal objection against such an approach. If A and B are bialgebraic lattices, then $A \otimes B$ is not bialgebraic in general. To see why that is true, let A_2 be the domain shown in Figure 2. This is a bialgebraic lattice with $A_2{}^{\perp} \cong (A_2)^{op} \cong A_2$. If $A_2 \otimes A_2$ were bialgebraic, then $(A_2 \otimes A_2)^{op} \cong (A_2 \otimes A_2)^{\perp} \cong (A_2{}^{\perp} \otimes A_2)^{\perp}$ would have to be algebraic as well. The isomorphism [8, 14]

$$(A_2{}^{\perp} \otimes A_2)^{\perp} \cong A_2 \multimap A_2 \tag{11}$$

would then ensure that the function space $A_2 \multimap A_2$ is algebraic.

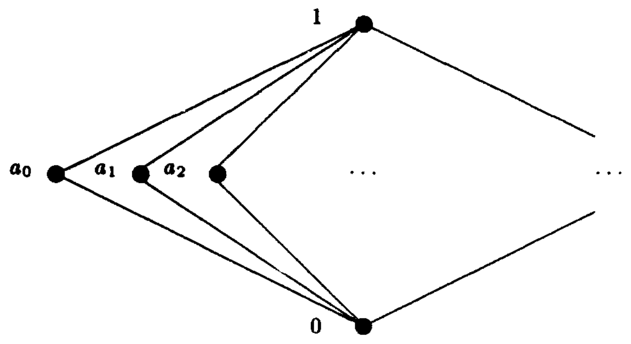


Fig. 2. A Scott-domain A_2 such that $A_2 \multimap A_2$ is not a Scott-domain

Proposition 3 *The function space $A_2 \multimap A_2$ is not algebraic.* □

In particular, the full subcategory of bialgebraic lattices in BC is not a category of Scott-domains with linear types. Looking again at the criterion of algebraicity in Proposition 2, we could ask what changes if we assume all maps $d \in \mathcal{D}$ to preserve *all* suprema not just directed ones?

Definition 4 *For an object A in BC set*

$$\mathcal{L}(A) := \{d \in A \multimap A \mid dd = d \sqsubseteq id_A,\ im(d)\ finite\}. \quad (12)$$

We call A linear *iff there exists a directed set $\mathcal{D}$ in $\mathcal{L}(A)$ such that $\sqcup\mathcal{D} = id_A$ holds in $A \multimap A$. Let LIN denote the full subcategory of BC which has all linear domains as objects. Let LL be the full subcategory of BC which has all linear lattices as objects.* □

We want to show that LIN and LL are categories of Scott-domains with linear types.

Theorem 1 *1. LIN contains all finite objects in BC and LL contains all finite objects in SUP,*
2. every object in LIN is algebraic,
3. not every object in LIN, respectively LL, is distributive,
4. LIN, respectively LL, is closed under inverse limits of projection-embedding pairs in BC, respectively SUP,
5. LIN, respectively LL, is closed under $\multimap$,
6. LIN, respectively LL, is closed under $[\rightarrow]$,
7. LIN, respectively LL, is closed under [illegible].

□

It can be shown that PRIME is a proper subcategory of LIN; because of Theorem 1.3, we only need to show that every prime-algebraic domain is linear. The intuition behind a prime-algebraic domain A is that every element $a \in A$ is the supremum of complete primes below it.

Definition 5 *Let A be a bounded complete domain. An element $p \in A$ is called a* complete prime *of A iff for all bounded sets $X \subseteq A$ the relation $p \sqsubseteq \sqcup_A X$ implies $p \sqsubseteq x$ for some $x \in X$. Let $Pr(A)$ denote the poset of all complete primes of a bounded complete domain A. The domain A is* prime-algebraic *iff the supremum of $\downarrow(a) \cap Pr(A)$ equals a for all $a \in A$. Let PRIME be the full subcategory of BC with all prime-algebraic domains as objects. Let PAL be the full subcategory of BC with all prime-algebraic lattices as objects.* □

The condition for $p \in \mathrm{Pr}(A)$ reads like the criterion for being a finite element, $p \in \mathrm{K}(A)$, except that we now quantify over *all* bounded sets X, not only directed ones.

Proposition 4 *Every prime-algebraic domain is linear; in particular, PRIME is a full subcategory of LIN and PAL is a full subcategory of LL.* □

Theorem 1 has a corresponding version for the categories PRIME and PAL.

Theorem 2 *1. PRIME contains all finite, distributive objects in BC and PAL contains all finite, distributive objects in SUP.*
2. every object in PRIME is algebraic.
3. every object in PRIME is distributive.
4. PRIME, respectively PAL, is closed under inverse limits of projection-embedding pairs in BC, respectively SUP.
5. PRIME, respectively PAL, is closed under $\multimap$.
6. PRIME, respectively PAL, is closed under $[\rightarrow]$.
7. PRIME, respectively PAL, is closed under $\otimes$.

□

The category LL is also closed under the *additive* type constructors of SUP, introduced in [14].

Definition 6 *Let A and B be objects in BC. Then define*

- *$A + B$ to be the* coalesced sum *[12, 18, 20] of A and B,*
- *$A \oplus B$ to be $(A+B)^{\perp\perp}$,*
- *$A\&B$ to be the order-theoretic product of A and B and*
- *$\wp$ to be the De Morgan Dual of $\otimes$:*

$$A \wp B := (A^{\perp} \otimes B^{\perp})^{\perp}. \tag{13}$$

□

Proposition 5 *1. If A and B are linear domains, then so are $A + B$, $A \oplus B$, $A\&B$ and $A\wp B$.*
2. If A and B are linear lattices, then so are $A \oplus B$, $A\&B$ and $A\wp B$.

$\square$

Note that $+$ is the categorical product in BC and LIN, whereas $\oplus$ is the categorical *biproduct* in SUP and LL [14, Lemma 4.5 & 4.10]; also, $A + B$ is a lattice iff A or B is a singleton domain. Since LIN and LL are closed under & and $[\to]$, we obtain:

Corollary 1 *The category LIN^{sc}, respectively LL^{sc}, of linear domains, respectively linear lattices, and* Scott-continuous *maps as morphisms is cartesian closed.*

$\square$

3 Linear Lattices modeling Linear Logic

We have shown that LL is closed under $()^{\perp}, \odot, \wp, \multimap, \oplus$ and $\&$. We define the four domains modeling the constants of classical linear logic to be

$$\begin{aligned} \bot &:= \{0 < 1\} \\ \mathbf{1} &:= \bot^{\perp} \cong \bot \\ \top &:= \{*\} \\ \mathbf{0} &:= \top^{\perp} \cong \top. \end{aligned} \tag{14}$$

Since SUP and BC are symmetric monoidal closed categories [14] and LL and LIN are closed under all the constructions discussed so far in SUP and BC, we conclude that LL and LIN are symmetric monoidal closed categories as well.

Theorem 3 *LIN and LL are symmetric monoidal closed categories.* $\square$

The category LL gives us a model of linear logic in the standard fashion of [23]. Moreover, we have a natural isomorphism $A \oplus B \cong A\&B$ in SUP and LL [14, Lemma 4.10]. Therefore, the category LL can give us only an incomplete semantics of classical linear logic: we have morphisms $f \in (A \oplus B) \multimap (A\&B)$ but $(A \oplus B) \multimap (A\&B)$ is not a theorem of classical linear logic [25].

In the introduction, we already gave good categorical reason for the choice of $!(A)$ as the lattice of all Scott-closed subsets of A, ordered by inclusion.

Definition 7 *For an object A in SCOTT and any poset P, define*

- *$L(P)$ to be the poset of* lower sets *of P ordered by inclusion.*
- *$\wp_l(A)$ to be $L(K(A) \setminus \{0_A\})$,*
- *$\epsilon_A : A \to \wp_l(A)$ by $\epsilon_A := \lambda a. \downarrow(a) \cap (K(A) \setminus \{0_A\})$,*
- *$!(A)$ as the lattice of all Scott-closed subsets of A, ordered by inclusion and*
- *$?(A)$ as $(!(A^{\perp}))^{\perp}$.* $\square$

We realize $?(A)$ as the Scott-topology on $A^\perp$.

Proposition 6 *Let A be an object in SCOTT. Then we have the following:*

1. *$\langle \wp_l(A), \epsilon_A\rangle$ is a lower power domain for A,*
2. *$\wp_l(A)_\perp \cong L(K(A)) \cong !(A)$,*
3. *$L(K(A))$ is an object in PAL with $Pr(L(K(A))) \cong K(A)$,*
4. *$!(A) \cong \sigma(A)^\perp$ and*
5. *$?(A) \cong \sigma(A^\perp)$.*

□

The isomorphism $L(K(A)) \cong !(A)$ had been stated in the non-lifted version in [26]. We still have to ensure that $!(A)$ and $?(A)$ are linear for a linear domain A.

Proposition 7 *The categories PRIME, PAL, LIN and LL are closed under $!()$ and $?()$.* □

The functor $!()$ transforms products into tensor products.

Theorem 4 *1. For objects A and B in a category C of Scott-domains with linear types, we have*

$$!(A\&B) \cong !(A) \otimes !(B) \tag{15}$$

2. and for lattices A and B in C, we have

$$[A - B] \cong !(A) \multimap B. \tag{16}$$

□

Let us summarize what we have demonstrated so far.

Theorem 5 *The categories of Scott-domains PRIME, LIN, PAL and LL have linear types and PRIME $\subseteq$ LIN.* □

Since all objects in PRIME, respectively LIN, are isomorphic to inverse limits of *finite* objects in PRIME, respectively LIN, the category PRIME, respectively LIN, is 'determined' by its class of objects of finite size. For PRIME, the constraint on a finite domain is the distributivity axiom, for LIN, we allow *all* finite domains in BC. The situation is similar for PAL and LL. One might ask whether this is typical for categories of Scott-domains with linear types.

In [14, Proposition 5.9], we showed the natural isomorphism

$$A \otimes B \cong A \wp B \tag{17}$$

for objects A and B in PAL. Therefore, PAL is a *compact closed* category. The situation changes if we consider the larger category LL. The example of two finite lattices where $\wp$ differs from $\otimes$ is due to Michael Barr [4, page 100].

Theorem 6 *1. For all categories C of Scott-domains with top with linear types such that every object in C is distributive, we have a natural isomorphism $A \otimes B \cong A \wp B$.*

2. There exist linear lattices C and D such that $C \odot D \ncong C \wp D$.

□

Theorem 6 states that the distributivity in a category C of Scott-domains with top with linear types implies that $\wp$ equals $\otimes$. One might wonder how strong the link between distributivity of all objects in C and the compact closedness of such a category is; and what about the existence of categories of Scott-domains with linear types other than PRIME, PAL, LIN and LL?

Question 1 *Given a compact closed category C of Scott-domains with top with linear types, is every object in C distributive?* □

Question 2 *Are there categories C of Scott-domains with linear types other than PRIME, PAL, LIN and LL: if so, is every such category C in BC contained in LIN?* □

4 Quasi-prime Algebraic Domains

In this section, we want to show that a linear domain is distributive iff it is prime-algebraic. In this sense, linear domains constitute a generalization of prime-algebraic domains by abandoning the distributivity axiom but at the same time preserving the richness of the available type structure. The proof of that requires the notion of a *completely sup-irreducible element* [7] and of *quasi-prime algebraic domains* introduced by Guo-Qiang Zhang in [30].

In [30], it was noted that each $p \in \mathrm{Pr}(A)$ has a unique element $p^* < p$ in A such that $a \sqsubseteq p^*$ for all $a < p$ in A. Guo-Qiang Zhang calls elements p which have such a p^* *quasi-primes* and defines a quasi-prime algebraic domain A to be a Scott-domain, in which each element is the supremum of quasi-primes below it. Clearly, $p \in A$ is quasi-prime iff for all $X \subseteq \downarrow(p)$ the relation $p \sqsubseteq \sqcup_A X$ implies $p \sqsubseteq x$ for some $x \in X$.

Comparing this to the definition of a complete prime, has lead Guo-Qiang Zhang to the name *quasi-prime*, for one quantifies only over all bounded sets X in $\downarrow(p)$, not in all of A. Such elements have also been studied in lattice theory [5, 7].

Definition 8 *Let A be a bounded complete domain. An element $q \in A$ is called a* completely sup-irreducible element *of A iff for all bounded $X \subseteq A$ the equation $q = \sqcup_A X$ implies $q \in X$. Let $Si(A)$ denote the poset of completely sup-irreducible elements of A. Let SI be the full subcategory of BC with objects all A such that the supremum of $\downarrow(a) \cap Si(A)$ equals a for all $a \in A$.* □

The next lemma compares the notions of quasi-primes, complete primes and completely sup-irreducible elements. It had been shown in [30] for Scott-domains.

Lemma 2 *Let A be an object in BC and $p \in A$. Then, the following are equivalent:*

1. *p is a completely sup-irreducible element in A,*
2. *p is a quasi-prime in A and*
3. *p is a complete prime in $\downarrow(p)$.*

□

Since $\mathrm{Pr}(A) \subseteq \mathrm{K}(A)$ holds for all objects A in BC, and since finite suprema of finite elements are finite [7], we know that every prime-algebraic domain A is also a Scott-domain. Moreover, $k \in A$ is then finite in A iff k is the supremum of a finite set $F \subseteq \mathrm{Pr}(A)$ in A. This does not hold if we replace $\mathrm{Pr}(A)$ by $\mathrm{Si}(A)$. In Figure 1, the lattice $A_1{}^\perp$ is readily seen to be an object in SI with

$$\mathrm{Si}(A_1{}^\perp) = \{a_n \mid n > 1\} \cup \{b\}, \tag{18}$$

but $A_1{}^\perp$ is not algebraic. In particular, b is an element of $\mathrm{Si}(A_1{}^\perp) \setminus \mathrm{K}(A_1{}^\perp)$. For objects in SI, the absence of such elements is equivalent to the algebraicity of the domain.

Proposition 8 *Let A be an object in SI. Then*

1. *A is algebraic iff $Si(A) \subseteq K(A)$ and*
2. if *A is algebraic, then $k \in A$ is finite in A iff k is the supremum of a finite set $F \subseteq Si(A)$ in A.*

□

It is time to define quasi-prime algebraic domains as a category. Also, we need a name for the category of all distributive domains in BC.

Definition 9 *Let QP be the full subcategory of SI which has all Scott-domains in SI as objects. Let dBC be the full subcategory of BC which has all distributive domains in BC as objects.* □

Note that the objects in QP are exactly Guo-Qiang Zhang's quasi-prime algebraic domains [30]. By definition, QP is contained in SI, yet, $A_1{}^\perp$ is an object in SI but not in QP, for it is not an algebraic domain. Since

$$\mathrm{Si}(A_1) = \{a_n \mid n > 0\} \cup \{b\}, \tag{19}$$

we see that A_1 is an object in QP.

Remark 2 *The category QP is not closed under $()^\perp$ in BC; QP is not a category of Scott-domains with linear types.* □

In [30], one obtains a symmetric monoidal closed category with finite products such that its class of objects equals the class of objects in QP (=all quasi-prime algebraic domains). The approach differs from the one taken in this paper, for Guo-Qiang Zhang considers *quasi-linear maps*, a special class of Scott-continuous functions, as elements of the internal hom [30].

This is interesting as most models of intuitionistic logic rest on the notion of a function preserving all suprema. The technical price being paid, however, is the absence of a dualizing object [30]. But another point in favor of working with non-linear functions is the possibility of representing all objects as *information systems* [30].

For linear domains, this seems only possible after one has established an order-theoretic axiomatization of the posets $K(A)$ for linear domains A; such an axiomatization has been given for SFP-objects [19] and bifinite domains [10, 15]. We have been unable to suggest such an axiomatization in the linear setting. If such a logical description of linear domains can be found, it is likely not to be a first-order theory on posets $K(A)$. It would be of interest to investigate, whether there is a greatest category of Scott-domains with linear types which has a first-order axiomatization—see [11] for the situation of Scott-continuous maps.

We pointed out that QP does not support the linear types in BC; but every category of Scott-domains with linear types is contained in QP.

Theorem 7 *Let C be a category of Scott-domains with linear types. Then C is a full subcategory of QP.* □

This theorem uses only the fact that all objects in C are algebraic and that C is closed under $()^{\perp}$ in BC. Theorem 7 is not a vacuous statement, for there exist Scott-domains which are not quasi-prime algebraic. Before we give an example, let us draw a conclusion for linear domains.

Corollary 2 *The category LIN is a full subcategory of QP.* □

Note that our example of a Scott-domain A_3, which is not quasi-prime algebraic, must be infinite; otherwise, A_3 would be linear and therefore an object in QP. Consider the lifted, full binary tree with its root as top element as depicted in Figure 3. This describes a Scott-domain A_3 with

$$\mathrm{Si}(A_3) = \emptyset, \tag{20}$$

so A_3 is not quasi-prime algebraic—this example is due to Guo-Qiang Zhang [30]. Since ${A_3}^{\perp} \cong (A_3)^{op}$ is not algebraic, we conclude that $A_3 \multimap A_3$ is not algebraic, for there exists a canonical closure-embedding pair $\langle c, e\rangle\colon A_3 \multimap A_3 \to (A_3)^{op}$ in BC and the image of a Scott-continuous closure operator of an algebraic domain is algebraic [7]. The same reasoning applies to the Scott-domain A_1.

Now, we are in a position to prove that distributivity and prime-algebraicity are the same concept in LIN and in QP.

Lemma 3 *For A in BC, we have*

1. *$Pr(A) \subseteq Si(A)$ and*
2. *if A is distributive and algebraic, then $Si(A) \subseteq Pr(A)$.*

□

Theorem 8 *The categories PRIME, $dBC \cap LIN$ and $dBC \cap QP$ are equal.* □

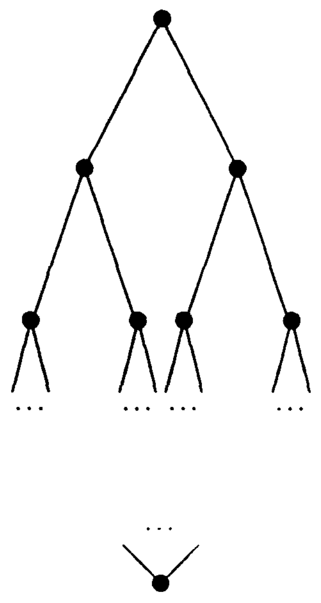


Fig. 3. A Scott-domain A_3 which is not quasi-prime algebraic

5 Conclusion

We have developed the concept of a category of Scott-domains with linear types as a subcategory of Scott-domains of BC which supports the type operations of linear logic in BC such that it is closed under inverse limits of projection-embedding pairs in BC. We gave four examples of such categories: prime-algebraic d[illegible]ains, prime-algebraic lattices, linear domains and linear lattices.

We showed [illegible] every prime-algebraic domain is linear (and distributive) and that every distributive linear domain is prime-algebraic. In this sense, linear domains can be viewed as a *generalization* of prime-algebraic domains.

Every linear lattice is bialgebraic and bialgebraic lattices are ill-behaved under the linear type constructors if they are not linear [Proposition 3]. Hence, we can construe linear lattices as a *benign* subcategory of the full subcategory of bialgebraic lattices in SUP.

6 Future Work

The questions stated in this article form the ground on which future work should take off. We did not discuss linear domains A with a countable basis $K(A)$. It can be shown that all the type constructors presented in this paper preserve countability of the basis $K(A)$.

We also did not include an analysis of the fine structure of linear domains. This will be done in a subsequent piece of work. Further, it would be interesting to investigate ***linear, stable domains*** A which are obtained by assuming that all maps in Definition 4 are stable as well and that the directed sets are directed with respect to the stable order in [9].

Acknowledgements

I would like to thank Samson Abramsky for his continuous advice and encouragement in this work. Thanks to Mike Mislove and the MFPS'93 organizers for such a wonderful conference. I am grateful to G. Q. Zhang for having communicated to me his work on quasi-prime algebraic domains. It inspired me to write down the definition of a linear domain. Achim Jung and Phillip Sünderhauf discussed with me some non-linear domains with non-algebraic linear function spaces. At the MFPS'93 meeting, Michael Barr was extremely helpful in pointing out to me his example of two finite lattices where $\wp$ differs from $\otimes$ [4, page 100]. The two referee reports helped shaping and clarifying this paper.

References

1. S. Abramsky *Domain theory in logical form*, in: *Annals of Pure and Applied Logic* **51**, pp. 1–77, 1991
2. J. Adámek, H. Herrlich and G. Strecker *Abstract And Concrete Categories*, J. Wiley & Sons, Inc., 1990
3. B. Banaschewski and E. Nelson *Tensor Products And Bimorphisms*, Canad. Math. Bull., vol. **19 (4)**, pp. 385–402, 1976
4. M. Barr **-Autonomous categories*, Springer Lecture Notes in Mathematics, SLNM **752**, 1979
5. G. Birkhoff *Lattice Theory*, American Mathematical Society Colloquium Publications, volume 25, third edition, third printing, reprinted with corrections, 1984
6. P.-L. Curien *Categorical Combinators, Sequential Algorithms and Functional Programming*, Research Notes in Theoretical Computer Science, Pitman, London, 1986
7. G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. W. Mislove and D. Scott *A Compendium of Continuous Lattices*, Springer Verlag, New York, 1980
8. J.-Y. Girard *Linear Logic*, Theoretical Computer Science **50**, pp.1–102, North-Holland, Amsterdam, 1987
9. J.-Y. Girard, Y. Lafont and P. Taylor *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science **7**, Cambridge University Press, Cambridge, 1989
10. C. A. Gunter *Profinite Solutions For Recursive Domain Equations*, Doctoral Dissertation, University of Wisconsin at Madison, 1985
11. C. A. Gunter *The largest first-order axiomatizable cartesian closed category of domains*, in: *Logic and Computer Science*, A. Meyer (editor), pp.142–148, IEEE Computer Society, June 1986
12. C. A. Gunter *Semantics of Programming Languages*, Foundations in Computing Series, The MIT Press, Cambridge, Massachusetts, 1992

13. M. Huth *Cartesian Closed Categories of Domains and the Space Proj(D)*, in: *Proceedings of the Seventh Workshop on the Mathematical Foundations of Programming Semantics'*, Carnegie-Mellon University, Pittsburgh, March 1991, LNCS **598**, pp. 259–271, 1992
14. M. Huth *A Maximal Monoidal Closed Category Of Distributive Algebraic Domains*, Technical Report 1992, Department of Computing and Information Sciences, Kansas State University, accepted for publication in the journal of *Information and Computation*
15. A. Jung *Cartesian Closed Categories of Domains*, CWI Tract **66**, Amsterdam, 110pp., 1989
16. S. Mac Lane *Categories for the Working Mathematician*, Springer Verlag, New York, 1971
17. M. W. Mislove *When Are Order Scattered And Topologically Scattered The Same?*, Annals of Discrete Mathematics **23**, pp.61–80, North-Holland, Amsterdam, 1984
18. H. R. Nielson and F. Nielson *Semantics With Applications*, Wiley Professional Computing, England, 1992
19. G. Plotkin *The category of complete partial orders: a tool of making meaning*, in: *Proceedings of the Summer School on Foundations of Artificial Intelligence and Computer Science*, Instituto di Scienze dell'Informazione, Universita di Pisa, 1978
20. D. A. Schmidt *Denotational Semantics*, Allyn and Bacon, Inc., 1986
21. D. Scott *Continuous Lattices*, Lecture Notes in Mathematics, vol. **274**, pp.97–136, Springer, New York, 1972
22. D. Scott *Domains for Denotational Semantics*, in: *ICALP82*, M.Nielsen and E.M.Schmidt (eds.), Springer Verlag, LNCS, vol.**140**, 1982
23. R. Seeley *$*$-autonomous categories, cofree coalgebras and linear logic*, In J.W. Grey and A. Scedrov, editors, *Categories in Computer Science and Logic*, volume **92** of *Contemporary Mathematics*, pp.371–382, American Mathematical Society, 1989
24. M. B. Smyth *The Largest Cartesian Closed Category of Domains*, Theoretical Computer Science **27**, pp.109–119, 1983
25. A. S. Troelstra *Lectures On Linear Logic*, CSLI Lecture Notes, No.**29**, 1992
26. G. Winskel *On Power Domains And Modality*, Theoretical Computer Science, **36**, pp.127–137, 1985
27. G. Winskel *An Introduction to Event Structures*, in: *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, LNCS, vol.**354**, pp.364–399, 1988
28. G. Q. Zhang *Logic of Domains*, Birkhäuser, Boston, 1991
29. G. Q. Zhang *DI-Domains as Prime Information Systems*, to appear in the journal of *Information And Computation*, **100 (2)**, pp.151–177, 1992
30. G. Q. Zhang *Quasi-prime Algebraic Domains*, manuscript, personal communication, 1993

Universal Quasi-Prime Algebraic Domains (Extended Abstract)

Guo-Qiang Zhang*
Department of EECS, AI Laboratory
University of Michigan, Ann Arbor, MI 48109
E-mail: gqz@cs.uga.edu

Summary. This paper demonstrates the existence of a saturated quasi-prime algebraic domain. It also presents a cpo of quasi-prime generated information systems for solving domain equations.

Key Words: Domain theory, semantics of programming languages, universal structures, lattices, universal algebra.

1 INTRODUCTION

Quasi-prime algebraic domains are a class of cpos within the Scott domains. They are introduced by the author in [11] as a new domain-theoretic model for linear logic. Quasi-prime algebraic domains with quasi-linear functions form a monoidal closed category. The unique

*On leave from Department of Computer Science, The University of Georgia, Athens, GA 30602 characteristic of the category is that the morphisms are not 'linear', as the term 'quasi-linear' suggests. This is a bit surprising, since all other known domain theoretic linear categories all use linear functions as morphisms [11].

However, how robust and useful the concept of quasi-prime algebraic domains is depends on whether or not they have other nice domain theoretic properties. One of the desirable properties is the existence of a universal (or even saturated) domain [4] in a certain category. The other related property to have is a framework for solving domain equations by fixed point construction, as in [5, 8]. It is the purpose of the paper to establish these results for quasi-prime generated information systems which represent quasi-prime algebraic domains.

One of the most useful results on universal domains is given in the work of Droste and Göbel [1], who introduced the Fraissé-Jónsson theorem in model theory into the area of domain theory. This makes it much easier to show the existence of certain universal domains because it reduces the existence of a saturated structure to the amalgamation property of the finite objects of a certain category.

We apply the result of Droste and Göbel for showing the existence of a saturated (universal, homogeneous) quasi-prime algebraic domain. Our main definition here is the notion of q-embeddings for quasi-prime algebraic domains. The appropriate notion of embeddings for Scott domains (call them s-embeddings) [5, 2] and for dI-domains (call them r-embeddings – 'r' for rigid) [8] are well-known . However, none of the these embeddings works for quasi-prime algebraic domains, for the following reasons:

- The s-embeddings are too general: under this embedding the colimit of an ω-chain of finite Scott domains (which are quasi-prime algebraic) need not be quasi-prime algebraic, because any Scott domain can be seen as a colimit of this kind.

- The r-embeddings are too specific: there are certain quasi-prime algebraic domains which cannot be represented as a colimit of any chain of finite Scott domains, although the r-embeddings are suitable for the I-domains (dI-domains without axiom d).

With q-embeddings we get the desired algebroidal category of quasi-prime generated information systems. The finite objects of the category are shown to have the amalgamation property. This implies the existence of a saturated quasi-prime algebraic domain.

Based on the notion of q-embeddings, we will also introduce a cpo of quasi-prime generated information systems on which various constructions are shown to induce continuous functions. This implies the existence of recursively defined quasi-prime generated information systems.

Here is the outline of the structure of the paper. In Section 2 we introduce the notions of quasi-primes and quasi-prime algebraic domains. In Section 3 we represent quasi-prime algebraic domains as information systems. This will bring technical convenience to the rest of the paper. Section 4 recalls the result of Droste and Göbel on the existence of universal domains. Section 5 presents a category of quasi-prime algebraic information system with q-embeddings as the morphisms. Section 6 verifies the amalgamation property of the finite objects of the category introduced in Section 5. In the final section, we introduce a cpo of quasi-prime generated information systems and the continuity of various constructions on this cpo.

2 Quasi-Prime Algebraic Domains

In a Scott domain D, an element p is called a *complete prime* if

$$p \sqsubseteq \bigsqcup X \Rightarrow \exists x \in X.\, p \sqsubseteq x.$$

On the other hand, an element q is called a *quasi-prime* if

$$q = \bigsqcup X \Rightarrow \exists x \in X.\, p = x.$$

A Scott domain is ***prime algebraic*** if every element is the least upper bound of complete primes below (less than or equal to) it. Similarly, a Scott domain is ***quasi-prime algebraic*** if every element is the least upper bound of quasi-primes below it. As far as functions are concerned, for prime algebraic domains, a function f is *linear* if and only if

$$p \sqsubseteq f(x) \Rightarrow \exists r \sqsubseteq x.\, p \sqsubseteq f(r), \quad \text{where } p, r \text{ are complete primes.}$$

For quasi-prime algebraic domains, a function f is *quasi-linear* if and only if

$$q \sqsubseteq f(x) \Rightarrow \exists s \sqsubseteq x.\, q \sqsubseteq f(s), \quad \text{where } q, s \text{ are quasi-primes.}$$

The following table summarizes the relationships between complete primes and quasi-primes, linear functions and quasi-linear functions. For comparison, we also include isolated elements and continuous functions.

	Definition
Isolated Elements	$d \sqsubseteq \bigsqcup X \Rightarrow \exists x \in X.\, d \sqsubseteq x$ (X directed)
Complete Primes	$p \sqsubseteq \bigsqcup X \Rightarrow \exists x \in X.\, p \sqsubseteq x$ (X bounded)
Quasi-Primes	$q = \bigsqcup X \Rightarrow \exists x \in X.\, p \sqsubseteq x$
Continuous Functions	$f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ (X directed)
Linear Functions	$f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ (X bounded)
Quasi-Linear Functions	$q \sqsubseteq f(x) \Rightarrow \exists s \sqsubseteq x.\, q \sqsubseteq f(s)$ (q, s quasi-primes)

It is helpful to note that all *finite* Scott domains are quasi-prime algebraic (we need the next theorem for this). Of course not all finite Scott domains are prime algebraic. Moreover, not all Scott domains are quasi-prime algebraic. The following theorem can help us find such an example. This theorem is extremely helpful in identifying quasi-primes. This, I believe, is also the key advantage of working with quasi-prime algebraic domains.

Theorem 2.1 *Let D be a Scott domain. An element $q \in D$ is a quasi-prime iff there is a unique element $q^{\prec}$ immediately below q:*

$$x \sqsubset q \Rightarrow x \sqsubseteq q^{\prec}.$$

It is worth pointing out that, as a consequence of the theorem, if x has a unique element immediately below it, then x is an isolated element. Bottoms are never quasi-primes.

It is now easy to see that, if one turns the complete binary tree upside down and adjoining a bottom element, one gets a Scott domain which is not quasi-prime algebraic, since there is no quasi-primes in this domain. Also note that it is easy to show, by mathematical induction on the number of elements below an isolated element, that finite Scott domains are quasi-prime algebraic.

We end this section by remarking that a dual concept of quasi-primes was mentioned in [3] (pages 92-93), called the *completely irreducible elements.* Quasi-primes were introduced in [11] as a by-product of studying quasi-prime algebraic domains. Therefore, our motivation, objectives, and results are, in any case, totally different from that of [3]. More important, we have gone far beyond a particular class of elements. We consider domains generated by these elements, and we consider categories of quasi-prime algebraic domains. We have introduced quasi-linear functions (detail presented in a forthcoming paper) which are the corresponding morphisms for quasi-primes.

3 Quasi-Prime Generated Information Systems

This section introduces a representation of quasi-prime algebraic domains as information systems. This will bring technical convenience for the presentation of the rest of the paper.

An information system [6] is a structure $\underline{A} = (A, Con, \vdash)$ where

- A is a countable set of propositions (tokens),
- Con is a collection of finite subsets of A (the consistent sets),
- $\vdash \subseteq Con - \{\emptyset\} \times A$, the entailment relation,

which satisfy

- $(X \subseteq Y \;\&\; Y \in Con) \Rightarrow X \in Con$,
- $a \in A \Rightarrow \{a\} \in Con$,
- $(X \vdash a \;\&\; X \in Con) \Rightarrow X \cup \{a\} \in Con$,
- $(a \in X \;\&\; X \in Con) \Rightarrow X \vdash a$,
- $(X \vdash Y \;\&\; Y \vdash c) \Rightarrow X \vdash c$.

Here $X \vdash Y$ is the abbreviation for $X \vdash b$ for every $b \in Y$. Thus $X \vdash \emptyset$ is vacuously true. Note that the information systems we consider here are not exactly the same as those introduced by Scott. We do not assume a distinguished element Δ, standing for *true*. To compensate for this, we require that X is non-empty when we write $X \vdash a$. This has the effect that the bottom element of the corresponding domain is always the empty set.

The elements $|\,\underline{A}\,|$, of information system $\underline{A} = (A, Con, \vdash)$ consists of subsets x of propositions which are

- consistent: $X \subseteq^{fin} x \Rightarrow X \in Con$, and
- deductively closed: $X \subseteq x \;\&\; X \vdash a \Rightarrow a \in x$.

Let $X \dashv\vdash Y$ be the abbreviation for $X \vdash Y$ and $Y \vdash X$. For technical convenience, we only consider information systems which are *antisymmetric* in this paper:

$$\forall a, b \in A\ [\{a\} \dashv\vdash \{b\} \Rightarrow a = b].$$

Definition 3.1 *Let $\underline{A} = (A, Con, \vdash)$ be an information system. A token $a \in A$ is a quasi-prime token if*

$$X \dashv\vdash a \Rightarrow a \in X.$$

We write A^q for the set of quasi-prime tokens of $\underline{A}$. The information system is called quasi-prime generated iff for each $a \in A$, there is a finite set $X \subseteq A^q$ such that $X \dashv\vdash \{a\}$.

Let $\overline{S}$ stand for the deductive closure of any token set S, i.e.

$$\overline{S} = \{a \mid \exists X \subseteq^{fin} S . X \vdash a\}.$$

We first show that for each quasi-prime token a, $\overline{a}$ is a quasi-prime element. For this purpose, let $y = \overline{a} - \{a' \mid a' \vdash a\}$. Then y is again an ideal element. This is because from the assumption that a is a quasi-prime token we know that for any $X \subseteq^{fin} y$ (therefore $a \vdash X$), $X \vdash a$ implies $b = a$. Now let $z \sqsubset \overline{a}$ be an ideal element. It is clear that $a \notin z$. Therefore $z \sqsubseteq y$, and y is the unique element immediately below $\overline{a}$. By Theorem 2.1, $\overline{a}$ is a quasi-prime.

Suppose x is a quasi-prime in the domain determined by a quasi-prime generated information system. That means there is a unique element y covered by x. Let $a \in x - y$. Clearly $\overline{a} \subseteq x$. If $\overline{a} \neq x$, we must have $\overline{a} \subseteq y$, since y is the unique element immediately below x, and every element strictly below x must therefore below y. This would lead to $a \in y$, contradicting our assumption $a \in x - y$. The only alternative is $\overline{a} = x$. Let $X \subseteq x$ be such that $X \vdash a$. We can also assume that propositions in X does not entail each other unless they are the same. If for each $b \in X$, $\overline{b} \subset x$, then $\overline{b} \subseteq y$ for each $b \in X$, and $\bigsqcup\{\overline{b} \mid b \in X\} \subseteq y$. This is impossible because $X \vdash a$.

Therefore $\overline{b} = \overline{a}$ for some $b \in X$, which means, by antisymmetry, $b = a$.

The above two paragraphs show that an ideal element x of an information system is a quasi-prime if and only if $x = \overline{a}$ for some quasi-prime token a.

In general, we have the following theorem, whose proof can be found in [11].

Theorem 3.1 *For each quasi-prime generated information system $\underline{A}$,*

$$(|\underline{A}|, \subseteq)$$

is a quasi-prime algebraic domain. On the other hand, for any quasi-prime algebraic domain D, there is a quasi-prime generated information system $\underline{A}$ such that $D \cong |\underline{A}|$.

We remark that this representation theorem can be put in a stronger form: we can require every token of a quasi-prime generated information system to be a quasi-prime.

4 Universality and Amalgamation

A unified theory of universal objects can be found in [3]. The basic theorem is that in any algebroidal category in which all morphisms are monic, the existence of a universal, homogeneous object is equivalent to the amalgamation property of the (finite objects of the) category. For reference purposes we recall some of the relevant definitions.

Let **C** be a category where all the morphisms are monic (corresponding to the intuitive notion of one-to-one). Let $\mathbf{C}_f$ be the finite objects of **C**. An object U of **C** is universal in **C** (or, **C**-universal) if for any object A in **C**, there is an arrow $f : A \rightarrow U$. U is homogeneous (or, $\mathbf{C}_f$-homogeneous) if for any finite object A with a pair of arrows $f, g : A \to U$, there is an isomorphism $h : U \to U$ such that $f = h \circ g$. U is saturated if for any A, B of $\mathbf{C}_f$ and arrows $f : A \to U$, $g : A \to B$, there is an $h : B \to U$ such that $h \circ g = f$.

A category $\mathbf{C}$ is said to have the amalgamation property if for any arrows $f_1 : A \to B_1$, $f_2 : A \to B_2$ in $\mathbf{C}$, there are arrows $g_1 : B_1 \to B$, $g_2 : B_2 \to B$ in $\mathbf{C}$ such that the following diagram commute.

$$\begin{tikzcd}
A \arrow[r, "f_1"] \arrow[d, "f_2"'] & B_1 \arrow[d, "g_1"] \\
B_2 \arrow[r, "g_2"'] & B
\end{tikzcd}$$

The result of Droste and Göbel is based on the notion of an algebroidal category.

Definition 4.1 *An algebroidal category is one which has the following properties:*

It has an initial object,

Every object of the category is a colimit of an ω-chain of finite objects,

Every ω-chain of finite objects has a colimit, and

The number of (up to isomorphism) finite objects is countable.

Theorem 4.1 *(Droste and Göbel) Let $\mathbf{C}$ be an algebroidal category with all morphisms monic. Let $\mathbf{C}_f$ be the full subcategory of finite objects of $\mathbf{C}$. The existence of a $\mathbf{C}$-universal, $\mathbf{C}_f$-homogeneous object is equivalent to the amalgamation property of $\mathbf{C}_f$, which is in turn equivalent to the existence of a $\mathbf{C}_f$ saturated object.*

Note that in various categories of information systems, finite objects are often exactly those with a finite token set.

5 Q-EMBEDDINGS

To be able to apply Theorem 4.1, we need to introduce an algebroidal category of quasi-prime generated information systems. The morphisms for this category are q-embeddings.

Definition 5.1 *Let $\underline{A} = (A, Con_A, \vdash_A)$, $\underline{B} = (B, Con_B, \vdash_B)$ be quasi-prime generated information systems. A function $f : A \to B$ is a q-embedding of $\underline{A}$ into $\underline{B}$ if*

1. f *is one-to-one*;
2. $\forall X \subseteq A \forall a \in A$
   $X \in Con_A \Longleftrightarrow f(X) \in Con_B$,
   $X \vdash_A a \Longleftrightarrow f(X) \vdash_B f(a)$;
3. $f(A^q) \subseteq B^q$.

We remark that given a q-embedding from $\underline{A}$ to $\underline{B}$, if $f(a)$ is a quasi-prime for some $a \in A$, then a itself must be a quasi-prime. Indeed, suppose $X \dashv\vdash_A a$. Then $f(X) \dashv\vdash_B f(a)$. But $f(a)$ is a quasi-prime; so $f(a) \in f(X)$, which means $a \in X$ for f is one-one.

It is informative to show an example which is a usual embedding on information systems (i.e., that satisfies conditions 1, 2 above) [5, 2] but not a q-embedding.

Example. Let

$$\underline{A} = (\{1,3\}, Con_A, \vdash_A)$$

where Con_A includes $\{1,3\}$, and $3 \vdash_A 1$, and let

$$\underline{B} = (\{1,2,3\}, Con_B, \vdash_B)$$

where Con_B includes $\{1,2,3\}$, and $3 \vdash_B 1$, $3 \vdash_B 2$, and $\{1,2\} \vdash_B 3$. It is clear that $\underline{A}$ embeds (by identity) into $\underline{B}$ in the usual sense, but not in the sense of a q-embedding. This is because $3 \in A^q$, but $3 \notin B^q$.

Proposition 5.1 *Quasi-prime generated information systems with q-embeddings form a category, written as* $\mathbf{Q}$.

We now present several propositions leading to the main conclusion that $\mathbf{Q}$ is algebroidal.

Proposition 5.2 *Colimits exist in* $\mathbf{Q}$ *for* ω*-chains of finite information systems.*

Proposition 5.3 *Every quasi-prime generated information system is the colimit of an* ω*-chain of finite information systems in* $\mathbf{Q}$.

The above propositions, together with the observation that the empty information system is initial, implies the following.

Theorem 5.1 *Quasi-prime generated information systems with q-embeddings form an algebroidal category.*

It is easy to see that, corresponding to q-embeddings, there is a notion of embedding-projection pairs on quasi-prime algebraic domains. These are just the usual embedding-projection pairs with the additional requirement that they preserve quasi-primes.

6 Existence of a Saturated Quasi-Prime Algebraic Domain

The purpose of this section is to show that the finite objects of the category $\mathbf{Q}$ have the amalgamation property. In light of Theorem 4.1 and Theorem 5.1, this means there exists a saturated quasi-prime generated information system. Note that our proof below follows the style of [2].

Let

$$\underline{A} = (A, Con_A, \vdash_A),$$

$$\underline{B}_1 = (B_1, Con_1, \vdash_1),$$

and

$$\underline{B}_2 = (B_2, Con_2, \vdash_2)$$

be finite quasi-prime generated information systems such that $f_1 : \underline{A} \to \underline{B}_1$ and $f_2 : \underline{A} \to \underline{B}_2$ are q-embeddings. By renaming the tokens, it is enough to consider the case where f_i's are inclusions (partial identities) and $A = B_1 \cap B_2$.

Definition 6.1 *We construct the information system*

$$\underline{B} = (B, Con, \vdash)$$

from $\underline{B}_1$ and $\underline{B}_2$, where

- $B = B_1 \cup B_2$;
- $X \in Con$ *if and only if*

$$\exists Z \exists Y \supseteq X \cap B_1. \, [Y \vdash_1 Z \vdash_2 X \cap B_2] \;\; or$$
$$\exists Z \exists Y \supseteq X \cap B_2. \, [Y \vdash_2 Z \vdash_1 X \cap B_1];$$

- $\vdash = \bigcup_{i \geq 0} \vdash^i$, *where $\vdash^i$'s are specified as follows:*

$$\begin{aligned} \vdash^0 = \{(X, a) \mid X \in Con \; \& \; & either \; B_1 \cap X \vdash_1 a \\ & or \;\; B_2 \cap X \vdash_2 a\}, \\ \vdash^{i+1} = \{(X, a) \mid \exists Y. \, X \vdash^i Y \vdash^i a\}. & \end{aligned}$$

Before getting into technical details, we would like to give the reader an intuitive feeling of what is going on. $\underline{B}_1$ and $\underline{B}_2$ are two given structures sharing a common substructure $\underline{A}$. The question is whether $\underline{B}_1$ and $\underline{B}_2$ can both be seen living in a larger structure. The larger structure must respect the already-existing relationships between $\underline{B}_1$, $\underline{B}_2$, and $\underline{A}$. In particular, $\underline{A}$ serves as a "bridge" between $\underline{B}_1$ and $\underline{B}_2$, so some X from B_1 may entail some Y from A, which

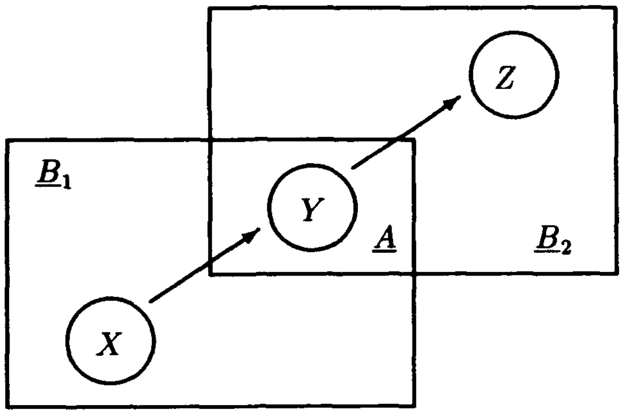


Figure 1: $\underline{A}$ acting as a bridge between $\underline{B}_1$ and $\underline{B}_2$

may in turn entail some Z in B_2. Entailment must be transitive; therefore, in the larger structure we should have X entail Z (see Figure 1). That is why our construction for $\vdash$ is a kind of transitive closure.

The consistency predicate *Con*, however, must come before $\vdash$ can be defined: the $\vdash$ relation is only defined on consistent sets. On the surface, it may look that there can be several choices for specifying *Con*. Letting $X \in Con$ if and only if $X \cap B_1 \in Con_1$ and $X \cap B_2 \in Con_2$, for example, may seem to be a reasonable choice. A second thought, however, reveals that this is not the case.

Example. Let $A = \{1,2\}$, with a trivial consistency predicate and a trivial entailment relation. Let $B_1 = \{\alpha, 1, 2\}$, with $\alpha \vdash_1 1$, and let $B_2 = \{1, 2, a, b\}$ such that $\{a, b\} \notin Con_2$, and $\{1,2\} \vdash_2 b$, $a \vdash_2 2$. It is easy to check that we have three quasi-prime generated information systems in this way, and A q-embeds into B_1 and B_2. In the bigger system to be constructed, do we want to have $\{\alpha, a\} \in Con$ because $\{\alpha\} \in Con_1$ and $\{a\} \in Con_2$? The answer is no: the given conditions $\alpha \vdash_1 1$, $a \vdash_2 2$, and $\{1,2\} \vdash_2 b$ would imply $\{\alpha, a\} \vdash \{a, b\}$, which must be inconsistent.

□

Our original specification in Definition 6.1 seems to be the only right choice. We now check that $\underline{B}$ is indeed a quasi-prime generated information system which makes the diagram required by the amalgamation commute. This is achieved by a sequence of lemmas.

Lemma 6.1 *The structure $\underline{B}$ given in Definition 6.1 is an information system.*

Our next lemma is the key to the proofs of various results later.

Lemma 6.2 *For the information system $\underline{B}$ given in Definition 6.1, $X \subseteq B_1$ and $X \vdash Y$ imply*

$$X \vdash_1 Y \cap B_1 \quad \textit{and}$$
$$\exists Z \subseteq A.\ X \vdash_1 Z \vdash_2 Y \cap B_2.$$

There are a couple of ways Lemma 6.2 can be interpreted. It says that for X's from B_1, $\vdash$ is the same as $\vdash^1$. This means when X is restricted to B_1, $X \vdash Y$ if and only if $X \vdash^1 Y$. It also says that

$$X \subseteq B_1 \ \&\ X \vdash Y \Rightarrow \exists X'.\ X \vdash_1 X' \vdash^0 Y,$$

by taking X' to be $(Y \cap B_1) \cup Z$. These corollaries are sometimes more handy to use.

Lemma 6.3 *For $X \subseteq B_1$ and $a \in B_1$,*

$$X \in Con_1 \Leftrightarrow X \in Con \quad \textit{and} \quad X \vdash_1 a \Leftrightarrow X \vdash a.$$

We now come to another important lemma which deals with quasi-primes. It shows that quasi-primes in $\underline{B}_1$ or $\underline{B}_2$ remain to be quasi-primes in $\underline{B}$.

Lemma 6.4 *Let $\underline{B}$ be the information system specified in Definition 6.1. Then every quasi-prime of its component remains to be a quasi-prime, i.e.,*

$$B_1^q \cup B_2^q \subseteq B^q.$$

Note that the equality $B_1^q \cup B_2^q = B^q$ follows easily from this lemma.

As a corollary of the previous lemmas, we have

Theorem 6.1 *$\underline{B}$ is a quasi-prime generated information system. Moreover, both $\underline{B}_1$ and $\underline{B}_2$ q-embed into $\underline{B}$.*

In summary, we have proved

Theorem 6.2 *The finite objects of the category* **Q** *has the amalgamation property.*

As a consequence, we have shown the following.

Theorem 6.3 *There exists a saturated quasi-prime algebraic domain.*

7 A CPO of Quasi-Prime Generated Information Systems

One of the purposes of a universal domain is to ensure the existence of solutions to domain equations for denotational semantics of programming languages. Another way to achieve the same goal is to introduce a cpo of quasi-prime generated information systems using a substructure relation. Various constructions on information systems can be shown to induce continuous functions on the cpo. The existence of the least fixed point for continuous functions then guarantees the existence of solution to domain equations. The advantage of this approach is that the solution is up to equality, rather than up to an isomorphism. We describe this approach now, which uses the idea first introduced in [5].

Definition 7.1 *Let* $\underline{A} = (A, Con_A, \vdash_A)$ *and* $\underline{B} = (B, Con_B, \vdash_B)$ *be quasi-prime generated information systems.* $\underline{A} \trianglelefteq \underline{B}$ *if*

1. $A \subseteq B$
2. $X \in Con_A \iff X \subseteq A \;\&\; X \in Con_B$,
3. $X \vdash_A a \iff X \cup \{a\} \subseteq A \;\&\; X \vdash_B a$, *and*
4. $A^q \subseteq B^q$.

When $\underline{A} \trianglelefteq \underline{B}$ we call $\underline{A}$ a subsystem of $\underline{B}$. Our definition of subsystem is similar to that of Larsen and Winskel [5], with item 4 the only extra requirement. Note that for quasi-prime generated information system $\underline{A}$ and $\underline{B}$, if $A = B$ and $\underline{A} \trianglelefteq \underline{B}$, then $\underline{A} = \underline{B}$.

The relation $\trianglelefteq$ is a complete partial order on the *class* of quasi-prime generated information systems.

Theorem 7.1 *The relation* $\trianglelefteq$ *is a partial order with the least element*

$$\underline{\perp} = (\emptyset, \{\emptyset\}, \emptyset).$$

If $\underline{A_0} \trianglelefteq \underline{A_1} \trianglelefteq \cdots \trianglelefteq \underline{A_i} \trianglelefteq \cdots$ *is an increasing chain of stable information systems where* $\underline{A_i} = (A_i, Con_i, \vdash_i)$, *then their least upper bound is*

$$\bigcup_i \underline{A_i} = \left(\bigcup_i A_i,\ \bigcup_i Con_i,\ \bigcup_i \vdash_i\right).$$

Write **CPO** for the class of quasi-prime generated information systems under $\trianglelefteq$. **CPO** is not a cpo in the usual sense simply because they are not a set but a class. The subsystem relation $\trianglelefteq$ can be easily extended to $n-$tuples in the same way as described in [8]. A useful observation is that a unary operation F is continuous iff it is monotonic with respect to $\trianglelefteq$ and continuous on proposition sets, *i.e.* for any ω-chain

$$\underline{A}_1 \trianglelefteq \underline{A}_2 \cdots \trianglelefteq \underline{A}_i \trianglelefteq \cdots,$$

each proposition of $F(\bigcup_i \underline{A}_i)$ is a proposition of $\bigcup_i F(\underline{A}_i)$.

Many constructions can be introduced on quasi-prime generated information systems such as sum, product, lifting, function space, and even quasi-linear function space.

In the rest of the section we illustrate that function space $\rightarrow$ corresponds to a continuous operation

$$\rightarrow: \mathbf{CPO}^2 \rightarrow \mathbf{COP}.$$

Other constructions can be shown, in a similar way, to induce continuous operations on **CPO**.

The function space construction is given as follows.

$$\begin{aligned}
&\underline{A} \rightarrow \underline{B} = (Con_A \times B^q, Con_{A\rightarrow B}, \vdash_{A\rightarrow B}), \quad \text{where}\\
&\qquad X \in Con_{A\rightarrow B} \ \text{ iff } \ \bigcup \pi_A(X) \in Con_A \Rightarrow \text{[illegible]}(X) \in Con_B,\\
&\qquad X \vdash_{A\rightarrow B} (u, b) \text{ iff } \ \{b' \mid \exists u'. u \vdash_A u' \ \& \ (u', b') \in X\} \vdash_B b
\end{aligned}$$

Note that $\rightarrow$ preserves quasi-prime generated information systems. $\rightarrow$ is monotonic in its first argument. Suppose $\underline{A} \trianglelefteq \underline{A'}$. Write

$$\underline{C} = (C, Con, \vdash) = [\underline{A} \rightarrow \underline{B}]$$

and

$$\underline{C'} = (C', Con', \vdash') = [\underline{A'} \rightarrow \underline{B}].$$

We check condition 4 in Definition 7.1, to show that $\underline{C} \trianglelefteq \underline{C'}$. However, this is trivial because every token of the function space is a quasi-prime.

Let

$$\underline{A_0} \trianglelefteq \underline{A_1} \trianglelefteq \cdots \trianglelefteq \underline{A_i} \trianglelefteq \cdots$$

be a chain of quasi-prime generated information systems. Let (u, a) be a token of $[(\bigcup_i \underline{A_i}) \rightarrow \underline{B}]$. Then clearly (u, a) is a token of $[\underline{A_j} \rightarrow \underline{B}]$ for some j, and thus is a token of $\bigcup_i [\underline{A_i} \rightarrow \underline{B}]$. ¿From this we can deduce that $\rightarrow$ is continuous in its first argument. By a similar but easier proof we get that $\rightarrow$ is continuous in its second argument hence it is continuous.

As an application, we illustrate how to find a solution to the equation

$$X = X_{\uparrow} \rightarrow X$$

within quasi-prime generated information systems. Here $(\ )_{\uparrow}$ is the lifting construction specified as follows. Its use here makes sure that a non-trivial solution is obtained.

Let $\underline{A} = (A, Con, \vdash)$ be a quasi-prime generated information system. Define the *lift* of $\underline{A}$ to be $\underline{A}_{\uparrow} = (A', \vdash')$, where

- $A' = (\{0\} \times A) \cup \{0\}$,
- $X \vdash' Y \Leftrightarrow [0 \in Y$ or $\{c \mid (0, c) \in X\} \vdash_{\underline{A}} \{b \mid (0, b) \in Y\}]$.

It is easy to show that lifting preserves quasi-prime generated systems. It is an operation which, given a structure, produces a new one by joining a new token weaker than all the old ones. One can easily check that it gives a continuous operation.

Since the composition of continuous functions remains continuous, and any continuous function $F(x, y)$ of two variables gives rise to a continuous function $F(x, x)$ of one variable, the operation

$$X \longmapsto [X_{\uparrow} \rightarrow X]$$

is a continuous operation on quasi-prime generated information systems. It has a least fixed point

$$\underline{A} = \underline{A}_{\uparrow} \rightarrow \underline{A}.$$

This can be used as a model for the un-typed lambda calculus.

ACKNOWLEDGMENTS Thanks go to Michael Huth for pointing out to me that, I had, as a by-product of considering quasi-prime algebraic domains, rediscovered a notion of completely irreducible elements. I am grateful to Michael Mislove who provided travel support for me to present the results in the 9th International Conference on the Foundations of Programming Semantics.

References

[1] Droste, M., and Göbel, R. Universal domains in the theory of denotational semantics of programming languages. Proc. of the IEEE 5th annual symposium on logic in computer science, 1990.

[2] Droste, M., and Göbel, R. Universal information systems. *International Journal of Foundations of Computer Science* vol. 1, no. 4, 1991.

[3] Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M., and Scott, D.S. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.

[4] Gunter, C. Universal profinite domains. *Information and Computation* 72, 1987.

[5] Larsen, K. and Winskel, G. Using information systems to solve recursive domain equations effectively. In *Lecture Notes in Computer Science 173*, 1984.

[6] Scott, D.S. Domains for denotational semantics. In *Lecture Notes in Computer Science 140*, 1982.

[7] Winskel, G. An introduction to event structures. *Lecture Notes in Computer Science 354*. 1988.

[8] Zhang, G.-Q. DI-domains as prime information systems. *Information and Computation* 100, 1992. Also in Lecture Notes in Computer Science 372 (ICALP-1989, Italy).

[9] Zhang, G.-Q. Some monoidal closed categories of domains and event structures *Mathematical Structures in Computer Science*, to appear. Also: Proceedings of the 7th Conference on Mathematical Foundations of Programming Semantics, Pittsburgh, 1991.

[10] Zhang, G.-Q. *Logic of Domains.* Birkhauser, Boston, 1991.

[11] Zhang, G.-Q. Quasi-prime algebraic domains: a linear category of non-linear functions. To appear. 1992.

Holomorphic Models of Exponential Types in Linear Logic

R.F. Blute[*]
Department of Mathematics
University of Ottawa
Ottawa, Canada K1N 6N5
rblute@acadvm1.uottawa.ca

Prakash Panangaden[†]
School of Computer Science
McGill University
Montreal, Canada H3A 2A7
prakash@cs.mcgill.ca

R.A.G. Seely[‡]
Department of Mathematics
McGill University
Montreal, Canada H3A 2K6
rags@math.mcgill.ca

Abstract

In this paper we describe models of several fragments of linear logic with the exponential operator ! (called OF COURSE) in categories of linear spaces. We model ! by the *Fock space* construction in Banach (or Hilbert) spaces, a notion originally introduced in the context of quantum field theory. Several variants of this construction are presented, and the representation of Fock space as a space of holomorphic functions is described. This also suggests that the "non-linear" functions we arrive at *via* ! are not merely continuous, but analytic.

Keywords: Fock space, linear logic, Banach spaces, holomorphic functions, quantum field theory.

[*]Research supported in part by FCAR of Quebec.
[†]Research supported in part by an NSERC Research Grant.
[‡]Research supported in part by an NSERC Research Grant and by FCAR of Quebec.

0 Introduction

Linear logic was introduced by Girard [G87] as a consequence of his analysis of the traditional connectives of logic into more primitive connectives. The resulting logic is more resource sensitive; this is achieved by placing strict control over the structural rules of contraction and weakening, introducing a new "modal" operator OF COURSE (denoted $!$) to indicate when a formula may be used in a resource-*insensitive* manner—*i.e.* when a resource is renewable. Without the $!$ operator, the essence of linear logic is carried by the multiplicative connectives; at its most basic level, linear logic is a logic of monoidal-closed categories (in much the same way that intuitionistic logic is a logic of cartesian-closed categories). In modelling linear logic, one begins with a monoidal-closed category, and then adds appropriate structure to model linear logic's additional features. To model linear negation, one passes to the $*$-autonomous categories of Barr [B79]. To model the additive connectives, one then adds products and coproducts. Finally, to model the exponentials, and so regain the expressive strength of traditional logic, one adds a triple and cotriple, satisfying properties to be outlined below. This program was first outlined by Seely in [Se89].

Linear logic bears strong resemblance to linear algebra (from which it derives its name), but one significant difference is the difficulty in modelling $!$. The category of vector spaces over an arbitrary field is a symmetric monoidal closed category, indeed in some sense the prototypical monoidal category, and as such provides a model of the intuitionistic variant of multiplicative linear logic. Furthermore, this category has finite products and coproducts with which to model the additive connectives. It thus makes sense to look for models of various fragments of linear logic in categories of vector spaces. However, modelling the exponentials is more problematic. It is the primary purpose of this paper to present methods of modelling exponential types in categories arising from linear algebra. We study models of the exponential connectives in categories of linear spaces which have monoidal (but generally not monoidal-closed) structure. (We shall also include a model in finite-dimensional vector spaces.)

To model the finer distinctions achieved by linear logic, one ought to consider vector spaces enriched with appropriate additional structure. For example, to model linear negation, one considers vector spaces enriched with an additional topological structure. These are the *linear topologies* of Lefschetz and Barr [Le41, B76a]. The relationship to linear logic is discussed in [Bl93a]. To model the noncommutative [Ab91] or braided

[Bl93b] variants of linear logic, one considers the linear representations of certain Hopf algebras [Bl93a]. Finally, to model the exponentials, it is necessary to consider normed vector spaces.

Vector spaces are inherently finitary structures in the sense that every vector is a finite sum of multiples of basis vectors, and one is allowed only to take finite sums of vectors. To model the notion of infinitely renewable resources, one would like to be able to take infinite sums of vectors. But to do this, one needs a notion of convergence, and to define convergence one needs a notion of topology. The most heavily studied topological vector spaces are Hilbert and Banach spaces which derive their topologies from a norm; either defined indirectly *via* an inner product, as in Hilbert spaces, or directly, as in Banach spaces. Once a vector space is normed, then all of the familiar notions from analysis, such as limit and Cauchy sequence can be defined. What we wish to suggest in this paper is that while the multiplicative and additive fragment MALL of linear logic corresponds to the linear structure of a vector space, the exponentials correspond to its analytic structure.

We begin by introducing the two main notions of complete normed vector space, Banach spaces and Hilbert spaces. The construction which will be used to model the exponential formulas $!A$ arose originally in quantum field theory, and is known as *Fock space*. It was designed as a framework in which to consider many particle states. The key point of departure for quantum field theory was the realization that so-called "elementary" particles are created and destroyed in physical processes and that the mathematical formalism of ordinary quantum mechanics needs to be revised to take this into account. The physical intuitions behind the Fock construction will be sketched in the penultimate section. The formula for Fock space will also be familiar to mathematicians in that it corresponds to the free symmetric algebra on a space. As a free construction, Fock induces a pair of adjoint functors, and hence a cotriple. It is this cotriple which will be used to model $!$. It should be noted that this category of algebras inherits the monoidal structure from the underlying category of spaces but there is no hope that this category could have a monoidal-closed structure.

While Fock space has an abstract representation in terms of an infinite direct sum, physicists such as Ashtekar, Bargmann, Segal and others, see [AM-A80, Ba61, S62] have analyzed concrete representations of Fock space as certain classes of holomorphic functions on the base space. Thus, these models further the intuition that the exponentials correspond to the analytic properties of the space. In fact, there is a clear sense in which morphisms in the Kliesli category for the cotriple can be viewed as generalized holomorphic functions. Thus, there should be an analogy to coherence spaces where the Kliesli category corresponds to the stable maps.

Fock space also has two additional features which correspond to additional structure, not expressible in the syntax of linear logic. These are the *annihilation* and *creation* operators, which are used to model the annihilation and creation of particles in a field. These may give a tighter control of resources not expressible in the pure linear logic. Thus, these models may be closer to the bounded linear logic of Girard, Scedrov and Scott [GSS91].

The results of this paper suggest that analyticity may provide new insights into computability not captured by the traditional notions of continuity. Continuity has been enormously successful in capturing the idea that computable functions process information a finite piece at a time. On the other hand, there are many continuous functions that are not computable. Despite the tremendous clarifications brought about by Scott's ideas, a precise characterization of computability still appeals to notions of encoding from classical recursion theory. With the notion of analytic function one has the notion of convergent power series which represents the function. This is nothing more than an encoding of a continuous function with a discrete string. Thus the notion of encoding *may* be captured by analyticity. Of course, we are far from offering any such theory yet.

Another possible application of this work is that the refined connectives of linear logic may lend insight into certain aspects of quantum field theory. For example, there are two distinct methods of combining particle states. One can superimpose two states onto a single particle, or one can have two particles coexisting. The former seems to correspond to additive conjunction and the latter to the multiplicative. This physical imagery is missing in quantum mechanics, which was specially designed to handle a single particle; it only shows up in quantum field theory.

In this paper, we begin by reviewing the categorical structure necessary to model linear logic, and specifically exponential types. We then give the relevant definitions pertaining to normed vector spaces, as well as a number of examples. We also discuss the monoidal structure of these categories. Then, the various ingredients which go into the construction of Fock space are presented and the resulting adjointness is described. Finally, the holomorphic function representation of Fock space is presented, and a brief description of its physical interpretation is given.

1 Linear Logic and Monoidal Categories

We shall begin with a few preliminaries concerning linear logic. We shall not reproduce the formal syntax of linear logic, nor the usual discussion of its intuitive interpretation or utility—for this the reader is referred to the standard references, such as [G87]. We do recall [Se89] that a categorical semantics for linear logic may be based on Barr's notion of $*$-autonomous categories [B79]. If only to establish notation, here is the definition.

Definition 1 *A category C is $*$-autonomous if it satisfies the following:*

1. *C is symmetric monoidal closed; that is, C has a tensor product $A \otimes B$ and an internal hom $A \multimap B$ which is adjoint to the tensor in the second variable*

$$Hom(A \otimes B, C) \cong Hom(B, A \multimap C)$$

2. *C has a dualizing object $\perp$; that is, the functor $(\ )^{\perp}: C^{op} \longrightarrow C$ defined by $A^{\perp} = A \multimap \perp$ is an involution (*viz.* the canonical morphism $A \longrightarrow ((A \multimap \perp) \multimap \perp)$ is an isomorphism).*

In addition various coherence conditions must hold—a good account of these may be found in [M-OM89]. Coherence theorems may be found in [BCST, Bl91, Bl92]. An equivalent characterization of $*$-autonomous categories is given in [CS91], based on the notion of weakly distributive categories. That characterization is useful in contexts where it is easier to see how to model the tensor $\otimes$, the "par" $\parr$ and linear negation, and the coherence conditions may be expressed in terms of those operations.

The structure of a $*$-autonomous category models the evident eponymous structure of linear logic: the categorical tensor $\otimes$ is the linear multiplicative $\otimes$ and the internal hom $\multimap$ is linear implication. The dualizing object $\perp$ is the unit for linear "par" $\parr$, or equivalently, is the dual of the unit I for the tensor[1].

There are a number of variants of linear logic whose categorical semantics is based on this. First is full "classical" linear logic, which includes the additive operations. These correspond to requiring that the category

[1]In other papers we have used the notation $\top$ for the unit for $\otimes$, and $\oplus$ instead of $\parr$. Here we shall try to avoid controversy by using notation traditional in the context of Banach spaces, and by generally ignoring the "par". So in this paper, $\oplus$ means direct sum, which coincides with Girard's notation. We use $\times$ for cartesian product, corresponding to Girard's &. And we shall use the usual notation for the appropriate spaces when referring to the units.

$\mathcal{C}$ have products and coproducts. (If $\mathcal{C}$ is $*$-autonomous, one of these will imply the other by de Morgan duality.) There is also Girard's notion of "intuitionistic" linear logic [GL87], which omits linear negation and "par"—this corresponds to merely requiring that $\mathcal{C}$ be *autonomous*, that is to say, symmetric monoidal closed (with or without products and coproducts, depending on whether or not the additives are wanted). There is an intermediate notion, "full intuitionistic linear logic" due to de Paiva [dP89], in which the morphism $A \longrightarrow A^{\perp\perp}$ need not be an isomorphism. And as mentioned above, there is the notion of weakly distributive category [CS91, BCST], where negation and internal hom are not required.

One classically important class of $*$-autonomous categories are the *compact* categories [KL80] where the tensor is self-dual: $(A \otimes B)^{\perp} \cong A^{\perp} \otimes B^{\perp}$. Linear logicians often regard with derision those models in which "tensor" and "par" coincide, but from some mathematical points of view these are very natural.

In this paper we shall model various fragments of linear logic; we shall describe the fragments in terms of the categorical structure present, without explicitly identifying the fragments.

Finally, in order to be able to recapture the full strength of classical (or intuitionistic) logic, one must add the "exponential" ! (and its de Morgan dual ?). (All our structures will model !.) We saw in [Se89] that this amounts to the following.

Definition 2 *A monoidal category $\mathcal{C}$ with finite products admits (Girard)* **storage** *if there is a cotriple* $!: \mathcal{C} \longrightarrow \mathcal{C}$ *(with the usual structure maps* $A \xleftarrow{\epsilon_A} \,!A \xrightarrow{\delta_A} \,!!A$*), satisfying the following:*

1. *for each object $A \in \mathcal{C}$, $!A$ carries (naturally) the structure of a (cocommutative) $\otimes$-comonoid* $\top \xleftarrow{e_A} \,!A \xrightarrow{d_A} \,!A \otimes \,!A$ *(and the coalgebra maps are comonoid maps), and*

2. *there are natural comonoidal isomorphisms*
$$\mathrm{I} \xrightarrow{\sim} \,!1 \quad \text{and} \quad !A \otimes \,!B \xrightarrow{\sim} \,!(A \times B)\ .$$

Some remarks: First, it is not hard to see that the first condition above is redundant, the comonoidal structure on $!A$ being induced by the isomorphisms of the second condition. However, the first condition is really the key point here, as may be seen from several generalizations of this definition, to the intuitionistic case without finite products in [BBPH], and to the weakly distributive case, again without finite products, [BCS93].

The main point here is that without products one replaces the second condition with the requirement that the cotriple ! (and the natural transformations ϵ, δ) be comonoidal. And second, one ought not drown in the categorical terminology—terms like "comonoidal" in essence refer to various coherence (or commutativity) conditions which may be looked up when needed. Readers not interested in coherence questions can follow the discussion by just noting the existence of appropriate maps, and believe that all the "right" diagrams will commute. They can regard it as somebody else's business to ensure that this is indeed the case.

In the mid-1980's, Girard studied coherence spaces as a model of system F, and realized the following fact, which led directly to the creation of linear logic. Of course Girard did not put the matter in these categorical terms at the time, but the essential content remains the same—ordinary implication factors through linear implication *via* the cotriple !. (Another way of expressing this is to say that a model of full classical linear logic induces an interpretation of the typed λ-calculus.)

Theorem 1 *If C is a $*$-autonomous category with finite products admitting Girard storage !, then the Kleisli category $C_!$ is cartesian closed.*

This result is virtually folklore, but a proof may be found in [Se89].

One of the problems with finding models of linear logic comes from the difficulty of finding well-behaved (in the above sense) cotriples on $*$-autonomous categories. For example, one of the main problems with vector spaces as a model of linear logic is the lack of any natural interpretation of !. (We shall soon return to this point, and indeed, in a sense this is the main point of this paper.) This question seems closely bound up with questions of completeness. Barr [B91] has shown how in certain cases one can get appropriate cotriples (*via* cofree coalgebras) from a subcategory of the Chu construction [B79]. One case where this route works out fairly naturally is if the $*$-autonomous category is compact: in that case, one can construct cofree coalgebras by the familiar formula

$$!A = \top \times A \times (A \otimes_s A) \times (A \otimes_s A \otimes_s A) \times \cdots$$

(where the tensors $\otimes_s$ are the symmetric tensor powers). We shall see an echo of this construction in the Fock space construction below.

2 Normed Vector Spaces

As discussed in the introduction, we will be primarily working in normed vector spaces. Normed spaces seem necessary to capture correctly the intuition behind Girard's exponentials. Vector spaces are, in some sense, intrinsically finitary structures. Every vector is a finite sum of multiples of basis vectors, and one is only allowed to take finite sums of arbitrary vectors. It seems likely that to correctly model ! and ?, one should be able to take infinite sums of vectors, thereby capturing the idea of infinitely renewable resource. However, to do this, one needs a notion of convergence. And to define convergence, one needs a notion of norm. Once a space is normed, then it is possible to define limits and Cauchy sequences, and so on. Normed vector spaces, which are the principal objects of study in functional analysis, should be considered as the meeting ground of concepts from linear algebra and analysis. They are also an ideal place to model linear logic.

We will now briefly review the basic concepts of the subject. For more complete discussions, see [KR83, C90, CLM79].

Henceforth all vector spaces are assumed to be over the complex numbers and are allowed to be infinite-dimensional. We will use Greek letters for complex numbers and lower-case Latin letters from the end of the alphabet for vectors.

Definition 3 *A* **norm** *on a vector space V is a function, usually written $\| \, \|$, from V to $\mathbf{R}$, the real numbers, which satisfies*

1. $\| v \| \geq 0$ *for all* $v \in V$,
2. $\| v \| = 0$ *iff* $v = 0$,
3. $\| \alpha v \| = | \alpha | \| v \|$,
4. $\| v + w \| \leq \| v \| + \| w \|$.

For finite dimensional vector spaces the norm usually used is the familiar Euclidean norm. As soon as one has a norm one obtains a metric by the equation $d(u, v) = \| u - v \|$. One can ask whether the resulting space is complete or not as a metric space. It turns out that the spaces that are complete play a central role in functional analysis.

2.1 Banach Spaces

Definition 4 *A* **Banach space** *is a complete, normed vector space.*

Example 1 Consider the space of sequences of complex numbers. We write a for such a sequence, $a = \{a_n\}_{n=1}^{\infty}$ and we write $\| a \|_\infty$ for the supremum of the $| a_i |$.

$$l_\infty = \{a : \| a \|_\infty < \infty\}$$

This is a Banach space with $\| a \|_\infty$ as the norm.

Another norm is obtained on sequences as follows. Define:

$$\| a \|_1 = \Sigma_{n=1}^{\infty} | a_i |$$

Then let:

$$l_1 = \{a : \| a \|_1 < \infty\}$$

More generally, if $p \geq 0$, we may define:

$$l_p = \{a : \| a \|_p \equiv (\Sigma_{n=1}^{\infty} | a |^p)^{1/p} < \infty\}$$

All of these will be examples of Banach spaces. Furthermore, these can be defined not only for sequences of complex numbers, but for sequences obtained from any Banach space.

Example 2 Let X be a compact Hausdorff space. The vector space of complex-valued continuous functions on X is generally denoted $\mathbf{C}(X)$. Since X is compact, such functions must have a supremum, and from this it is straightforward to obtain a norm. Now convergence in this norm is the familiar notion of uniform convergence. As is well known from elementary analysis, sequences of uniformly bounded, continuous functions converge to a bounded continuous function. Thus, we have a Banach space. On the other hand if we looked at functions that vanish outside some closed, bounded interval (the functions of compact support) then we do not get a Banach space since these could converge to a function that does not have compact support.

The following theorem shows one common way in which Banach spaces arise. First we need a definition.

Definition 5 *Suppose that B_1, B_2 are Banach spaces and that T is a linear map from B_1 to B_2. We say that T is* **bounded** *if $sup_{x \neq 0} \frac{\| Tx \|}{\| x \|}$ exists. We define the* **norm of** *T, written $\| T \|$, to be this number.*

If T is indeed bounded, then a standard argument [KR83], establishes

Lemma 2 $sup_{\|x\|=1} \| Tx \| = \| T \|$.

Thus one can use vectors of unit norm to calculate the norm of a linear function rather than having to look for the sup over all nonzero vectors.

Linear maps from a Banach space to itself are traditionally called *operators*, and the norm of such maps is called the *operator norm*.

Since a Banach space is also a metric space under the induced metric described above, one can also ask to characterize which linear maps are also continuous. In this regard, we have the following result.

Lemma 3 *A linear map from $f : A \rightarrow B$ is continuous if and only if it is bounded.*

The following theorem shows that the category of Banach spaces and bounded linear maps is enriched over itself.

Theorem 4 *If A is a normed vector space and B is a Banach space then the space of bounded linear maps with the norm above is a Banach space.*

We will denote this space $A \multimap B$.

There are several possible categories of interest with Banach spaces as the objects. The most obvious one is the category with bounded linear maps as the morphisms. However, it turns out that the category with *contractive maps*[2] is of greater interest and has nicer categorical properties. These properties are discussed in [B76a] and below.

Definition 6 *A* **contractive map**, *T, from A to B is a bounded linear map satisfying the condition, $\| Tx \| \leq \| x \|$. Equivalently, the contractive maps are those of norm less than or equal to 1.*

We will write $\mathcal{BANCON}$ for the category of Banach spaces and contractive maps and $\mathcal{BANACH}$ for the category of Banach spaces and bounded linear maps. While $\mathcal{BANCON}$ has a richer categorical structure, for the purposes of modelling the exponential types of linear logic, we will be forced to work in $\mathcal{BANACH}$.

2.2 Monoidal Structure of $\mathcal{BANACH}$

We first point out that $\mathcal{BANACH}$ has a canonical symmetric monoidal closed structure. We begin by constructing a tensor product. Let A and B be objects in $\mathcal{BANACH}$. Begin by forming the tensor of A and B, $A \otimes_{\mathbf{C}} B$, as complex vector spaces. We first define a partial norm for elements of the form $a \otimes b$ by the equation:

$$\| a \otimes b \| = \| a \| \| b \|$$

[2]Strictly speaking, they should be called "non-expansive" maps.

We would like to extend this partial norm to a norm on all of $A \otimes_{\mathbf{C}} B$. Such a norm is called a *cross norm.* It turns out that there are many such cross norms, a number of which were discovered by Grothendieck. The one we will use in this paper is called the *projective cross norm.* It is in some sense the least such. A detailed discussion of these issues is contained in [T79]. The projective cross norm is defined for an arbitrary element, x, of $A \otimes_{\mathbf{C}} B$ by the following formula:

$$\| x \| = inf\{\| a \| \| b \| \text{ such that } x = \Sigma a \odot b\}$$

One can verify that this is in fact a cross norm on $A \otimes_{\mathbf{C}} B$. Now, the resulting normed space will not be complete in general, so one obtains a Banach space by completing it. This will act as the tensor product in the category $\mathcal{BANACH}$. It will be denoted simply by $A \odot B$. Furthermore, we have the following adjunction in $\mathcal{BANACH}$.

Lemma 5 *The functor $B \otimes (\ )$ is left adjoint to $B \multimap (\ )$.*

Corollary 6 *$\mathcal{BANACH}$ is a symmetric monoidal closed category.*

Analogously, $\mathcal{BANCON}$ is also a monoidal closed category. Note that although one only uses contractive maps in this category, the internal hom is still given by all bounded linear maps.

As such, they are models of (at least) the multiplicative fragment of intuitionistic linear logic. To obtain a model of the classical linear logic, one possibility is the topological construction of Barr in [B76a]. See also [Bl93a]. The idea is to add an additional topological structure to the space, and then only consider maps which are also continuous with respect to this topology. If the topology is chosen carefully, one obtains a large class of reflexive objects, *i.e.* objects which are isomorphic to their double dual space. Such objects can be used to model the negation of classical linear logic.

2.3 Completeness Properties of $\mathcal{BANCON}$ and $\mathcal{BANACH}$

The main advantage of studying the category of contractions is in its completeness properties. While $\mathcal{BANACH}$ has very weak completeness properties, $\mathcal{BANCON}$ is complete and cocomplete. These constructions exist in $\mathcal{BANACH}$ but some lose the universal property. We will describe some of these universal properties. We begin with finite coproducts.

Definition 7 *Let A and B be Banach spaces. The direct sum, $A \oplus B$, is the Cartesian product equipped with the norm $\| a \oplus b \| = \| a \| + \| b \|$.*

Then we have the distributivity property of $\otimes$ over $\oplus$.

Proposition 7 $A \otimes (B \oplus B') \cong (A \otimes B) \oplus (A \otimes B')$.

We now discuss finite products.

Definition 8 *The product of two Banach spaces, $A \times B$, has as its underlying space $A \oplus B$, but now with norm given by:*

$$\| a \oplus b \| = max\{\| a \|, \| b \|\}$$

As a category of vector spaces, $\mathcal{BANCON}$ is fairly unique in this respect. While most such categories model the additive fragment of linear logic, they invariably equate the two connectives, since finite products and coproducts coincide. In other words, $\mathcal{BANCON}$ does not share the familiar property of being an additive category.

We now present countably infinite products and coproducts.

Definition 9 *Let $\{A_i\}_{i=1}^{\infty}$ be a sequence of Banach spaces. Define $\Pi(A_i)$ to be those sequences which converge in the l_∞ norm,* i.e. *bounded sequences equipped with the obvious norm.*

Define $\Sigma(A_i)$ to be all sequences which converge in the l_1 norm.

This gives countable products and coproducts in $\mathcal{BANCON}$. Similar constructions can be applied for uncountable products and coproducts.

Equalizers in $\mathcal{BANCON}$ correspond to equalizers in the underlying category of vector spaces. The fact that bounded maps are continuous implies that the subspace will be complete. Coequalizers are obtained as a quotient, with the induced norm being the infimum of the norms of the elements of the equivalence class. See [C90] for a discussion of quotients of Banach spaces.

Theorem 1 *$\mathcal{BANCON}$ is complete and cocomplete.*

All of the above constructions exist in $\mathcal{BANACH}$, but some of them will lose their universal property. $\mathcal{BANACH}$ is an additive category, with sums and products given by the coproducts in $\mathcal{BANCON}$. (Note that the two spaces $A \times B$ and $A \oplus B$ are isomorphic in $\mathcal{BANACH}$, but not in $\mathcal{BANCON}$.) In $\mathcal{BANACH}$, the above infinite products and coproducts exist, but do not share the universal property. They only have this property for bounded families of maps. Equalizers and coequalizers are as in $\mathcal{BANCON}$.

2.4 Hilbert Spaces

An alternate approach to defining a norm on a vector space is *via* an inner product. An inner product has the property that it induces a norm on the underlying space.

Definition 10 *Given a complex vector space, V, an* **inner product** *for V is a function from $V \times V$ to the complex numbers which is conjugate linear in its first argument and linear in its second argument. This is written $\langle u|v\rangle$.*

Furthermore, an inner product must have the following properties.

- $\langle x|x\rangle \geq 0$
- $\langle x|y\rangle = \overline{\langle y|x\rangle}$
- *if* $\langle x|x\rangle = 0$, *then* $x=0$

Here, $\bar{z}$ refers to complex conjugation. Real Hilbert spaces are defined analogously, with conjugation being taken to be the identity.

Given an inner product we immediately get a norm by $\| x \| = (\langle x|x\rangle)^{1/2}$. As with Banach space what turns out to be crucial is the property of being complete.

Definition 11 *A* **Hilbert space** *is a vector space equipped with an inner product such that the vector space is complete in the induced norm.*

Example 3 The space l_2 of all sequences of complex numbers such that:

$$\Sigma_{i=1}^{\infty} | a_i |^2 \leq \infty$$

One defines an inner product by:

$$\langle x|y\rangle = \Sigma_{i=1}^{\infty} x_i \overline{y_i}$$

Every finite dimensional complex vector space is a Hilbert space with the usual inner product.

The category of Hilbert spaces and bounded linear maps will be denoted by $\mathcal{HILBERT}$. This category has a tensor product which can be constructed in a manner analogous to the construction for Banach spaces. $\mathcal{HILBERT}$ also has finite products and coproducts, in both cases these are given by direct sum, with the evident inner product. $\mathcal{HILBERT}$ does not have very many infinite limits or colimits.

3 Symmetric and Antisymmetric Tensors

We introduce two further constructions in the category $\mathcal{BANACH}$. These will be quotients of the tensor product. Since the category has coequalizers such quotients will be well-defined.

3.1 Symmetric Tensor Products

First, we introduce the symmetric tensor product of a Banach space with itself.

Definition 12 *Let A be a Banach space. The Banach space $A \otimes_s A$ is defined to be the following coequalizer:*

$$\begin{tikzcd} A \otimes A \arrow[r, shift left, "id"] \arrow[r, shift right, "\tau"'] & A \otimes A \arrow[r] & A \otimes_s A \end{tikzcd}$$

Note that τ is the twist map, $a \otimes b \mapsto b \otimes a$.

This is the general definition of symmetrized tensor. It turns out that in categories of vector spaces, this quotient is canonically isomorphic to the equalizer of these two maps, and that this equalizer is split by the map:

$$a \otimes b \mapsto \frac{1}{2}(a \otimes b + b \otimes a)$$

We will frequently use this representation in the sequel.

The n^{th} symmetric power is defined analogously. The Banach space $\bigotimes^n A$ has $n!$ canonical endomorphisms, and the Banach space $\bigotimes^n_S$ is the coequalizer of all of these. Again, it is isomorphic to the equalizer, and there is a splitting, as above. A good way to view the symmetrized tensor is to observe that the symmetric group acts on the space $\bigotimes^n A$, and that the symmetrized tensor is the invariant subspace. As such, an appropriate notation for the symmetrized tensor is:

$$\frac{\bigotimes^n A}{n!}$$

We will also freely use this representation, as well.

3.2 Antisymmetric Tensor Products

This will be defined in a similar fashion. Again, we first define the antisymmetric tensor of a Banach space B with itself. It will be denoted $B \otimes_A B$. It is the coequalizer of the following diagram:

$$\begin{tikzcd} B \otimes B \arrow[r, shift left, "id"] \arrow[r, shift right, "-\tau"'] & B \otimes B \arrow[r] & B \otimes_A B \end{tikzcd}$$

Here, $-\tau$ is the map $a \otimes b \mapsto -b \otimes a$.

Members of this space can canonically viewed as elements of the ordinary tensor product, of the form:

$$x = a \otimes b - b \otimes a$$

The n^{th} antisymmetric power is defined analogously.

4 Fock space and categories of algebras

4.1 Fock space

We are now ready to define the Fock spaces. They are traditionally defined in $\mathcal{HILBERT}$; we will, however, define them in $\mathcal{BANACH}$.

Definition 13 *Let B be a Banach space. The* **symmetric Fock space** *of B is the infinite direct sum of the spaces $\bigotimes_s^n B$, where, when n is zero we use the complex numbers. The* **antisymmetric Fock space** *of B is the infinite direct sum of the spaces $\bigotimes_A^n B$.*

$$\mathcal{F}(B) = C \oplus B \oplus \cdots \oplus \bigotimes\nolimits_s^n B \oplus \cdots$$

$$\mathcal{F}_A(B) = C \oplus B \oplus \cdots \oplus \bigotimes\nolimits_A^n B \oplus \cdots$$

Since Fock is defined using infinite direct sums and coequalizers it is clear that Fock defines a functor.

We can think of an element of $\mathcal{F}(B)$ as an infinite sequence $\langle c, v_1, v_2, \ldots \rangle$ where c is a complex number and $v_i \in B_i$.

Now we check that the Fock space actually satisfies all the properties that need to be satisfied by an OF COURSE type, *i.e.* satisfies the properties of [Se89], discussed in Section 1. This consists of two parts, verifying that Fock spaces form a cotriple on the category of Banach algebras and verifying the so-called exponential law, *viz.* $!(A \times B) \cong\ !A \otimes\ !B$. We check the former by displaying a suitable adjunction in the next subsection.

Proposition 8 *Let A and B be Banach spaces.*

$$\mathcal{F}(A \times B) \cong \mathcal{F}(A) \otimes \mathcal{F}(B).$$

Here the product is what is called the direct sum by analysts. The isomorphism is in the category $\mathcal{BANACH}$.

Proof – We need to exhibit maps in both directions and show that all the conditions required for an isomorphism are satisfied. The isomorphism is based on the following "formal calculation".

$$\begin{aligned}
\mathcal{F}(A\times B) &= \mathcal{F}(A\oplus B)\\
&= \mathbf{C}\oplus(A\oplus B)\oplus\tfrac{1}{2}((A\oplus B)\otimes_s(A\oplus B))\cdots\\
&= \mathbf{C}\oplus A\oplus B\oplus\tfrac{1}{2}(A\otimes_s A)\oplus\tfrac{1}{2}(B\otimes_s B)\oplus(A\otimes_s B)\cdots\\
&= \mathcal{F}(A)\otimes\mathcal{F}(B).
\end{aligned}$$

The rigorous argument is as follows. We call an element of $\mathcal{F}(B)$ a *pure* tensor if it is of the form $\langle 0,0,\ldots,v,0,0,\ldots\rangle$ and a *finite-rank* tensor if it is of the form $\langle v_0,v_1,\ldots,v_n,0,0,\ldots\rangle$; *i.e.* zero after some finite stage. Now the pure tensors form a basis for $\mathcal{F}(B)$. In order to define the iso from $\mathcal{F}(A\times B)$ to $\mathcal{F}(A)\otimes\mathcal{F}(B)$ we need only specify the map on the pure tensors. A pure tensor, p, in $\mathcal{F}(A\times B)$ looks like $p=\Sigma x_1\otimes\ldots\otimes x_n$ where $x_i=y_i+z_i, y_i\in A, z_i\in B$. Using distributivity of $\otimes$ over $+$ we have

$$p=\Sigma[(y_1+z_1)\otimes\ldots\otimes(y_n+z_n)]$$

$$=\Sigma[y_1\otimes\ldots\otimes y_n+\ldots+(y_{i_1}\otimes\ldots\otimes y_{i_k})\otimes(z_{j_1}\otimes\ldots\otimes z_{j_l})+\ldots+z_1\otimes\ldots\otimes z_n]$$

The last expression is a sum of elements of $\mathcal{F}(A)\otimes\mathcal{F}(B)$. The iso in the other direction is obtained by viewing the pure elements of $\mathcal{F}(A)$ and $\mathcal{F}(B)$ as polynomials and carrying out polynomial multiplication. ∎

The units are easily identified.

Lemma 9 *The complex numbers,* $\mathbf{C}$, *viewed as a Banach space form a unit for tensor product. The one point space, written* $\mathbf{0}$, *is the unit for the direct sum.*

The effect of $\mathcal{F}$ on the units is given below. The proofs are immediate from the definitions. Equality means isomorphism in $\mathcal{BANACH}$.

Lemma 10 *1.* $\mathcal{F}(\mathbf{0})=\mathbf{C}$.

2. $\mathcal{F}(\mathbf{C})=l_1$.

Proof – The proof of the first assertion is immediate. For the second assertion, note that, since $\mathbf{C}$ is the unit for tensor all the terms in the infinite direct sum are just **C**. Thus we have infinite sequences of members of **C** with the same convergence criterion as for l_1. ▮

This lemma shows that one cannot use this construction in categories of finite-dimensional spaces.

Now we consider the antisymmetrized Fock space[3]. It turns out that one gets a model of the exponential types in the category of finite-dimensional vector spaces using the antisymmetrized Fock space.

Proposition 11 *If V is a finite-dimensional vector space of dimension n, then $\mathcal{F}_A(V)$ is also a finite-dimensional vector space with dimension 2^n.*

Proof – Consider the vector space $\bigotimes_A^p V$ with $p > n$. We claim that this space is the zero vector space. Since $\otimes$ is adjoint to internal hom in $\mathcal{VEC}_{fd}$, the space $\bigotimes_A^p V$ is isomorphic to the space of completely antisymmetric p-linear maps from V to the scalars. Let f denote such a map. Since V is only n-dimensional one cannot have p linearly independent arguments to such maps. Thus one of the arguments must be a linear combination of the others. Thus on any arguments f becomes a combination of terms of the form $f(\ldots, u, \ldots, u, \ldots)$ where two arguments must be equal. But antisymmetry makes such a term zero. Thus f is the zero vector and the vector space $\bigotimes_A^p V$ is the one-point space. Thus the infinite direct sum becomes a finite direct sum. Now consider $p \leq n$. It is clear that one can only choose C_p^n sets of p linearly independent vectors given a basis. Thus the dimensionality of the space $\bigotimes_A^p V$ is C_2^n and hence, adding the dimensions to get the dimension of the direct sum, we conclude that the dimension of $\mathcal{F}_A(V)$ is 2^n. ▮

The exponential law for the antisymmetric case can be argued similarly. The detailed verification can be found in [BSZ92] in Section 3.2 on exponential laws.

4.2 Categories of algebras

In this section we shall review some basic facts about categories of algebras, and see in particular how these fit into the current context. (See [M71] for a review of the basic categorical facts, and [L65] for the basic algebra, for instance.) For reference, we do give the following definition here.

[3]The arguments below are well-known to differential geometers. Prakash Panangaden would like to thank Steve Vickers for reminding him about these facts.

Definition 14 *A triple consists of a functor $F: \mathcal{B} \longrightarrow \mathcal{B}$, together with natural transformations $\eta: id \longrightarrow F$ and $\mu: FF \longrightarrow F$, such that $\mu \circ \eta F = \mu \circ F\eta = id$ and $\mu \circ \mu T = \mu \circ T\mu$.*

One simple point to recall is that categories of algebras and of coalgebras are closely connected to the existence of triples and cotriples. Given a triple $F: \mathcal{B} \longrightarrow \mathcal{B}$, (with structure morphisms η, μ), an F-algebra is an object B and a morphism $h: F(B) \longrightarrow B$ (subject to two commutativity conditions, corresponding to the associative and unit laws). (This notion can be generalized to arbitrary functors.) There is a canonical category of such algebras, the Eilenberg-Moore category $\mathcal{C}^F$, and an adjunction $\mathcal{C} \rightleftarrows \mathcal{C}^F$. Any adjunction canonically induces a triple, and this one canonically induces the original triple. The category of free F-algebras is the Kleisli category $\mathcal{C}_F$ of the triple; again, there is a canonical adjunction $\mathcal{C} \rightleftarrows \mathcal{C}_F$ which induces the original triple. Of course this dualizes for cotriples, with the corresponding notion of coalgebras. (We shall avoid the unpleasant use of terms like "coEilenberg-Moore" and "coKleisli".)

Usually mathematicians have been more interested in the Eilenberg-Moore category of a triple (or cotriple) than in the Kleisli category; although there has been some interest in Kleisli categories recently (for instance in the context of linear logic, as mentioned earlier in this paper), we shall follow this tradition and shall work in Eilenberg-Moore categories. Indeed, it is there that we shall find some of our models. One reason for this is quite practical: it is often simpler to recognize the category of algebras and so derive the triple (similarly, once one has a candidate for a triple, it is often simpler to construct the category of algebras and verify the adjunction than to directly show the original functor is a triple). But there is another reason: we want to show that the Fock space functor is a cotriple (so as to model $!$), but on the categories of spaces we consider, this is not the case—rather it is a triple. By passing to the algebras, we can fix this, because of the following fact:

Fact Given an adjunction $\mathcal{C} \underset{U}{\overset{F}{\rightleftarrows}} \mathcal{D}$, $F \dashv U$, the composite UF is a triple on $\mathcal{C}$, and so (dually) the composite FU is a cotriple on $\mathcal{D}$.

So we obtain our model of $!$ on the category of algebras.

4.2.1 Algebras for the symmetric (bosonic) Fock space construction

We begin with a more traditional notion of algebra; the connection between these comes *via* the triple induced by the adjunction given by the free algebra construction, as outlined above. In other words, the category of (traditional) algebras is equivalent to the category of UF algebras.

Definition 15 *An algebra* A *is a space* A *equipped with morphisms*

$$m\colon A \otimes A \longrightarrow A \text{ and } i\colon \mathbf{C} \longrightarrow A$$

satisfying

$$\begin{tikzcd}
A \otimes A \otimes A \arrow[r, "m \otimes id"] \arrow[d, "id \otimes m"'] & A \otimes A \arrow[d, "m"] \\
A \otimes A \arrow[r, "m"] & A
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{C} \otimes A \arrow[r, "i \otimes id"] \arrow[dr, "\cong"'] & A \otimes A \arrow[d, "m"] & A \otimes \mathbf{C} \arrow[l, "id \otimes i"'] \arrow[dl, "\cong"] \\
& A &
\end{tikzcd}$$

Here we are supposing the base field to be $\mathbf{C}$; otherwise replace $\mathbf{C}$ with the base field k. If in addition the following diagram commutes, then the algebra A is said to be *symmetric* or commutative. (τ is the canonical "twist" morphism.)

$$\begin{tikzcd}
A \otimes A \arrow[r, "\tau"] \arrow[dr, "m"'] & A \otimes A \arrow[d, "m"] \\
& A
\end{tikzcd}$$

An example of such an algebra comes from the Fock space of a Banach space: the multiplication m is defined by "multiplication of series" in an evident manner. The use of the symmetrized tensor in the definition of Fock space guarantees that this will indeed be a symmetric algebra, and it is standard that this description gives the free such algebra. In other words, we have the following proposition.

Proposition 12 *Given a Banach space B, the Fock space $\mathcal{F}(B)$ canonically carries an algebra structure, and indeed is the free symmetric algebra generated by B.*

It follows from this (or rather from the adjunction $\mathcal{BANCON} \rightleftarrows \mathcal{SALG}$) that we have a cotriple on the category $\mathcal{SALG}$ of symmetric algebras, given by taking the Fock algebra on the underlying space of an algebra. As the details of this are both standard and similar to the case of the antisymmetric Fock space construction, which we shall discuss in more detail next, we shall leave the details here to the reader.

4.3 Algebras for the antisymmetric (fermionic) Fock space construction

Recall that we work in the context of $\mathcal{VEC}_{fd}$ finite dimensional vector spaces when considering the antisymmetric Fock construction. This category is self-dual, and is compact with biproducts: the product and coproduct coincide. This duality also implies that a triple is also a cotriple, so we can model ! in the category of spaces. However, to show that the Fock space construction defines a triple (or cotriple), it is again simpler to consider the category of algebras. Although we are not familiar with any previous consideration of this category of algebras as such, the context is familiar: the antisymmetric Fock space construction is usually called (when thought of as an algebra) the Grassman algebra, or the "alternating" or "interior" algebra; the multiplication defined on it is called the "wedge product" (a term derived from the usual notation for this product).

Definition 16 *An alternating algebra* A *is a graded algebra* A *(with unit) whose multiplication map satisfies the property that, if x, y are of degree m, n respectively, then $xy = (-1)^{nm}yx$ (which by the grading must be of degree $n + m$).*

Note that the unit must be of degree 0. Morphisms of alternating algebras are just homomorphisms as algebras.

Proposition 13 *There is a canonical alternating algebra structure on $\mathcal{F}_A(V)$, for any finite dimensional vector space V. The antisymmetric Fock construction is left adjoint to the forgetful functor $U: \mathcal{VEC}_{fd} \overset{\mathcal{F}_A}{\underset{U}{\rightleftarrows}} \mathcal{AALG}$, where $\mathcal{AALG}$ is the category of alternating algebras. As a consequence, $\mathcal{F}_A$ defines a triple (and so cotriple) on $\mathcal{VEC}_{fd}$.*

Proof– (Sketch) The multiplication on $\mathcal{F}_A(V)$ is the standard "wedge" product [L65], which to elements $x_1 \otimes_A \ldots \otimes_A x_n, y_1 \otimes_A \ldots \otimes_A y_m$ gives the product $x_1 \otimes_A \cdots \otimes_A x_n \otimes_A y_1 \otimes_A \cdots \otimes_A y_m$. Here $x \otimes_A y$ means the equivalence class of $x \otimes y$ in $A \otimes_A A$. (Essentially this is the same "multiplication of power series" we had in the symmetric case, with the alternating product used in place of the usual tensor.) For a vector space V, define $\eta: V \longrightarrow U\mathcal{F}_A(V)$ as the canonical injection. Given an alternating algebra A, define $\epsilon: \mathcal{F}_A(UA) \longrightarrow A$ by "adding the terms of the series": $\langle x_0, x_1, x_2^1 \otimes_A x_2^2, \ldots \rangle \mapsto i(x_0) + x_1 + m(x_2^1, x_2^2) + \cdots$, where i, m are the algebra maps.

To verify that we have an adjunction we must show the following commute:

$$\begin{tikzcd}
\mathcal{F}_A(V) \arrow[r, "\mathcal{F}_A(\eta)"] \arrow[dr, "1"'] & \mathcal{F}_A(U\mathcal{F}_A(V)) \arrow[d, "\epsilon\mathcal{F}_A"] \\
& \mathcal{F}_A(V)
\end{tikzcd}$$

$$\begin{tikzcd}
UA \arrow[r, "\eta U"] \arrow[dr, "1"'] & U\mathcal{F}_A(UA) \arrow[d, "U\epsilon"] \\
& UA
\end{tikzcd}$$

The second diagram is obvious; to verify the first, notice that $\mathcal{F}_A(\eta(x))$ maps

$$\begin{array}{lll}\langle x_0, x_1, x_2^1 \otimes_A x_2^2, \ldots\rangle & \mapsto & \langle x_0, \\ & & \langle 0, x_1, 0, \ldots\rangle, \\ & & \langle 0,0,x_2^1,0,\ldots\rangle \otimes_A \langle 0,0,x_2^2,0,\ldots\rangle, \\ & & \vdots \\ & & \rangle \end{array}$$

and it is clear that "adding up this series" just returns the original term. ∎

It now follows that we can model ! in $\mathcal{VEC}_{fd}$ with $\mathcal{F}_A$, *via* the formula $!\,V = (\mathcal{F}_A(V^\perp))^\perp$.

5 The Holomorphic-Function Representation of Fock Space

A possible reaction to the results of the last section is that the Fock space construction works purely fortuitously, in the sense that the proper notions of tensor products and infinite direct-sums happen to exist and conspire to make the construction of internal comonoids possible. In the present section we argue that in fact this construction is linked to much deeper mathematics. The symmetrized Fock space on a Banach space B, turns out to be a space of holomorphic functions (analytic functions) on B, properly defined. This hints at possible deeper connections between analyticity and computability which need to be explored.

The ideas here stem from early work by Bargmann [Ba61] on Hilbert spaces of analytic functions in quantum mechanics. This was extended by Segal [S62, BSZ92] to quantum field theory and Segal's extension was used by Ashtekar and Magnon [AM-A80] to develop quantum field theory in curved spacetimes. (A brief summary of the ideas is contained in an appendix to [P80] and in [P79].) The latter work involved making sense of the familiar Cauchy-Riemann conditions on infinite-dimensional spaces.

We quickly recapitulate the basic notion of analytic function in terms of one complex variable before presenting the infinite-dimensional case. A very good elementary reference is *Complex Analysis* by Ahlfors [Ah66]. Given the complex plane, $\mathbf{C}$, one can define functions from $\mathbf{C}$ to $\mathbf{C}$. Let z be a complex variable; we can think of it as $x + iy$ and thus one can think of functions from $\mathbf{C}$ to $\mathbf{C}$ as functions from $\mathbf{R}^2$ to $\mathbf{R}^2$. An *analytic* or ***holomorphic*** function is one that is everywhere differentiable. In the notion of differentiation, the limit being computed, *viz.*

$$\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$$

allows h to be an arbitrary complex number and hence this limit is required to exist no matter in what direction h approaches 0. This much more stringent requirement makes complex differentiability much stronger than the usual notion of differentiability. If a complex function is differentiable at a point it can be represented by a convergent power series in a suitable open region about the point. If one uses the fact that h can approach zero along either axis one can derive the *Cauchy-Riemann* equations for a complex valued function $f = u(x,y) + iv(x,y)$ of the complex variable $z = x + iy$,

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y}, \quad \frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}.$$

What is remarkable about complex functions is that this definition of analyticity yields the result that a complex-analytic function can be expressed by a convergent power-series in a region of the complex plane. This is remarkable because only one derivative is involved in the Cauchy-Riemann equations whereas the statement that a power-series representation exists is stronger, for real-valued functions, even than requiring infinite differentiability. In real analysis one has examples of functions that are infinitely differentiable at a point, but do not have a power series representation in any neighbourhood of that point. A function may have a power series representation that is valid everywhere, a so-called *entire* holomorphic function; the complex exponential function is an example.

There is a formal perspective, due to Wierstrass, that is rather more illuminating. Think of a complex variable $z = x + iy$ and its conjugate $\overline{z} = x - iy$ as being, formally, independent variables. A function could depend on z and on its complex conjugate, $\overline{z}$, for example, the function that maps each z to $z\overline{z} + iz\overline{z}$. An *analytic* or *holomorphic* function is one which has no dependance on $\overline{z}$. This is expressed formally by $df/d\overline{z} = 0$. When expressed in terms of the real and imaginary parts of f and z, this equation becomes the familiar Cauchy-Riemann equations. Thus this reinforces the view that a holomorphic function is properly thought of as a single complex-valued function of a single variable rather than as two real-valued functions of two real variables.

The theory of functions of finitely many complex variables is a nontrivial extension of the theory of functions of a single complex variable.

Entirely new phenomena occur, which have no analogues in the theory of a single complex variable. An excellent recent text is the three volume treatise by Gunning [Gu90]. For our purposes we need only the barest beginnings of the theory. Given $\mathbf{C}^n$, we can have functions from $\mathbf{C}^n$ to $\mathbf{C}$. One can introduce complex coordinates on $\mathbf{C}^n$, $z_1, \ldots, z_n$. One can define a holomorphic function here as one having a convergent power-series expansion in $z_1, \ldots, z_n$. The key lemma that allows one to mimic some of the results of the one-dimensional case is Osgood's lemma[4].

Lemma 14 *If a complex-valued function is continuous in an open subset D of $\mathbf{C}^n$ and is holomorphic in each variable seperately, then it is holomorphic in D.*

From this one can conclude that a holomorphic function in n variables satisfies the Cauchy-Riemann equations $\frac{\partial f}{\partial \bar{z}_i} = 0$. One is free to take either one of (a) satisfying Cauchy-Riemann equations or (b) having convergent power-series representations as the definition of holomorphicity.

Now we describe how to define holomorphic functions on infinite-dimensional, complex, Banach spaces. The basic intuition may be summarized thus. One starts with subspaces of finite *codimension.* Thus the quotient spaces are isomorphic to some $\mathbf{C}^n$. One can define what is meant by a holomorphic function on these quotient spaces as in the preceding paragraph. By composing a holomorphic function with the canonical surjection from the original Banach space to the quotient space we get a function on the original Banach space. These functions can all be taken to be holomorphic.

$$\begin{tikzcd} & B \arrow[d] \arrow[dr, dotted] & \\ B/{\sim} \; = & \mathbf{C}^n \arrow[r] & \mathbf{C} \end{tikzcd}$$

Intuitively these are the functions that are constant along all but finitely many directions, and holomorphic in the directions along which they do vary. These functions are called *cylindric holomorphic functions.* Because the sequence of coefficients of a power-series is absolutely convergent, we can define an l_1 norm on these functions in terms of the power-series. Finally the collection of all holomorphic funcitons is defined by taking the l_1-norm completion of the cylindric holomorphic functions.

[4]There is a considerably harder theorem, called Hartog's theorem, which drops the requirement of continuity.

Given a Banach space B, let U be a subspace with finite codimension n, *i.e.* the quotient space B/U is an n complex-dimensional vector space. The space B/U is isomorphic to $\mathbf{C}^n$. Let $\phi : B/U \rightarrow \mathbf{C}^n$ be an isomorphism; such a map defines a choice of complex coordinates on B/U. Let π_U be the canonical surjection from B to B/U.

Definition 1 *A* **cylindric holomorphic** *function on B is a function of the form $f \circ \phi \circ \pi_U$, where U, π_U and ϕ are as above and f is a holomorphic function from $\mathbf{C}^n$ to $\mathbf{C}$.*

We need to argue that the choice of coordinates does not make a real difference. Of course which functions get called holomorphic does depend on the choice of coordinates, but the space of holomorphic functions has the same structure[5]. Suppose that U and V are both subspaces of B and that U is included in V. Suppose that both these spaces are spaces of finite codimension, say n and m respectively. Clearly $n \geq m$. Now we have a linear map $\pi_{UV} : B/U \rightarrow B/V$ given by $x + U \mapsto x + V$; clearly this is a surjection. Now given coordinate functions $\phi : B/U \rightarrow \mathbf{C}^n$ and $\psi : B/V \rightarrow \mathbf{C}^m$ we can define a function $\alpha : \mathbf{C}^n \rightarrow \mathbf{C}^m$, given by $\psi \circ \pi_{UV} \circ \phi^{-1}$, which makes the diagram commute. Thus we do not have to impose "coherence" conditions on the choice of coordinates, we can always translate back and forth between different coordinate systems.

We will suppress these translation functions in what follows and assume that the coordinates have been serendipitously chosen to make the form of the functions simple. In other words, we can fix a family of subspaces $\{W_n | n \in N\}$ with W_n having codimension n and $W_{n+1} \subset W_n$. The coordinates can be chosen so that the space B/W_n has coordinates $z_1, \ldots, z_n$.

Suppose that f is a cylindric holomorphic function on B. This means that there is a finite-codimensional subspace W, and a holomorphic function f_W, from W to $\mathbf{C}$, such that $f = f_W \circ \pi_W$. The function f_W regarded as a function of n complex variables has a power-series representation

$$f_W(z_1, \ldots, z_n) = \Sigma a_{i_1 \ldots i_k} z_1^{i_1} \ldots z_k^{i_k}$$

and furthermore we have the following convergence condition

$$\Sigma |a_{i_1 \ldots i_k}| < \infty.$$

[5]This happens even in the one dimensional case. The function $\overline{z}$ is considered anti-holomorphic traditionally, but one could have called it holomorphic by interchanging the role of z and $\overline{z}$.

Thus with each such cylindric holomorphic function we can define the sum of the absolute values of the coefficients in the power-series expansion as the norm of the function. Viewing the sequences of coefficients as the elements of a complex vector space, we have an l_1 norm. We write $\| f \|$ for this norm of a cylindric holomorphic function.

Definition 2 *An* l_1**-holomorphic** *function on B is the limit of a sequence of cylindric holomorphic function in the above norm.*

The l_1 emphasizes that the holomorphic functions are obtained by a particular norm completion. In the corresponding theory of holomorphic functions on Hilbert spaces, one uses the inner-product to define polynomials and then perform a completion in the L_2 norm. A key difference is that our norm is defined on the sequence of coefficients whereas in the Hilbert space case, one uses the L_2 norm which is defined in terms of integration.

In the resulting Banach space there are several formal entities that were adjoined as part of the norm-completion process. We need to discuss in what sense these formally-defined entities can be regarded as bona-fide functions. Let $W_1, \ldots, W_r, \ldots$ be an infinite sequence of subspaces of B, each embedded in the previous. Assume, in addition, that all these spaces have finite codimension. Now assume that there is a sequence of cylindric holomorphic functions, f_n, on B obtained from a holomorphic function, $f^{(n)}$ on each of the quotient spaces B/W_i. Finally, assume that the sequence $\| f_n \|$ of (real) numbers is convergent. Such a sequence of cylindric holomorphic functions defines a holomorphic function on B. We call this function f. We need to exhibit f as a map from B to $\mathbf{C}$. Accordingly, let x be a point of B. For each of the functions f_n we have $|f_n(x)| \leq \| f_n \|$. Since the sequence of norms converges we have the sequence $f_n(x)$ converges absolutely and hence converges. Thus the function f *qua* function is given at each x of B by $\lim_{n\to\infty} f_n(x)$. However, in order to use the word "function" we need to show that the power-series has a domain of convergence. Unfortunately, it may not have a non-trivial domain of convergence but, in a sense to be made precise, it comes close to having a non-trivial domain of convergence.

The power-series representation of the function f is given as follows. It depends, in general, on infinitely many variables but each term in the power series will be a monomial in finitely many variables. Consider the coefficient of $z_{i_1}^{j_1} \ldots z_{i_k}^{j_k}$ in the expansion of f. In all but finitely many of the f_n all the indicated variables will appear in their power-series expansions. Consider the coefficients of this term in each power series; this forms a sequence of complex numbers α_n where α_n is 0 if there is no such term in the expansion of f_n. Since $|\alpha_n| \leq \| f_n \|$ the sequence α_n converges absolutely and hence converges to, say, α. This is the coefficient of $z_{i_1}^{j_1} \ldots z_{i_k}^{j_k}$ in the power-series expansion of f.

Consider the coordinates $z_1, \ldots, z_n$. This defines an n-dimensional subspace of the Banach space, which we call U_n. Now consider the power-series for f. It defines a family of holomorphic functions f^n where f^n is defined on the subspace U_n and is obtained by retaining only those terms in the power-series expansion of f which involve variables among $z_1, \ldots, z_n$. These are analytic functions on the U_n and, as such, have non-trivial domains of convergence. However, as n increases the radii of convergence could tend to 0. So we have the slightly weaker statement than the usual finite-dimensional notion; instead of having a non-zero radius of convergence in the Banach space we have a non-zero radius of convergence on every finite-dimensional subspace. If one uses *entire* functions, rather than analytic functions, at the starting point of the construction, then one can show that the resulting functions are entire; see page 67, theorem 1.13, of the book by Baez, Segal and Zhou [BSZ92]. Unfortunately when using the representation of elements of Fock space one may carry out simple operations that do not produce entire functions so we cannot just choose to work with entire functions. Nevertheless, many common functions, most notably the exponential, are entire.

Given a bona fide holomorphic function one can express it as a power series. The coefficients are calculated in the usual way, *viz.* by using Taylor's theorem

$$f = \Sigma_n \Sigma_{i_1 + \ldots + i_k = n} \frac{1}{i_1! \ldots i_k!} \left(\frac{\partial^n}{\partial^{i_1} z_1 \ldots \partial^{i_k} z_k} \right) z_1^{i_1} \ldots z_k^{i_k}$$

Since the mixed partial derivatives commute (the functions are holomorphic and hence certainly differentiable enough) the partial derivatives are, concretely speaking, symmetric arrays. Abstractly speaking this just means that they are elements of the symmetrized tensor product.

We can write this as follows.

Theorem 15 *A holomorphic function can be represented by its power-series expansion where the n^{th} term in the power-series expansion is a symmetrized n^{th} derivative:*

$$f = \Sigma(1/k!) D^{(k)} f$$

where the notation $D^{(k)} f$ means symmetrized k^{th} derivative of f.

The symmetrized derivatives live in the symmetrized tensor products of B with itself. One thus has a correspondence with the standard Fock representation and the notion of holomorphic function since in each case one has a string of symmetrized vectors.

5.1 A Digression on Complex Structures

This section can be skipped on a first reading[6]; however, the reader who feels queasy about all the explicit coordinate dependence in the definitions so far may find this section comforting. There is no need to start with complex vector spaces. One could have used real vector spaces from the outset. In order to sketch this briefly we begin with the notion of a complex structure on a vector space.

Definition 17 *Let V be a real vector space. A complex structure is a linear operator $J : V \to V$ such that $J^2 = -I$.*

An example of a complex structure on $\mathbf{R}^2$ is

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

It is immediate that V must be even-dimensional or infinite-dimensional if a complex structure exists on it. A given vector space may have several different complex structures defined on it.

One can go back and forth between real vector spaces equipped with complex structures and complex vector spaces in the following way. Suppose that (V, J) is a real vector space equipped with a complex structure. Now we can formally define the "complexification" of V as a vector space $V_{\mathbf{C}} = V' \oplus V''$ where V' and V'' are copies of V and multiplication by complex numbers is given by $(x + iy) * \langle a, b\rangle = \langle xa - by, xb + ay\rangle$. Now we can define a linear operator P on $V_{\mathbf{C}}$ by the formula $P\langle a, b\rangle = (1/2)\langle a + Jb, b - Ja\rangle$. It is easy to verify that P defines a projection operator on $V_{\mathbf{C}}$. It defines a subspace of $V_{\mathbf{C}}$, which is, as a real vector space, isomorphic to V. Similarly, given a complex vector space W we can construct a real vector space which is isomorphic to W, as a real vector space, and equip it with a complex structure which will give us back W when we apply the construction above to it. We first form the direct sum $W \oplus W$. Now we define a complex structure on this space by $J\langle a, b\rangle = \langle ia, -ib\rangle$. It is easy to check the claims made. The upshot is that one can talk about

[6]and every subsequent reading as well.

real vector spaces equipped with complex structures or about complex vector spaces interchangably.

The final piece of mathematics that we need is the *Lie derivative* from classical differential geometry. Developing the definitions from scratch would involve a long digression. Fortunately the ideas are simple so we will give an intuitive account. In what follows the word "smooth" is meant to signify infinitely differentiable. Consider a smooth manifold; a curved surface is an excellent model to keep in mind. Suppose that one has a smooth vector field on this manifold; that is to say a smooth assignment of a vector at every point on the manifold. Classical results from differential equations say that there is a family of nonintersecting curves that fill the manifold and such that the curves are everywhere tangent to the given vector field. Now these curves are all parametrized by a real parameter say t. If we fix a value for t, we can define a smooth bijective map ψ_t of the manifold to itself (a so-called "diffeomorphism") which is defined by moving each point t units along the unique curve passing through it. We can make the map ψ_t act on functions defined on V as follows: $\psi_t^*(f) = f \circ \psi_t$, for f a complex-valued function defined on V. We can now define the *Lie derivative of f along the vector field u at the point p* as the limit

$$\pounds_v f = \lim_{t \to 0} \frac{\psi_t^*(f)(p) - f(p)}{t}$$

This gives another function from V to the complex numbers. Intuitively we imagine that the given vector field, u, defines a flowing fluid. The vector at each point defines the velocity of the fluid locally and the stream-lines of the fluid give the family of curves mentioned above. The Lie derivative measures changes that an observer flowing with the fluid would see.

For us the Lie derivative tells us how to define changes seen "when travelling along the direction defined by a vector field". Now recall what is meant by an analytic function in ordinary complex analysis. A complex-valued function of two real variables, x and y, is analytic if it depends only on the complex variable $z = x + iy$ and not on the conjugate variable $\overline{z} = x - iy$. The Cauchy-Riemann equations say this precisely. The Lie derivative is what we need in order to do this in the infinite-dimensional case.

Definition 18 *Let B be a Banach space over the complex numbers. Now let J be a complex structure on this space. We call a vector v* **holomorphic** *if $Jv = iv$ and* **anti-holomorphic** *if $Jv = -iv$. If we have a* *real vector space V, equipped with a complex structure, we can define a holomorphic or an anti-holomorphic vector in the same way.*

A holomorphic vector plays roughly the role of a complex variable while an anti-holomorphic vector plays the role of a complex-conjugate variable. Now we can state the infinite-dimensional analogue of the Cauchy-Riemann conditions.

Definition 19 *A function $f : V \to \mathbf{C}$ is* **holomorphic** *if (i) it is differentiable and (ii) for every anti-holomorphic vector field v we have $\mathcal{L}_v f = 0$. An equivalent condition is $\mathcal{L}_{Jv} f = i\mathcal{L}_v f$ for holomorphic vector fields v.*

It is easy to check that the latter form of the second condition gives the usual Cauchy-Riemann equations in the one-dimensional case by choosing the vector fields appropriately.

6 The Physical Origin of Fock Space

The Fock space constructions described in the previous sections were independently invented by physicists and mathematicians. The symmetric Fock space (called the bosonic Fock space by physicists) is well known to mathematicians as the symmetric tensor algebra whereas the antisymmetric Fock space (fermionic Fock space) was invented by Grassman, at least in the finite-dimensional case, under the name of exterior algebra or alternating algebra. In this section we describe the role of Fock space in quantum field theory. In order to prevent intolerable regress in definitions we assume that the reader has an at least intuitive grasp of differential equations, the definition of a smooth manifold and associated concepts like that of a smooth vector field[7]

We begin with a brief discussion of quantum mechanics and classical mechanics. In classical mechanics one has systems which vary in time. The role of theory is to describe the temporal evolution of systems. Such temporal evolution is governed by a differential equation. The fact that one uses differential equations says something fundamental about the local nature of the dynamics of physical systems, at least according to conventional classical mechanics. In dealing with differential equations one has to distinguish between quantities that are determined and quantities that may be freely specified: the so called "initial conditions". Experiment tells one that systems are described by second-order differential

[7]Remarks requiring a more sophisticated vocabulary will appear as footnotes.

equations and hence that the functions being described and their first derivatives, at a given point of time, are part of the initial conditions. The space of all possible initial conditions is called the space of possible states or "phase" space, and is the kinematical arena on which dynamical evolution occurs[8]. A fundamental mathematical assumption is that the phase space is a $2n$ dimensional smooth manifold[9]. The points of phase space are called states. If the system is a collection of, say 7, particles, the states will correspond to the 42 numbers required to specify the positions and the velocities of each of the particles in three-dimensional space.

Through each point in phase space is a vector giving rise to a smooth vector field called the Hamiltonian vector field. One can draw a family of curves such that at every point there is exactly one curve passing through that point and the Hamiltonian is tangent to the curve at that point. Roughly speaking, the vector field defines a differential equation and the curves represent the family of solutions where each point represents a possible specification of initial conditions. An *observable* is a physical quantity that is determined by the state. As such it corresponds to a real-valued function on phase space. A typical example is the total energy of a system. Most of experimental mechanics is aimed at determining the Hamiltonian. In the formal development of analytical mechanics there is a special *antisymmetric 2-form* called the *symplectic form* which plays a fundamental mathematical role but is hard to describe in an intuitive or purely physical way.

In quantum mechanics, the above picture changes in the following fundamental ways. The observables become the fundamental physical entities. These are defined to form a particular subalgebra of an algebraic structure called a $\mathbf{C}^*$-algebra. The key point is that this algebra is not commutative, unlike the algebra of smooth functions on a manifold. Furthermore, the failure of commutativity is directly linked to the symplectic form; this was Dirac's contribution to the theory of quantum mechanics. Thus, structures available at the classical level provide guidance as to what the "correct" $\mathbf{C}^*$-algebra should be.

There is a representation of this algebra as the algebra of operators on a Hilbert space. The space of states acquires the structure of a Hilbert space and becomes the carrier of the representation of the $\mathbf{C}^*$-algebra. One presentation of this abstract Hilbert space is as the space of square-

[8]Sometimes one has a more complicated situation in which the phase space is constrained in such a way that it cannot be simply defined as a manifold. These are called non-holonomic constraints and correspond to such familiar situations as skating and rolling.

[9]Actually it has the structure of the cotangent bundle of a smooth n-manifold.

integrable complex-valued functions on a suitable underlying space; for example the space of possible configurations of a system. The space of states has acquired linear structure; this means that one can add states reflecting the intuition that in quantum mechanics a system can be in the superposition of two (or more) states. The inner product measures the extent to which two states resemble each other. Finally the fact that one has complex functions is strongly suggested by the observation of interference phenomena in nature.

An observable is a self-adjoint operator. The link between the mathematics and experiment is the following. If one attempts to measure the observable O for a system in state ψ one will obtain an eigenvalue of O. Self-adjoint operators have real eigenvalues so we will get a real-valued result. If ψ is an eigenvector with eigenvalue α, then, with no indeterminacy or uncertainty, one will obtain the value α. If ψ is not an eigenvector, one can express ψ as a linear combination of eigenvectors in the form $\psi = \Sigma a_i \psi_i$ where the ψ_i are assumed to be eigenvectors with eigenvalues α_i. The result of measuring O will be α_i with probability $|a_i|^2$. It is important to keep in mind that the absolute squares of the a_i correspond to probabilities but it is the a_i themselves that enter into the linear combinations of states. This interplay between the complex coefficients and the interpretation of their squares as probabilities is what distinguishes the probabilistic aspects of quantum mechanics from statistical mechanics which also has a probabilistic aspect but where one directly manipulates probabilities.

The dynamics of systems is described by a *first-order* differential equation called Schroedinger's equation. Thus, the evolution of states in quantum mechanics is determinate, just as in classical mechanics. The indeterminacy usually associated with quantum mechanics appears in the fact that the state of a system may not be an eigenstate of the observable being measured so the outcome of the measurement may be indeterminate.

Quantum mechanics is designed to handle systems in which the number of interacting entities (usually called "particles") is fixed. On the other hand, experiment tells us that at sufficiently high energies particles may be created or destroyed. Quantum field theory was invented to account for such processes. The original formulations of this theory due to Dirac, Heisenberg, Fock, Jordan, Pauli, Wigner and many others was quite heuristic. Now a reasonably rigourous theory is available; see the book by Baez, Segal and Zhou [BSZ92] for a recent exposition of quantum field theory.

The first need in a many-particle theory is a space of states which can describe variable numbers of particles; this is what Fock space is [Ge85]. The second ingredient is the availability of operators that can describe the creation and annihilation of particles. Of course, there is much more that needs to be said in order to see how all this formalism translates into calculations of realistic physical processes but that would require a very thick book which, in any case, has been written many times over.

Given a Hilbert space H in quantum mechanics representing the states of a single particle one can construct a many-particle Hilbert space as $\mathcal{F}(H)$. Suppose that $\psi, \phi \in H$; one interprets the element $\psi \otimes_s \phi$ of $H \otimes_s H$ as a two-particle state with one particle in the state ψ and the other in the state ϕ. Similarly for the other summands of $\mathcal{F}(H)$. The reason for the symmetrization is that one is dealing with indistinguishable particles so that the n-particle states have to carry representations of the permutation group. Thus one could have particle states that were symmetric or antisymmetric under interchange leading to the bosonic or fermionic Fock spaces respectively. It is a remarkable fact that both types of particles are observed in nature. Notice that $\psi \wedge \psi$ is identically zero hence one cannot have many-particle states in the antisymmetric Fock space in which both particles are in the same one-particle state. This is observed in nature as the exclusion principle. Fock space is the space of states for quantum field theory and is constructed from the space of states for quantum mechanics.

The following interesting operators are defined on Fock space. Let $\psi = \langle \psi_0, \psi_1, \psi_2, \ldots, \psi_n, \ldots \rangle$ be an element of $\mathcal{F}(H)$. Now let σ be an element of H. We define the operator $C(\sigma)$ by

$$C(\sigma)\psi = \langle 0, \psi_0\sigma, \sqrt{2}\, \psi_1 \otimes_s \sigma, \ldots, \sqrt{n+1}\, \psi_n \otimes_s \sigma, \ldots \rangle$$

This operator creates a particle in the state σ. There is an analogous operator $A(\sigma)$ which destroys a particle in state σ. These two operators are adjoint to each other. The fundamental algebraic relation between them is $A(\sigma)C(\sigma) - C(\sigma)A(\sigma) = I$ where I is the identity operator. From these two we can define the operator $N(\sigma) = C(\sigma)A(\sigma)$. Let v_n be a state with n particles in the state σ and with no other particles. For the rest of the paragraph we drop explicit mention of σ. Now $A\, v_n = \sqrt{n}\ v_{n-1}$ and $C\, v_n = \sqrt{n+1}\ v_{n+1}$, hence we have $N\, v_n = n\, v_n$. Thus v_n is an eigenstate of N with eigenvalue n; for this reason N is called the number operator. Now we also have $NA\, v_n = (AC - I)A\, v_n = A(CA - I)v_n = A(N - I)v_n = (n-1)A\, v_n$. In other words, $A\, v_n$ is also an eigenstate of N with eigenvalue $(n-1)$. This justifies the name "annihilation" operator. A similar calculation can be done for the creation operator. If we are successful in developing a theory of reduction of proof nets in terms of operator algebras, in the sense of Girard's geometry of interaction, we will have the A and C operators available. We hope that these can be used to give a quantitative handle on resource consumption during computation.

The presentation of Fock space above emphasized the concept of many-particle states. Mathematically, however, $\mathcal{F}(H)$ is just a Hilbert space and can be presented differently. As we have shown in the last section, it can be presented as the space of holomorphic functions of a Hilbert space (the details are somewhat different from the Banach space case but the ideas are essentially the same). The space of holomorphic functions has as its inner product

$$\langle g, f\rangle = \frac{1}{2\pi i}\int f(z)g(\overline{z})e^{-z\overline{z}}dz\,d\overline{z}.$$

(See [IZ80] page 435, for example.) What do the creation and annihilation operators look like from this perspective? For simplicity, let us look at power series in a single variable z. This amounts to only looking at the many-particle states of the form σ tensored with itself. The creation operator is just $z*(.)$ while the annihilation operator is just $d(.)/dz$. One can easily check that $(AC - CA)f = d(z*f)/dz - z*df/dz = f$; in other words the basic algebraic relation holds. Furthermore one can ask what the eigenstates of A and C look like. Clearly the eigenstate of C is just the zero vector. The eigenstate of A is the state represented by the holomorphic function e^z. These states actually exist in nature and are called "coherent" states; they occur, for example, in lasers. The key point about coherent states is that they "look classical"; one can remove a particle without changing the state. As such they bear a superficial resemblance to the role of ! formulas in linear logic.

7 Conclusion

To summarize the results we have claimed in this paper, we have produced models of the following fragments of linear logic. First, in finite-dimensional vector spaces we have a complete model of classical linear logic, albeit with a compact category, so that the tensor and par are identified, as are ! and ?. In the category of symmetric Banach *algebras* we have a model of the $\otimes, \times, \oplus, !$ fragment. This category cannot be endowed with closed structure, since $Hom(X, Y) = Hom(\mathrm{I}\otimes X, Y) = Hom(\mathrm{I}, X \multimap Y)$; in this category the unit I for $\otimes$ is also the initial object so the last hom set would have to be a singleton, clearly not the case for arbitrary X, Y. The closely related $\mathcal{BANCON}$ has several very pleasant features as a model of the multiplicative and additive fragment of linear logic—it is a rare example of a category of linear spaces which is neither additive nor compact—but unfortunately it is not possible to extend this to a model of ! on the algebra category as we did with $\mathcal{BANACH}$ as the exponential isomorphism fails there. In $\mathcal{HILBERT}$ we get results analagous to those with $\mathcal{BANACH}$, modelling the $\otimes, \times, \oplus, !$ fragment in the category of algebras. In addition to producing these models, we have described a mathematical representation for ! using holomorphic functions which suggests that one might profitably think of computability in terms of analyticity rather than continuity. Furthermore the mathematical structures described in this paper arise from quantum field theory and are suggestive of links with that subject.

It is crucial that one appreciate the differences between our work and that of Girard in [G89]. He has also used Banach algebras but all proofs are represented in a single Banach algebra, whereas we model formulas as individual algebras, with proofs as algebra homomorphisms. That is to say, we work in the category of Banach algebras, rather than inside a particular algebra. His major achievement is modelling cut elimination in terms of operator algebras. We on the other hand model provability in the appropriate fragment of linear logic.

Our next goal is to model the proof theory of linear logic in the spirit of Geometry of Interaction. Rather than following Girard, we will be guided by the following intuitions which are suggested by the physical interpretation of Fock space. We think of formulas as representing *states*, that is to say elements of a Fock space; a proof represents the process of interaction between particles in the initial state resulting in the particles observed in the final state. Mathematically the process is described by a combination of creation and annihilation operators. Proof normalization transforms processes into "observably equivalent" processes. In particular, we hope that our version of such a theory will permit a sharper analysis of complexity of computations.

Acknowledgements

We wish to thank several collegues for helpful conversations: John Baez, Mike Barr, Jim Otto, Steve Vickers, and especially Abhay Ashtekar.

References

[Ab91] Abrusci, V.M. "Phase semantics and sequent calculus for pure noncommutative classical linear propositional logic", Journal of Symbolic Logic **56** (1991) 1403–1451.

[Ah66] Ahlfors, L.V. *Complex Analysis.* McGraw-Hill, New York, 1966, second edition. (First edition 1953).

[AM-A80] Ashtekar, A. and A. Magnon-Ashtekar "A geometrical approach to external potential problems in quantum field theory", Journal of General Relativity and Gravitation **12** (1980) 205-223.

[BSZ92] Baez, J.C., I.E. Segal and Z. Zhou *Introduction to algebraic and constructive quantum field theory.* Princeton University Press, Princeton, New Jersey, 1992.

[Ba61] Bargmann, V. "On a Hilbert space of analytic functions and an associated integral transform", Communications in Pure and Applied Mathematics **14** (1961) 187-214.

[B76a] Barr, M. "Duality of vector spaces", Cahiers de Topologie et Géometrie Differentielle **17** (1976) 3–14.

[B76b] Barr, M. "Duality of Banach spaces", Cahiers de Topologie et Géometrie Differentielle **17** (1976) 15–52.

[B79] Barr, M. **-Autonomous Categories.* Lecture Notes in Mathematics **752**, Springer-Verlag, Berlin, Heidelberg, New York, 1979.

[B91] Barr, M. "*-autonomous categories and linear logic", Mathematical Structures in Computer Science **1** (1991) 159–178.

[BBPH] Benton, B.N., G. Bierman, V. de Paiva, and M. Hyland "Term assignment for intuitionistic linear logic (preliminary report)", Preprint, University of Cambridge, 1992.

[Bl91] Blute, R.F. "Proof nets and coherence theorems", in *Categories and Computer Science*, Paris 1991, Lecture Notes in Computer Science **530**, Springer-Verlag, Berlin, Heidelberg, New York, 1991.

[Bl92] Blute, R.F. "Linear Logic, Coherence and Dinaturality", to appear in Theoretical Computer Science.

[Bl93a] Blute, R. "Linear Topology, Hopf Algebras and $*$-autonomous Categories", preprint, McGill University (1993)

[Bl93b] Blute, R. "Braided Proof Nets and Categories", in preparation, (1993).

[BCST] Blute, R.F., J.R.B. Cockett, R.A.G. Seely, and T.H. Trimble "Natural deduction and coherence for weakly distributive categories", Preprint, McGill University, 1992. (Submitted to JPAA)

[BCS93] Blute, R.F., J.R.B. Cockett, and R.A.G. Seely "**!** and **?** for weakly distributive categories: Storage as tensorial strength", preprint, McGill University, in preparation.

[CLM79] Cigler, J., V. Losert, P. Michor *Banach Modules and Functors on Categories of Banach Spaces.* Dekker, 1979

[CS91] Cockett, J.R.B. and R.A.G. Seely "Weakly distributive categories", in M.P. Fourman, P.T. Johnstone, A.M. Pitts, eds., *Applications of Categories to Computer Science*, London Mathematical Society Lecture Note Series **177** (1992) 45–65.

[C90] Conway, J. *A Course in Functional Analysis.* Springer Verlag Graduate Texts in Mathematics **96**, (1990)

[F64] Freyd, P.J. *Abelian Categories: An introduction to the theory of functors.* Harper and Row, New York, 1964.

[Ge85] Geroch, R. *Mathematical Physics.* University of Chicago Press, 1985.

[G87] Girard, J.-Y. "Linear logic", Theoretical Computer Science **50** (1987) 1–102.

[G89] Girard, J.-Y. "Geometry of interaction I: Interpretation of system F", Proceedings of ASL meeting, Padova, 1988.

[GL87] Girard, J.-Y. and Y. Lafont "Linear logic and lazy computation", in TAPSOFT'87, **2**, Lecture Notes in Computer Science **250**, Springer-Verlag, Berlin, Heidelberg, New York, 1987, 52–66.

[GSS91] Girard, J.-Y., A. Scedrov, and P.J. Scott "Bounded linear logic", Theoretical Computer Science **97** (1992) 1–66.

[Gu90] Gunning, R. *Introduction to Holomorphic functions of several variables, Volume 1: Function theory.* Mathematics Series, Wadsworth and Brooks/Cole, Belmont CA, 1990.

[IZ80] Itzykson, J, and J. Zuber *Introduction to Quantum Field Theory.* McGraw-Hill, New York, 1980.

[KR83] Kadison, R.V. and J.R. Ringrose *Fundamentals of the Theory of Operator Algebras.* Academic Press, New York, 1983

[KL80] Kelly, G.M. and M. La Plaza, "Coherence for Compact Closed Categories", Journal of Pure and Applied Algebra **19** (1980) 193–213.

[L65] Lang, S. *Algebra.* Addison-Wesley Pub. Co., Reading, Menlo Park, London, Don Mills, 1965.

[Le41] Lefschetz, S. *Algebraic Topology.* AMS Colloquium Publications **27**, 1941.

[M71] MacLane, S. *Categories for the Working Mathematician.* Graduate Texts in Mathematics **5**, Springer-Verlag, Berlin, Heidelberg, New York, 1971

[M-OM89] Martí-Oliet, N. and J. Meseguer "From Petri nets to linear logic", in D.H. Pitt *et al.*, eds. *Category Theory and Computer Science*, Manchester 1989, Lecture Notes in Computer Science **389**, Springer-Verlag, Berlin, Heidelberg, New York, 1989.

[dP89] de Paiva, V.C.V. "A Dialectica-like model of linear logic", in D.H. Pitt *et al.*, eds., *Category Theory and Computer Science*, Lecture Notes in Computer Science **389**, Springer-Verlag, Berlin, Heidelberg, New York, 1989.

[P79] Panangaden, P. "Positive and negative frequency decompositions in curved spacetimes", Journal of Mathematical Physics **20** (1979) 2506–2514.

[P80] Panangaden, P. "Propagators and Renormalization of Quantum Field Theory in Curved Spacetimes.", Ph.D. Thesis, Physics Department, University of Wisconsin-Milwaukee,

1980. Available through University Microfilms International, Ann Arbor.

[Se89] Seely, R.A.G. "Linear logic, $*$-autonomous categories and cofree coalgebras", in J. Gray and A. Scedrov (eds.), *Categories in Computer Science and Logic*, Contemporary Mathematics **92** (Am. Math. Soc. 1989).

[S62] Segal, E. "Mathematical characterization of the physical vacuum for linear Bose-Einstein fields", Illinois Journal of Mathematics **6** (1962) 500–523.

[T79] Takesaki M. *Theory of Operator Algebras.* Springer-Verlag, Berlin, Heidelberg, New York, 1979.

A syntax for linear logic

Philip Wadler (wadler@dcs.glasgow.ac.uk)

Department of Computing Science, University of Glasgow, G12 8QQ, Scotland

Abstract. There is a standard syntax for Girard's linear logic, due to Abramsky, and a standard semantics, due to Seely. Alas, the former is incoherent with the latter: different derivations of the same syntax may be assigned different semantics. This paper reviews the standard syntax and semantics, and discusses the problem that arises and a standard approach to its solution. A new solution is proposed, based on ideas taken from Girard's Logic of Unity. The new syntax is based on pattern matching, allowing for concise expression of programs.

1 Introduction

Somewhere inside linear logic, there is a programming language struggling to get out. We wish to define an analogue of lambda calculus to solve the following equation

$$\frac{\textit{lambda calculus}}{\textit{intuitionistic logic}} = \frac{?}{\textit{linear logic}}$$

What does this language look like?

One would think the answer should be straightforward by now. There is the linear logic of Girard [Gir87], there is the syntax of Abramsky [Abr90], and there is the semantics of Seely [See89]. Each of these has become a standard.

Abramsky was inspired by the earlier work of Lafont [Laf88] and Holmström [Hol88], and in turn inspired related systems by Chirimar, Gunter, and Riecke [CGR92], Lincoln and Mitchell [LM92], Mackie [Mac91], Troelstra [Tro92], and Wadler [Wad90, Wad91].

Seely provided a categorical model, that subsumes other models such as coherence spaces [Gir87], event spaces [Pra91], games [LS91], and the Geometry of Interaction [AJ92].

Unfortunately, Abramsky's syntax is *incoherent* with Seely's semantics: different derivations of the same term may yield different semantics. The basic problem is that Promotion does not commute with substitution. All of the above syntaxes suffer from a similar problem in one form or another, meaning that it is difficult to assign them a meaning in any of the above models. (While the above rightly credits Abramsky's influence, it would be wrong to burden him with too much blame. His syntax is coherent with the operational model he uses.)

This difficulty was spotted previously by myself [Wad92]. Other researchers have not only observed the problem, but also proposed a solution in the form of a syntax that 'boxes' the Promotion rule, in much the same way that boxes are used in proof nets. Notable in this regard is the work of Benton, Bierman, de

Paiva, and Hyland [BBdPH92], which provides a thorough introduction to natural deduction and sequent versions of linear logic, their categorical semantics, and the associated proof theory.

This paper presents a new syntax for linear logic that resolves the Promotion problem. The new syntax follows naturally from the idea of using patterns in sequents to represent destructors. It is closely related to Girard's Logic of Unity, LU (though without the polarities) [Gir91]. Indeed, the syntax presented here is based on a suggestion from Jean-Yves Girard, who pointed out to me that the problems I had noted with the standard syntax are resolved in the syntax of LU. The syntax also bears a passing resemblance to Moggi's calculus for monads [Mog89].

The syntax has been expressed in a way such that Dereliction and Promotion are made explicit, but Contraction and Weakening are left implicit. Even though linear logic is a 'resource conscious' logic, it seems adequate to be conscious of Dereliction and Promotion alone. The semantics introduces sufficient coherence properties so that the precise order in which Contraction and Weakening is applied is irrelevant. Such details may safely be omitted from the programme, yielding a more economic mode of expression. For those who truly desire to control all the details, a variant syntax that makes Contraction and Weakening explicit is given at the end.

Another approach to giving a syntax for linear logic based on LU appears in more recent work [Wad93]. That paper presents a more tutorial introduction: it is based on natural deduction rather than sequent calculus, so it takes less advantage of pattern matching, and it stresses the syntactic aspects of proof reduction while ignoring the semantics.

The remainder of this paper is organised as follows. Section 2 presents Abramsky's syntax. Section 3 presents Seely's semantics. Section 4 presents the new syntax. Section 5 compares the new syntax with Girard's Logic of Unity. Section 6 sketches some variations on the new syntax.

2 Old syntax

For simplicity, we restrict ourself to the connectives $\otimes$ (tensor product), $\multimap$ (linear implication), & (product), and ! (of course). A *type* (or proposition) is built from these connectives and base types.

$$A, B, C ::= X \mid (A \otimes B) \mid (A \multimap B) \mid (A \,\&\, B) \mid\, !A$$

Let A, B, C range over types, and X range over base types.

For each of these types, there are *terms* to construct and destruct values of that type.

$$\begin{aligned} t, u ::= {} & x \mid (t, u) \mid (\text{let } (x, y) = t \text{ in } u) \mid (\lambda x.\, t) \mid (t\ u) \mid \\ & \langle t, u\rangle \mid (\text{let } \langle x, \_\rangle = t \text{ in } u) \mid (\text{let } \langle \_, y\rangle = t \text{ in } u) \mid \\ & !t \mid (\text{let } !x = t \text{ in } u) \mid (\text{let } (x@y) = t \text{ in } u) \mid (\text{let } \_ = t \text{ in } u) \end{aligned}$$

Let t, u range over terms, and f, x, y, z range over variables. The use here of 'let $(x, y) = t$ in u' in comparison with Abramsky's 'let t be $x \otimes y$ in u' merely reflects a preference for the traditional notation, not any significant difference.

An ***assumption*** has the form $x_1 : A_1, \ldots, x_n : A_n$ where all the variables are distinct, and $n \geq 0$. Let Γ and Δ range over assumptions. Write Γ, Δ for the catenation of two assumptions; whenever this appears it is assumed that the variables of Γ and Δ are disjoint. Finally, a ***judgement*** has the form $\Gamma \vdash t : A$.

The rules for this version of linear logic are shown in Figure 1. Each rule has zero or more hypotheses above the horizontal line, and a conclusion below. There is one rule for each term form, with the exception of the two rules Exchange and Cut. The Exchange rule expresses that the order of assumptions is irrelevant. The Cut rule uses the notation $u[t/x]$ to stand for the term derived from u by substituting t for all occurrences of x.

$$\text{Id}\ \frac{}{x : A \vdash x : A} \qquad \text{Exchange}\ \frac{\Gamma, x : A, y : B, \Delta \vdash t : C}{\Gamma, y : B, x : A, \Delta \vdash t : C}$$

$$\text{Cut}\ \frac{\Gamma \vdash t : A \qquad x : A, \Delta \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B}$$

$$\otimes\text{-R}\ \frac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma, \Delta \vdash (t, u) : (A \otimes B)} \qquad \otimes\text{-L}\ \frac{\Gamma, x : A, y : B \vdash t : C}{\Gamma, z : (A \otimes B) \vdash (\text{let } (x, y) = z \text{ in } t) : C}$$

$$\multimap\text{-R}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x.\, t) : (A \multimap B)} \qquad \multimap\text{-L}\ \frac{\Gamma \vdash t : A \qquad y : B, \Delta \vdash u : C}{\Gamma, f : (A \multimap B), \Delta \vdash u[(f\, t)/y] : C}$$

$$\&\text{-R}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : (A \,\&\, B)}$$

$$\&\text{-L}\ \frac{\Gamma, x : A \vdash t : C}{\Gamma, z : (A \,\&\, B) \vdash (\text{let } \langle x, \_ \rangle = z \text{ in } t) : C} \qquad \frac{\Gamma, y : B \vdash t : C}{\Gamma, z : (A \,\&\, B) \vdash (\text{let } \langle \_, y \rangle = z \text{ in } t) : C}$$

$$\text{Promotion}\ \frac{x_1 : !A_1, \ldots, x_n : !A_n \vdash t : B}{x_1 : !A_1, \ldots, x_n : !A_n \vdash !t : !B} \qquad \text{Dereliction}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma, z : !A \vdash (\text{let } !x = z \text{ in } t) : B}$$

$$\text{Contraction}\ \frac{\Gamma, x : !A, y : !A \vdash t : B}{\Gamma, z : !A \vdash (\text{let } (x@y) = z \text{ in } t) : B} \qquad \text{Weakening}\ \frac{\Gamma \vdash t : B}{\Gamma, z : !A \vdash (\text{let } \_ = z \text{ in } t) : B}$$

Fig. 1. Old syntax

The rules are given in sequent calculus style, so constructors are represented by rules (such as $\otimes$-R) where the connective appears in the consequent of the conclusion (to the right of $\vdash$), and destructors are represented by rules (such as $\otimes$-L) where the connective appears in the antaceedent of the conclusion (to the left of $\vdash$). Promotion constructs a term with 'of course' type: it is a !-R rule. Dereliction uses a variable with 'of course' type once, Contraction duplicates it, and Weakening discards it: we refer to these collectively as !-L rules.

The $\multimap$-L rule only allows one to apply a variable to a term. Readers may be more familiar with the application rule of Natural Deduction, which allows one to apply a term to a term.

$$\multimap\text{-E}\ \frac{\Gamma \vdash t : (A \multimap B) \qquad \Delta \vdash u : A}{\Gamma, \Delta \vdash (t\,u) : B}$$

This rule is derived as follows.

$$\frac{\Gamma \vdash t : (A \multimap B) \qquad \dfrac{\Delta \vdash u : A \qquad \dfrac{}{y : B \vdash y : B}\,\mathrm{Id}}{\Delta, f : (A \multimap B) \vdash (f\,u) : B}\,{\multimap}\text{-L}}{\Gamma, \Delta \vdash (t\,u) : B}\,\mathrm{Cut}$$

Note the central role played here by Cut. Sequent and natural deduction versions of linear calculus are presented and shown equivalent by Lincoln and Mitchell [LM92]. Various mixtures of the two systems have been used by various researchers [BBdPH92, CGR92, Wad90, Wad91].

Here are a few example judgements.

$\vdash (\lambda x.\,\lambda y.\,\text{let }\_ = y \text{ in } x) : A \multimap\, !B \multimap A$
$\vdash (\lambda r.\,\lambda s.\,\lambda x.\,\text{let }!f = r \text{ in let }!g = s \text{ in let }(y@z) = x \text{ in } f\ y\ !(g\ z)) :$
$\quad !(!A \multimap\, !B \multimap C) \multimap\, !(!A \multimap B) \multimap\, !A \multimap C$
$\vdash (\lambda x.\,\text{let }(y, z) = x \text{ in}$
$\quad !(\text{let }!r = y \text{ in let }\_ = z \text{ in } r,\ \text{let }!s = z \text{ in let }\_ = x \text{ in } s)) :$
$\quad (!A \otimes !B) \multimap\, !(A \,\&\, B)$

Because of the Cut rule, an unnerving property of this system is that terms do *not* uniquely encode derivations. For example, the judgement

$$z : !!A \vdash !(\text{let }!x = z \text{ in } x) : !!A$$

has the derivation

$$(*)\qquad \frac{\dfrac{\dfrac{}{x : !A \vdash x : !A}\,\mathrm{Id}}{z : !!A \vdash (\text{let }!x = z \text{ in } x) : !A}\,\text{Dereliction}}{z : !!A \vdash !(\text{let }!x = z \text{ in } x) : !!A}\,\text{Promotion}$$

and also the derivation

$$(**)\qquad \frac{\dfrac{\dfrac{}{x : !A \vdash x : !A}\,\mathrm{Id}}{z : !!A \vdash (\text{let }!x = z \text{ in } x) : !A}\,\text{Dereliction} \qquad \dfrac{\dfrac{}{y : !A \vdash y : !A}\,\mathrm{Id}}{y : !A \vdash !y : !!A}\,\text{Promotion}}{z : !!A \vdash !(\text{let }!x = z \text{ in } x) : !!A.}\,\mathrm{Cut}$$

At first this may seem vaguely disturbing. We shall see shortly that it is profoundly disturbing, because each of these derivations is attached to a *different* semantics.

3 Semantics

This section presents Seely's model of linear logic, restricted to the case of intuitionistic linear logic. Seely's model is normally thought of as deriving from $*$-autonomous categories, but the dualising object $*$ is only required to model classical linear logic.

Anticipating that objects will model types and assumptions, and that arrows will model terms, let A, B, C and Γ, Δ range over objects, and t, u, v range over arrows.

A model of intuitionistic linear logic is provided by a category with the following structure.

- It is symmetric monoidal closed, with unit object 1, tensor $\otimes$, and internal hom $\multimap$. The transpose of $t : \Gamma \otimes A \to B$ is $\mathit{curry}(t) : \Gamma \to (A \multimap B)$, and the counit is $\mathit{apply} : (A \multimap B) \otimes A \to B$.
- It possesses finite products, with terminal $\top$ and product $\&$. The unique arrow to the terminal is $\langle\rangle : \Gamma \to \top$, the mediating morphism of $t : \Gamma \to A$ and $u : \Gamma \to B$ is $\langle t, u\rangle : \Gamma \to A \& B$, and the projections are $\mathit{fst} : A \& B \to A$ and $\mathit{snd} : A \& B \to B$.
- It possesses a comonad !. The Kleisli operator of $t : !A \to B$ is $\mathit{kleisli}(t) : !A \to !B$, and the counit is $\mathit{counit} : !A \to A$.
- There are isomorphisms $1 \simeq !\top$ and $!A \otimes !B \simeq !(A \& B)$. These induce a comonoid structure on each object $!A$ that is natural in A, given by

$$!A \xrightarrow{\mathit{discard}} 1 = !A \xrightarrow{!\langle\rangle} !\top \simeq 1,$$
$$!A \xrightarrow{\mathit{duplicate}} !A \otimes !A = !A \xrightarrow{!\langle id, id\rangle} !(A \& A) \simeq !A \otimes !A.$$

A categorical model is obtained by associating with each base type an object in our category, inducing a map from types to objects. Write A for both a type and its corresponding object. Each assumption $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ possesses a corresponding object $\Gamma = A_1 \otimes \cdots \otimes A_n$; the empty assumption corresponds to the unit object 1.

Each judgement $\Gamma \vdash t : A$ corresponds to an arrow $t : \Gamma \to A$. Figure 2 shows how each derivation induces an arrow which is its semantics.

Since a given judgement may have more than one derivation, we must verify that all possible derivations of a judgement assign it the same semantics. This property is called *coherence*, and its importance was noted by Breazu-Tannen, Coquand, Gunter and Scedrov [BCGS91]. In our case, two derivations of a judgement can differ only in their use of the Exchange or Cut rules, since uses of all other rules are encoded in the term. Coherence is guaranteed for Exchange by the fact that $\otimes$ is symmetric monoidal.

Unfortunately, the Cut rule does indeed introduce incoherence, when used in conjunction with Promotion. The derivation $(*)$ given previously induces the

$$\text{Id}\ \frac{}{A \xrightarrow{id} A} \qquad \text{Exchange}\ \frac{\Gamma \otimes A \otimes B \otimes \Delta \xrightarrow{t} C}{\Gamma \otimes B \otimes A \otimes \Delta \simeq \Gamma \otimes A \otimes B \otimes \Delta \xrightarrow{t} C}$$

$$\text{Cut}\ \frac{\Gamma \xrightarrow{t} A \qquad A \otimes \Delta \xrightarrow{u} B}{\Gamma \otimes \Delta \xrightarrow{t \otimes id} A \otimes \Delta \xrightarrow{u} B}$$

$$\otimes\text{-R}\ \frac{\Gamma \xrightarrow{t} A \qquad \Delta \xrightarrow{u} B}{\Gamma \otimes \Delta \xrightarrow{t \otimes u} A \otimes B} \qquad \otimes\text{-L}\ \frac{\Gamma \otimes A \otimes B \xrightarrow{t} C}{\Gamma \otimes (A \otimes B) \simeq \Gamma \otimes A \otimes B \xrightarrow{t} C}$$

$$\multimap\text{-R}\ \frac{\Gamma \otimes A \xrightarrow{t} B}{\Gamma \xrightarrow{curry(t)} (A \multimap B)}$$

$$\multimap\text{-L}\ \frac{\Gamma \xrightarrow{t} A \qquad B \otimes \Delta \xrightarrow{u} C}{\Gamma \otimes (A \multimap B) \otimes \Delta \xrightarrow{t \otimes id \otimes id} A \otimes (A \multimap B) \otimes \Delta \simeq (A \multimap B) \otimes A \otimes \Delta \xrightarrow{apply \otimes id} B \otimes \Delta \xrightarrow{u} C}$$

$$\&\text{-R}\ \frac{\Gamma \xrightarrow{t} A \qquad \Gamma \xrightarrow{u} B}{\Gamma \xrightarrow{\langle t, u \rangle} (A \& B)}$$

$$\&\text{-L}\ \frac{\Gamma \otimes A \xrightarrow{t} C}{\Gamma \otimes (A \& B) \xrightarrow{id \otimes fst} \Gamma \otimes A \xrightarrow{t} C} \qquad \frac{\Gamma \otimes B \xrightarrow{t} C}{\Gamma \otimes (A \& B) \xrightarrow{id \otimes snd} \Gamma \otimes B \xrightarrow{t} C}$$

$$\text{Promotion}\ \frac{!A_1 \otimes \cdots \otimes !A_n \simeq !(A_1 \& \cdots \& A_n) \xrightarrow{t} B}{!A_1 \otimes \cdots \otimes !A_n \simeq !(A_1 \& \cdots \& A_n) \xrightarrow{Kleisli(t)} B}$$

$$\text{Dereliction}\ \frac{\Gamma \otimes A \xrightarrow{t} B}{\Gamma \otimes !A \xrightarrow{id \otimes counit} \Gamma \otimes A \xrightarrow{t} B}$$

$$\text{Contraction}\ \frac{\Gamma \otimes !A \otimes !A \xrightarrow{t} B}{\Gamma \otimes !A \xrightarrow{id \otimes duplicate} \Gamma \otimes (!A \otimes !A) \simeq \Gamma \otimes !A \otimes !A \xrightarrow{t} B}$$

$$\text{Weakening}\ \frac{\Gamma \xrightarrow{t} B}{\Gamma \otimes !A \xrightarrow{id \otimes discard} \Gamma \otimes I \simeq \Gamma \xrightarrow{t} B}$$

Fig. 2. Semantics

semantics

$$(*)\qquad \dfrac{\dfrac{\dfrac{}{!A \xrightarrow{id} !A}\,\text{Id}}{!!A \xrightarrow{counit} !A}\,\text{Dereliction}}{!!A \xrightarrow{kleisli(counit)} !!A.}\,\text{Promotion}$$

The derivation (**) given previously induces the semantics

$$(**)\qquad \dfrac{\dfrac{\dfrac{}{!A \xrightarrow{id} !A}\,\text{Id}}{!!A \xrightarrow{counit} !A}\,\text{Dereliction} \qquad \dfrac{\dfrac{}{!A \xrightarrow{id} !A}\,\text{Id}}{!A \xrightarrow{kleisli(id)} !!A}\,\text{Promotion}}{!!A \xrightarrow{counit} !A \xrightarrow{kleisli(id)} !!A.}\,\text{Cut}$$

These are *not* necessarily equal. The arrow for (*) is necessarily the identity, but the arrow for (**) is not. We thus have the following.

> **Counterexample.** The syntax of Figure 1 is not coherent with the semantics of Figure 2.

This problem arises only with the Promotion rule.

> **Theorem.** The syntax of Figure 1 is coherent with the semantics of Figure 2 if Promotion is not used. If a term does not contain ! as a constructor, then all derivations of it will have the same semantics, even if they use Cut.

The proof is by examination of overlapping rules.

All of the variations of Abramsky's syntax cited above suffer from this problem in one form or another. In a natural deduction system, this problem reveals itself in a failure of the Substitution Lemma: substitution does not commute with Promotion [Wad92]. The same difficulty is at the root of problems that Lincoln and Mitchell [LM92] and Chirimar, Gunter, and Riecke [CGR92] encountered with Subject Reduction theorems, forcing them to be restricted in various ways.

One way to fix the problems is to restrict the class of categorical models. In an earlier paper [Wad92], it was shown that substitution commutes with Promotion if and only if the categorical model satisfies $counit; kleisli(id) = id$. This is not very satisfactory, as none of the models cited at the beginning of this paper satisfy this restriction. Nonetheless, similar restrictions appears in the work of O'Hearn [O'He91] and Filinski [Fil92], and this may explain why.

Another fix is to revise the syntax of Promotion, so that it records explicitly what substitutions have occured. This suggestion has been made by Benton, Bierman, de Paiva, and Hyland [BBdPH92] and by Reddy [Red91]. The syntax of promotion is changed so that the term $!t$ is replaced by $![u_1/x_1, \ldots, u_n/x_n]t$, where $x_1, \ldots, x_n$ are all the free variables of t. Here the square brackets are concrete syntax; this concrete syntax is chosen to resemble the meta-syntax for substitution, since the roles are similar. The revised Promotion rule is as follows.

$$\text{Promotion}'\;\dfrac{x_1 : !A_1, \ldots, x_n : !A_n \vdash t : B}{z_1 : !A_1, \ldots, z_n : !A_n \vdash ![z_1/x_1, \ldots, z_n/x_n]t : B}$$

After promotion, the free variables of the term are $z_1, \ldots, z_n$, and any substitutions for these variables will be explicit in the term. By acting as a barrier to substitution, the new syntax performs much the same role that boxing does in proof nets [Gir87]. It is possible to show that this 'boxed' syntax is coherent: all derivations of a term have the same semantics.

Returning to our example, the first derivation becomes

$$(*)\qquad \dfrac{\dfrac{\dfrac{}{x : !A \vdash x : !A}\,\text{Id}}{y : !!A \vdash (\text{let } !x = y \text{ in } x) : !A}\,\text{Dereliction}}{z : !!A \vdash ![z/y](\text{let } !x = y \text{ in } x) : !!A}\,\text{Promotion}'$$

and the second becomes

$$(**)\qquad \dfrac{\dfrac{\dfrac{}{x : !A \vdash x : !A}\,\text{Id}}{z : !!A \vdash (\text{let } !x = z \text{ in } x) : !A}\,\text{Dereliction} \qquad \dfrac{\dfrac{}{y : !A \vdash y : !A}\,\text{Id}}{w : !A \vdash ![w/y]y : !!A}\,\text{Promotion}'}{z : !!A \vdash ![(\text{let } !x = z \text{ in } x)/y]y : !!A.}\,\text{Cut}$$

Now the terms are different, so it is not a problem that they are assigned different semantics.

The key idea here is that there is a barrier around Promotion indicating what substitutions occur. The next section will reveal a different syntax that erects a similar barrier.

4 New syntax

The new syntax makes three significant changes. First, it introduces a notion of ***pattern***. Whereas previously assumptions paired variables with types, now they will pair patterns with types. Second, the various instances of 'let' that appeared previously, associated with the ⊗-L, &-L, and !-L rules, are now all consolidated into a single 'let'. Third, there is no explicit indication of Contraction or Weakening in the terms. (This third change is convenient but not essential, and we will see how to undo it in the next section.)

For each type, there is now a ***term*** to construct values of that type, and a ***pattern*** to destruct values of that type. The exception is $\multimap$, which has terms for both construction and destruction. There is also a 'let' term.

$$\begin{aligned} p, q &::= x \mid (p, q) \mid \langle p, \_\rangle \mid \langle \_, q\rangle \mid !x \\ t, u &::= x \mid (t, u) \mid (\lambda p.\, t) \mid (t\ u) \mid \langle t, u\rangle \mid !t \mid (\text{let } p = t \text{ in } u) \end{aligned}$$

Let p, q range over patterns, t, u range over terms, and f, x, y, z range over variables. Note that patterns for the types $\otimes$ and $\&$ may be nested, but patterns for the type ! may not. We will see below that this system guarantees coherent semantics, but that if nested ! patterns were allowed then coherence would again be lost.

An ***assumption*** now has the form $p_1 : A_1, \ldots, p_n : A_n$ where $n \geq 0$ and no variable appears more than once in all of the patterns combined. Again, let Γ, Δ range over assumptions, and judgements have the form $\Gamma \vdash t : A$.

The rules for this version of linear logic are shown in Figure 3. With the exception of the new rule Let, there is a one-to-one correspondence between rules in the old syntax and rules in the new syntax. The $\otimes$-L, &-L, and !-L rules now all introduce patterns rather than 'let' terms. The introduction of 'let' terms has been factored out into a separate Let rule. The three !-L rules all introduce the same pattern, so there is no explicit indication of Contraction or Weakening. The appearance of ! patterns in Contraction helps to explain the restriction to variables, since this makes the substitution associated with Contraction easier to express. Promotion is changed so that in addition to requiring that all types in the assumption begin with a !, all patterns in the assumption must also do so.

This last change is the critical step – the ! patterns will act as a barrier to substitution, just as the 'boxed' syntax at the end of the last section did. What was written $![u_1/x_1, \ldots, u_n/x_n]t$ in the boxed syntax is here written

$$\text{let } !y_1 = u_1 \text{ in } \cdots \text{ let } !y_n = u_n \text{ in } t[!y_1/x_1, \ldots, !y_n/x_n].$$

Note that $![u_i/x_i]t$ is concrete syntax, whereas $t[!y_i/x_i]$ is meta-syntax for substitution. Although here the new syntax appears less compact than the boxed syntax, in practice the new syntax will often be more compact because of pattern matching, and because Contraction and Weakening are not explicitly indicated.

The Let rule has no logical content, as erasing the terms from the hypothesis or the conclusion gives the same logical judgement, $\Gamma, A \vdash B$. Indeed, the Let rule can be simply considered a convenient abbreviation, as it can be derived from the $\multimap$ rules and Cut.

$$\dfrac{\dfrac{\Gamma, p : A \vdash u : B}{\Gamma \vdash (\lambda p.\, u) : (A \multimap B)}\ {\multimap}\text{-R} \qquad \dfrac{\dfrac{}{x : A \vdash x : A}\ \text{Id} \qquad \dfrac{}{y : B \vdash y : B}\ \text{Id}}{f : (A \multimap B),\ x : A \vdash (f\, x) : B}\ {\multimap}\text{-L}}{\Gamma,\ x : A \vdash ((\lambda p.\, u)\, x) : B}\ \text{Cut}$$

Thus, we can take (let $p = x$ in u) as an abbreviation for $((\lambda p.\, u)\, x)$.

The rules in Figure 2 for assigning a semantics to the derivation of a term still apply. The Let rule assigns the judgement in the conclusion the same semantics as the judgement in the hypothesis.

Theorem. The syntax of Figure 3 is coherent with the semantics of Figure 2.

The proof is by examining the possible overlaps between rules.

$$\text{Id}\ \frac{}{x : A \vdash x : A} \qquad \text{Exchange}\ \frac{\Gamma, p : A, q : B, \Delta \vdash t : C}{\Gamma, q : B, p : A, \Delta \vdash t : C}$$

$$\text{Cut}\ \frac{\Gamma \vdash t : A \qquad x : A, \Delta \vdash u : B}{\Gamma, \Delta \vdash u[t/x] : B} \qquad \text{Let}\ \frac{\Gamma, p : A \vdash u : B}{\Gamma, x : A \vdash (\text{let } p = x \text{ in } u) : B}$$

$$\otimes\text{-R}\ \frac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma, \Delta \vdash (t, u) : (A \otimes B)} \qquad \otimes\text{-L}\ \frac{\Gamma, p : A, q : B \vdash t : C}{\Gamma, (p, q) : (A \otimes B) \vdash t : C}$$

$$\multimap\text{-R}\ \frac{\Gamma, p : A \vdash t : B}{\Gamma \vdash (\lambda p.\, t) : (A \multimap B)} \qquad \multimap\text{-L}\ \frac{\Gamma \vdash t : A \qquad y : B, \Delta \vdash u : C}{\Gamma, f : (A \multimap B), \Delta \vdash u[(f\,t)/y]) : C}$$

$$\&\text{-R}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : (A \mathbin{\&} B)}$$

$$\&\text{-L}\ \frac{\Gamma, p : A \vdash t : C}{\Gamma, \langle p, \_ \rangle : (A \mathbin{\&} B) \vdash t : C} \qquad \frac{\Gamma, q : B \vdash t : C}{\Gamma, \langle \_, q \rangle : (A \mathbin{\&} B) \vdash t : C}$$

$$\text{Promotion}\ \frac{!x_1 : !A_1, \ldots, !x_n : !A_n \vdash t : B}{!x_1 : !A_1, \ldots, !x_n : !A_n \vdash !t : !B} \qquad \text{Dereliction}\ \frac{\Gamma, z : A \vdash t : B}{\Gamma, !z : !A \vdash t : B}$$

$$\text{Contraction}\ \frac{\Gamma, !x : A, !y : A \vdash t : B}{\Gamma, !z : A \vdash t[z/x, z/y] : B} \qquad \text{Weakening}\ \frac{\Gamma \vdash t : B}{\Gamma, !z : !A \vdash t : B}$$

Fig. 3. New syntax

Here are the example judgements of Section 2 revisited.

$$\vdash (\lambda x.\, \lambda !y.\, x) : A \multimap !B \multimap A$$
$$\vdash (\lambda !f.\, \lambda !g.\, \lambda !x.\, f\, !x\, !(g\, !x)) : !(!A \multimap !B \multimap C) \multimap !(!A \multimap B) \multimap !A \multimap C$$
$$\vdash (\lambda(!r, !s).\, !\langle r, s \rangle) : (!A \otimes !B) \multimap !(A \mathbin{\&} B)$$

The new syntax is considerably more compact.

Returning to our main example, the first derivation becomes

$$(*) \qquad \cfrac{\cfrac{\cfrac{}{z : !A \vdash z : !A}\ \text{Id}}{!z : !!A \vdash z : !A}\ \text{Dereliction}}{!z : !!A \vdash !z : !!A.}\ \text{Promotion}$$

The second derivation is no longer valid. The Promotion rule no longer applies, because it contains patterns not in the proper form. In order to obtain the same semantics as previously, the derivation must be rewritten. The old use of the Id rule, which yielded $x : !A \vdash x : !A$, is replaced with a use of Id, Dereliction, and Promotion, which yields $!y : !A \vdash !y : !A$. Both derivations have the same semantics (the identity arrow), but further promotion is only possible for the latter.

(**)
$$\begin{prooftree}
\AxiomC{}
\RightLabel{Id}
\UnaryInfC{$z : !A \vdash z : !A$}
\RightLabel{Dereliction}
\UnaryInfC{$!z : !!A \vdash z : !A$}
\AxiomC{}
\RightLabel{Id}
\UnaryInfC{$x : A \vdash x : A$}
\RightLabel{Dereliction}
\UnaryInfC{$!x : !A \vdash x : A$}
\RightLabel{Promotion}
\UnaryInfC{$!x : !A \vdash !x : !A$}
\RightLabel{Promotion}
\UnaryInfC{$!x : !A \vdash !!x : !!A$}
\RightLabel{Let}
\UnaryInfC{$w : !A \vdash \text{let } !x = w \text{ in } !!x : !!A$}
\RightLabel{Cut}
\BinaryInfC{$!z : !!A \vdash (\text{let } !x = z \text{ in } !!x) : !!A.$}
\end{prooftree}$$

The new (*) and (**) have the same semantics as the old. As with the boxed semantics, we now have distinct terms yielding distinct semantics. Every old derivation carries into a new derivation with the same semantics; the only change needed may be to replace some uses of Id with Id, Dereliction, and Promotion, as above; and to add some uses of Let.

If nested ! patterns were allowed, the coherence property would again be lost. Consider the (illegal) judgement $!!x : !!A \vdash !x : !A$. There are two different proof trees that yield this judgement. The first applies rules in the order Id, Derelict, Promote, Derelict and has semantics *counit*; *kleisli*(*counit*), which simplifies to *counit*. The second applies rules in the order Id, Derelict, Derelict, Promote and has semantics *kleisli*(*counit*; *counit*), which does *not* simplify to *counit*. Hence the restriction that ! patterns cannot be nested. There is no similar problem for $\otimes$ or & patterns.

Since there are no longer explicit terms for Contraction and Weakening, these must be checked for coherence. Coherence here is guaranteed by the fact that *discard* and *duplicate* form a comonoid: duplicating and then discarding is the same as the identity; two duplications in different orders have the same meaning, and so on. The situation is very similar to that for Exchange, and indeed there appears to be no more reason for textually indicating each use of Contraction or Weakening than there is for indicating each use of Exchange.

The new syntax satisfies a pleasing number of equivalences. In the case where the 'let' is simply binding a variable, it can be replaced by substitution. Further, whenever a constructor meets a corresponding destructor, it can be substituted out. Finally, 'let' satisfies a pair of familiar laws. All these points are summarised in the following.

Theorem. The following equations hold for the syntax of Figure 3 with the semantics of Figure 2.

$$\begin{array}{ll}
(1) & (\text{let } x = t \text{ in } u) = u[t/x] \\
(2) & (\text{let } (p, q) = (t, u) \text{ in } v) = (\text{let } p = t \text{ in } (\text{let } q = u \text{ in } v)) \\
(3) & ((\lambda p.\, u)\, t) = (\text{let } p = t \text{ in } u) \\
(4) & (\text{let } \langle p, \_\rangle = \langle t, u\rangle \text{ in } v) = (\text{let } p = t \text{ in } v) \\
(5) & (\text{let } \langle \_, q\rangle = \langle t, u\rangle \text{ in } v) = (\text{let } q = u \text{ in } v) \\
(6) & (\text{let } !x = !t \text{ in } u) = u[t/x] \\
(7) & (\text{let } p = t \text{ in } p) = t \\
(8) & (\text{let } q = (\text{let } p = t \text{ in } u) \text{ in } v) = (\text{let } p = t \text{ in } (\text{let } q = u \text{ in } v))
\end{array}$$

These laws assume no collision of bound variables; e.g., in law (2), the free variables of u must not be bound in p.

Law (1) is immediate from coherence. Laws (2)–(6) and (8) follow immediately from the categorical semantics. Law (7) is proved by induction on the pattern.

Here are equations (6)–(8) again, with the last two instantiated to the special case of ! patterns.

$$\begin{array}{c}
(\text{let } !x = !t \text{ in } u) = u[t/x] \\
(\text{let } !x = t \text{ in } !x) = t \\
(\text{let } !y = (\text{let } !x = t \text{ in } u) \text{ in } v) = (\text{let } !x = t \text{ in } (\text{let } !y = u \text{ in } v))
\end{array}$$

These are reminiscent of the three equations satisfied by Moggi's calculus for monads [Mog89]. For our syntax the first equation depends on the right counit law for comonads and the second equation depends on the left counit law for comonads; while for Moggi's calculus the first equation depends on the left unit law for monads, and the second equation depends on the right unit law for monads. However, the analogy goes awry with the third equation. Moggi's last equation depends on the associative law for monads, while our last equation has nothing to do with the associative law for comonads. (However, the associative laws for comonads is important in verifying the coherence of the new syntax.)

5 Logic of Unity

The system described here is closely related to Girard's Logic of Unity (LU) [Gir91]. Indeed, it was inspired by it: the trick that avoids coherence problems was stolen from LU. To clarify the relation, this section present an appropriately simplified version of LU. Major differences from Girard's LU are that this version is restricted to the intuitionistic fragment, and there are no polarities.

In this variant of LU, there are two sorts of assumptions, linear and intuitionistic. Linear assumptions pair patterns with types, so they have the form $p_1 : A_1, \ldots p_n : A_n$, while intuitionistic assumptions pair variables with types, so they have the form $x_1 : A_1, \ldots x_n : A_n$. Linear assumptions may not be contracted or weakened, while intuitionistic assumptions may. The Contraction rule is much more neatly expressed in terms of variables because it involves substitution, which partly explains the restriction to variables in intuitionistic assumptions. Let Γ, Δ range over linear assumptions, and Φ, Ψ range over intuitionistic assumptions. A judgement has the form $\Gamma;\ \Phi \vdash t : A$, where the linear and intuitionistic assumptions are separated by a semicolon.

The rules for this variant of LU are shown in Figure 4. There is a close correspondence with our new syntax of Figure 3, here called LL for short. The previous Id rule is split into two rules, Id and Id-Int, the first dealing with a linear assumption and the second dealing with an intuitionistic one. Similarly, the previous Exchange rule is split into Exchange and Exchange-Int. The logical rules for $\otimes$ and $\multimap$ deal with linear assumptions. Promotion and Dereliction are logical rules of ! and deal with the relation between the two sorts of assumptions, while Contraction and Weakening have metamorphosed from logical rules of ! to structural rules dealing with intuitionistic assumptions.

$$\text{Id}\ \frac{}{x : A;\ \vdash x : A} \qquad \text{Id-Int}\ \frac{}{;\ x : A \vdash x : A}$$

$$\text{Exchange}\ \frac{\Gamma,\ p : A,\ q : B,\ \Delta;\ \Phi \vdash t : C}{\Gamma,\ q : B,\ p : A,\ \Delta;\ \Phi \vdash t : C} \qquad \text{Exchange-Int}\ \frac{\Gamma;\ \Phi,\ x : A,\ y : B,\ \Psi \vdash t : C}{\Gamma;\ \Phi,\ y : B,\ x : A,\ \Psi \vdash t : C}$$

$$\text{Cut}\ \frac{\Gamma;\ \Phi \vdash t : A \qquad x : A,\ \Delta;\ \Psi \vdash u : B}{\Gamma,\ \Delta;\ \Phi,\ \Psi \vdash u[t/x] : B} \qquad \text{Let}\ \frac{\Gamma,\ p : A;\ \Phi \vdash u : B}{\Gamma,\ x : A;\ \Phi \vdash (\text{let } p = x \text{ in } u) : B}$$

$$\otimes\text{-R}\ \frac{\Gamma;\ \Phi \vdash t : A \qquad \Delta;\ \Psi \vdash u : B}{\Gamma,\ \Delta;\ \Phi,\ \Psi \vdash (t, u) : (A \otimes B)} \qquad \otimes\text{-L}\ \frac{\Gamma,\ p : A,\ q : B;\ \Phi \vdash t : C}{\Gamma,\ (p, q) : (A \otimes B);\ \Phi \vdash t : C}$$

$$\multimap\text{-R}\ \frac{\Gamma,\ p : A;\ \Phi \vdash t : B}{\Gamma;\ \Phi \vdash (\lambda p.\ t) : (A \multimap B)} \qquad \multimap\text{-L}\ \frac{\Gamma;\ \Phi \vdash t : A \qquad y : B,\ \Delta;\ \Psi \vdash u : C}{\Gamma,\ f : (A \multimap B),\ \Delta;\ \Phi,\ \Psi \vdash u[(f\ t)/y]) : C}$$

$$\&\text{-R}\ \frac{\Gamma;\ \Phi \vdash t : A \qquad \Gamma;\ \Phi \vdash u : B}{\Gamma;\ \Phi \vdash \langle t, u \rangle : (A \,\&\, B)}$$

$$\&\text{-L}\ \frac{\Gamma,\ p : A;\ \Phi \vdash t : C}{\Gamma,\ \langle p, \_ \rangle : (A \,\&\, B);\ \Phi \vdash t : C} \qquad \frac{\Gamma,\ q : B;\ \Phi \vdash t : C}{\Gamma,\ \langle \_, q \rangle : (A \,\&\, B);\ \Phi \vdash t : C}$$

$$\text{Promotion}\ \frac{;\ \Phi \vdash t : B}{;\ \Phi \vdash !t : !B} \qquad \text{Dereliction}\ \frac{\Gamma,\ !z : !A;\ \Phi \vdash t : B}{\Gamma;\ z : A,\ \Phi \vdash t : B}$$

$$\text{Contraction}\ \frac{\Gamma;\ \Phi,\ x : A,\ y : A \vdash t : B}{\Gamma;\ \Phi,\ z : A \vdash t[z/x, z/y] : B} \qquad \text{Weakening}\ \frac{\Gamma;\ \Phi \vdash t : B}{\Gamma;\ \Phi,\ z : A \vdash t : B}$$

Fig. 4. A version of the Logic of Unity

It is possible to translate LU into LL. A judgement of the form $\Gamma;\ \Phi \vdash t : A$ in LU corresponds to a judgement $\Gamma,\ !\Phi \vdash t : A$ in LL, where if Φ is

$x_1 : A_1, \ldots, x_n : A_n$ then $!\Phi$ is $!x_1 : !A_1, \ldots, !x_n : !A_n$.

Each rule in LU corresponds to the rule of the same name in LL, with two spectacular exceptions. Id-Int in LU translates to a combination of Id and Dereliction in LL.

$$\text{Id-Int}\ \frac{}{;\ x : A \vdash x : A} \quad \mapsto \quad \dfrac{\dfrac{}{x : A \vdash x : A}\ \text{Id}}{!x : !A \vdash x : A}\ \text{Dereliction}$$

On the other hand, both the hypothesis and conclusion of the Dereliction rule of LU translate to the same judgement of LL.

$$\text{Dereliction}\ \frac{\Gamma, !z : !A;\ \Phi \vdash t : B}{\Gamma;\ z : A, \Phi \vdash t : B} \quad \mapsto \quad \Gamma, !z : !A, !\Phi \vdash t : B$$

Thus Id-Int in LU corresponds to Dereliction in LL, while Dereliction in LU corresponds to nothing at all!

The translation induces the obvious semantics: the semantics of a judgement in LU is the the same as the semantics of the corresponding judgement in LL. Analogues of the theorems of Section 4 hold.

There are a number of rules which one would expect of LU, which can be derived from the rules given here. The most important of these is Cut-Int.

$$\text{Cut-Int}\ \frac{;\ \Phi \vdash t : A \qquad \Delta;\ x : A, \Psi \vdash u : B}{\Delta;\ \Phi, \Psi \vdash (\text{let } !x = !t \text{ in } u) : B}$$

This rule is derived as follows.

$$\dfrac{\dfrac{;\ \Phi \vdash t : A}{;\ \Phi \vdash !t : !A}\ \text{Promotion} \qquad \dfrac{\dfrac{\Delta;\ x : A, \Psi \vdash u : B}{\Delta, !x : !A;\ \Psi \vdash u : B}\ \text{Dereliction}}{\Delta, y : !A;\ \Psi \vdash \text{let } !x = y \text{ in } u : B}\ \text{Let}}{\Delta;\ \Phi, \Psi \vdash (\text{let } !x = !t \text{ in } u) : B}$$

Observe that the semantics of (let $!x = !t$ in u) is identical to the semantics of $u[t/x]$, which may offer further scope for simplification.

6 Variations

Many programmers are unfamiliar with the $\multimap$-L rule of the sequent calculus, and may find the $\multimap$-E rule of natural deduction more natural. On the other hand, the use of sequent calculus seems to naturally capture the pattern matching in the $\otimes$ and & rules, so there may be some value in exploring a hybrid of the two systems. One variation would simply replace the $\multimap$-L rule by $\multimap$-E. This might be easier for programmers to follow, though important logical properties such as cut-elimination would be lost.

The work presented here extends straightforwardly to handle sums.

$$\oplus\text{-R}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash (\mathrm{inl}\, t) : (A \oplus B)} \qquad \frac{\Gamma \vdash u : B}{\Gamma \vdash (\mathrm{inr}\, u) : (A \oplus B)}$$

$$\oplus\text{-L}\ \frac{\Gamma \vdash z : (A \oplus B) \qquad \Delta, p : A \vdash t : C \qquad \Delta, q : B \vdash u : C}{\Gamma, \Delta \vdash (\mathrm{case}\, z\, \mathrm{of}\, \{\mathrm{inl}\, p \rightarrow t; \mathrm{inr}\, q \rightarrow u\}) : C}$$

These rules do not exploit the power of pattern matching as thoroughly as one might hope; for instance, patterns of the form (inl p) and (inr q) cannot appear nested inside other patterns. An open question is whether there is a different approach that allows for such nested patterns. One path in this direction is indicated by the work of Breazu-Tannen, Kesner, and Puel [BTKP93].

Another variation is to include patterns to indicate Contraction and Weakening. The grammar of patterns is divided into *patterns* and *of-course patterns*, the former being a superset of the latter.

$$\begin{aligned} p, q &::= x \mid (p, q) \mid \langle p, \_\rangle \mid \langle \_, q\rangle \mid o \\ o, r &::= (o@r) \mid \_ \mid !x \end{aligned}$$

Let p, q range over patterns, and o, r range over of-course patterns. The new rules are as follows.

$$\text{Promotion}\ \frac{o_1 : !A_1, \ldots, o_n : !A_n \vdash t : B}{o_1 : !A_1, \ldots, o_n : !A_n \vdash !t : !B} \qquad \text{Dereliction}\ \frac{\Gamma, z : A \vdash t : B}{\Gamma, !z : !A \vdash t : B}$$

$$\text{Contraction}\ \frac{\Gamma, o : A, r : A \vdash t : B}{\Gamma, (o@r) : A \vdash t : B} \qquad \text{Weakening}\ \frac{\Gamma \vdash t : B}{\Gamma, \_ : !A \vdash t : B}$$

Dereliction, Contraction, and Weakening introduce the three different sorts of of-course pattern, while Promotion allows any of-course pattern. This variation is included simply to illustrate that the approach used here does not preclude the use of specific patterns to indicate Contraction and Weakening. However, in practice there does not seem to be much value in including such detailed information.

Acknowledgements. I am grateful to Jean-Yves Girard, Samson Abramsky, Robert Seely, Martin Hyland, Valeria de Paiva, and Uday Reddy for their insights into linear logic. One of the MFPS referees made detailed and valuable suggestions for improvement.

References

[Abr90] S. Abramsky, Computational interpretations of linear logic. Presented at *Workshop on Mathematical Foundations of Programming Language Semantics*, 1990. To appear in *Theoretical Computer Science*.

[AJ92] S. Abramsky and R. Jagadeesan, New foundations for the geometry of interaction. In *7'th Symposium on Logic in Computer Science*, IEEE Press, Santa Cruz, California, June 1992.

[Bar79] M. Barr, **-Autonomous Categories*. Lecture Notes in Mathematics 752, Springer Verlag, 1979.

[BBdPH92] N. Benton, G. Bierman, V. de Paiva, and M. Hyland, Type assignment for intuitionistic linear logic. Draft paper, August 1992.

[BCGS91] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov, Inheritance as explicit coercion. *Information and Compututation*, 93:172–221, 1991. (An earlier version appeared in *Symposium on Logic in Computer Science*, IEEE Press, Asilomar, California, June 1989.)

[BTKP93] V. Breazu-Tannen, D. Kesner, L. Puel, A typed pattern calculus. In *8'th Symposium on Logic in Computer Science*, Montreal, June 1993.

[CGR92] J. Chirimar, C. A. Gunter, and J. G. Riecke. Linear ML. In *Symposium on Lisp and Functional Programming*, ACM Press, San Francisco, June 1992.

[Fil92] A. Filinski, Linear continuations. In *Symposium on Principles of Programming Languages*, ACM Press, Albuquerque, New Mexico, January 1992.

[Gir87] J.-Y. Girard, Linear logic. *Theoretical Computer Science*, **50**:1–102, 1987.

[Gir91] J.-Y. Girard, On the unity of logic. Manuscript, 1991.

[Hol88] S. Holmström, A linear functional language. Draft paper, Chalmers University of Technology, 1988.

[Laf88] Y. Lafont, The linear abstract machine. *Theoretical Computer Science*, **59**:157–180, 1988.

[LM92] P. Lincoln and J. Mitchell, Operational aspects of linear lambda calculus. In *7'th Symposium on Logic in Computer Science*, IEEE Press, Santa Cruz, California, June 1992.

[LS91] Y. Lafont and T. Streicher. Game semantics for linear logic. In *6'th Symposium on Logic in Computer Science*, IEEE Press, Amsterdam, July 1991.

[Mac91] I. Mackie, Lilac: a functional programming language based on linear logic. Master's Thesis, Imperial College London, 1991.

[Mog89] E. Moggi, Computational lambda-calculus and monads. In *4'th Symposium on Logic in Computer Science*, IEEE Press, Asilomar, California, June 1989.

[O'He91] P. W. O'Hearn, Linear logic and interference control. In *Conference on Category Theory and Computer Science*, Paris, September 1991. LNCS, Springer Verlag.

[Pra91] V. Pratt, Event spaces and their linear logic. In *AMAST '91: Algebraic Methodology And Software Technology*, Iowa City, Springer Verlag LNCS, 1992.

[Red91] U. Reddy, Acceptors as Values. Manuscript, December 1991.

[See89] R. A. G. Seely, Linear logic, *-autonomous categories, and cofree coalgebras. In *Categories in Computer Science and Logic*, June 1989. AMS Contemporary Mathematics 92.

[Tro92] A. S. Troelstra, *Lectures on Linear Logic*. CSLI Lecture Notes, 1992.

[Wad90] P. Wadler, Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, North Holland, April 1990.

[Wad91] P. Wadler, Is there a use for linear logic? In *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, New Haven, Connecticut, ACM Press, June 1991.

[Wad92] P. Wadler, There's no substitute for linear logic. Presented at *Workshop on Mathematical Foundations of Programming Language Semantics*, Oxford, April 1992.

[Wad93] P. Wadler, A taste of linear logic. In *Mathematical Foundations of Computer Science*, Gdansk, Poland, LNCS, Springer Verlag, August 1993.

A Complete Axiomatisation for Trace Congruence of Finite State Behaviors

Alexander Rabinovich
Department of Computer Science
The University of Texas at El Paso
El Paso TX 79968
e.mail: alex@cs.ep.utexas.edu

1 Introduction

Salomaa in [4] presented a complete axiomatisation for language equivalence of regular expressions. In [2] Milner proposed a complete proof system for bisimulation equivalence over finite state behaviors. Bisimulation equivalence is a very discriminating equivalence. Later, a weaker equivalence, called observational congruence, was considered by Milner and its complete axiomatisation was provided in [3]. Unlike language equivalence, both bisimulation equivalence and observational congruence distinguish between finite state behaviors on the basis of their branching structure.

In this paper trace congruence is considered. Trace congruence ignores branching structure and is close to language equivalence of classical automata theory. We provide a complete proof system for trace congruence over finite state behaviors presented as μ-expressions.

The paper is organized as follows: In Section 2 finite state behaviors are described as μ-expressions. Section 3 introduces trace equivalence. Unfortunately, trace equivalence is not substitutive. Section 3 provides a characterization of the fully abstract refinement of trace equivalence which we call trace congruence. A proof system for trace congruence is presented in section 4. The proof of its completeness is given in section 6. Section 5 contains some definitions which are helpful for the completeness proof.

In section 7 we comment about the relationship between our axiomatisation and Salomaa's classical axiomatisation of language equivalence for regular expressions. One of the open questions mentioned in [2] is to find an axiomatisation of bisimulation equivalence for finite state behaviors presented as regular expressions. We provide such an axiomatisation for a variant of regular expressions. Two appendixes contain the proofs of technical lemmas.

2 μ-Expressions

We are dealing in this paper with finite state behaviors presented as μ-expressions. Let us first recall some definitions and facts about μ-expressions and their behaviors. The presentation is based on [2, 3].

We presuppose two fixed sets

$$\begin{array}{rl} Act = \{a,\ a_1, \ldots b,\ b_1, \ldots\} & \text{the actions} \\ Var = \{X,\ X_1, \ldots Y,\ Y_1, \ldots\} & \text{the variables.} \end{array}$$

μ-expressions are defined by the following grammar:

$$E ::= 0|X|aE|E + E|\mu X.E, \text{ where } X \in Var,\ a \in Act.$$

μ stands for recursion (binding the variable which follows it). The notions of *free* and *bound* occurrences of a variable in an expression are defined as usual. An occurrence of X is *guarded* in E if it occurs within subexpression aF of E. A variable is guarded in E if all its occurrences in E are guarded.

We write $E\{E_1/X_1, \cdots E_n/X_n\}$ for the expression obtained by simultaneous substitution of E_i for each free occurrence of X_i in E, renaming bound variables as necessary. The definition is standard and is omitted.

An expression can evolve to another expression by performing an action. We write $E \xrightarrow{a} E'$ if E can evolve to E' by performing action a. The definition of the transition relation $\xrightarrow{a}$ is provided by the following inference rules:

Definition 1 $E \xrightarrow{a} E'$ *if it can be shown by the following inference rules:*

1. $aE \xrightarrow{a} E$

2. $\dfrac{E_1 \xrightarrow{a} E_2}{E_1 + E \xrightarrow{a} E_2} \qquad \dfrac{E_1 \xrightarrow{a} E_2}{E + E_1 \xrightarrow{a} E_2}$

3. $\dfrac{E\{\mu X.E/X\} \xrightarrow{a} E_1}{\mu X.E \xrightarrow{a} E_1}$

Charts generalize automata and are defined as follows:

Definition 2 *A* chart C *is a quadruple* $< Q,\ s,\ D,\ E >$ *where:*

$$\begin{array}{c} Q \textit{ is a nonempty set (the nodes)} \\ s \in Q \textit{ (the initial state)} \\ D \subset Q \times Act \times Q \textit{ (the derivations)} \\ Ex \subset Q \times Var \textit{ (the extensions)} \end{array}$$

C *is* finite *if* Q, D, Ex *are finite.*

With μ-expression E the chart (notations $Chart(E)$) is associated as follows:
Nodes: all μ-expressions.
The initial node: E.

$(E_1, a, E_2) \in D$ if $E_1 \xrightarrow{a} E_2$.
$(E, X) \in Ex$ if there is a free occurrence of X in E, which is not guarded.

Let a be an action and s, s' be states of chart C; we shall write $s \xrightarrow{a} s'$ if $(s, a, s') \in D$.

Charts are too concrete objects. Usually a behavior equivalence $\sim$ is introduced on charts and the objects are $\sim$-equivalences classes of charts. One of the most important equivalences studied in concurrency is bisimulation equivalence $\sim_{bis}$. Milner showed

Fact 2.1 *(μ-expressions represent finite behaviors)*

1. *For every finite chart C there exists an expression E such that $Chart(E) \sim_{bis} C$.*

2. *For every expression E there exists a finite chart C such that $Chart(E) \sim_{bis} C$.*

Bisimulation equivalence is a very discriminating equivalence and most equivalences studied in the literature are coarser than it. Clearly, fact 2.1 holds when $\sim_{bis}$ is replaced by any equivalence which is coarser than bisimulation. All these justify the use of an adjective 'finite state behavior' with μ-expressions.

3 Trace Equivalence and Trace Congruence

In this section we define trace equivalence on charts. Then, we find a fully abstract refinement of trace equivalence wrt the operations: sum, prefixing, substitution and recursion.

Definition 3 *(Traces and generalized traces.) Let C be a chart and s_0 be its initial state.*

- *A trace of C is a sequence $a_1 \dots a_n$ of actions such that there exist nodes $s_1 \dots s_n$ in C and $s_{i-1} \xrightarrow{a_i} s_i$ for $i = 1 \dots n$.*

- *A generalized trace of chart C is a pair consisting of a sequence $a_1 \dots a_n$ of action and a variable X such that there exist nodes $s_1 \dots s_n$ in C. and $s_{i-1} \xrightarrow{a_i} s_i$ for $i = 1 \dots n$ and $X \in Ex(s_n)$.*

Remark: (1) The set of traces of chart C is a subset of Act^*; the set of generalized traces of C is a subset of $Act^* \times Var$. (2) The set of traces of C is prefix closed. i.e., if $a_1 \dots a_n$ is a trace of C then for every $m \le n$ its prefix $a_1 \dots a_m$ is a trace of C. In particular, every chart contains the empty trace ϵ.

We say that expressions E and E' are trace equivalent if their corresponding charts have the same set of traces. For an expression E, we denote by $trace(E)$ the set of traces of the corresponding chart.

Lemma 3.1 *(Operations on traces)*

1. *$trace(0) = \{\epsilon\}$ - the empty string.*
2. *$trace(X) = \{\epsilon\}$ - the empty string.*
3. *$trace(E + E') = trace(E) \cup trace(E')$.*
4. *$trace(aE) = \{as \ : \ s \in trace(E)\}$.*

Unfortunately, trace equivalence is not a congruence wrt substitution and the μ operator. For example, 0 and X have the same traces, but $0\{a0/X\}$ and $X\{a0/X\}$ have different traces. We are looking for the maximal equivalence which refines trace equivalence and is a congruence wrt prefixing, sum, substitution and fixed point. Such an equivalence is called the **fully abstract** refinement of traces wrt the above operations.

Theorem 3.2 *(Full abstractness.) The fully abstract refinement of trace equivalence with respect to sum, prefixing, substitution and recursion is characterized as follows: E and E' are equivalent iff they have the same set of traces and the same set of generalized traces.*

We will use the term 'trace congruence' (notation $\sim_{trace}$) for this fully abstract equivalence.

Proof: The proof that this equivalence is a congruence is quite straightforward and is omitted.

To show that it is fully abstract, assume that the set of variables which are *free in* E and E' is a subset of $\{X_1, \ldots X_n\}$. Let $a_1 \ldots a_n$ be different actions which do not appear in E, E'. Then $E\{a_1 0/X_1 \ldots a_n 0/X_n\}$ and $E'\{a_1 0/X_1 \ldots a_n 0/X_n\}$ are trace equivalent only if E and E' have the same set of generalized traces. Therefore, if E and E' are related by a congruence $\sim$ which refines trace equivalence, they have the same set of generalized traces. Since $\sim$ refines trace equivalence, E and E' should also have the same set of traces.

To summarize, $\sim_{trace}$ is a congruence and any congruence $\sim$ which refines trace equivalence should refine $\sim_{trace}$. Therefore, $\sim_{trace}$ is the fully abstract refinement of trace equivalence. □

Remark: (1) Trace congruence is also the fully abstract refinement of trace equivalence wrt to the operation prefix, sum and recursion. (2) If every node of charts C, C' has the empty extension, then C and C' are trace congruent iff they are trace equivalent. In particular, trace congruence and trace equivalence coincide for μ-expressions without free variables.

4 Proof System

Our proof system for trace congruence in addition to the standard equivalence and congruence inference rules has the following axioms and fixed point inference rule.

Axioms
S1 $E + F = F + E$
S2 $(E + F) + G = E + (F + G)$
S3 $E + E = E$

S4 $E + 0 = E$
R1 $\mu X.\ E = \mu Y.(E\{Y/X\})$, Y not free in E
R2 $\mu X.\ E = E\{\mu X.\ E/X\})$
R3 $\mu X.\ E = \mu X.\ (E + X)$
P1 $aX + aY = a(X + Y)$

Fixed point Inference Rule
R4 From $E = F\{E/X\}$, X guarded in F, infer $E = \mu X.\ F$.

The set of axioms *S1-S4* is sound and complete for bisimulation equivalence over the recursive free subset of μ-expressions [1]. It is well known that by augmenting this set by *P1* a sound and complete system is obtained for trace equivalence over the variable free subset of μ-expressions. Moreover, it is not difficult to check that *S1-S4, P1* are also sound and complete for trace congruence over the recursive free subset of μ-expressions.

The main result of [2] is that the set of axioms *S1-S4, R1-R3* and fixed point inference rule *R4* is a sound and complete system for bisimulation equivalence. Our system is obtained by augmenting this set by *P1*; we will show that it is sound and complete for trace congruence.

Notations: We write $\vdash E = E'$ if $E = E'$ is provable in our system. We write $\vdash_M E = E'$ if $E = E'$ is provable without using axiom *P1*, i.e. it is provable in Milner's system.

5 Systems of Equations

This section contains some definitions which are needed for the completeness proof given in the next section.

Definition 4 *(an $(\tilde{X};\ \tilde{Y})$ system of equations) Let $\tilde{X} = \{X_1, \ \ldots X_n\}$, $\tilde{Y} = \{Y_1, \ \ldots Y_m\}$ be different variables and $\tilde{F} = \{F_1 \ldots F_n\}$ expressions with free variables in $\tilde{X} \cup \tilde{Y}$. A sequence of formal equations $Sys := < X_1 = F_1, \ldots X_n = F_n >$ is called an $(\tilde{X};\ \tilde{Y})$ system of equations; $\tilde{X}$ are called the bound variables of the system and X_1 is the principal variable of the system.*

Definition 5 *(Guarded, standard and deterministic systems)*

- *An $(\tilde{X};\ \tilde{Y})$ system is* guarded *if all $\tilde{X}$ variables are guarded in the expressions F_i of the system.*
- *A system is* standard *if all F_i are of the form $\sum_{j \in I_i} a_{i,j} X_{f(i,j)} + \sum_{j \in K_i} Y_{g(i,j)}$.*
- *A standard system is called* deterministic *if $a_{i,j} = a_{i,j'}$ implies that $j = j'$.*

Note that standard and deterministic systems of equations are guarded.

Definition 6 *(Solutions of a system) Consider an $(X_1 \ldots X_n : Y_1 \ldots Y_m)$ system Sys with equations $X_i = F_i$, for $i = 1, \ldots n$.*

- *A sequence E_1, E_2,... E_n of expressions is a solution of Sys iff for every i $E_i \sim_{trace} F_i\{E_1/X_1, \ldots E_n/X_n\}$; expression E_1 is called a principal solution of the system.*
- *E_1, E_2,... E_n is an M-provable solution of Sys iff $\vdash_M E_i = F_i\{E_1/X_1, \ldots E_n/X_n\}$ for every i; expression E_1 is called a principal M-provable solution of the system.*
- *E_1, E_2,... E_n is a provable solution of Sys iff $\vdash E_i = F_i\{E_1/X_1, \ldots E_n/X_n\}$ for every i; expression E_1 is called a principal provable solution of the system.*

6 Soundness and Completeness

Theorem 6.1 *(Soundness) The axioms and the inference rules are sound for trace congruence.*

As usual, the proof of soundness theorem is simple and we concentrate here on the completeness proof. Our completeness proof is based on Milner's completeness proof for bisimulation equivalence [2]. Many of our arguments are modifications of his ideas.

The following theorem was proved by Milner (theorem 5.7 in [2]).

Theorem 6.2 *(Unique M-provable solution of equations.) Every guarded ($\tilde{X}$; $\tilde{Y}$) system of equations Sys has a unique M-provable solution, i.e., Sys has an M-provable solution and if both $E_1 \cdots E_n$ and $E'_1 \cdots E'_n$ are M provable solutions of Sys then $\vdash_M E_i = E'_i$ for $i = 1, \cdots, n$.*

In [2] it was shown (theorem 5.8)

Theorem 6.3 *(Equational characterization of μ-expressions.) Every μ-expression is a principal M-provable solution of a standard system of equations.*

M-provable equations are provable, therefore,

Corollary 6.4 *Every μ-expression is a principal provable solution of a standard system of equations.*

We strengthen the corollary and show

Theorem 6.5 *Every μ-expression is a principal provable solution of a deterministic system of equations.*

Proof: The proof is given in appendix A. □

Our proof of the completeness theorem uses the following

Claim 6.6 *Assume (1) $E \sim_{trace} E'$ (2) E is a provable principal solution of deterministic system of equations Sys (3) E' is a provable principal solution of deterministic system of equa-*

tions Sys'. Then there exists a guarded system Sys'' such that both E and E' are its principal provable solutions.

Proof: The proof is given in appendix B. □

Theorem 6.7 *(Completeness) If $E \sim_{trace} E'$ then $\vdash E = E'$.*

Proof: Actually, Milner's proof of theorem 6.2 also gives the *unique provable solution theorem*: every guarded system of equations has a unique provable solution.

Now, by theorem 6.5 there exist deterministic systems of equations Sys and Sys' such that E and E' are principal provable solutions of Sys and Sys' respectively.

Therefore, by claim 6.6, there exists a guarded system of equations Sys'' such that both E and E' are its principal provable solutions. Therefore, by the unique provable solution theorem, $\vdash E = E'$. □

7 Relationship to Salomaa's axiomatisation

In [4] Salomaa presented a complete axiom system for language equivalence over regular expressions. Trace congruence is conceptually close to language equivalence. Our proof has the same structure as Salomaa's (unique provable solution theorem. equational characterization theorem), but in many technical arguments it is closer to Milner's proof for bisimulation equivalence and surprisingly, we were unable to adopt Salomaa's proof for a complete axiomatisation of trace congruence.

The main obstacle for extending Salomaa's proof lies in the fact that the empty language is present explicitly in his axiomatisation and plays a very important role there. However, for trace congruence there is no expression which 'corresponds' to the empty language. In particular, unlike the language law $b\emptyset = \emptyset = a\emptyset$, there exists no expression E for which aE is trace congruent to bE.

In the rest of this section we consider an interesting subset of regular expressions. Adopting Salomaa's results, we provide an axiomatisation of bisimulation equivalence and an axiomatisation of trace congruence over this subset. The proofs are omitted and will be given in the full paper.

7.1 #-Expressions

In remark 5 of [4] it is explained how to axiomatise 'regular expressions' without the empty language.

More exactly, Salomaa considers expressions constructed from an alphabet by the following operations: concatenation, sum and positive iteration # ($A^{\#} = A + AA + AAA + \ldots$). Let us call such expressions - #-expressions. It is claimed there that the following proof system is complete for #-expressions:

Axioms:

A1:	Associativity of sum.
A2:	Associativity of concatenation.
A3:	Commutativity of sum.
A4:	$A(B+C) = AB + AC$.
A5:	$(A+B)C = AB + AC$.
A6:	$A + A = A$.
A'10:	$A^{\#} = A + AA^{\#}$

Inference rules:

R1:	Usual Congruence rules.
R2:	$A = BA + C$ implies $A = B^{\#}C + C$.

7.2 Embedding of #-expressions into μ-expressions

Similar to Milner's [2] embedding of the standard regular expressions into μ-expressions one can embed #-expressions into μ-expressions.

Let X be a distinguished variable. Define:

1. $Em(a) = aX$ for every action a.
2. $Em(A+B) = Em(A) + Em(B)$.
3. $Em(AB) = Em(A)\{Em(B)/X\}$.
4. $Em(A^{\#}) = \mu Y.A + A\{Y/X\}$

Remark 1. In the chart of $Em(A)$ the nodes may have either an extension X or the empty extension. From every node, a node with extension X is reachable. The initial node has the empty extension. Such a chart can be considered as an automaton whose accepting states are the nodes with extension X. The language accepted by this automaton coincides with the language defined by expression A.

Remark 2. (Axiomatisation of trace congruence over #-expressions.) This embedding is adequate for trace congruence, i.e., $A = B$ is provable in Salomaa's axiomatisation iff $Em(A) = Em(B)$ is provable in our axiomatisation of trace congruence.

7.3 Axiomatisation of #-expression wrt bisimulation

Bisimulation equivalence, due to Milner and Park, is one of the most fundamental equivalences that has emerged in concurrency. Its definition is omitted here, but let us note that axiom A'10 holds even for bisimulation equivalence, i.e. $Em(A^{\#})$ is bisimilar to $Em(A) + Em(A)Em(A^{\#})$.

It is also easy to see that the axioms A1, A2, A3, A5, A6 and the inference rules R1, R2 are valid for bisimulation, but axiom A4 fails.

We do not know whether the set A1, A2, A3, A5, A6, A'10, R1, R2 is complete for bisimulation equivalence over #-expressions. One can show that there exists no #-expression which solves equation $Y =_{bis} aY + b$. We do not know how to characterize the systems of equations with the solutions bisimilar to #-expressions.

However, we can prove that a complete proof system for bisimulation equivalence is obtained when R2 is replaced by the following inference rule:

R3: Every system of equations of the form $Y_i = \sum a_{i,j} Y_{f(i,j)} + \sum b_{i,j}$ has at most one solution.

8 Further Results

(a) Axiomatisation of trace *approximation* - straightforward.

(b) τ -action. The addition of axiom $E = \tau E$ provides a complete axiomatisation of trace congruence with unobservable action τ.

(c) *Divergence.* Trace congruence identifies expressions 0 and $\mu X.\, X$. We can introduce the notion of divergence and provide a complete axiomatisation for the refinement of trace congruence which properly takes into account divergence.

Acknowledgments

Our completeness proof is based on Milner's proof [2]. I am grateful to Robin Milner for his remarks on the draft of the paper. The results of section 7 were inspired by these comments.

References

[1] M. Hennesy and R. Milner. Algebraic laws for nondeterminism and concurrency. In *J. Assoc. Comp. Mach.* , volume 32, pp. 137-161, 1985.

[2] R. Milner. A complete proof system for a class of regular behaviors. In *JCSS* , volume 28, pp. 439-466, 1984.

[3] R. Milner. A complete axiomatisation for observational Congruence of finite state behaviors. In *Information and Computation*, volume 81. pp. 227-247, 1989.

[4] A. Salomaa. Two complete axiom systems for the algebra of regular events. In *J. Assoc. Comp. Mach.*, volume 13(1) pp. 158-169. 1965.

Appendix

The appendix contains two sections. In section A theorem 6.5 is proved; in section B claim 6.6 is proved.

A Determinization

In this appendix we show that every expression is a principal provable solution of a deterministic system of equations. This fact follows immediately from the corollary 6.4 and the following

Claim A.1 *If E_1 is a principal provable solution of a standard system of equations, then E_1 is a principal provable solution of a deterministic system of equations.*

First, we introduce some notations which are helpful in the proof of the claim.

Let $X_i = \sum_{j\in J_i} a_{i,j}X_{f(i,j)} + \sum_{k\in K_i} Y_{g(i,k)}$ be an equation.

Define

$S(i) = \{a_{i,j} : j \in J_i\}$ - successor actions of X_i.
$D(i,a) = \{r : aX_r \textit{ is a summand of the equation}\}$ - a-derivatives of X_i.
$Ex(i) = \{r : Y_r \textit{ is a summand of the equation}\}$ - extensions of X_i.

Extend point-wise these functions to the subsets of $\{1, \ldots p\}$, i.e.

$S(Q) = \cup_{i\in Q} S(i)$,
$D(Q,a) = \cup_{i\in Q} D(i,a)$,
$Ex(Q) = \cup_{i\in Q} Ex(i)$.

Now, if $X_i = \sum_{j\in J_i} a_{i,j}X_{f(i,j)} + \sum_{k\in K_i} Y_{g(i,k)}$ is an equation in our system, it can be easily shown, by the axioms for sum, that

$$\vdash_M \sum_{j\in J_i} a_{i,j}X_{f(i,j)} + \sum_{k\in K_i} Y_{g(i,k)} = \sum_{a\in S(i)}(\sum_{r\in D(i,a)} aX_r) + \sum_{r\in Ex(i)} Y_r.$$

Therefore, every standard system of equations is $\vdash_M$ equivalent to a system with equations of the form $X_i = \sum_{a\in S(i)}(\sum_{r\in D(i,a)} aX_r) + \sum_{r\in Ex(i)} Y_r$. From now on, we will only consider such systems.

Now we are ready to start the proof of claim A.1.

Since E_1 is a principal provable solution of a standard system of equations there exist $E_2, \ldots E_p$ and a system of equations:

$$Sys = \begin{cases} X_1 &= \sum_{a\in S(1)}(\sum_{r\in D(1,a)} aX_r) + \sum_{r\in Ex(1)} Y_r \\ \vdots & \vdots \qquad\qquad \vdots \\ X_i &= \sum_{a\in S(i)}(\sum_{r\in D(i,a)} aX_r) + \sum_{r\in Ex(i)} Y_r \\ \vdots & \vdots \qquad\qquad \vdots \\ X_p &= \sum_{a\in S(p)}(\sum_{r\in D(p,a)} aX_r) + \sum_{r\in Ex(p)} Y_r \end{cases}$$

such that for every i

$$\vdash E_i = \sum_{a\in S(i)} (\sum_{r\in D(i,a)} aE_r) + \sum_{r\in Ex(i)} Y_r \tag{1}$$

We are going to define a deterministic system of equations Sys' and to show that E_1 is its principal solution. The bound variables of Sys' are indexed by the subsets of $\{1,\ldots p\}$; X_1 is its principal variable. For a set $Q \subset \{1,\ldots p\}$ the equation for X_Q is:

$$X_Q = \sum_{a\in S(Q)} aX_{D(Q,a)} + \sum_{r\in Ex(Q)} Y_r$$

It is clear that this is a deterministic system.

Define $E_Q \equiv \sum_{i\in Q} E_i$. We claim that $\vdash E_Q = \sum_{a\in S(Q)} aE_{D(Q,a)} + \sum_{r\in Ex(Q)} Y_r$.

Note that

$$\vdash E_{Q_1} + E_{Q_2} = E_{Q_1\cup Q_2} \tag{2}$$

Recall that by the assumption of the claim

$$\vdash E_i = \sum_{a\in S(i)} (\sum_{r\in D(i,a)} aE_r) + \sum_{r\in Ex(i)} Y_r \tag{3}$$

Therefore, adding the equations for $i \in Q$

$$\vdash \sum_{i\in Q} E_i = \sum_{i\in Q}(\sum_{a\in S(i)} (\sum_{r\in D(i,a)} aE_r) + \sum_{r\in Ex(i)} Y_r) \tag{4}$$

The left-hand side is E_Q by the definition of E_Q. The right-hand side can be rearranged, by the axioms for $+$, and gives

$$\vdash E_Q = \sum_{a\in \cup_{i\in Q} S(i)} (\sum_{r\in \cup_{i\in Q} D(i,a)} aE_r) + \sum_{r\in \cup_{i\in Q} Ex(i)} Y_r \tag{5}$$

By the idempotence of $+$, and the fact that S, D. Ex are extended point-wise to sets we obtain:

$$\vdash E_Q = \sum_{a\in S(Q)} (\sum_{r\in D(Q,a)} aE_r) + \sum_{r\in Ex(Q)} Y_r \tag{6}$$

Now applying the prefix axiom, $aX + aY = a(X + Y)$, we derive

$$\vdash E_Q = \sum_{a\in S(Q)} a \sum_{r\in D(Q,a)} E_r + \sum_{r\in Ex(Q)} Y_r \tag{7}$$

By the definition, $E_{D(Q,a)} \equiv \sum_{i\in D(Q,a)} E_i$, therefore

$$\vdash E_Q = \sum_{a \in S(Q)} aE_{D(Q,a)} + \sum_{r \in Er(Q)} Y_r \tag{8}$$

Hence, the equations obtained by substitution of E_Q for X_Q in the equations of Sys' are provable. In particular, E_1 is a principal provable solution of Sys'.

B Proof of Claim 6.6

In this section we prove claim 6.6. We recall claim 6.6.

Claim B.1 *Assume (1) $E_1 \sim_{trace} E_1'$ (2) E_1 is a provable principal solution of deterministic system of equations Sys (3) E_1' is a provable principal solution of deterministic system of equations Sys'. Then there exists a guarded system Sys'' such that both E_1 and E_1' are its principal provable solutions.*

The proof is based on

Lemma B.2 *Assume that (1) $E \sim_{trace} G$ (2) $E \sim_{trace} \sum_{i \in I} a_i E_i + \sum_{k \in K} Y_k$ and (3) $G \sim_{trace} \sum_{j \in J} b_j G_j + \sum_{m \in M} Z_m$, where a_i are different actions and b_j are different actions. Then (a) $\{Y_k : k \in K\} = \{Z_m : m \in M\}$; (b) $|I| = |J|$ and there exists a one-one function $h : I \to J$ such that $a_i = b_{h(i)}$ and $E_i \sim_{trace} G_{h(i)}$. (c) $G \sim_{trace} \sum_{i \in I} a_i G_{h(i)} + \sum_{k \in K} Y_k$ and $\vdash \sum_{j \in J} b_j G_j + \sum_{m \in M} Z_m = \sum_{i \in I} a_i G_{h(i)} + \sum_{k \in K} Y_k$.*

Proof: The initial node s_0 of the chart for $\sum_{i \in I} a_i E_i + \sum_{k \in K} Y_k$ has the extension $\{Y_k : k \in K\}$ and its set of successor actions is $\{a_i : i \in I\}$. The initial node s_0' of the chart for $\sum_{j \in J} b_J G_j + \sum_{m \in M} Z_k$ has the extension $\{Z_m : m \in M\}$ and its set of successor actions is $\{b_j : j \in J\}$. In equal charts the initial nodes have the same extension and the same set of successor actions, therefore $\{Y_k : k \in K\} = \{Z_m : m \in M\}$ and $\{a_i : i \in I\} = \{b_j : j \in J\}$. Moreover, since all a_i are different and all b_j are different the relation $R(i, j) =_{def} a_i \equiv b_j$ is a graph of a one-one function h between sets I and J. In particular, using associativity and commutativity of $+$, the equation for G can be rewritten into $G \sim_{trace} \sum_{i \in I} a_i G_{h(i)}) + \sum_{m \in M} Z_m$ and $\vdash \sum_{j \in J} b_j G_j + \sum_{m \in M} Z_m = \sum_{i \in I} a_i G_{h(i)}) + \sum_{k \in K} Y_k$.

In the case when all actions a_i are different, it can be easily shown that $(a_i s, X)$ is a generalized trace of $\sum_{i \in I} a_i E_i + \sum_{k \in K} Y_k$ iff (s, X) is a generalized trace of E_i. Similarly, $(a_i s, X)$ is a generalized trace of $\sum_{i \in I} a_i G_{h(i)} + \sum_{k \in K} Y_k$ iff (s, X) is a generalized trace of $G_{h(i)}$. Since $\sum_{i \in I} a_i E_i + \sum_{k \in K} Y_k$ and $\sum_{i \in I} a_i G_{h_i} + \sum_{k \in K} Y_k$ have the same set of generalized traces, it follows that E_i and $G_{h(i)}$ have the same set of generalized traces. Similar arguments show that E_i and $G_{h(i)}$ have the same set of traces. Hence $E_i \sim_{trace} G_{h(i)}$. □

Now we proceed with the proof of claim B.1. Let $E_1, \ldots E_p$ be a provable solution of a deterministic system of equations:

$$\vdash E_i = \sum_{j \in J_i} a_{i,j} E_{f(i,j)} + \sum_{j \in K_i} Y_{g(i,j)} \quad i = 1, \ldots p. \tag{9}$$

Let $E'_1, \dots E'_m$ be a provable solution of deterministic system of equations:

$$\vdash E'_{i'} = \sum_{j \in J'_{i'}} b_{i',j} E'_{f'(i',j)} + \sum_{j \in K'_{i'}} Y'_{g'(i',j)} \quad i' = 1, \dots m. \tag{10}$$

All provable equations are valid, therefore in the above equations = can be replaced by $\sim_{trace}$. Define set $R = \{(i, i') : E_i \sim_{trace} E'_{i'}\}$. R is not empty, because $(1,1) \in R$.

For $(i, i') \in R$, all the assumptions of lemma B.2 hold, therefore there exists a one-one function $h_{i,i'} : J_i \to J'_{i'}$ such that $a_{i,j} = b_{i',h_{i,i'}(j)}$ and $E_{f(i,j)} \sim_{trace} E'_{f'(i',h_{i,i'}(j))}$.

Consider the following system Sys'' of equations with bound variables indexed by the elements of R. The equation for $X_{i,i'}$ is:

$$X_{i,i'} = \sum_{j \in J_i} a_{i,j} X_{f(i,j), f'(i',h_{i,i'}(j))} + \sum_{j \in K_i} Y_{g(i,j)} \tag{11}$$

Note that all bound variables in this system are indexed by elements of R. Indeed $E_{f(i,j)} \sim_{trace} E'_{f'(i',h_{i,i'}(j))}$ by the conclusion (c) of lemma B.2. Therefore, the pair $(f(i,j),\ f'(i', h_{i,i'}(j)))$ is in R.

Define a sequence of expressions indexed by elements of R: $E_{i,i'} \equiv E_i$. We claim that when $E_{i,i'}$ are substituted for $X_{i,i'}$ in Sys'', then the resulting equations are provable. Indeed, after substitution, equations (i, i') become

$$E_{i,i'} = \sum_{j \in J_i} a_{i,j} E_{f(i,j), f'(i',h_{i,i'}(j))} + \sum_{j \in K_i} Y_{g(i,j)} \tag{12}$$

and by the definition of $E_{i,i'}$ it is equivalent to

$$E_i = \sum_{j \in J_i} a_{i,j} E_{f(i,j)} + \sum_{j \in K_i} Y_{g(i,j)} \tag{13}$$

which by our assumption is provable. Therefore (12) is also provable.

Now define another sequence of expressions indexed by elements of R: $E'_{i,i'} \equiv E'_{i'}$. Again we claim that when $E'_{i,i'}$ are substituted for $X_{i,i'}$ in Sys'' then the resulting equations are provable. Indeed, after substitution, the equations for $X_{i,i'}$ become

$$E'_{i,i'} = \sum_{j \in J_i} a_{i,j} E'_{f(i,j), f'(i',h_{i,i'}(j))} + \sum_{j \in K_{i'}} Y_{g(i,j)} \tag{14}$$

and by the definition of $E'_{i,i'}$ it is provable iff

$$\vdash E'_{i'} = \sum_{j \in J_i} a_{i,j} E'_{f'(i',h_{i,i'}(j))} + \sum_{j \in K_i} Y_{g(i,j)} \tag{15}$$

By (c) of the lemma $\vdash \sum_{j \in J'_{i'}} b_{i',j} E'_{f'(i',j)} + \sum_{j \in K'_{i'}} Y'_{g'(i,j)} = \sum_{j \in J_i} a_{i,j} E'_{f'(i',h_{i,i'}(j))} + \sum_{j \in K_i} Y_{g(i,j)}$ therefore (14) is provable iff the following is provable:

$$\vdash E'_{i'} = \sum_{j \in J'_{i'}} b_{i',j} E'_{f'(i',j)} + \sum_{j \in K'_{i'}} Y'_{g'(i',j)} \quad i' = 1, \dots m. \tag{16}$$

However, this is provable by our assumption (see equation (10) above). Hence, both $E_{1,1}$ and $E'_{1,1}$ are principal provable solutions of Sys''. But, by the definition $E_{1,1} \equiv E_1$ and $E'_{1,1} \equiv E'_1$. Therefore E_1 and E'_1 are principal provable solutions of Sys''.

Finally, note that Sys'' is a guarded system and its size is polynomial in the sizes of Sys and Sys'.

THE ASYMMETRIC TOPOLOGY OF COMPUTER SCIENCE

R. C. Flagg, Mathematics
U. of Southern Maine, Portland, ME 04103

R. D. Kopperman, Mathematics
City Coll. of New York, New York, NY 10031

1. Continuity Spaces and Asymmetric Topology

Computer science tends to need analysis of "topological" situations in which orderings and generalized metrics, both symmetric and asymmetric, play a role. Many of its natural topologies are not T_1 (for definition see a text, such as [Ke]), for example:
the lower and Scott topologies on a continuous lattice and
the topology of digital n-space.

In fact, these spaces fail to satisfy the *weak symmetry* axiom:

$$x \in \mathrm{cl}\{y\} \Rightarrow y \in \mathrm{cl}\{x\}.$$

Here, as often in mathematics, the lack of symmetry leads to a straightforward duality. Some examples elsewhere in mathematics include:
For noncommutative rings and algebras, define $R^* = (R, +, \circ)$, where $a \circ b = ba$, (the same works for monoids and categories),
for partial orders $P^* = (P, \leq^{-1})$,
for quasimetrics, $d^*(x,y) = d(y,x)$,
for quasiuniformities, $\mathcal{Q}^* = \{Q^{-1} \mid Q \in \mathcal{Q}\}$.

The case of distances came to our attention because every topology is in a natural sense a generalized metric topology. Our generalized metric spaces, called continuity spaces, are given a leisurely, elementary discussion in [Kp]. For our use, the following will suffice:

1. Definition. $\mathcal{M} = (X, d, A, P)$ is a *continuity space* if X is any set, A a value semigroup (a generalization of $[0,\infty]$, defined in [Kp]), P a set of positives on A (generalizing $(0,\infty] \subseteq [0,\infty]$), and $d : X \times X \to A$ satisfies:

(m1) $d(x,x) = 0$,
(m2) $d(x,z) \leq d(x,y) + d(y,z)$.

The *dual* of d is d^*, defined by $d^*(x,y) = d(y,x)$; that of $\mathcal{M}$ is $\mathcal{M}^* = (X, d^*, A, P)$. The *symmetrization* of d is $d^S = d \vee d^*$; that of $\mathcal{M}$ is $\mathcal{M}^S = (X, d^S, A, P)$.

The *closed ball of radius a* is $N_a(x) = \{y \mid d(x,y) \leq a\}$, the *topology induced by* $\mathcal{M}$ is $\mathcal{T}_{\mathcal{M}} = \{T \mid x \in T \Rightarrow (\exists r \in P)(N_r(x) \subseteq T)\}$.

A continuity space is c_0 if $d(x,y) + d(y,x) = 0 \Rightarrow x = y$; it is c_1 if $d(x,y) = 0 \Rightarrow x = y$, and it is *symmetric* if for each x, y, $d(x,y) = d(y,x)$ (equivalently, if $d = d^*$). Finally, a continuity space is *Boolean* if for each $a \in A, a = a + a$.

The precise result is:

2. Theorem. [Kp]: Each continuity space yields a topology, and every topology arises from some continuity space.

This result was improved in [Fl], where it was shown that it suffices to consider certain special value semigroups called value quantales. Since the latter are cocontinuous lattices, they are complete and allow a straightforward completion theory for their continuity spaces; for these, the positives can be defined as the elements way above 0. We use only these restricted continuity spaces in section 2, when completeness becomes an issue, and for a fixed such value quantale V, denote $\mathcal{M} = (X, d_X)$ (since V, P are fixed).

Of course, by the notation introduced in 1, $\mathcal{T}_{\mathcal{M}^*}$ denotes the topology induced by the dual of $\mathcal{M}$, and $\mathcal{T}_{\mathcal{M}^s}$ that arising from the symmetrization. We call them the dual and symmetrization topologies, and below we use the simple-to-establish equation: $\mathcal{T}_{\mathcal{M}^s} = \mathcal{T}_{\mathcal{M}} \vee \mathcal{T}_{\mathcal{M}^*}$.

Certainly, a metric space is a symmetric c_0 continuity space in which $A = [0, \infty]$, $P = (0, \infty]$ and $d(x, y)$ is never ∞. For such, the induced topology is routinely seen to be a the usual metric topology.

The c_0 assumption is equivalent to the requirement that the induced topology be T_0. Another equivalent condition is that the symmetrization topology is Hausdorff, and thus that symmetrically convergent nets (or filters) have unique limits. These are desirable properties, and in section 2 many of our continuity spaces will be c_0. The c_1 assumption is equivalent to the requirement that the induced topology be T_1. This requirement is not satisfied by many spaces that interest us below, thus is not made.

For two reasons, a consideration of bitopological spaces is needed for an understanding of topological duality which provides a context for the above, and on which we base our approach (see [Ko], or, for alternate viewpoints, see [Lw], [Sm]):

(a) Bitopological spaces have an obvious duality: $(X, \mathcal{T}, \mathcal{T}^*)^* = (X, \mathcal{T}^*, \mathcal{T})$. Further, there is a natural identification of a topological space $(X, \mathcal{T})$ with the self-dual bitopological space $(X, \mathcal{T}, \mathcal{T})$. This self-duality means that many bitopological theorems involving this duality look like topological theorems.

(b) Any dual, $\mathcal{T}^*$, of a topology, $\mathcal{T}$, on the same set, X, must be recognized by its relationship with the original, and this relationship can certainly be stated as a property of a bitopological space.

Many useful such relationships "look like" separation axioms, or are related to compactness. Indeed, for each usual topological separation axiom T_i (see [Ke]) there is a duality-motivated bitopological separation axiom (see [Ko] or [Ky]).

For these, as usual: $T_i \Rightarrow T_j$ if $j < i$.

Further, a topological space $(X, \mathcal{T})$ satisfies T_i iff $(X, \mathcal{T}, \mathcal{T})$ does.

In addition, there is a compactness-related axiom in whose presence some of these implications are reversed (see [Ko]):

3. Definition. A bitopological space $\mathcal{X} = (X, \mathcal{T}, \mathcal{T}^*)$ is:

stable if each proper *-closed set is quasicompact,

quasicompact if so with respect to $\mathcal{T}$,

joincompact if $\mathcal{X}, \mathcal{X}^*$ are both quasicompact, stable and T_2.

A topological space $(X, \mathcal{T})$ is *acompact (short for asymmetrically compact Hausdorff)* if for some topology $\mathcal{T}^*$ on X, $(X, \mathcal{T}, \mathcal{T}^*)$ is joincompact (this $\mathcal{T}^*$ is unique).

4. Fact. (a) The lower and the Scott topology on a continuous lattice, are acompact, and the dual of each is the other. (Some of this can be found in [G&]; the rest in [Ko] and [H&].)

(b) If X is joincompact, then $(X, \mathcal{T} \vee \mathcal{T}^*)$ is a compact Hausdorff space. (That the Lawson topology is compact T_2 is a special case of (a).)

(c) A topology is compact T_2 $\Leftrightarrow$
it is acompact and T_1 $\Leftrightarrow$
it is acompact, T_0 and self-dual ($\mathcal{T}^* = \mathcal{T}$).

(d) For a T_0 bitopological space $\mathcal{X}$:
both $\mathcal{X}$ and $\mathcal{X}^*$ are $T_{3.5}$ $\Leftrightarrow$
there is a continuity space $\mathcal{M}$ for which $\mathcal{T}$ is $\mathcal{T}_{\mathcal{M}}$, and $\mathcal{T}^* = \mathcal{T}_{\mathcal{M}^*}$.

Remark: Each joincompact space is $T_{3.5}$, thus a continuity space exists as in (d). Lawson, in [La] asserted that for each continuous lattice L, there is a continuity space $\mathcal{M}$ valued in that lattice (that is, $A = L$), for which the lower topology is $\mathcal{T}_{\mathcal{M}}$ and the Scott topology is $\mathcal{T}_{\mathcal{M}^*}$.

Such a continuity space is Boolean. As a result, its symmetrization topology must be 0-dimensional (its $N_r^S(x)$ are simultaneously closed and open by [Kp], proposition 10).

But in this case, $\mathcal{T}_{\mathcal{M}^S} = \mathcal{T}_{\mathcal{M}} \vee \mathcal{T}_{\mathcal{M}^*}$ must be the Lawson topology, which is often not 0-dimensional (eg., on the unit interval), a contradiction. In [FK] we show that the Lawson assertion (and his proof) holds for algebraic cpo's, and give a related characterization for all continuous cpo's.

2. Examples of Categories of Domains

In the remainder of this paper we investigate how continuity spaces can be used to construct 'convenient categories for denotational semantics'. By exploiting the possible asymmetry of the distance function on a continuity space, we are able to include important aspects of both the metric space and cpo approaches to denotational semantics. Moreover, we provide new examples which may be suitable for modelling language constructs that occur in concurrent and probabilistic programming. There are a number of important issues still to be resolved before the theory presented here can be considered satisfactory.

By a *net* in X we understand a family of elements $(x_\lambda)_{\lambda\in\Lambda}$ of X indexed by a directed set $(\Lambda, \leq)$. A net $(x_\lambda)_{\lambda\in\Lambda}$ is *Cauchy* if for every $\epsilon >> 0$ there exists a λ_0 such that for all $\mu, \nu \geq \lambda_0$, $\epsilon \geq d(x_\mu, x_\nu)$. X is *complete* if every Cauchy net has a limit in the symmetric topology on X.(1)

For Cauchy nets $(x_\lambda)_{\lambda\in\Lambda}$, $(y_\lambda)_{\lambda\in\Lambda}$, we need the equation:

$$d^S(\lim x_\lambda, \lim y_\lambda) = \lim d^S(x_\lambda, y_\lambda).$$

The right-hand side requires the existence of well-behaved limits in the value semigroup, so henceforth we restrict ourselves to a class of value semigroups introduced in [Fl].

5. Definition. A *value quantale* $< V, \leq, + >$ is a complete distributive lattice $< V, \leq >$ together with a binary operation $+$ such that the following conditions are satisfied:

(vq1) $< V, +, 0 >$ is a commutative monoid;
(vq2) for all $p \in V$ and all $S \subseteq V$, $p + \bigwedge S = \bigwedge_{s\in S}(p + s)$;
(vq3) for all $p \in V$, $p = \bigwedge\{q \in V \mid q >\!\!>^{\cdot} p\}$.

In (vq3) we have written $q >> p$ to indicate that q is *way above* p; that is, for any subset $W \subseteq V$, if $p \geq \bigwedge W$, then for some finite $F \subseteq W$, $q \geq \bigwedge F$. Thus (vq3) is equivalent to the requirement that V^{op} be a continuous lattice.

Now let $< V, \leq, + >$ be a value quantale. By a V-continuity space, (X, d_X) we mean a continuity space (X, d_X, A, P) for which $A = V$ and $P = V^+ = \{p \in V \mid p >> 0\}$. A value quantale V can itself be regarded as a complete V-continuity space, with $d_V(x, y) = x \dot{-} y$, where $x \dot{-} y = \bigwedge\{z \mid x \leq y + z\}$ (see [Fl]). (Notice that by this definition, $\dot{-}$ left adjoint to $+$: that is, $a - c \leq b \Leftrightarrow a \leq b + c$.) We call X a *V-domain* if $X = (X, d_X)$ is a complete, c_0 V-continuity space.

Each V-continuity space X has a completion; that is, there is a V-domain $X^\sim$ and an isometric map $\iota : X \to X^\sim$ with the following universal property: for any V-domain Y and

(1) For each $\epsilon >> 0$, let $D_\epsilon = \{< x, y > \in X \times X \mid \epsilon >> d^S(x, y)\}$. The family $\{D_\epsilon\}_{\epsilon>>0}$ is a base for a uniformity, $\mathcal{V}^S$, on X and the uniform topology generated by $\mathcal{V}^S$ is the symmetric topology on X. X is complete in the sense just described iff the uniform space $(X, \mathcal{V}^S)$ is complete.

any nonexpansive mapping $f : X \to Y$ there is a unique nonexpansive map $f^{\sim} : X^{\sim} \to Y$ such that $f = f^{\sim} \circ \iota$. Notice that ι is one-one iff X is c_0; in general, $\iota(x) = \iota(y) \Leftrightarrow d(x,y) + d(y,x) = 0$. $X^{\sim}$ has a number of additional properties that we will need below.

(11) for every $\alpha \in X^{\sim}$, there is a Cauchy net $(x_\epsilon)_{\epsilon>>0}$ in X such that $\alpha = \lim_\epsilon \iota(x_\epsilon)$.

(12) For $\alpha, \beta \in X^{\sim}$, if $\alpha = \lim_\epsilon \iota(x_\epsilon)$ and $\beta = \lim_\epsilon \iota(y_\epsilon)$, then $(d(x_\epsilon, y_\epsilon))_\epsilon$ is a Cauchy net in V and $d_{\sim}(\alpha,\beta) = \lim_\epsilon d(x_\epsilon, y_\epsilon)$.

Let V-**Dm** denote the category with objects the V-domains and morphisms the nonexpansive maps between such.

SOME EXAMPLES

Scott Domains. In the classical approach to domain theory, introduced by Scott and Strachey [SSt], a domain is a certain type of complete partially ordered set (cpo). We illustrate how this approach can be included in our general theory by considering two examples: algebraic cpo's and continuous cpo's.

Let K be a set. Then the power set of K, $\mathcal{P}(K)$, is a value quantale with $+ = \cup$. We make a cpo A into a $\mathcal{P}(K)$-continuity space $\mathcal{A}_\mathcal{P} = (A, d_\mathcal{P})$, where K is the set of compact elements of A, by defining $d_\mathcal{P}(x,y) = (\downarrow(x) \cap K) \setminus (\downarrow(y) \cap K)$, for $x, y \in A$.

6. Theorem. ([FK]) Assume A is an algebraic cpo. Then $\mathcal{A}_\mathcal{P}$ is c_0, the induced topology on $\mathcal{A}_\mathcal{P}$ is the Scott topology on A, the dual topology on $\mathcal{A}_\mathcal{P}$ is the lower topology on A, and the symmetric topology on $\mathcal{A}_\mathcal{P}$ is the Lawson topology on A.

Notice that $\mathcal{P}(K)^+ = \{r \mid r \text{ is a cofinite subset of } K\}$. As a result, the continuity space $\mathcal{A}_\mathcal{P}$ is ***totally bounded***; that is, for all $\epsilon >> 0$ there is a finite subset $\{a_1, a_2, \ldots, a_n\}$ of A such that $A = N^s_\epsilon(a_1) \cup N^s_\epsilon(a_2) \cup \ldots \cup N^s_\epsilon(a_n)$. Thus $\mathcal{A}_\mathcal{P}$ is an $\mathcal{P}(K)$-domain iff the Lawson topology on A is compact. This condition is equivalent to the requirement that A be a '2/3-SFP' domain (see [Pl]). This is the case if A is bounded complete. In particular, for a Scott domain (i.e., an ω-algebraic, bounded complete cpo) A, $\mathcal{A}_\mathcal{P}$ is an $\mathcal{P}(K)$-domain.

Note that a Scott continuous map $f : A \to A$ need not be nonexpansive from $\mathcal{A}_\mathcal{P}$ to $\mathcal{A}_\mathcal{P}$. It is an open problem to provide a construction from Scott domains to continuity spaces which will send a wide class of continuous maps to nonexpansive ones.

Continuous Cpo's. For a general continuous cpo, where the Lawson topology may not be zero-dimensional, we give a "fuzzy" version of the above construction; that is, we replace the two point set $\{0,1\}$ by the u[illegible]erval.

A ***character*** on a CPO A is [illegible]on $k : A \to [0,1]$ which preserves directed suprema and has a left-adjoint. This is a [illegible] generalization of compact element, since $k \in A$ is compact iff the characteristic function $\chi_{\uparrow(k)} : A \to \{0,1\}$ preserves directed suprema and has a left-adjoint.

Assume A is a cpo and let K be the set of characters on A. $[0,1]^K$ is a value quantale with the componentwise ordering and operation of addition. Define $d_\mathcal{I} : A \times A \to [0,1]^K$ by $d_\mathcal{I}(x,y)(k) = k(x) \dot{-} k(y)$, for $x, y \in A$, $k \in K$. Then $\mathcal{A}_\mathcal{I} = (A, d_\mathcal{I})$ is a $[0,1]^K$-continuity space.

7. Theorem. ([FK]) Assume A is a continuous cpo. Then $\mathcal{A}_\mathcal{I}$ is c_0, the induced topology on $\mathcal{A}_\mathcal{I}$ is the Scott topology on A, the dual topology on $\mathcal{A}_\mathcal{I}$ is the lower topology on A, and the symmetric topology on $\mathcal{A}_\mathcal{I}$ is the Lawson topology on A.

Again, the continuity space $\mathcal{A}_\mathcal{I}$ is totally bounded and so is an $\mathcal{I}(K)$-domain iff the Lawson topology on A is compact. This condition is equivalent to the requirement that A be supersober (cf., [G&] p. 310). Thus if A is bounded complete, then $\mathcal{A}_\mathcal{I}$ is an $\mathcal{I}(K)$-domain.

Metric Spaces. In much of the work concerned with modelling concurrent processes, complete metric spaces ([dZ], [AR]) have proved to be a useful tool. This example is easily captured in the present framework. Let $\mathcal{R}$ be the extended nonnegative reals $[0, \infty]$ with the usual ordering and the standard operation of addition. Then $\mathcal{R}$ is a value quantale, which we call the *value quantale of distances*. A symmetric $\mathcal{R}$-domain is a complete metric space. The induced topology on a symmetric $\mathcal{R}$-domain is, of course, the usual metric topology.

Probabilistic Domains. Neither domains of cpo's nor complete metric spaces seem adequate to model, in a natural way, languages which involve probabilistic constructs. We consider in this example a value quantale whose corresponding notion of semantic domain may be more adequate for this purpose.

Let $\mathcal{M}$ be the set of monotone maps from $[0, \infty)$ to $[0, 1]$, ordered pointwise. We call $F \in \mathcal{M}$ a *distance distribution function* (d.d.f.) iff F is *left-continuous*: for all $x \in [0, \infty)$, $\sup_{y<x} F(y) = F(x)$. Let Δ be the collection of all d.d.f.'s with the opposite of the pointwise ordering. Since the sup of d.d.f's is still a d.d.f., Δ is a complete lattice. The operation $+ : \Delta \times \Delta \rightarrow \Delta$, defined by $(F+G)(x) = \sup_{u+v=x} \min\{F(u), G(v)\}$, makes Δ a value quantale. A symmetric Δ-domain is a complete probabilistic quasimetric space [SSk]. Moreover the induced topology on a symmetric Δ-domain X is the strong topology on X.

3. Closure of Categories of Domains under Elementary Operations

The category V–**Dm** is closed under a number of basic operations, which are needed to build up complex data types from primitive ones.

Products. There are two natural notions of the product of two V-continuity spaces $\mathcal{A} = (A, d)$ and $\mathcal{B} = (B, d)$: the *Cartesian product*, $\mathcal{A} \times \mathcal{B} = (A \times B, d_\times)$, where

$$d_\times(< x_1, y_1 >, < x_2, y_2 >) = d(x_1, x_2) \vee d(y_1, y_2),$$

and the *tensor product*, $\mathcal{A} \otimes \mathcal{B} = (A \otimes B, d_\otimes)$, where $A \otimes B = A \times B$ and

$$d_\otimes(< x_1, y_1 >, < x_2, y_2 >) = d(x_1, x_2) + d(y_1, y_2).$$

$\mathcal{A} \times \mathcal{B}$ has the familiar universal property of the Cartesian product. $\mathcal{A} \otimes \mathcal{B}$ also satisfies a natural universal property. Call a map $f : A \times B \rightarrow C$ *separately nonexpansive* if for each $a \in A$, the function $b \rightarrow f(a, b)$ is nonexpansive from B to C and for each $b \in B$, the function $a \rightarrow f(a, b)$ is nonexpansive from A to C. The indentity map $I : A \times B \rightarrow A \otimes B$ is clearly separately nonexpansive. Moreover for any separately nonexpansive map $f : A \times B \rightarrow C$ there is a unique nonexpansive map $\hat{f} : A \otimes B \rightarrow C$ (namely, f itself) such that $f = \hat{f} \circ I$. If V is Boolean, then these two notions of product are identical.

It should be noted here for use in the power domain discussion later, that $+, \vee, \wedge : (V, d_V) \otimes (V, d_V) \rightarrow (V, d_V)$ and $\dot{-} : (V, d_V) \otimes (V, d_V^*) \rightarrow (V, d_V)$ are all nonexpansive functions. The required inequalities can be shown in a straightforward manner using the adjointness which holds between $\dot{-}$ and $+$.

Coproducts. For V-continuity spaces $\mathcal{A} = (A, d)$ and $\mathcal{B} = (B, d)$, their coproduct is $\mathcal{A} \oplus \mathcal{B} = (A \dot{\cup} B, d_\oplus)$, where $A \dot{\cup} B$ is disjoint union and for $x, y \in A \dot{\cup} B$,

$$d_\oplus(x, y) = \begin{cases} d_A(x, y) & \text{if } x, y \in A \\ d_B(x, y) & \text{if } x, y \in B \\ \infty & \text{otherwise.} \end{cases}$$

Function Spaces. The *function space* $[\mathcal{A} \rightarrow \mathcal{B}]$ consists of all nonexpansive maps from $\mathcal{A}$ to $\mathcal{B}$, where for $f, g \in [\mathcal{A} \rightarrow \mathcal{B}]$, $d_{[\mathcal{A} \rightarrow \mathcal{B}]}(f, g) = \bigvee\{d_B(f(x), g(x)) \mid x \in X\}$. For each V-domain $\mathcal{A}$, the functor $\mathcal{A} \otimes -$ is left-adjoint to $[\mathcal{A} \rightarrow -]$. Thus, if V is Boolean, then V-**Dm** is Cartesian closed, in general, V-**Dm** is a symmetric monoidal closed category.

Power Domains. To model nondeterminism, in addition to the above domain constructors, we also need a form of *power domain*. The standard constructions of the lower, upper, and convex power domains can easily be adapted to the continuity space setting.

For a V-continuity space (A,d), let $\mathcal{P}'_f(A)$ denote the set of finite nonempty subsets of A and define the three functions $d_{\mathcal{U}}$, $d_{\mathcal{L}}$, $d_{\mathcal{C}} : \mathcal{P}'_f(A) \times \mathcal{P}'_f(A) \to V$ by

$$d_{\mathcal{U}}(U,V) = \bigvee_{v \in V} \bigwedge_{u \in U} d(u,v),$$
$$d_{\mathcal{L}}(U,V) = \bigvee_{u \in U} \bigwedge_{v \in V} d(u,v), \text{ and}$$
$$d_{\mathcal{C}}(U,V) = d_{\mathcal{U}}(U,V) \vee d_{\mathcal{L}}(U,V).$$

It is easy to show that $(\mathcal{P}'_f(A), d_{\mathcal{U}})$, $(\mathcal{P}'_f(A), d_{\mathcal{L}})$, and $(\mathcal{P}'_f(A), d_{\mathcal{C}})$ are V-continuity spaces (although they are not necessarily c_0). The ***upper power space***, $A^{\mathcal{U}}$, is the completion of $(\mathcal{P}'_f(A), d_{\mathcal{U}})$, the ***lower power space***, $A^{\mathcal{L}}$, is the completion of $(\mathcal{P}'_f(A), d_{\mathcal{L}})$, and the ***convex power space***, $A^{\mathcal{C}}$, is the completion of $(\mathcal{P}'_f(A), d_{\mathcal{C}})$. We consider the universal property of the convex power space in detail and indicate briefly how to modify the discussion for the other two cases.

8. Lemma. The union operation $\cup : \mathcal{P}'_f(A) \times \mathcal{P}'_f(A) \to \mathcal{P}'_f(A)$ is nonexpansive (with respect to $d_{\mathcal{U}}, d_{\mathcal{L}}$, or $d_{\mathcal{C}}$).

9. Definition. A ***convex V-algebra*** is a V-domain E together with a nonexpansive binary operation $\star : E \times E \to E$ which is

(1) associative: $(x \star y) \star z = x \star (y \star z)$;
(2) commutative: $x \star y = y \star x$; and
(3) idempotent: $x \star x = x$.

A ***homomorphism*** from the convex V-algebra E_1 to the convex V-algebra E_2 is a nonexpansive map $h : E_1 \to E_2$ such that for all $x, y \in E_1$, $h(x \star y) = h(x) \star h(y)$.

Let $\iota^{\mathcal{C}} : \mathcal{P}'_f(A) \to A^{\mathcal{C}}$ be the canonical isometric mapping from $\mathcal{P}'_f(A)$ to $A^{\mathcal{C}}$. For $\alpha, \beta \in A^{\mathcal{C}}$ choose $(U_\epsilon)_\epsilon$ and $(V_\epsilon)_\epsilon$ Cauchy nets in $\mathcal{P}'_f(A)$ such that

$$\alpha = \lim_\epsilon \iota^{\mathcal{C}} U_\epsilon \quad \text{and} \quad \beta = \lim_\epsilon \iota^{\mathcal{C}} V_\epsilon.$$

It follows at once from Lemma 8 that $(U_\epsilon \cup V_\epsilon)_\epsilon$ is also a Cauchy net. Let

$$\alpha \cup^{\mathcal{C}} \beta = \lim_\epsilon \iota^{\mathcal{C}}(U_\epsilon \cup V_\epsilon).$$

$\alpha \cup^{\mathcal{C}} \beta$ is well-defined by Lemma 8 and c_0.

10. Lemma. $(A^{\mathcal{C}}, \cup^{\mathcal{C}})$ is a convex V-algebra. Moreover, for all $U, V \in \mathcal{P}'_f(A)$,

$$\iota^{\mathcal{C}}(U \cup V) = \iota^{\mathcal{C}}(U) \cup^{\mathcal{C}} \iota^{\mathcal{C}}(V).$$

PROOF.

Let $\alpha, \alpha', \beta, \beta' \in A^{\mathcal{C}}$. Choose $(U_\epsilon)_\epsilon$, $(U'_\epsilon)_\epsilon$, $(V_\epsilon)_\epsilon$, $(V'_\epsilon)_\epsilon$ Cauchy nets in $\mathcal{P}'_f(A)$ so that $\alpha = \lim_\epsilon \iota^{\mathcal{C}} U_\epsilon$, $\alpha' = \lim_\epsilon \iota^{\mathcal{C}} U'_\epsilon$, $\beta = \lim_\epsilon \iota^{\mathcal{C}} V_\epsilon$, and $\beta' = \lim_\epsilon \iota^{\mathcal{C}} V'_\epsilon$.

Then by (12), Lemma 8 and the nonexpansiveness (thus continuity) of $\vee$, we get

$$\begin{aligned} d_{\mathcal{C}}(\alpha \cup^{\mathcal{C}} \alpha', \beta \cup^{\mathcal{C}} \beta') &= \lim_\epsilon d_{\mathcal{C}}(U_\epsilon \cup U'_\epsilon, V_\epsilon \cup V'_\epsilon) \\ &\le \lim_\epsilon (d_{\mathcal{C}}(U_\epsilon, V_\epsilon) \vee d_{\mathcal{C}}(U'_\epsilon, V'_\epsilon)) \end{aligned}$$

$$= \lim_\epsilon d_C(U_\epsilon, V_\epsilon) \vee \lim_\epsilon d_C(U'_\epsilon, V'_\epsilon)$$
$$= d_C(\alpha, \beta) \vee d_C(\alpha', \beta').$$

Q.E.D.

We write $\{x\}^C$ for $\iota^C(\{x\})$, where $x \in A$. Clearly, $\{\}^C : A \to A^C$ is an isometric embedding.

In the proof of the main theorem below, we need the following consequence of the distributivity of V: for any finite family $\{x_{i,j} \mid i \in I,\ j \in J\}$ of elements of V,

$$\bigvee_{i \in I} \bigwedge_{j \in J} x_{i,j} = \bigwedge_{\phi : I \to J} \bigvee_{i \in I} x_{i\ \phi(i)}.$$

11. Theorem. Assume A is a V-domain, $(E, \star)$ is a convex V-algebra, and $f : A \to E$ is nonexpansive. Then there is a unique homomorphism $f^C : A^C \to E$ such that $f = f^C \circ \{\}^C$.

PROOF.

Existence.

Define $f' : \mathcal{P}'_f(A) \to E$ by

$$f'(\{x_1, \ldots, x_n\}) = f(x_1) \star \ldots \star f(x_n).$$

Then for all $U, V \in \mathcal{P}'_f(A)$, $f'(U \cup V) = f'(U) \star f'(V)$, and $f' \circ \{\} = f$.

CLAIM 1. $f' : \mathcal{P}'_f(A) \to E$ is nonexpansive.

Let $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_m\}$ be finite subsets of A. Then for any functions $\phi : \{1, 2, \ldots, m\} \to \{1, 2, \ldots, n\}$ and $\psi : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, m\}$,

$$\begin{aligned} d_E(f(U), f(V)) &= d_E(f(u_1) \star \ldots \star f(u_n), f(v_1) \star \ldots \star f(v_m)) \\ &= d_E(f(u_1) \star \ldots \star f(u_n) \star f(u_{\phi(1)}) \star \ldots f(u_{\phi(m)}), \\ &\quad f(v_{\psi(1)}) \star \ldots \star f(v_{\psi(n)}) \star f(v_1) \star \ldots \star f(v_m)) \\ &\leq \bigvee_{1 \leq i \leq n} d_E(f(u_i), f(v_{\psi(i)})) \vee \bigvee_{1 \leq j \leq m} d_E(f(u_{\phi(j)}), f(v_j)) \\ &\leq \bigvee_{1 \leq i \leq n} d(u_i, v_{\psi(i)}) \vee \bigvee_{1 \leq j \leq m} d(u_{\phi(j)}, v_j) \end{aligned}$$

Hence

$$\begin{aligned} d_E(f(U), f(V)) &\leq \bigwedge_{\psi} \bigvee_{i} d(u_i, v_{\psi(i)}) \vee \bigwedge_{\phi} \bigvee_{j} d(u_{\phi(j)}, v_j) \\ &= \bigvee_{i} \bigwedge_{j} d(u_i, v_j) \vee \bigvee_{j} \bigwedge_{i} d(u_i, v_j) \\ &= d_C(U, V). \end{aligned}$$

Claim 1 follows.

Let $f^C : A^C \to E$ be the unique nonexpansive extension of f'.

CLAIM 2. f^C is a homomorphism.

Let $\alpha, \beta \in A^C$ and choose $(U_\epsilon)_\epsilon$ and $(V_\epsilon)_\epsilon$ Cauchy nets in $\mathcal{P}'_f(A)$ such that

$$\alpha = \lim_\epsilon \iota^C U_\epsilon \quad \text{and} \quad \beta = \lim_\epsilon \iota^C V_\epsilon.$$

Then

$$\begin{aligned} f^C(\alpha \cup^C \beta) &= f^C(\lim_\epsilon \iota^C(U_\epsilon \cup V_\epsilon)) \\ &= \lim_\epsilon f^C \iota^C(U_\epsilon \cup V_\epsilon) \\ &= \lim_\epsilon f'(U_\epsilon \cup V_\epsilon) \\ &= \lim_\epsilon f'(U_\epsilon) \star f'(V_\epsilon) \\ &= \lim_\epsilon f'(U_\epsilon) \star \lim_\epsilon f'(V_\epsilon) \\ &= f^C(\alpha) \star f^C(\beta). \end{aligned}$$

Claim 2 follows.

Since $f^C \circ \{\}^C = (f^C \circ \iota^C) \circ \{\} = f' \circ \{\} = f$, f^C is the required homomorphism.

Uniqueness.

Since f^C is required to be a homomorphism, its restriction to $\mathcal{P}'_f(A)$ must be equal to f'. This determines f^C uniquely.

Q.E.D.

Thus A^C is the *free* convex V-algebra on the V-domain A. Corresponding results hold for $A^{\mathcal{U}}$ and $A^{\mathcal{L}}$. For $A^{\mathcal{U}}$ the notion of convex V-algebra is replaced by that of *upper* V-algebra, which is obtained by adding the axiom $(4^{\mathcal{U}})$ $d(x \star y, x) = 0$. For $A^{\mathcal{L}}$, the axiom $(4^{\mathcal{L}})$ $d(x, x \star y) = 0$ is added to the definition of convex V-algebra yielding the notion of *lower* V-algebra.

4. Fixed Points

FIXED POINTS OF MORPHISMS

In denotational semantics the meanings of many language constructs are given as solutions of equational specifications. The existence of such solutions depends on the existence of fixed points of certain morphisms. To specify such a class of morphisms for V-**Dm**, we assume additional structure on V.

12. Definition. An *action* on V is a monotone map, $\odot : [0,\infty] \times V \to V$ satisfying the following conditions for all $\alpha, \beta \in [0,\infty]$ and $p, q \in V$:

(a) $1 \odot p = p$;
(b) $(\alpha\beta) \odot p = \alpha \odot (\beta \odot p)$; and
(c) $(\alpha + \beta) \odot p = \alpha \odot p + \beta \odot p$.

On the value quantale of distances, ordinary multiplication is an action. This is, of course, the motivating example. Another important example is provided by the value quantale of distance distribution functions. For $\alpha \in [0,\infty]$ and $F \in \Delta$, $\alpha \odot F$ is defined by $(\alpha \odot F)(x) = F(\frac{x}{\alpha})$.

13. Definition. Assume X is a V-continuity space and $f : X \to X$. Then f is a *contraction mapping* if there is an $\alpha \in [0,1)$ such that for all $x_1, x_2 \in X$, $\alpha \odot d(x_1, x_2) \geq d(f(x_1), f(x_2))$.

To adapt the standard proof of Banach's Fixed Point Theorem to the continuity space setting, we need one more notion. An element $p \in V$ is $\odot$-*finite* if $\bigwedge_{\epsilon>0}(\epsilon \odot p) = 0$. In the two examples mentioned above, being finite can be characterized in familiar terms. An element $p \in \mathcal{R}$ is finite iff $p < \infty$. An element $F \in \Delta$ is finite iff it is the distribution of a random variable; that is, iff $\lim_{x\to\infty} F(x) = 1$.

Fixed-Point Theorem for Contraction Mappings. Assume X is a V-domain and $f : X \to X$ is a contraction mapping. If there is an element x in X such that $d^S(x, f(x))$ is finite, then

f has a fixed point; that is, there is an element $x_\infty \in X$ such that $f(x_\infty) = x_\infty$. Moreover, if x_1 and x_2 are fixed points of f and $d^S(x_1, x_2)$ is finite, then $x_1 = x_2$.

In the two examples mentioned above, this theorem specializes to the Banach Fixed Point Theorem and a version of Sherwood's fixed point theorem for probabilistic metric spaces [Sh].

SOLUTIONS TO REFLEXIVE DOMAIN EQUATIONS

Many data types are also naturally specified using fixed points. For example to model the λ-calculus, one needs a solution to the equation: $D \cong At \oplus [D \to D]$. Equations of this type are called *reflexive domain equations* and can generally be reduced to fixed point equations of the form $F(D) \cong D$, where F is an endofunctor on the category of domains. The presence of an action on V allows us to define a notion of *contractive functor* for which fixed-points can be found, using the construction of [AR] for solving reflexive domain equations in categories of metric spaces. In this approach, the standard notion of *projection pair* is replaced by that of *retraction pair* together with a measure of how close such a pair is to an isometry. Precisely, if X_1 and X_2 are V-domains, then a *retraction pair* from X_1 to X_2 is a pair (f, g) of nonexpansive maps $f : X_1 \to X_2$ and $g : X_2 \to X_1$ such that $g \circ f = Id_{X_1}$. The *norm* of (f, g) is $|(f, g)| = d_{[X_2 \to X_2]}(f \circ g, Id_{X_2})$.

14. Definition. If $F : V - \mathbf{Dm} \to V - \mathbf{Dm}$ is a functor, then F is *contractive* if there is an $\alpha \in [0, 1)$ such that for all retraction pairs $(f, g) : D \to D'$. $\alpha \odot |(f, g)| \geq |(F(f), F(g))|$.

Fixed-Point Theorem for Contractive Functors. Assume $F : V - \mathbf{Dm} \to V - \mathbf{Dm}$ is a contractive functor and there is a retraction pair $(f, g) : D \to F(D)$ such that $|(f, g)|$ is finite. Then F has a fixed point; that is, there is a V-domain D_∞ such that D_∞ is isomorphic to $F(D_\infty)$ in $V - \mathbf{Dm}$.

We have stated this theorem in its simplest form. To solve domain equations such as $D \cong At \oplus [D \to D]$, where D appears both covariantly and contravariantly, certain modifications must be made; however, this presents no real difficulties.

REFERENCES

AR P. America and J.J.M.M. Rutten, "Solving Reflexive Domain Equations in a Category of Complete Metric Spaces", *J. Comput. System. Sci.*, **39**(3), 1989, pp. 343–375.

dZ J. deBakker and J. Zucker, "Processes and the Denotational Semantics of Concurrency", *Information and Control*, **54**, 1982, pp. 70–120.

Fl R. C. Flagg, "Quantales and Continuity Spaces", to appear in *Algebra Universalis*.

FK Flagg, R. C. and R. Kopperman, "Continuity Spaces and Continuous Lattices", preprint.

G& Gierz, G., K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove and D. S. Scott, *A Compendium of Continuous Lattices*, Springer-Verlag, Berlin, 1980.

Gn C. A. Gunter, "Comparing Categories of Domains", in: *Mathematical Foundations of Programming Semantics*, ed. A. Melton, LNCS 239, Springer-Verlag, 1985, pp. 101–121.

H& Henriksen, M., R. Kopperman, J. Mack, and D. W. B. Somerset, "Joincompact Spaces, Continuous Lattices, and C*-algebras", preprint.

Ke Kelley, J. L., *General Topology*, van Nostrand, Princeton, 1955.

Ky Kelly, W. C., "Bitopological Spaces", *Proc. London Math. Soc.* **13** (1963), 71-89.

Ko Kopperman, R., "Asymmetry and Duality in Topology", preprint.

Kp Kopperman, R., "All Topologies Come From Generalized Metrics", *Am. Math. Monthly* **95** (1988), 89-97.

La J. D. Lawson, "The Versatile Continuous Order", in: ***Mathematical Foundations of Programming Language Semantics***, (ed. M. Mislove), Springer-Verlag, 1989, pp. 134-160.

Lw Lawson, J. D., "Order and strongly sober compactifications", ***Topology and Category Theory in Computer Science***, G. M. Reed, A. W. Roscoe and R. F. Wachter, eds., Oxford University Press, Oxford, 1991, pp. 179-205.

Pl G. D. Plotkin, "Post-graduate lecture notes in advanced domain theory", Dept. of Computer Science, Univ. of Edinburgh, 1981.

Sh H. Sherwood, "Complete probabilistic metric spaces", *Z. Wahrsch. Verw. Geb.* **20**, (1971), 117-128.

SSk B. Schweizer and A. Sklar, *Probabilistic Metric Spaces*, North Holland, Amsterdam, 1983.

SSt D. Scott and C. Strachey, "Toward a mathematical semantics for computer languages", *Proceedings of the Symposium on Computers and Automata*, Polytechnic Institute Press, New York, 1971, 19-46.

Sm Smyth, M. B., "Stable Local Compactification I", *J. London Math. Soc.*.

Wi Wilson, W. A., "On Quasi-metric Spaces". Amer. J. Math. **53** (1931), 675-684.

Ultimately Periodic Words of Rational ω-Languages

Hugues Calbrix [*] Maurice Nivat [*] Andreas Podelski [†]

Abstract

In this paper we initiate the following program: Associate sets of finite words to Büchi-recognizable sets of infinite words, and reduce algorithmic problems on Büchi automata to simpler ones on automata on finite words. We know that the set of ultimately periodic words $UP(L)$ of a rational language of infinite words L is sufficient to characterize L, since $UP(L_1) = UP(L_2)$ implies $L_1 = L_2$. We can use this fact as a test, for example, of the equivalence of two given Büchi automata. The main technical result in this paper is the construction of an automaton which recognizes the set of all finite words $u \cdot \$ \cdot v$ which naturally represent the ultimately periodic words of the form $u \cdot v^\omega$ in the language of infinite words recognized by a given Büchi automaton.

1 Introduction

Büchi automata recognizing sets of infinite words appear as a major tool in modelizing the behavior of a number of computing systems including distributed and real-time systems and circuits. The standard theoretical results about the decidability of the equivalence of two Büchi automata do not lead to efficient algorithms for equality test or optimisation of such automata, see e.g. Safra[5] or Sistla, Vardi and Volper [6] (a question about which almost nothing is known). The basic idea underlying the present paper is that a set of infinite words recognized by a Büchi automaton is entirely known when we know the subset of ultimately periodic words (of the form $u \cdot v^\omega$) it contains, and we prove that this set is finitely representable since the set of finite words $u \cdot \$ \cdot v$ corresponding to all the $u \cdot v^\omega$ is rational, i.e. recognizable by a finite automaton. This fact brings the hope that a number of constructions which are presently outwardly performed on Büchi automata can be performed on simple dfa's. This is already the case for the S1S logic (see [7]) for which this method brings an described in [2].

Two main theorems are proved in this paper. The first one states the rationality of $L_\$$, the language of finite representations of ultimately peri-

[*]LITP, Université de Paris 7, 2,place Jussieu ,75251 Paris Cedex 05, France. e-mail: {calbrix,tcsmn}@litp.ibp.fr

[†]Digital PRL, 85, avenue Victor Hugo, 92563 Rueil-Malmaison, France. e-mail: podelski@prl.dec.com

odic words of an arbitrary rational ω-language L, and its proof brings a construction of an automaton that recognizes $L_\$$. The second one states a nice characterization of the languages K that are $L_\$$ for a given rational ω-language L and also brings a construction of a Büchi automaton recognizing L.

Section 3 describes informally the first construction and two representative examples are shown. The formal proof of the first result lies in Section 4. In Section 5, we study the determinisation of the previous construction and we give an upper bound of its number of states. Section 6 is devoted to the proof of the second theorem and also gives a bound to the number of states of the second construction. In Section 7, we raise some questions about rational languages contained in $A^* \cdot \$ \cdot A^+$ and the set of ultimately periodic words that they represent. Section 8 concludes this paper.

2 Basic Definitions

Let A be a finite set called the *alphabet*. We denote A^* the set of *finite words* on A — finite sequence of elements of A. We note ϵ the empty sequence, which is called the *empty word*. We denote A^+ the set of non-empty words, i.e. $A^+ = A^* \setminus \{\epsilon\}$. Let u be a finite word. We denote by $|u|$ the length of the sequence u. The length of the empty word ϵ is thus 0. We denote by A^ω the set of *infinite words* on A — infinite sequences of elements of A. A *language* is a subset M of A^*, and *ω-language* a subset L of A^ω.

A *finite automaton* $\mathcal{A}$ is a tuple (Q, I, D, E) made of a finite set Q, the elements of which are the *states* of the automaton, a subset I of Q of *initial* states, a subset D of Q of *distinguished* states, and a subset E of $Q \times A \times Q$, the elements of which are the *edges* of the automaton. It will be convenient to number the elements of Q. We will then write $Q = \{q_1, \dots q_m\}$.

Let $u = u(1) \cdot \ldots \cdot u(k)$ be a finite word. A word $c = c(0) \cdot \ldots \cdot c(k+1)$ of Q^+ is a calculus of $\mathcal{A}$ on u if $(c(i), u(i), c(i+1)) \in E$ for each i such that $1 \leq i \leq k$. This calculus is *successful* if $c(0) \in I$ and $c(k+1) \in D$. We denote $L(\mathcal{A})$ the language of finite words u such that there is a successful calculus of $\mathcal{A}$ on u. In this case, the elements of D are called *final* states and D is denoted F. The set of languages $L(\mathcal{A})$ for some automaton $\mathcal{A}$, is denoted $\mathcal{R}at(A^*)$, and its elements are called *rational languages*.

Let $\mathcal{A}$ be a finite automaton and $v \in A^+$ a non-empty finite word. When there exists a calculus $c \in Q^+$ of $\mathcal{A}$ on v such that $c = p \cdot c' \cdot q$, we will write $p \xrightarrow[\mathcal{A}]{v} q$. When in addition $c' \cdot q$ contains a distinguished state, we will write $p \overset{v}{\underset{\mathcal{A}}{-\!\circ\!\rightarrow}} q$, and on the other hand, when $c' \cdot q$ contains no distinguished state, we will write $p \overset{v}{\underset{\mathcal{A}}{\rightsquigarrow}} q$.

Let $\alpha = \alpha(0) \cdot \alpha(1) \cdot \ldots$ be an infinite word. A word $\chi = \chi(0) \cdot \chi(1) \cdot \ldots$ of Q^ω is a *calculus* of $\mathcal{A}$ on α if $(\chi(i), \alpha(i), \chi(i+1)) \in E$ for each integer i. This calculus is *successful* if $\chi(0) \in I$ and if there exists a distinguished state q of D such that $\chi(k) = q$ for infinitely many integers k. We denote $L^\omega(\mathcal{A})$ the ω-language of infinite words α such that there is a successful calculus of $\mathcal{A}$ on α. In this case, the elements of D are called *repeated* states and D is denoted R and $\mathcal{A}$ is called a *Büchi automaton*. The set of ω-languages $L^\omega(\mathcal{A})$ for some automaton $\mathcal{A}$ is denoted $\mathcal{R}at(A^\omega)$ and its elements are called *rational* ω-languages.

We denote $UP(A^\omega)$ the set $\{u \cdot v^\omega \mid (u,v) \in A^* \times A^+\}$, the elements of which are the *ultimately periodic* words. Let L be an ω-language, we denote $UP(L)$ the set $L \cap UP(A^\omega)$ of all ultimately periodic words of L. Let α be an ultimaltely periodic word of A^ω. A word $v \in A^+$ is a *period* of α if there exists a word $u \in A^*$ such that $\alpha = u \cdot v^\omega$. Similarly, a word $u \in A^*$ is a *prefix* of α if there is a period v of α such that $\alpha = u \cdot v^\omega$. This definition of a prefix is thus more restrictive than the usual one. Indeed, a isn't a prefix of $aa \cdot b^\omega$, for there is no word $v \in A^+$ such that $aa \cdot b^\omega = a \cdot v^\omega$.

Fact 1 *Let L_1 and L_2 be two rational ω-languages such that $UP(L_1) = UP(L_2)$, then $L_1 = L_2$.*

Proof The ω-language $(L_1 \cup L_2) \setminus (L_1 \cap L_2)$ does not contain any ultimately periodic word and it is a rational ω-language, because the set $\mathcal{R}at(A^\omega)$ is closed under boolean combinations. However, every non-empty rational ω-language contains at least one ultimately periodic word. Thus $(L_1 \cup L_2) \setminus (L_1 \cap L_2)$ is the empty set and $L_1 = L_2$. ◇

The set of ultimately periodic words of a rational ω-language is thus characteristic of this ω-language. The ultimately periodic word $u \cdot v^\omega$ on the alphabet A may be represented by the finite word $u \cdot \$ \cdot v$ on the alphabet $A \cup \$$, where $\$$ is a dummy symbol which is not already in A. Let L be a rational ω-language. We define the language $L_\$ = \{u \cdot \$ \cdot v \mid u \cdot v^\omega \in L\}$ on the alphabet $A \cup \$$, to be the set of all the finite words which represent ultimately periodic words of L. The Fact 1 allows us to say that $L_\$$ characterizes the rational ω-language L.

3 Finite Words

Let L be a rational ω-language and $\mathcal{A} = (Q, I, R, E)$ a Büchi automaton which recognizes it (we set $Q = \{q_1, \ldots, q_m\}$.) For each r such that $1 \leq r \leq m$, we set $M_r = \{u \in A^* \mid \exists q \in I, q \xrightarrow{u} q_r\} = L(Q, I, \{q_r\}, E)$ and $N_r = \{v \in A^+ \mid v^\omega \in L^\omega(Q, \{q_r\}, R, E)\}$. It is clear that for each pair of words $(u,v) \in M_r \times N_r$, $u \cdot v^\omega \in L$, because a successful calculus of $\mathcal{A}$ on $u \cdot v^\omega$ may be built from a calculus of $(Q, I, \{q_r\}, E)$ on u leading to q_r and a successful calculus of $(Q, \{q_r\}, R, E)$ on v^ω. Moreover, for each ultimately periodic word $u \cdot v^\omega \in L$, there exists a $q_r \in Q$ such that $u \in M_r$ and $v \in N_r$. This q_r is the state reached after the reading of u in a successful calculus of $\mathcal{A}$ on $u \cdot v^\omega$. We may decompose the previously defined language $L_\$$, using the languages M_r and N_r in the following way.

$$L_\$ = \bigcup_{r=0}^{m} M_r \cdot \$ \cdot N_r \tag{1}$$

Languages M_r are made of prefixes of ultimately periodic words of L and these languages are rational, because they are recognized by automata $(Q, I, \{q_r\}, E)$. Languages N_r are made of periods of ultimately periodic words of L. We will build automata which recognize languages N_r to show that they are rational too. The rationality of $L_\$$ will follow from this fact.

It might be noticed that there are various ways to show the rationality of $L_\$$. We can show that the syntactic congruence of $L_\$$ and Arnold's congruence of L (defined in [1]) are the same on the set A^+ (see [3]). Then the syntactic congruence of $L_\$$ is of finite index and $L_\$$ is thus rational. It is also possible to use the equivalence between S1S-logical definability and rationality for ω-languages (see e.g. [7]) to construct an automaton recognizing $L_\$$ from a logical formula defining L (this procedure is described in [2]). However, the direct construction of an automaton is the most efficient way to produce a recognizing device for $L_\$$.

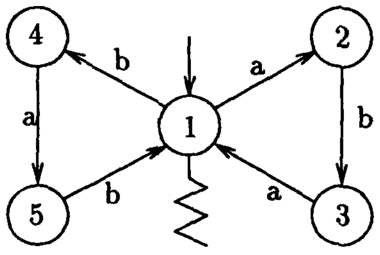


figure 1

A word v^ω is recognized by the automaton $(Q, \{q_r\}, R, E)$ if there is a successful calculus of this automaton on v^ω. This calculus runs along one or several loops—i.e. cyclic sequences of states—which contains repeated states of R. For exemple, let $L = (aba + bab)^\omega$ be the language recognized by the automaton of the *Figure 1*. The word $(aba)^\omega$ is recognized, and the infinite sequence of states $(123)^\omega$ is a successful calculus of the automaton on $(aba)^\omega$. This calculus defines a loop—1231—which runs through a repeated state—1.

The word *aba* is in the language N_1 of periods of ultimately periodic words of L recognized from the state 1, and we just need to know the calculus of the automaton on the word *aba* to find the loop 1231. This is not always the case, as the next example will show it. The word $(ab)^\omega$ is recognized too, the infinite sequence of states $(123145)^\omega$ is a successful calculus of the automaton on this word and the loop found here is the sequence 1231451. The word *ab* is member of the language N_1 but the calculus of the automaton on the word *ab* doesn't permit us to find the loop, which only appears in the calculus of $(ab)^3$.

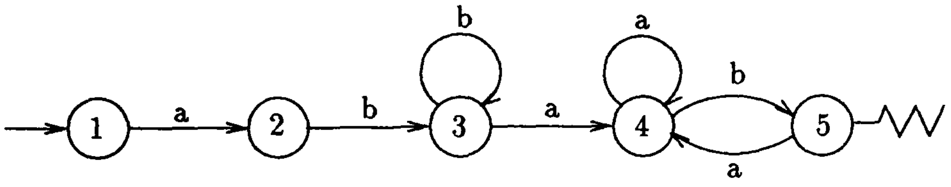


figure 2

Another example is given by the language $L = ab^+ \cdot (a^+b)^\omega$, recognized by the automaton of the *Figure 2*. The word $(ab)^\omega$ is recognized by this automaton and the infinite sequence of states $123(45)^\omega$ is a successful calculus of the automaton on the word $(ab)^\omega$. The word *ab* is element of the language N_1 of the periods of L which can be read from the state 1 of the automaton, but a calculus on *ababab* is necessary to find a loop—here 545. It becomes clear with this example that the first state of the calculus isn't necessarily involved in the loop found in this calculus.

The principle of the construction that we are going to describe is to simulate calculi of the automaton $\mathcal{A}$ which recognize the language L, starting from each state of the automaton $\mathcal{A}$. This simulation leads to a vector-state which contains as components ends of simulated calculi with an element of the set $\{0,1\}$ which is 1 if and only if the simulated calculus contains a repeated state. Final states are those from which a loop of $\mathcal{A}$ containing a repeated state can be built. For the first example, the calculus of the automaton recognizing N_1 on the word *ab* may be—the state denoted $\square$ is added to $\mathcal{A}$ to make it complete

$$\begin{pmatrix} 1,0 \\ 2,0 \\ 3,0 \\ 4,0 \\ 5,0 \end{pmatrix} \xrightarrow{a} \begin{pmatrix} 2,0 \\ \square \\ 1,1 \\ 5,0 \\ \square \end{pmatrix} \xrightarrow{b} \begin{pmatrix} 3,0 \\ \square \\ 4,1 \\ 1,1 \\ \square \end{pmatrix}$$

From this calculus, we can build the following calculi of the automaton $\mathcal{A}$ on the word ab,

$$1 \underset{\mathcal{A}}{\overset{ab}{\rightsquigarrow}} 3,\ 3 \underset{\mathcal{A}}{\overset{ab}{-\!\circ\!\!\rightarrow}} 4 \text{ and } 4 \underset{\mathcal{A}}{\overset{ab}{-\!\circ\!\!\rightarrow}} 1,$$

which permit us to find a loop containing a repeated state in a calculus of $\mathcal{A}$ on the word $(ab)^\omega$. Moreover, we can make the calculus begin with the state 1, 3 or 4, which shows that the word ab is element of N_1, N_3 and N_4.

4 First Construction

Formally, let L be an ω-language recognized by an automaton $\mathcal{A} = (Q, I, R, E)$ such that $Q = \{q_1, \ldots, q_m\}$. We suppose without loss of generality that the automaton $\mathcal{A}$ is complete, i.e. that $\{q \mid (p, a, q) \in E\} \neq \emptyset$ for each pair $(q, a) \in Q \times A$. For each state q_r of the automaton $\mathcal{A}$, we will build an automaton $\mathcal{A}_{N_r}$ which recognizes the language N_r previously defined.

The automaton $\mathcal{A}_{N_r}$ is built from the set of states $(Q \times \{0,1\})^{|Q|}$. The initial state is the vector-state $\vec{q_0} = ((q_1, 0), \ldots, (q_m, 0))$. The tuple $(\vec{p}, a, \vec{p}')$— with $\vec{p} = ((p_1, f_1), \ldots, (p_m, f_m))$ and $\vec{p}' = ((p'_1, f'_1), \ldots, (p'_m, f'_m))$— is an edge of $\mathcal{A}_{N_r}$ if, for each i such that $1 \leq i \leq m$, $(p_i, a, p'_i) \in E$ and if $p'_i \in R$ then $f'_i = 1$ else $f'_i = f_i$. A state $\vec{p} = ((p_1, f_1), \ldots, (p_m, f_m))$ is a final state if the following condition is verified. Let $(i_k)_{1 \leq k \leq m}$ be the finite sequence of integers defined by the relation $p_k = q_{i_k}$, and $(j_k)_{k \geq 0}$ be the infinite sequence defined recursively by $j_0 = r$ and $j_{k+1} = i_{j_k}$ for each $k \geq 0$. This sequence ranges only over a finite set of values. Let thus s be the smallest integer satisfying $j_s \in \{j_k \mid 0 \leq k < s\}$ and s' the only integer such that $s' < s$ and $j_{s'} = j_s$. Then, the state $\vec{p}$ is final if and only if $1 \in \{f_{j_k} \mid s' \leq k \leq s\}$.

The following lemma states the fundamental property of the automaton $\mathcal{A}_{N_r}$. A calculus of $\mathcal{A}_{N_r}$ begining with $\vec{q_0}$ on a word v contains for each state q_i a calculus of the automaton $\mathcal{A}$ begining with q_i on the word v. Moreover, a state of the automaton $\mathcal{A}_{N_r}$ reached by the reading of v from the state $\vec{q_0}$ can be built from the calculi of $\mathcal{A}$ on v starting from each state of $\mathcal{A}$.

Lemma 2 *Let $v \in A^*$ and $\vec{p} = ((p_1, f_1), \ldots, (p_m, f_m))$ be a state of $\mathcal{A}_{N_r}$. Then $\vec{q_0} \xrightarrow[\mathcal{A}_{N_r}]{v} \vec{p}$ if and only if, for each i such that $1 \leq i \leq m$, $q_i \underset{\mathcal{A}}{\overset{v}{\rightsquigarrow}} p_i$ if $f_i = 0$ and $q_i \underset{\mathcal{A}}{\overset{v}{-\!\circ\!\!\rightarrow}} p_i$ if $f_i = 1$.*

Proof Let $v \in A^*$ and $\vec{p}$ a state of $\mathcal{A}_{N_r}$. We will show the lemma with an induction on the length of the word v. If $v = \epsilon$, then the lemma trivially is true. Thus, we assume that $v \neq \epsilon$ and we set $v = u \cdot a$ with $u \in A^*$ and $a \in A$.

Let us assume that $\vec{q}_0 \xrightarrow[\mathcal{A}_{N_r}]{v} \vec{p}$. Let $\vec{p}\,'$ a state of $\mathcal{A}_{N_r}$ such that $\vec{q}_0 \xrightarrow[\mathcal{A}_{N_r}]{u} \vec{p}\,'$ and $\vec{p}\,' \xrightarrow[\mathcal{A}_{N_r}]{a} \vec{p}$. We set $\vec{p}\,' = ((p_1', f_1'), \ldots, (p_m', f_m'))$. We deduce from the induction hypothesis, that $q_i \xrightarrow[\mathcal{A}]{u} p_i'$ for each i such that $1 \leq i \leq m$. Since $(\vec{p}\,', a, \vec{p})$ is an edge of $\mathcal{A}_{N_r}$, we deduce from the definition of $\mathcal{A}_{N_r}$ that (p_i', a, p_i) is an edge of $\mathcal{A}$, and that $q_i \xrightarrow[\mathcal{A}]{v} p_i$, for each i such that $1 \leq i \leq m$. Moreover, if $f_i = 1$ then $f_i' = 1$ and in this case $q_i \overset{u}{\underset{\mathcal{A}}{-\!\circ\!\!\rightarrow}} p_i'$, or $f_i' = 0$ and in that case p_i is a repeated state. In both cases, we get that $q_i \overset{v}{\underset{\mathcal{A}}{-\!\circ\!\!\rightarrow}} p_i$. To conclude, if $f_i = 0$ then $f_i' = 0$, and we deduce from induction hypothesis that $q_i \overset{u}{\underset{\mathcal{A}}{\rightsquigarrow}} p_i'$. The state p_i isn't a repeated state and thus $q_i \overset{v}{\underset{\mathcal{A}}{\rightsquigarrow}} p_i$.

Let us now assume that $q_i \overset{v}{\underset{\mathcal{A}}{\rightsquigarrow}} p_i$ for each i such that $f_i = 0$ and that $q_i \overset{v}{\underset{\mathcal{A}}{-\!\circ\!\!\rightarrow}} p_i$ for each i such that $f_i = 1$. For each i such that $1 \leq i \leq m$, there is a state p_i' from Q such that $q_i \xrightarrow[\mathcal{A}]{u} p_i'$ and (p_i', a, p_i) is an edge of $\mathcal{A}$. Moreover, if $f_i = 1$ and p_i isn't a repeated state, then we can choose p_i' such that $q_i \overset{u}{\underset{\mathcal{A}}{-\!\circ\!\!\rightarrow}} p_i'$, and we set then $f_i' = 1$. If $f_i = 1$ and p_i is a repeated state, we set $f_i' = 0$ if $q_i \overset{u}{\underset{\mathcal{A}}{\rightsquigarrow}} p_i'$ and $f_i' = 1$ in the other case. If $f_i = 0$, we can choose p_i' such that $q_i \overset{u}{\underset{\mathcal{A}}{\rightsquigarrow}} p_i'$, and we simply set $f_i' = 0$. We deduce from the induction hypothesis that the state $\vec{p}\,' = ((p_1', f_1'), \ldots, (p_m', f_m'))$ thus defined is such that $\vec{q}_0 \xrightarrow[\mathcal{A}_{N_r}]{u} \vec{p}\,'$. Then, we see from the definition of the automaton $\mathcal{A}_{N_r}$ that $(\vec{p}\,', a, \vec{p})$ is an edge of $\mathcal{A}_{N_r}$. Thus, we conclude that $\vec{q}_0 \xrightarrow[\mathcal{A}_{N_r}]{v} \vec{p}$. ◇

It remains to show the equality between N_r and $L(\mathcal{A}_{N_r})$. So, let v be a word of $L(\mathcal{A}_{N_r})$, let $\vec{p} = ((p_1, f_1), \ldots, (p_m, f_m))$ be a final state of $\mathcal{A}_{N_r}$ such that $\vec{q}_0 \xrightarrow[\mathcal{A}_{N_r}]{v} \vec{p}$, and let sequences $(i_k)_{1 \leq k \leq m}$ and $(j_k)_{k \geq 0}$ and integers s and s' be defined as previously. From the previous lemma, we deduce the existence of calculi $b_1, \ldots, b_m$ of the automaton $\mathcal{A}$ on the word v such that $b_k = q_k \cdot b_k' \cdot p_k$ for each k such that $1 \leq k \leq m$ (we set $c_k = q_k \cdot b_k'$.) From the definitions of sequences $(i_k)_{1 \leq k \leq m}$ and $(j_k)_{k \geq 0}$, we deduce the equalities $p_{j_k} = q_{i_{j_k}} = q_{j_{k+1}}$. The infinite word $c_{j_0} \cdot \ldots \cdot c_{j_{s'-1}} \cdot (c_{j_{s'}} \cdot \ldots \cdot c_{j_{s-1}})^\omega$ of Q^ω is thus a calculus of $\mathcal{A}$ on the infinite word v^ω. Moreover, $\vec{p}$ is a final state of $\mathcal{A}_{N_r}$, then $f_{j_k} = 1$ for an integer k such that $s' \leq k < s$, and we deduce from the Lemma 2 that $c_{j_k}' \cdot q_{j_{k+1}}$ contains a repeated state. Because the first state of the previous infinite calculus is $q_{j_0} = q_r$, this is a successful calculus of the automaton $(Q, \{q_r\}, R, E)$ on the infinite word v^ω. The inclusion $L(\mathcal{A}_{N_r}) \subseteq N_r$ is thus proved.

Conversely, let v be a word of N_r. If $\mathcal{A}$ isn't deterministic, there exist non-regular calculus of $\mathcal{A}$ on v^ω. However, we will show in the next lemma the existence of a particular ultimately periodic calculus of $\mathcal{A}$ on v^ω which can be used to build a successful calculus of $\mathcal{A}_{N_r}$ on v.

Lemma 3 *Let $v \in N_r$—i.e. such that $v^\omega \in L^\omega(Q, \{q_r\}, R, E)$. Then, there is a successful calculus $\pi \in Q^\omega$ of the automaton $(Q, \{q_r\}, R, E)$ on v^ω which satisfy the following property. There is two integers s and s', $s' < s$, and some words $c_0, \ldots, c_{s-1} \in Q^*$ such that $|c_k| = |v|$, $c_k = p_k \cdot c'_k$ with $p_k \in Q$ and $p_k \neq p_l$ for each pair of integers k, l such that $0 \leq k, l < s$ and $k \neq l$ and these words verify $\pi = c_0 \cdot \ldots \cdot c_{s'_1} \cdot (c_{s'} \cdot \ldots \cdot c_{s-1})^\omega$.*

Proof Let $v \in N_r$ and $\chi = c_0 \cdot c_1 \cdots$ be a successful calculus of $(Q, \{q_r\}, R, E)$ on v^ω with $|c_k| = |v|$ for each integer $k \geq 0$. This calculus isn't necessarily ultimately periodic because $\mathcal{A}$ can be non-deterministic. For each integer $k \geq 0$, we set $c_k = p_k \cdot c'_k$, with $p_k \in Q$. Let then s be the least integer satisfying $p_s \in \{p_k \mid 0 \leq k < s\}$—such an integer exists because Q is a finite set—and s' be the integer such that $s' < s$ and $p_{s'} = p_s$. There are two possibilities. In the first case, the word $c_{s'} \cdots c_{s-1}$ contains a repeated state and then $c_0 \cdots c_{s'_1} \cdot (c_{s'} \cdots c_{s-1})^\omega$ is a successful calculus of $(Q, \{q_r\}, R, E)$ on v^ω which satisfy hypothesis of the lemma. In the other case, $\chi' = c_0 \cdot \ldots \cdot c_{s'-1} \cdot c_s \cdot \ldots$ is a successful calculus of $(Q, \{q_r\}, R, E)$ on v^ω. Then we repeat the whole process with χ' until we are in the first case. Because we're removing a non-empty factor of χ at each step and χ is successful, we are sure that the process will stop in a finite number of steps. $\diamond$

Then, let π be a calculus of $\mathcal{A}$ on v^ω satisfying the hypothesis of Lemma 3. We can set $q_1 = p_0, \ldots, q_s = p_{s-1}$ without loss of generality, the other states of Q are numbered arbitrarily, and we also set $p_s = p_{s'}$. For all k such that $1 \leq k \leq s$, the word $c_{k-1} \cdot p_k$ is a calculus of $\mathcal{A}$ on v, and thus $q_k \xrightarrow[\mathcal{A}]{v} p_k$. On the other hand, $\mathcal{A}$ is complete, and then for all k such that $s < k \leq m$, there is a state p_k such that $q_k \xrightarrow[\mathcal{A}]{v} p_k$. For each k such that $1 \leq k \leq s$, we set $f_k = 1$ if $c'_{k-1} \cdot p_k$ contains a repeated state, $f_k = 0$ otherwise. For each k such that $s < k \leq m$, we set $f_k = 0$ if $q_k \overset{v}{\underset{\mathcal{A}}{\rightsquigarrow}} p_k$, and $f_k = 1$ otherwise. The state $\vec{p} = ((p_1, f_1), \ldots, (p_m, f_m))$ thus defined is such that $\vec{q_0} \xrightarrow[\mathcal{A}_{N_r}]{v} \vec{p}$. For this state, the sequence $(j_k)_{k \geq 0}$ is defined by $j_0 = 1$, $j_k = k+1$ for all $k < s$, and $j_k = j_{k-(s-s')}$ for all $k \geq s$. s is the least integer verifying $j_s \in \{j_k \mid 1 \leq k < s\}$ and s' is such that $s' < s$ and $j_{s'} = j_s$. Moreover, π is successful then $f_{j_k} = 1$ for an integer k such that $s' \leq k \leq s$. This shows that $\vec{p}$ is a final state of $\mathcal{A}_{N_r}$ and finishes the proof of the inclusion $N_r \subseteq L(\mathcal{A}_{N_r})$.

The languages N_r are recognized by the automata $\mathcal{A}_{N_r}$ and are thus rational. From the equality (1), we deduce that the language $L_\$$ is rational too. Finally, we have shown the following proposition.

Proposition 4 *Let L be a rational ω-language on the alphabet A and let $L_\$$ be the language of finite words on the alphabet $A \cup \$$, defined by $L_\$ = \{u \cdot \$ \cdot v \mid u \cdot v^\omega \in L\}$. Then $L_\$$ is rational.* $\diamond$

It is easy to construct an automaton recognizing $L_\$$ from the automata $\mathcal{A}$ and $\mathcal{A}_{N_r}$. Indeed, let $\mathcal{A}_\$$ be the disjoint union of automata $\mathcal{A}$ and $\mathcal{A}_{N_r}$, for each r, to which we're adding the edges $(q_r, \$, \vec{q_0}_r)$—$\vec{q_0}_r$ is the initial state of $\mathcal{A}_{N_r}$. The initial states of $\mathcal{A}_\$$ are those of $\mathcal{A}$ and the final states of $\mathcal{A}_\$$ are those of all $\mathcal{A}_{N_r}$. Then obviously $L(\mathcal{A}_\$) = L_\$$.

5 Determinising $\mathcal{A}_\$$

The automaton $\mathcal{A}_\$$ that we built in the previous paragraph isn't deterministic. One reason for this is that it contains $\mathcal{A}$, which itself is not generally deterministic. However, accessible states of the subset automata built from $\mathcal{A}_\$$ have a particular shape, which provides a simple representation of these states and a bound to its number.

We first build for each state q_r of $\mathcal{A}$ the subset automaton $\mathcal{P}(\mathcal{A}_{N_r})$ of the automaton $\mathcal{A}_{N_r}$. Its initial state is the singleton $\{\vec{q_0}\}$, and we denote δ its transition function. Let P be an accessible state of $\mathcal{P}(\mathcal{A}_{N_r})$ and v a word of A^* such that $\delta(\{\vec{q_0}\}, v) = P$. Let $\vec{p}$ and $\vec{p}'$ two states of $\mathcal{A}_{N_r}$, members of P — $\vec{p} = ((p_1, f_1), \ldots, (p_m, f_m))$, $\vec{p}' = ((p'_1, f'_1), \ldots, (p'_m, f'_m))$. Then, each state $\vec{p}'' = ((p''_1, f''_1), \ldots, (p''_m, f''_m))$ such that $(p''_k, f''_k) \in \{(p_k, f_k), (p'_k, f'_k)\}$ for each k such that $1 \leq k \leq m$ is a member of P. This is a direct consequence of Lemma 2. The state P is thus entirely defined by the sets $P_k = \{(p_k, f_k) \mid \vec{p} = ((p_1, f_1), \ldots, (p_m, f_m)) \in P\}$, i.e. P is the set of states $\vec{p} = ((p_1, f_1), \ldots, (p_m, f_m))$ such that $(p_k, f_k) \in P_k$ for each k such that $1 \leq k \leq m$. The set of states of the subset automaton is in bijective correspondence with the set $(\mathcal{P}(Q \times \{0, 1\}))^m$, which contains 2^{2m^2} elements.

The automata $\mathcal{A}_{N_r}$, and thus $\mathcal{P}(\mathcal{A}_{N_r})$, have the same stucture — the only thing that changes is final states — and there is a straightfoward construction to build a deterministic automaton recognizing a language such as $N = \cup_{i=1}^{k} N_{r_i}$, the union of languages N_r. This automaton is isomorphic to the common structure of $\mathcal{P}(\mathcal{A}_{N_r})$, and its set of final states is the union of the final states of automata $\mathcal{P}(\mathcal{A}_{N_{r_i}})$.

Now we build a deterministic automaton that recognizes $L_\$$. This automaton is the disjoint union of $\mathcal{P}(\mathcal{A})$, the subset automaton of the automaton $\mathcal{A}$, and of automata that we built previously recognizing each language N, the union of the languages N_r to which we add edges $(P, \$, \bar{q_0}_P)$, where $\bar{q_0}_P$ is the initial state of the automaton recognizing the language $\cup_{r\in P} N_r$. The automaton we have built is deterministic and recognizes the language $L_\$$. There are at most 2^m states in $\mathcal{P}(\mathcal{A})$, and there are at most 2^m unions of languages N_r, which are recognized by automata with at most 2^{2m^2} states. Finally, there are at most $2^m + 2^{2m^2+m}$ states in this automaton.

6 Infinite Words and Second Construction

Let L be a rational ω-language and $L_\$$ the rational language defined in the previous paragraphs. Let $u \cdot \$ \cdot v$ be a word in $L_\$$ and $u' \cdot \$ \cdot v'$ a word in $A^* \cdot \$ \cdot A^+$ such that $u \cdot v^\omega = u' \cdot v'^\omega$. It is then clear that $u' \cdot \$ \cdot v'$ is an element of $L_\$$. Let us define the equivalence relation $\overset{UP}{\equiv}$ on the language $A^* \cdot \$ \cdot A^+$ in the following way.

$$u \cdot \$ \cdot v \overset{UP}{\equiv} u' \cdot \$ \cdot v' \text{ if and only if } u \cdot v^\omega = u' \cdot v'^\omega,$$

Then, $L_\$$ is saturated by $\overset{UP}{\equiv}$.

Let K be a rational language of $(A \cup \$)^*$ contained in $A^* \cdot \$ \cdot A^+$. A necessary condition for K to be $L_\$$ for a rational ω-language L is that K is saturated by $\overset{UP}{\equiv}$. We will show that this condition is sufficient too, and we will construct an automaton that recognizes L. We first need the following lemma.

Lemma 5 *Let M and N be two languages of A^* such that $M \cdot N^* = M$ and $N^+ = N$. Then, for each infinite word $\alpha \in A^\omega$, $\alpha \in UP(M \cdot N^\omega)$ if and only if there exist two words $u \in M$ and $v \in N$ such that $u \cdot v^\omega = \alpha$.*

Proof It is clear that for each words $u \in M$ and $v \in N$, $u \cdot v^\omega \in M \cdot N^\omega$. Conversely, let $\alpha = u \cdot v^\omega$ be a ultimately periodic word of $M \cdot N^\omega$, and $u_0, u_1, \ldots$ a sequence of words such that $u_0 \in M$, $u_i \in N$ for each $i > 0$ and $u_0 \cdot u_1 \cdot \ldots = u \cdot v^\omega$. We set $l = |v|$, $l_i = |u_0 \cdot \ldots \cdot u_i|$ for each integer i, and $P = \{l_i \mid i \in \mathbb{N}\}$. P is an infinite subset of $\mathbb{N}$, thus there is an integer k such that $P \cap (l\mathbb{N} + k)$ is infinite. Let n_1 and n_2 be two integers such that $0 < n_1 < n_2$, $l_{n_1} > |u|$, and l_{n_j} is in $l\mathbb{N} + k$ for $j = 1$ and 2. We can then find two words v_1 and v_2 such that $v = v_1 \cdot v_2$, and two integers k_1 and k_2 such that $u_0 \cdot \ldots \cdot u_{n_1} = u \cdot v^{k_1} \cdot v_1$ and $u_{n_1+1} \cdot \ldots \cdot u_{n_2} = v_2 \cdot v^{k_2} \cdot v_1$. The two ultimately periodic words $u \cdot v^\omega$ and $u_0 \cdot \ldots \cdot u_{n_1} \cdot (u_{n_1+1} \cdot \ldots \cdot u_{n_2})^\omega$ are equal. We deduce from the hypothesis on languages M and N that $u_0 \cdot \ldots \cdot u_{n_1} \in M$ and $u_{n_1+1} \cdot \ldots \cdot u_{n_2} \in N$, and this ends the proof. $\diamond$

Let then $K \subseteq A^* \cdot \$ \cdot A^+$ be a rational language saturated by $\overset{UP}{\equiv}$. Let $\mathcal{A} = (Q, I, F, E)$ a deterministic automaton which recognizes K. We denote by δ the transition function of the automaton $\mathcal{A}$ and let q_0 be its initial state. We set $Q_d = \{q \in Q \mid \exists u \cdot \$ \cdot v \in K, q = \delta(q_0, u)\}$. For each state $q \in Q_d$ we denote by M_q the language of words u such that $\delta(q_0, u) = q$, and we denote by N_q the language of words v such that $\delta(q, v)$ if a final state. M_q and N_q are rational languages and $K = \cup_{q \in Q_d} M_q \cdot \$ \cdot N_q$ because K is a subset of $A^* \cdot \$ \cdot A^+$.

The language N_q is recognized by the automaton $\mathcal{A}_q = (Q, \{\delta(q, \$)\}, F, E)$, and for each final state q_f, we let the rational language N_{q,q_f} be the set of words v such that $\delta(q, v) = q$ and $\delta(q, \$ \cdot v) = q_f = \delta(q_f, v)$. This language is composed of words v of N_q that loop on both q and q_f, the final state of the calculus of $\mathcal{A}_q$ on v. Finally, we define the ω-rational language L by

$$L = \bigcup_{(q,q_f) \in Q_d \times F} M_q \cdot N_{q,q_f}^{\omega} \tag{2}$$

The languages M_q and N_{q,q_f} satisfy the hypothesis of Lemma 5 , i.e. $N_{q,q_f}^{+} = N_{q,q_f}$ and $M_q \cdot N_{q,q_f}^{*} = M_q$. Each ultimately periodic word α which is an element of $M_q \cdot N_{q,q_f}^{\omega}$ is equal to $u \cdot v^{\omega}$ with $u \in M_q$ and $v \in N_{q,q_f}$. Then, $u \cdot \$ \cdot v \in K$ and we deduce from the saturation of K by $\overset{UP}{\equiv}$ that all words $u \cdot \$ \cdot v$ such that $\alpha = u \cdot v^{\omega}$ are elements of K. We have thus shown the inclusion $L_\$ \subseteq K$.

Conversely, let $u \cdot \$ \cdot v$ be a word of K. For each integer k, words $u \cdot \$ \cdot v$ and $u \cdot v^k \cdot \$ \cdot v$ represent the same ultimately periodic word. K is saturated by $\overset{UP}{\equiv}$ thus, $u \cdot v^k \cdot \$ \cdot v \in K$ and $\delta(q_0, u \cdot v^k) \in Q_d$. Let the sequence of states $p_k \in Q_d$ be defined by $p_k = \delta(q_0, u \cdot v^k)$. Q_d is finite, so we can find two integers r and m such that $m \geq 1$, $p_r = p_{r+m}$ and for each integer $k \leq r + m$, $p_k \notin \{p_0, \ldots, p_{k-1}\}$. We may show by a simple induction that $p_{k+m} = p_k$ for each integer $k \geq r$. We set $r = sm + r'$, with $0 \leq r' < m$, and $k_1 = r + m - r' = (s+1)m$. Then, we get $p_{2k_1} = p_{k_1 + (s+1)m} = p_{k_1}$, because $k_1 > r$, and then $q_0 \xrightarrow[\mathcal{A}]{u \cdot v^{k_1}} p_{k_1} \xrightarrow[\mathcal{A}]{v^{k_1}} p_{k_1}$. We set $q = p_{k_1}$. With a similar argument on the sequence of final states p'_k defined by $p'_k = \delta(q, \$ \cdot (v^{k_1})^k)$, we show that there exists an integer k_2 such that $p \xrightarrow[\mathcal{A}]{\$ \cdot v^{k_1 k_2}} p'_{k_2} \xrightarrow[\mathcal{A}]{v^{k_1 k_2}} p'_{k_2}$. We set $q_f = p'_{k_2}$. We have thus showed that $u \cdot v^{\omega} \in M_q \cdot N_{q,q_f}^{\omega}$, because $u \cdot v^{\omega} = u \cdot v^{k_1} \cdot (v^{k_1 k_2})^{\omega}$ and the words $u \cdot v^{k_1}$ and $v^{k_1 k_2}$ are in M_q and in N_{q,q_f}, respectively . The infinite word $u \cdot v^{\omega}$ is in L, and this proves the set inclusion $K \subseteq L_\$$. Finaly, we have showed the following proposition.

Proposition 6 *Let $K \subseteq A^* \cdot \$ \cdot A^+$ a rational language. Then, there exists a rational ω-language L such that $K = L_\$$ if and only if K is saturated by the equivalence $\overset{UP}{\equiv}$.* ◇

We can build directly from $\mathcal{A}$ an automaton recognizing the ω-language L. The set Q_d can be effectively computed. For each state $q \in Q_d$, the language M_q is recognized by the automaton $(Q, I, \{q\}, E)$, which have m states. For each final state q_f, the language N_{q,q_f} is the intersection of the tree languages $L(Q, \{q\}, \{q\}, E), L(Q, \{\delta(q, \$)\}, \{q_f\}, E)$ and $L(Q, \{q_f\}, \{q_f\}, E)$, and this language is recognized by an automaton with m^3 states. Each ω-language $M_q \cdot N^{\omega}_{q,q_f}$ is thus recognized by an automaton with $m + m^3$ states. There are at most m^2 pairs $(q, q_f) \in Q_d \times F$, and then the ω-language L is recognized by an automaton which has at most $m^3 + m^5$ states.

7 Remarks

The set $\hat{K} = \{u \cdot v^{\omega} \mid u \cdot \$ \cdot v \in K\}$ of ultimately periodic words corresponding to a rational set K of finite words in $A^* \cdot \$ \cdot A^+$ needs not be equal to $UP(M)$ for any rational language $M \in \mathcal{R}at(A^{\omega})$. In fact, there exists $M \in \mathcal{R}at(A^{\omega})$ such that $\hat{K} = UP(M)$ if and only if the smallest language containing K saturated by $\overset{UP}{\equiv}$ is rational, and this is not always the case.

For example, $K = \$ \cdot A^+$ is a rational set of finite words include in $A^* \cdot \$ \cdot A^+$. $\hat{K}$ is the set of periodic words on the alphabet A and $(\hat{K})_\$$ is not a rational set if A has more than one letter. In fact, if a and b are distinct letters of A, $\$ \cdot a \cdot b^n \in K$ for each $n \in \mathrm{N}$ and then $a \cdot b^n \cdot \$ \cdot a \cdot b^n \in (\hat{K})_\$$ for each integer n. But for each integer n', $n' < n$, $a \cdot b^{n'} \cdot \$ \cdot a \cdot b^n \notin (\hat{K})_\$$ because the word $a \cdot b^{n'} \cdot (a \cdot b^n)^{\omega}$ is not periodic. The language $(\hat{K})_\$$ may not be rational because it does not even satisfy pumping lemma conclusions.

8 Conclusion

We have solved in principle the problem of building an effective one-to-one correspondance between Büchi automata and dfa's recognizing the languages $UP(M)_\$$. This raises two immediate natural questions. How can we decide efficiently that a rational language $K \subseteq A^* \cdot \$ \cdot A^+$ is saturated by $\overset{UP}{\equiv}$? How can we decide that $(\hat{K})_\$$ is rational ? More generally, the question is raised to derive from canonical forms of the dfa's recognizing the $UP(M)_\$$ canonical forms for the Büchi automata recognizing the M's and hopefully efficient practical algorithms for the manipulation of Büchi automata.

9 Bibliography

[1] Arnold, A., "A Syntactic Congruence for Rational ω-Languages," *Theoretical Computer Science,* **39**, (1985), 333-335.

[2] Calbrix, H., Nivat, M., Podelski, A., "Une méthode de décision de la logique monadique du second ordre d'une fonction successeur", submited to *Comptes Rendus de l'Académie des Sciences, Série I.*

[3] Calbrix, H., Nivat, M., Podelski, A., "Sur les mots ultimement périodiques des langages rationnels de mots infinis", submited to *Comptes Rendus de l'Académie des Sciences, Série I.*

[4] Eilenberg, S., "Automata, Languages and Machines," Vol. A, *Academic Press,* (1974).

[5] Safra, S., "On the complexity of ω-automata," in: *Proc. 29th Ann. IEEE Symp. on Foundations of Computer Science,* (1988), 319-327.

[6] Sistla, A.P., M.Y. Vardi and P. Volper, "The Complementation Problem for Büchi automata with Application to Temporal Logic," *Theoret. Comput. Sci.,* **49**, (1987), 217-237.

[7] Thomas, W., "Automata on Infinite Objects," in *Handbook of Theoretical Computer Science, J. van Leeuven ed., Elsevier,* (1990), 133-191.

Category of Δ–Functors

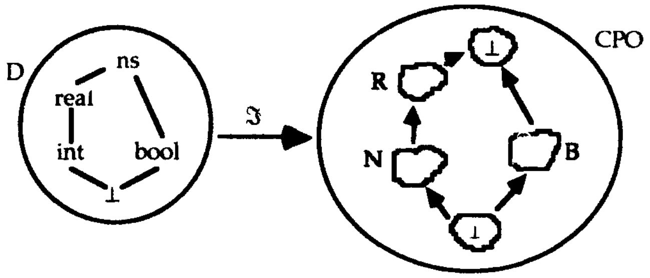
Adrian Fiech
Department of Computing and Information Sciences
Kansas State University
E–mail : fiech@cis.ksu.edu

Abstract :

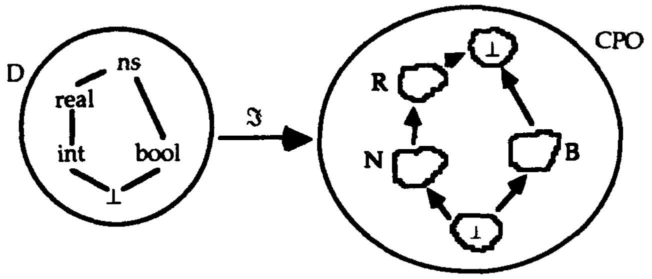
We define the category Func_Δ with functors $F:D_F \rightarrow \mathrm{SCOTT}$ ($D_F \in \mathrm{CPO}$) as objects and pairs ($f:D_F \rightarrow D_G, \eta : F \twoheadrightarrow G \circ f$) as morphisms ($\eta$ is a natural transformation). We show that this category is closed under the common domain theoretical operations $+, \times, \perp$ and $\rightarrow$. The category Func_Δ is an O–category and all the operations we define on it are continuous functors, so we will be able to solve recursive equations in Func_Δ. We also show that if we restrict Func_Δ to functors that preserve directed colimits then the category is not closed under the $\rightarrow$ operation. The category Func_Δ is a basis for a model of second–order lambda calculus with subtyping.

0 Introduction

The category of Δ–Functors is motivated by John Reynolds' work on category–sorted algebras [Reynolds 1980]. Reynolds' work addresses the problem of treatment of coercions between types. The key idea in category–sorted algebras can be expressed with the help of the following figure (see next page) :



The cpo D contains the type names and the functor $\Im$ maps type names to corresponding cpos. The fact that **int** is a subtype of **real** is described by the mapping $\Im[\text{int} \sqsubseteq \text{real}]$ from the set of natural numbers into the set of real numbers. The meaning of a polymorphic operator like **succ**, which takes an element x from int or real and returns x+1 (for all other types it returns error), can be expressed as a natural transformation between the functors $\Im$ and $\Im \circ f_{succ}$, where $f_{succ}(\text{int})=\text{int}$, $f_{succ}(\text{real})=\text{real}$ and $f_{succ}(d)=\text{ns}$ otherwise. This approach can be generalized by allowing recursive definition of the domain D, e.g. $D \cong B+D \rightarrow D$, where B is the set of base types. The function f_{succ} can now be treated as a type name and can be included in the domain D. This allows the polymorphic operators, like id or succ to be higher order and to self apply, which is of importance to models for lambda calculi. The cpo corresponding to the type name f_{succ} should be the cpo of natural transformations $\Im \xrightarrow{\bullet} \Im \circ f_{succ}$. This generalization of Reynolds' work was introduced by David Schmidt [Schmidt 1990]. As the functor $\Im$ is now defined recursively we must be able to solve recursive equations in a category Func_{CPO} of functors $F{:}D_F \rightarrow \text{CPO}$. The category Func_{CPO} is closed under the counterparts to domain theoretical operations $+$, $\times$, $\perp$, $\rightarrow$ and all these operations are continuous functors on Func_{CPO} [Schmidt 1990]. We can use the ideas developed by Schmidt to give a



The cpo D contains the type names and the functor $\Im$ maps type names to corresponding cpos. The fact that **int** is a subtype of **real** is described by the mapping $\Im[\text{int} \sqsubseteq \text{real}]$ from the set of natural numbers into the set of real numbers. The meaning of a polymorphic operator like **succ**, which takes an element x from int or real and returns x+1 (for all other types it returns error), can be expressed as a natural transformation between the functors $\Im$ and $\Im \circ f_{succ}$, where $f_{succ}(\text{int})=\text{int}$, $f_{succ}(\text{real})=\text{real}$ and $f_{succ}(d)=\text{ns}$ otherwise. This approach can be generalized by allowing recursive definition of the domain D, e.g. $D \cong B+D \rightarrow D$, where B is the set of base types. The function f_{succ} can now be treated as a type name and can be included in the domain D. This allows the polymorphic operators, like id or succ to be higher order and to self apply, which is of importance to models for lambda calculi. The cpo corresponding to the type name f_{succ} should be the cpo of natural transformations $\Im \xrightarrow{\bullet} \Im \circ f_{succ}$. This generalization of Reynolds' work was introduced by David Schmidt [Schmidt 1990]. As the functor $\Im$ is now defined recursively we must be able to solve recursive equations in a category Func_{CPO} of functors $F{:}D_F \rightarrow CPO$. The category Func_{CPO} is closed under the counterparts to domain theoretical operations $+$, $\times$, $\perp$, $\rightarrow$ and all these operations are continuous functors on Func_{CPO} [Schmidt 1990]. We can use the ideas developed by Schmidt to give a model for polymorphic λ2–calculus with subtyping [Fiech 1993; Fiech, Schmidt 1993]. Algebraic domains play an important role in domain theory and the λ2–calculus model benefits if we can work only with algebraic cpos. Motivated by this, we define the category Func_Δ with functors $F : D_F \rightarrow \mathrm{SCOTT}$ as objects and make sure that this category is closed under the previously mentioned operations. Most of the problems are caused by the $\rightarrow$ operator. In general the new functor $F \rightarrow G : [D_F \rightarrow D_G] \rightarrow \mathrm{CPO}$, with $F \rightarrow G[f] := F \twoheadrightarrow G \circ f$ (the cpo of natural transformations) doesn't produce an algebraic cpo. In part we can resolve this by requiring that $F[d_k \sqsubseteq d_l]$ preserves finite elements and $G[d_k \sqsubseteq d_l]$ preserves nonempty infimas [Fiech, Huth 1991]. Although now $F \rightarrow G[f]$ is a Scott–domain, the category of such functors is still not closed under the $\rightarrow$ operation as $F \rightarrow G[f_i \sqsubseteq f_j]$ doesn't necessary preserve finite elements. We can solve this problem if we require that for all $F \in \mathrm{Func}_\Delta$, $F[d_k \sqsubseteq d_l]$ is a lower embedding (p.4). From all the domain–theoretical operations on Func_Δ only the definition of the $\mathcal{P}$–operator (powerdomain constructor) isn't immediately clear. We define the $\mathcal{P}$–operator and give a justification for our choice. Another interesting question about the category Func_Δ is if we can require that all functors in it preserve directed colimits (lubs of directed sets). The answer to this question is unfortunately no, as again the functor $F \rightarrow G$ may not preserve directed colimits. This negative result holds for any category of functors $F : D_F \rightarrow C$, where C is a subcategory of CPO which contains the one- and two- element cpos.

The framework discussed in this paper also generalizes the functor–category semantics described by Oles, Reynolds, O'Hearn and Tennent [Oles 1982,1985; Reynolds 1980; O'Hearn and Tennent 1993].

1 Basic domain and category theory

This section is a brief review of the necessary definitions in domain and category theory.

A **partial order** $(D, \sqsubseteq)$ is a set D and a binary relation $\sqsubseteq$ on D, which is reflexive, antisymmetric and transitive. For a subset $M \subseteq D$ an element $x \in D$ is called an **upper bound** of M if for all $m \in M$, $m \sqsubseteq x$. An element $x \in D$ is called **least upper bound (lub)** of M, $\bigsqcup M$ if it is an upper bound of M and if for all upper bounds x^* of M $x \sqsubseteq x^*$. Analogously we can define **lower bound** and **greatest lower bound** $\sqcap$. The lower set of X, $\downarrow X$ is defined as $\downarrow X := \{d \in D \mid d \sqsubseteq x \text{ for some } x \in X\}$ (analogously $\uparrow X$). A subset $M \subseteq D$ is **directed** if for every finite subset $M' \subseteq M$ there exists an upper bound $m \in M$ for M'. A **complete partial order (cpo)** is a poset $(D, \sqsubseteq)$, st. every directed subset $M \subseteq D$ has a least upper bound $\bigsqcup^{\uparrow} M \in D$. For a subset M of a cpo X, the **$\bigsqcup^{\uparrow}$-closure** of M is defined as the smallest subset M^* of X that contains M, st. for every directed $N \subseteq M^*$, $\bigsqcup^{\uparrow} N \in M^*$. A function $f: A \to B$ between two posets A and B is **continuous** if for any directed set $M \subseteq A$, f(M) is also directed and $f(\bigsqcup^{\uparrow} M) = \bigsqcup^{\uparrow}\{f(m) \mid m \in M\}$ whenever $\bigsqcup^{\uparrow} M$ exists. A continuous function $e: D \to E$ is an **embedding** if there exists a continuous function $p: E \to D$, st. $p \circ e = id_D$ and $e \circ p \sqsubseteq id_E$. We call p a **projection** and denote it by e^R. If e(D) is a lower set in E ($e(D) = \downarrow e(D)$) then e is a **lower embedding**. An element $x \in (D, \sqsubseteq)$ is **finite** if for all directed sets M with $x \sqsubseteq \bigsqcup^{\uparrow} M$ there exists some element $m \in M$, st. $x \sqsubseteq m$. We denote the set of all finite elements in D as **K(D)**. The cpo $(D, \sqsubseteq)$ is **algebraic** if for all $x \in D$ the set $M = \{a \in K(D) \mid a \sqsubseteq x\}$ is directed and $x = \bigsqcup^{\uparrow} M$. $(D, \sqsubseteq)$ is **bounded**

complete if every nonempty, bounded subset $X \subseteq D$ has a lub in D. A **Scott domain** is a bounded complete, algebraic cpo. A subset $U \subseteq D$ is called **Scott–open** if $U = \uparrow U$ and if for every directed set M, $\bigsqcup M \in U \Rightarrow m \in U$ for some $m \in M$. For two nonempty subsets $A, B \subseteq D$ $A \sqsubseteq_R B$ if for every $a \in A$ and every Scott–open set U with $a \in U$ there exists some $b \in B \cap U$. $A \equiv_R B$ iff $A \sqsubseteq_R B$ and $B \sqsubseteq_R A$. For a nonempty set $A \subseteq D$ we define the equivalence class $[A] := \{B \subseteq D \mid B \equiv_R A\}$. The relational powerdomain **($\mathcal{P}_R(D), \sqsubseteq_R$)** is the cpo of the nonempty subsets of the elements in D quotiented by the relation $\equiv_R$ and partialy ordered by $\sqsubseteq_R$. If D is algebraic then $\mathcal{P}_R(D)$ is a Scott–domain. For any $A \subseteq D$ there is a canonical representation of [A] which is $\bigcup_{A^* \in [A]} A^* \in [A]$. For this canonical A, $A = \downarrow A$ holds and A is closed under lubs of directed sets. If $f : D \rightarrow E$ is a continuous function then $f^+ : \mathcal{P}_R(D) \rightarrow \mathcal{P}_R(E)$ defined as $f^+(A) := [\{f(a) \mid a \in A\}]$, is also continuous.

A **category** Ω is a quadruple $\Omega = (O, \mathrm{hom}, \mathrm{id}, \circ)$ where (i) **O** is a class whose members are Ω-objects (ii) for each pair (A,B) of Ω-objects **hom(A,B)** is a set whose members are called Ω–morphism from A to B (iii) for each Ω–object A $\mathbf{id}_A : A \rightarrow A$ is the A–identity (iv) $\circ$ is a composition operator assigning to each pair of morphism $f : A \rightarrow B$, $g : B \rightarrow C$ the composite morphism $g \circ f : A \rightarrow C$. We also require that $f \circ (g \circ h) = (f \circ g) \circ h$ $(h : C \rightarrow D)$ and $\mathrm{id}_A \circ f = f$, $g \circ \mathrm{id}_B = g$. The class O is usually denoted by **Ob(Ω)** and the class of Ω–morphisms **Mor(Ω)** is defined as the disjoint union of all the sets hom(A,B) in Ω. The category **CPO** has as objects complete partial orders and as morphisms continuous functions. In **CPO**$_\perp$ all cpos have a least element $\perp$. In the category **SCOTT** the objects are Scott–domains.

Let Ω,Φ be categories. A **functor** $F:\Omega \rightarrow \Phi$ is a function that assigns to each Ω–object A a Φ–object $F(A)$ and to each Ω-morphism $f:A \rightarrow B$ a Φ–morphism $F(f):F(A) \rightarrow F(B)$, st. $F(f \circ g)=F(f) \circ F(g)$ and $F(id_A)=id_{F(A)}$.
Let $F,G:\Omega \rightarrow \Phi$ be functors. A **natural transformation** $\tau:F \twoheadrightarrow G$ is a function that assigns to each Ω–object A a Φ–morphism $\tau_A:F_A \rightarrow G_A$, st. for each Ω-morphism $f:A \rightarrow B$ $G(f) \circ \tau_A=\tau_B \circ F(f)$. If $F,G : D \rightarrow CPO$ are two functors then $F \twoheadrightarrow G$, the set of natural transformations from F into G together with the ordering $\eta \sqsubseteq \phi \Leftrightarrow \forall d \in D:\eta_d \sqsubseteq \phi_d$ is a cpo.
A **sink** in a category Ω is a pair $((f_i)_{i \in I},A)$ consisting of an object $A \in Ob(\Omega)$ and a family of morphism $f_i:A_i \rightarrow A$ in Ω. If $F:\Omega \rightarrow \Phi$ is a functor then an Φ–sink $(F[i] \xrightarrow{f_i} A)_{i \in Ob(\Omega)}$ is **natural** for F if for each Ω–morphism $d : i \rightarrow j$, $f_j \circ F[d]=f_i$. A **colimit** of F is a natural sink $(F[i] \xrightarrow{f_i} C)_{i \in Ob(\Omega)}$, st. for any other natural sink $(F[i] \xrightarrow{g_i} A)_{i \in Ob(\Omega)}$ there exists a unique morphism $h:C \rightarrow A$ with $h \circ f_i=g_i$ for all $i \in Ob(\Omega)$. A category Φ is **cocomplete** if every functor F from a small category Ω into Φ has a colimit in Φ. The category CPO is cocomplete (but $CPO_\perp$ is not).

2 The category Func$_{CPO}$

The category Func$_{CPO}$ was first introduced by David Schmidt [Schmidt 1990]. The category Func$_\Delta$ is based on the definitions given by Schmidt. The difference between this two categories is that the functors in Func$_\Delta$ map elements in D into Scott–domains instead into arbitrary cpos. This restriction causes many problems when we want to close Func$_\Delta$ under the operations $+,\times,\perp,\rightarrow$ and $\mathcal{P}$. A solution to this problems will be given in the next section. In this section we adapt the definitions for Func$_{CPO}$ and the operations $+, \times, \perp, \rightarrow$ given by Schmidt. We also define the powerset–operator $\mathcal{P}$ on Func_{CPO}.

Definition 2.1

The category $\mathbf{Func_{CPO}}$ has as objects pairs $(D_F \in \mathrm{Ob}(CPO_\perp)$, $F : D_F \rightarrow CPO_\perp$), st. F is a functor, $F[d \sqsubseteq d']$ is a strict function for all $d \sqsubseteq d' \in D_F$ and $F[\perp] \cong \{\perp\}$. The morphisms between two objects F and G are pairs (f, η), where f is a strict function from D_F into D_G and η is a natural transformation in $F \twoheadrightarrow G \circ f$. Composition on morphisms $(f : D_F \rightarrow D_G,\ \eta : F \twoheadrightarrow G \circ f)$ and $(g : D_G \rightarrow D_H,$ $\gamma : G \twoheadrightarrow H \circ g)$ is defined as $(g \circ f, \lambda d \in D_F . \gamma_{f(d)} \circ \eta_d)$.

The category Func_{CPO} is closed under the common domain–theoretical operations which are defined in the following.

Definition 2.2

Let $F_1, F_2 \in \mathrm{Ob}(\mathrm{Func}_{CPO})$.

a) The bottom functor

$$\perp : \mathrm{Func}_{CPO} \rightarrow \mathrm{Func}_{CPO}$$

is defined as (we write $F_\perp$ instead of $\perp(F)$) :

$F_\perp : D_\perp \rightarrow CPO$

$F_\perp[\perp] = \{\perp\}$ the one element cpo

$F_\perp[d \in D] = F[d]$

$F_\perp[\perp \sqsubseteq d] = \lambda x.\ \perp_{F[d]}$

$F_\perp[d_1 \sqsubseteq d_2] = F[d_1 \sqsubseteq d_2]$

$\perp(f : D_1 \rightarrow D_2, \eta : F_1 \twoheadrightarrow F_2 \circ f)$

$= (\underline{\lambda} x.fx, \lambda d \in D_{1_\perp}.\text{if } d = \perp \text{ then } \lambda x.\perp \text{ else } \eta(d))$

b) The product functor

$$\times : \mathrm{Func}_{CPO} \times \mathrm{Func}_{CPO} \rightarrow \mathrm{Func}_{CPO}$$

is defined as

$F_1{\times}F_2 : D_1{\times}D_2 \to CPO$

$F_1{\times}F_2[(d_1,d_2)] = F_1[d_1]{\times}F_2[d_2]$

$F_1{\times}F_2[(d_1 \sqsubseteq d_1', d_2 \sqsubseteq d_2')] = (F_1[d_1 \sqsubseteq d_1'], F_2[d_2 \sqsubseteq d_2'])$

$\times((f_1,\eta_1),(f_2,\eta_2)) = (f_1{\times}f_2, \eta_1{\times}\eta_2)$, where

$\eta_1{\times}\eta_2 := \lambda(d_1,d_2)\in D_1{\times}D_2.\eta_1(d_1){\times}\eta_2(d_2)$

c) The sum functor

$$+ : Func_{CPO} \times Func_{CPO} \to Func_{CPO}$$

is defined as

$F_1{+}F_2 : D_1{+}D_2 \to CPO$

$F_1{+}F_2[\bot] = \{\bot\}$

$F_1{+}F_2[(1,d_1)] = F_1[d_1]$

$F_1{+}F_2[(2,d_2)] = F_2[d_2]$

$F_1{+}F_2[\bot \sqsubseteq (i,d)] = \lambda x. \bot_{F_i[d]}$

$F_1{+}F_2[(i,d_1) \sqsubseteq (i,d_2)] = F_i[d_1 \sqsubseteq d_2]$

$+((f_1,\eta_1),(f_2,\eta_2)) = (f_1{+}f_2, \eta_1{+}\eta_2)$, where

$\eta_1{+}\eta_2 := \lambda x\in D_1{+}D_2.$cases x of

$\bot \to \lambda x.\bot \mid (1,d) \to \eta_1(d) \mid (2,d) \to \eta_2(d)$

e) The exponentiation functor

$$\to : Func_{CPO}{}^{op} \times Func_{CPO} \to Func_{CPO}$$

is defined as

$F_1 \to F_2 : [D_1 \to D_2] \to CPO$

$F_1 \to F_2[f] = F_1 \twoheadrightarrow F_2 \circ f$ the cpo of natural transformations

$F_1 \to F_2[f_1 \sqsubseteq f_2] = \lambda\eta\in F_1 \twoheadrightarrow F_2\circ f_1.(\lambda t\in D_1.F_2[f_1 t \sqsubseteq f_2 t]\circ(\eta_t))$

$\to((f_1,\eta_1),(f_2,\eta_2)) = (\lambda g\in[D_1 \to D_2].f_2\circ g\circ f_1, \eta_1 \to \eta_2)$ where

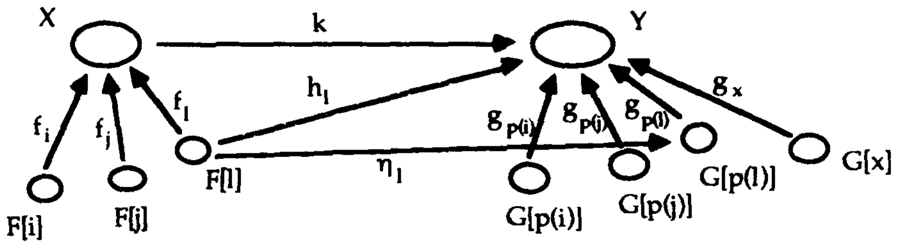
$\eta_1 \to \eta_2 = \lambda g\in[D_1 \to D_2].\lambda\phi\in F_1 \twoheadrightarrow F_2\circ g . \eta_2\circ\phi\circ\eta_1$

Proposition 2.3 [Schmidt 1990]

The category $Func_{CPO}$ is an O–category and the functors $+, \times, \perp, \rightarrow$ are locally continuous.

□

The only remaining operation we have to define is the powerset–operator $\mathcal{P}$.
Lets consider a simple functor $G:\{\bullet\} \rightarrow CPO$ with $G[\bullet]=Y$. When we apply $\mathcal{P}$ to G, as a result we would expect a functor G^P from $\{\bullet\}=\mathcal{P}_R(\{\bullet\})$ into CPO, where $G^P[\bullet]=\mathcal{P}_R(Y)$. Let now $F:D_F \rightarrow CPO$ be an arbitrary functor in $Func_{CPO}$. The functor F^P should be from $\mathcal{P}_R(D_F)$ into CPO. But what should the cpo $F^P[A]$ be, where $A \in \mathcal{P}_R(D_F)$? We should look at the union of all elements in F[a], $a \in A$. We can find all elements $x \in \bigcup_{a\in A} F[a]$ in the colimit of the functor F. This colimit $(F[i] \xrightarrow{f_i} X)_{i\in D_F}$ exists in CPO [Fiech 1992] and as $f_\perp(\perp)$ is the least element in X it also exists in $CPO_\perp$. In the colimit cpo X, elements which are essentially equal (like $2 \in int$ and $2 \in real$) are identified. We can define the poset $X_A := \bigcup_{a\in A} f_a(F[a])$ and then take for $F^P[A]$ the relational powerset domain on X_A^* (the $\bigsqcup^{\uparrow}$–closure of X_A in X), $F^P[A] := \mathcal{P}_R(X_A^*)$. When applying this to the functor $G:\{\bullet\} \rightarrow CPO$ we get the expected functor G^P. Because $A \sqsubseteq_R B \Leftrightarrow A \subseteq B$ we have $X_A^* \subseteq X_B^*$ and therefore there exists the obvious inclusion function $\iota: X_A^* \rightarrow X_B^*$. For $F^P[A \sqsubseteq_R B]$ we can take the function ι^+. It is clear that F^P preserves identities and composition, so F^P is indeed a functor. The functor $\mathcal{P}: Func_{CPO} \rightarrow Func_{CPO}$ still has to be defined on morphisms (p,η). $\mathcal{P}[(p,\eta)]$ must be a pair (p^+,η^+), where $p^+:\mathcal{P}_R(D_F) \rightarrow \mathcal{P}_R(D_G)$ and $\eta^+: F^P \twoheadrightarrow G^P \circ p^+$. It is obvious what the function p^+ should be $(p^+(A)=[\{p(a) \mid a \in A\}])$. To define the natural transformation η^+ we use the following figure (see next page) :



If $A \in \mathcal{P}_R(D_F)$ then $\eta_A^+ : F^P[A] \to G^P[p^+(A)]$. We have the two colimits $(F[i] \xrightarrow{f_i} X)_{i \in D_F}$ and $(G[i] \xrightarrow{g_i} Y)_{i \in D_G}$. We can construct the natural sink $(F[i] \xrightarrow{h_i} Y)_{i \in D_F}$, where $h_i := g_{p(i)} \circ \eta_i$. As X is the colimit for F we get a unique function $k : X \to Y$ which makes the diagrams commute: $k \circ f_i = h_i$ for all $i \in D_F$. Now we can define the function $q_A : X_A{}^* \to Y_{p^+(A)}{}^*$ as the restriction of k to $X_A{}^*$: $q_A := k\,|_{X_A{}^*}$. We can extend q_A in the obvious way into a continuous function from $\mathcal{P}_R(X_A{}^*)$ into $\mathcal{P}_R(Y_{p^+(A)}{}^*)$. The natural transformation η^+ can be defined now as $\eta^+{}_A := q_A{}^+$. It is clear that η^+ is indeed a natural transformation. It is easy to check that $\mathcal{P}$ preserves identities and composition. Now we can formally define the $\mathcal{P}$–functor.

Definition 2.4

Let $F : D_F \to \mathrm{CPO}$, $G : D_G \to \mathrm{CPO}$ be functors and let $(F[i] \xrightarrow{f_i} X)_{i \in D_F}$ resp. $(G[i] \xrightarrow{g_i} Y)_{i \in D_G}$ be the colimits for F resp. G.
The powerset functor

$$\mathcal{P} : \mathrm{Func}_{\mathrm{CPO}} \to \mathrm{Func}_{\mathrm{CPO}}$$

is defined as (we write F^P instead of $\mathcal{P}(F)$) :
$F^P : \mathcal{P}_R(D_F) \to \mathrm{CPO}$
$F^P[A] = \mathcal{P}_R(X_A{}^*)$
$F^P[A \sqsubseteq_R B] = \iota^+$ where ι is the inclusion from $X_A{}^*$ into $X_B{}^*$
$\mathcal{P}(p : D_F \to D_G, \eta : F \twoheadrightarrow G \circ p) = (p^+, \lambda A \in \mathcal{P}_R(D_F).(k\,|_{X_A{}^*})^+)$
where k is the unique morphism from X into Y with $k \circ f_i = g_{p(i)} \circ \eta_i$.

Lemma 2.5

The functor $\mathcal{P}:\mathrm{Func}_{CPO} \rightarrow \mathrm{Func}_{CPO}$ is locally continuous.

Proof

Let $F:D_F \rightarrow CPO$ and $G:D_G \rightarrow CPO$ be functors and let $(p^n:D_F \rightarrow D_G,\ \eta^n:F \twoheadrightarrow G\circ p)_n$ be a chain in the set hom(F,G) with $(\bigsqcup\{p^n\},\bigsqcup\{\eta^n\})$ the lub of this chain. We have to show that $\mathcal{P}[(\bigsqcup\{p^n\},\bigsqcup\{\eta^n\})] = \bigsqcup\{\mathcal{P}[(p^n,\eta^n)]\}$.
$\mathcal{P}[(\bigsqcup\{p^n\},\bigsqcup\{\eta^n\})]=((\bigsqcup\{p^n\})^+,\lambda A\in\mathcal{P}_R(D_F).(k^{\sqcup}|_{X_A^*})^+)=(\bigsqcup\{(p^n)^+\},\ \lambda A \in\mathcal{P}_R(D_F).(k^{\sqcup}|_{X_A^*})^+)$ as $(\_)^+$ is a continuous operation [Plotkin 1976]. $k^{\sqcup}$ is the mediating morphisms from X into Y with $k^{\sqcup}\circ f_i=g_{\sqcup\{p^n(i)\}}\circ\bigsqcup\{(\eta^n)_i\}$. Also $k^n\circ f_i=g_{p^n(i)}\circ\eta^n{}_i$. It is clear that $\bigsqcup\{g_{p^n(i)}\circ\eta^n{}_i\}=g_{\sqcup\{p^n(i)\}}\circ\bigsqcup\{(\eta^n)_i\}$. Therefore we have $k^{\sqcup}=\bigsqcup\{k^n\}$ and $(k^{\sqcup}|_{X_A^*})^+ = \bigsqcup\{(k^n|_{X_A^*})^+\}$. So $\mathcal{P}[(\bigsqcup\{p^n\},\bigsqcup\{\eta^n\})] = (\bigsqcup\{(p^n)^+\}, \bigsqcup\{\lambda A\in\mathcal{P}_R(D_F).(k^n|_{X_A^*})^+\}) = \bigsqcup\{(p^n)^+,(\lambda A\in\mathcal{P}_R(D_F).(k^n|_{X_A^*})^+)\} = \bigsqcup\{\mathcal{P}[(p^n,\eta^n)]\}$.

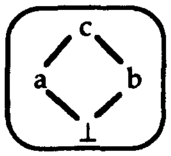
❑

3 Δ–Functors

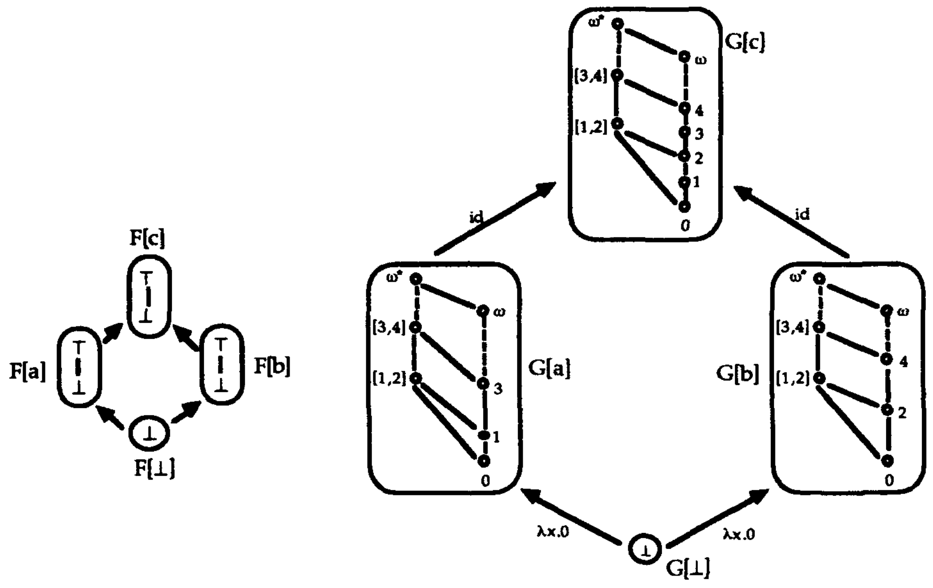
Algebraic cpos play an important role in domain theory. Also the model for polymorphic λ2–calculus mentioned in the introduction [Fiech 1993; Fiech,Schmidt 1993] benefits if we can work only with algebraic cpos. For these reasons we define the category Func_Δ which has as objects functors $F:D_F \rightarrow SCOTT$. We have to make sure that Func_Δ is closed under the operations defined in the previous section. It is easy to see that $+,\times$ and $\perp$ don't cause any problems. A different situation emerges when we consider the $\rightarrow$ functor. The new functor $F \rightarrow G$ must map any function $f\in D_F \rightarrow D_G$ into an algebraic cpo. So $F \twoheadrightarrow G\circ f$ must be algebraic. But this won't hold in general.

Example 3.1

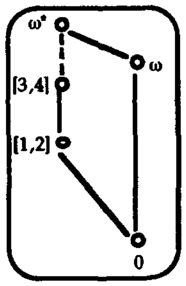
Let $F : D_F \to \mathrm{SCOTT}$ and $G : D_G \to \mathrm{SCOTT}$ be two functors where

$D_F = D_G =$ 

Define $F[a]=F[b]=F[c]=\{\perp \sqsubseteq \top\}$ and G[a], G[b], G[c] as in the following figure :



Consider the identity function on D_F. The cpo $F \twoheadrightarrow G \circ id_{D_F}$ is isomorphic to the nonalgebraic cpo



We can solve this problem if we require that all the functors in $Func_\Delta$ preserve finite elements and nonempty infimas.

Theorem 3.2 [Fiech, Huth 1991]

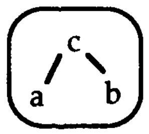
Let $F,G:\Omega \rightarrow SCOTT$ be two functors with a small category as source, st. for all morphisms $f \in \Omega$ the map $F[f]$ preserves finite elements and $G[f]$ preserves nonempty $\sqcap$'s. If $F \twoheadrightarrow G$ is nonempty then $F \twoheadrightarrow G$ is a Scott–domain.

❑

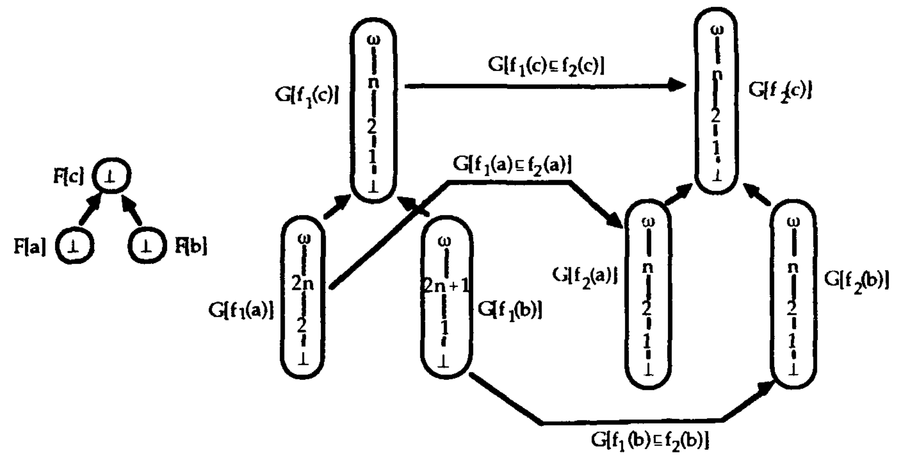
As embeddings preserve finite elements, functors with $F[d_k \sqsubseteq d_l]$ an embedding for all $d_k \sqsubseteq d_l$ would be good candidates for our category. But this isn't enough as again $F \rightarrow G[f_i \sqsubseteq f_j]$ may not be an embedding.

Example 3.3

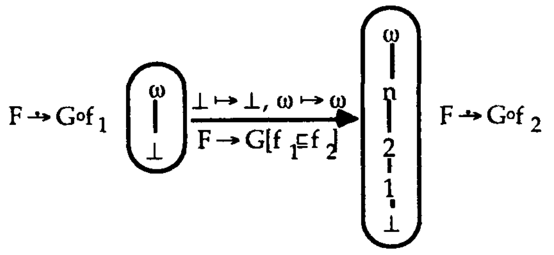
We define two functors $F:D_F \rightarrow CPO$ and $G:D_G \rightarrow CPO$.

$D_F = D_G =$ 

F maps all elements in D_F to the one element cpo $\{\perp\}$. G[a] and G[b] are the cpos of even resp. odd numbers and G[c] is the cpo of all natural numbers (with the natural ordering on integers). $G[a \sqsubseteq c]$ and $G[b \sqsubseteq c]$ are the obvious embedding mappings. The functions f_1, $f_2 \in [D_F \rightarrow D_G]$ are defined as $f_1 := id$ and $f_2 := \lambda x.c$ (see next page).



The cpo of natural transformations $F \twoheadrightarrow G\circ f_1$ has only two elements and $F \twoheadrightarrow G\circ f_2$ is isomorphic to $G[c]$.



It is clear that $F \rightarrow G[f_1 \sqsubseteq f_2]$ is not an embedding and that the category of functors with embedding morphisms is not closed under the $\twoheadrightarrow$ operation.

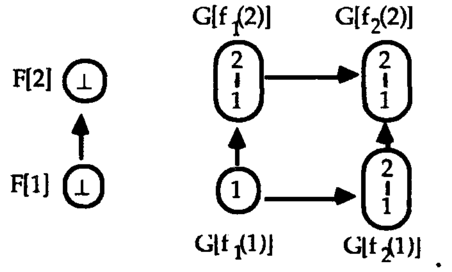
What could we change in this example? All the involved cpos are Scott–domains and the mappings are embeddings which preserve nonempty infimas. This doesn't leave much room for improvement. The only additional requirement we may impose seems to be that all the morphisms are lower embeddings. Fortunatelly this is also enough as we will see in the rest of this section. We require that the category Func_Δ has as objects functors from a domain D into the category of Scott–domains, st. all $F[d_i \sqsubseteq d_j]$ are lower embeddings.

Notice that if $e:D \rightarrow E$ is a lower embedding then e preserves finite elements, arbitrary $\sqcup$'s and nonempty $\sqcap$'s.

Next we have to make sure that $F \rightarrow G[f_1 \sqsubseteq f_2]$ is also a lower embedding. What is the projection mapping to $F \rightarrow G[f_1 \sqsubseteq f_2] = \lambda\eta \in F \twoheadrightarrow G \circ f_1 . \lambda d \in D_F . G[f_1(d) \sqsubseteq f_2(d)] \circ \eta_d$? The obvious guess might be that $F \rightarrow G[f_1 \sqsubseteq f_2]^R = \lambda\phi \in F \twoheadrightarrow G \circ f_2 . \lambda d \in D_F . G[f_1(d) \sqsubseteq f_2(d)]^R \circ \phi_d$. But this map doesn't always produce a natural transformation.

Example 3.4

In $F \twoheadrightarrow G \circ f_2$ we have the natural transformation ϕ, with $\phi_1 = \lambda\bot.2$ and $\phi_2 = \lambda\bot.2$. But η defined by $\eta_1 := G[f_1(1) \sqsubseteq f_2(1)]^R \circ \phi_1$, $\eta_2 := G[f_1(2) \sqsubseteq f_2(2)]^R \circ \phi_2$ is not a natural transformation as the diagrams don't commute.



In the above example the function $F \rightarrow G[f_1 \sqsubseteq f_2]$ is an embedding although the projection map is different from our guess. So we need a different way of defining the corresponding projection. But first we show that $F \rightarrow G[f_1 \sqsubseteq f_2](F \twoheadrightarrow G \circ f_1)$ is a lower set in $F \twoheadrightarrow G \circ f_2$.

Lemma 3.5

Let $F:D_F \rightarrow CPO$, $G:D_G \rightarrow CPO$ be functors, st. for all $d_i \sqsubseteq d_j \in D_G$, $G[d_i \sqsubseteq d_j]$ is a lower embedding. Then for the functor

$F \to G : [D_F \to D_G] \to CPO$, $F \to G[f_1 \sqsubseteq f_2](F \twoheadrightarrow G \circ f_1)$ is a lower set in $F \to G[f_2]$.

Proof

Let $\eta \in F \twoheadrightarrow G \circ f_1$ and $\phi \sqsubseteq F \to G[f_1 \sqsubseteq f_2](\eta) =: \eta^*$. It is easy to see that $\phi^* := \lambda d \in D_F . G[f_1(d) \sqsubseteq f_2(d)]^R \circ \phi_d$ is a natural transformation. The diagrams commute as all $G[d_i \sqsubseteq d_j]$'s are lower embeddings (For any $x \in F[d_i]$, $G[f_1(d_i) \sqsubseteq f_2(d_i)] \circ G[f_1(d_i) \sqsubseteq f_2(d_i)]^R (\phi_{d_i}(x)) = \phi_{d_i}(x)$ because $\phi_{d_i}(x) \sqsubseteq \eta_{d_i}(x)$). It is clear that $F \to G[f_1 \sqsubseteq f_2](\phi^*) = \phi$.

❑

Lemma 3.6 .

Let E be an algebraic cpo and $e : D \to E$ a continuous function, st. $e(D)$ is a lower set in E. If there exists a monotone function $p : E \to D$, st. $p \circ e = id_D$ and $e \circ p \sqsubseteq id_E$ then p is also continuous and therefore e is an embedding.

Proof

Let $\omega = \bigsqcup\!\!\uparrow \omega_i$ in E. E is algebraic, so there exists a directed set $Q \subseteq K(E)$ with $\bigsqcup\!\!\uparrow Q \doteq e(p(\omega)) \sqsubseteq \omega$. Because $e(D)$ is a lower set in E we must have $\bigsqcup\!\!\uparrow p(q_j) = p(e(p(\omega))) = p(\omega)$. For all $q_j \in Q$ there must exist some ω_i with $q_j \sqsubseteq \omega_i$. As p is monotone we get $p(q_j) \sqsubseteq p(\omega_i)$ and therefore every upper bound for $\{p(\omega_i)\}$ must also be an upper bound for $\{p(q_j)\}$. Hence $p(\omega) = \bigsqcup\!\!\uparrow \{p(q_j)\} \sqsubseteq \bigsqcup\!\!\uparrow \{p(\omega_i)\}$ and trivially $\bigsqcup\!\!\uparrow \{p(\omega_i)\} \sqsubseteq p(\omega)$. So $p(\omega) = \bigsqcup\!\!\uparrow \{p(\omega_i)\}$ and p is continuous.

❑

At this point we are ready to define the corresponding projection mapping to $F \to G[f_1 \sqsubseteq f_2]$.

Proposition 3.7

Let $F : D_F \to SCOTT$, $G : D_G \to SCOTT$ be two functors, st. for all $d_i \sqsubseteq d_j$ in D_G, $G[d_i \sqsubseteq d_j]$ is a lower embedding. For the functor $F \to G$

and for any $f_k \sqsubseteq f_l$ in $[D_F \rightarrow D_G]$, $F \rightarrow G[f_k \sqsubseteq f_l]$ is also a lower embedding.

Proof

First we show that for any $\eta \in F \twoheadrightarrow G \circ f_l$ the set $Q := \{\phi \in F \twoheadrightarrow G \circ f_k \mid F \rightarrow G[f_k \sqsubseteq f_l](\phi) \sqsubseteq \eta\}$ has a maximum element. For every $d \in D_F$ the set $\{\phi(d) \mid F \rightarrow G[f_k \sqsubseteq f_l](\phi) \sqsubseteq \eta\}$ is bounded by the function $G[f_k(d) \sqsubseteq f_l(d)]^R \circ \eta(d)$. Because $F[d] \rightarrow G[d]$ is bounded complete we have a lub for the set $\{\phi(d) \mid F \rightarrow G[f_k \sqsubseteq f_l](\phi) \sqsubseteq \eta\}$. All $G[d_i \sqsubseteq d_j]$ are embeddings and thus preserve arbitrary lubs. Therefore we can define the natural transformation ϕ^* as $\phi^*(d) := \bigsqcup\{\phi(d) \mid F \rightarrow G[f_k \sqsubseteq f_l](\phi) \sqsubseteq \eta\}$. The diagrams obviously commute and $F \rightarrow G[f_k \sqsubseteq f_l](\phi^*) \sqsubseteq \eta$.

Now we can define the corresponding projection mapping $F \rightarrow G[f_k \sqsubseteq f_l]^R := \bigsqcup\{\phi \in F \twoheadrightarrow G \circ f_k \mid F \rightarrow G[f_k \sqsubseteq f_l](\phi) \sqsubseteq \eta\}$. Obviously $(F \rightarrow G[f_k \sqsubseteq f_l]^R) \circ (F \rightarrow G[f_k \sqsubseteq f_l]) = id_{F \twoheadrightarrow G \circ f_k}$ and $(F \rightarrow G[f_k \sqsubseteq f_l]) \circ (F \rightarrow G[f_k \sqsubseteq f_l]^R) \sqsubseteq id_{F \twoheadrightarrow G \circ f_l}$. *It is clear that* $F \rightarrow G[f_k \sqsubseteq f_l]^R$ is monotone. From Lemma 3.6 we can conclude that $F \rightarrow G[f_k \sqsubseteq f_l]$ is an embedding. Together with Lemma 3.5 we get that $F \rightarrow G[f_k \sqsubseteq f_l]$ is a lower embedding.

❑

Now we are ready to define the category Func_Δ.

Definition 3.8

A functor $F : D_F \rightarrow CPO_\perp$ is called a **Δ–functor** if $F[d] \in SCOTT$ for all $d \in D_F$ and $F[d_k \sqsubseteq d_l]$ is a lower embedding for all $d_k \sqsubseteq d_l \in D_F$. We also require that $F[\perp] = \{\perp\}$.

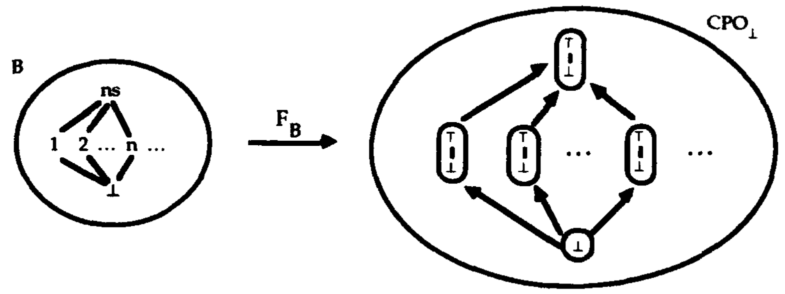
Definition 3.9

The category $\mathbf{Func}_\Delta$ has as objects Δ–functors. Morphisms in Func_Δ and operations $+, \times, \perp, \rightarrow$ on Func_Δ are defined as in the case of Func_{CPO}.

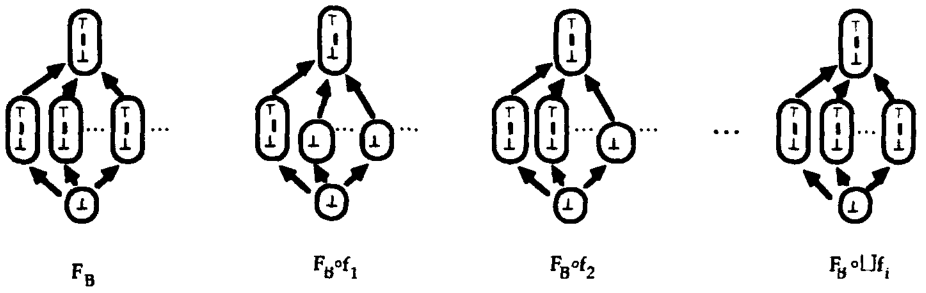
Another interesting question is if we can restrict Func_Δ only to functors which preserve directed colimits (lubs of directed sets). Under these conditions if we assume that D_F and D_G are both Scott–domains then any natural transformation $\eta \in F \twoheadrightarrow G$ is uniquely determined by the set of functions $\{\eta_a : F[a] \to G[a] \mid a \in K(D_F)\}$. Also given a set of functions $\{\phi_a : F[a] \to G[a] \mid a \in K(D_F)\}$, st. the corresponding diagrams commute, $G[a \sqsubseteq b] \circ \phi_a = \phi_b \circ F[a \sqsubseteq b]$ for all $a,b \in K(D_F)$, we can always complete this set (in a unique way) into a natural transforma
$\phi : F \twoheadrightarrow G$. Again we have to check if this new Func_Δ is clo
under $\to$. A related problem is the preservation of colimits on function spaces. Given a functor $F : D_F \to \mathrm{CPO}$ with the colimit $(F[i] \xrightarrow{f_i} X)_{i \in D_F}$ for any cpo Y we can construct the functor $F_Y : D_F \to \mathrm{CPO}$ where $F_Y[d] := Y \to F[d]$ and $F[d_1 \sqsubseteq d_2] := \lambda f. F[d_1 \sqsubseteq d_2] \circ f$. We may expect now that the cpo $[Y \to X]$ is the colimit for F_Y, but this is not the case in general. In this special case $[Y \to X]$ is the colimit if D_F is directed and all f_i's are embeddings. But in the case of Δ–functors and the $\to$ operation on Func_Δ the result is negative.

Example 3.10

We define the functor $F_B : B \to \mathrm{CPO}_\perp$ as :



The functor $F_B \to F_B : [B \to B] \to CPO_\perp$ with $F_B \to F_B[f] = F_B \twoheadrightarrow F_B \circ f$ maps functions in $[B \to B]$ into cpos of natural transformations. Next we *define the functions* $f_i : B \to B$ *as* $f_i(\perp)=\perp$, $f_i(ns)=ns$, $f_i(k)=ns$ for $k \in \{1,\dots,i\}$ and $f_i(k)=\perp$ for $k \in \{i+1,\dots\}$. Obviously $f_i \sqsubseteq f_{i+1}$ and $\overset{\uparrow}{\bigsqcup}\{f_i\} = \underline{\lambda} x.ns$. In all the cpos $F_B \twoheadrightarrow F_B \circ f_i$ we have only one natural transformation $\eta^\perp$ with $(\eta^\perp)_n = \lambda x.\perp$. But in $F_B \twoheadrightarrow F_B \circ (\overset{\uparrow}{\bigsqcup}\{f_i\})$ there are two natural transformations $\eta^\perp$ and $\eta^\top$ with $(\eta^\top)_n = id$. So $F_B \to F_B[\overset{\uparrow}{\bigsqcup}\{f_i\}]$ is not the colimit of the cpos $F_B \to F_B[f_i]$.



4 Acknowledgment

I would like to thank David Schmidt for many hours of productive discussions and helpful hints.

5 References

Adamek J., Herrlich H., Strecker G. [1990]
Abstract and Concrete Categories, John Wiley&Sons, 1990

Fiech A., Huth M. [1991]
Algebraic Domains of Natural Transformations, Technical Report, KSU, Comp. Sci., 1991

Fiech A. [1992]
Colimits in the Category CPO, Tech. Report, KSU, Comp. Sci.92
Fiech A. [1993]

A Denotational Model for Polymorphic Lambda Calculus with Subtyping, dissertation, Kansas State University, Dept. of Comp. and Inf. Sci., 1993

Fiech A., Schmidt D. [1993]
Polymorphic Lambda Calculus and Subtyping, in preparation

O'Hearn P., Tennent R. [1993]
Relational Parametricity and Local Variables, Proc 20th ACM Symp Princ Prog Lags, Charleston, SC, Jan 1993

Oles F. [1982]
A Category–Theoretic Approach to the Semantics of Programming Languages, Ph.D. thesis, Syracuse University, Syracuse, NY, 1982

Oles F. [1985]
Type Algebras, Functor Categories and Stock Structure. In M. Nivat and J. Reynolds, editors, Algebraic Methods in Semantics, Cambridge University Press, Cambridge 1985

Plotkin G. [1976]
A powerdomain construction, SIAM J. of Computing, vol.5, pp. 452-487

Reynolds J. [1980]
Using Category Theory to Design Implicit Conversions and Generic Operators, in LNCS 94, p. 211-258, Springer Verlag 1980

Schmidt D. [1986]
Denotational Semantics, Allyn and Bacon, Inc. 1986

Schmidt D. [1990]
Action Semantics Based Language Design, SOFSEM '90 Zotavovna Sirena, Janske Lazne, Krkonose, Nov. 1990.

A Categorical Interpretation of Landin's Correspondence Principle

Anindya Banerjee and David A. Schmidt
Department of Computing and Information Sciences
Kansas State University [*]
(banerjee, schmidt)@cis.ksu.edu

Abstract

Many programming languages can be studied by desugaring them into an intermediate language, namely, the simply-typed λ- calculus. In this manner Landin and Tennent discovered a "correspondence" between the semantics of definition bindings and parameter bindings such that the semantics of free identifiers becomes independent of their mode of definition.

In this paper we consider programming languages with modules and we desugar modules into records. A categorical model for the simply-typed λ- calculus with records is then freely generated. The record construction becomes a tensor product, the lambda abstraction construction becomes a function space, and if the language satisfies the correspondence principle, then the categorical exponentiation diagram commutes. A converse result is also proved. The framework for defining the model is of interest because it defines a hierarchy of call-by-value λ-calculi, of which call-by-name is the weakest form of call-by-value calculus.

Applications to compiling are given.

1 Introduction

In his seminal paper on the next 700 programming languages [9], Landin suggested that a programming language might satisfy a correspondence in the semantics of its definition and parameter constructions. That is, the semantics of binding a body, `U`, to a name, `i`, as seen in:

```
define i = U in V
```

should be the same as that of binding an actual parameter, `U`, to a formal parameter, `i`, as seen in:

```
define j(i) = V in call j(U)
```

where `j` is fresh.

[*]Manhattan, Kansas 66506, USA. Part of this work was supported by NSF under grant CCR-9102625.

Tennent [23] titled this the *correspondence principle* and suggested that it be used as a design guide for programming languages. The primary benefit from the correspondence principle is that a program phrase containing free identifiers can be understood without concern as to whether the identifiers were bound by definitions or parameters. For example, ... `A` ... means the same whether it appears in:

```
define A = 4 in ...A...
```

or in:

```
define G(A) = ...A... in G(4)
```

1.1 Correspondence in higher order, modular languages

The importance of the correspondence principle increases when a programming language is *higher-order*, that is, abstractions can be arguments and results of other abstractions. Consider the following example:

```
function g(a) = (function f(b) = ...a...b... in return f)
```

A call to `g(somevalue)` returns `f` with a binding to `a`. The semantics of `f` is explained as: `define a = somevalue in function f(b) = ...a...b....` For this explanation to make sense, correspondence *must* hold.

Finally, languages with modules need correspondence to ensure proper behavior: a module, in the sense of Ada and Standard ML, is a set of definitions. Modules can be built hierarchically, one module importing another (*cf.* SML's "functors" [11, 12]):

```
module m = (define i = something)
in  module n(x) = (use x; define j = ...i...)
in  ...use n(m)...
```

or they can be written "flat":

```
module n = (define i = something; define j = ...i...)
in  ...use n...
```

Correspondence ensures that the semantics of a hierarchical module equals the semantics of a flat one with the same set of definitions. For example, if the "`something`" in the above example was a looping expression, and module parameters like `m` were evaluated eagerly but module importations like `use n` were done lazily, then hierarchical module construction would be a futile endeavor.

1.2 This paper

We show that Landin's correspondence principle, as it arises in the above examples, can be formalized in Category Theory: a language's definition construct defines a tensor product construction, its parameter construct defines a function space construction, and correspondence ensures that the exponentiation diagram commutes. That is, the category freely generated by a programming language with correspondence has an associative, commutative tensor product (with a unit) and it has weak categorical exponentiation (the fill-in morphism might not be unique). Hence, it is a symmetric monoidal weakly closed category. The significance is that the tensor product ensures that sets of bindings (*i.e.*, modules) behave like sets, and exponentiation ensures that definition bindings behave like parameter bindings. Both properties are crucial to properly designed, modular, higher-order programming languages. We also show a converse result: if the semantics of a programming language fits a "usual" format, weak exponentiation in the model implies correspondence.

In this way, a fundamental intuitive programming language criterion is characterized as a fundamental categorical one. This fits within Reynolds' program of "Semantics ... [as] applied mathematics; it seeks profound definitions rather than difficult theorems ... the application of such concepts directly reveals regularity in linguistic behaviour and strengthens and objectifies our intuitions of simplicity and uniformity." ([25], page 3).

A programming language with correspondence can use call-by-name or call-by-value binding, so the framework for our proof must accommodate both. For this reason, we prove the result for a hierarchy of call-by-value λ-calculi, of which call-by-name is the weakest call-by-value calculus. In this light, our result can be viewed as a generalization of the cartesian closedness of models for call-by-name, simply-typed λ-calculus [7] to monoidal closedness for call-by-value λ-calculi. Our result also reveals that the reason why categorical exponentiation holds is because the form of binding defined by a product construction corresponds with the form of binding defined by the function space construction.

In the rest of this paper, we define our metalanguage, outline the proof of the correspondence theorem, and state an application to compiling.

2 The Metalanguage

One way to obtain correspondence is to force it upon a language. Landin [8], Reynolds [18, 19], and Tennent [23, 24] observed that correspondence *must* hold if both definition binding and parameter binding are desugared purely into λ-abstractions:

`define i = U in V`	desugars to	`(λi.V)U`
`define j(i) = V in j(U)`	desugars to	`define j = λi.V in j(U)` which desugars to `(λi.V)U`, when `λi.V` is copied for `j`

Thus, the semantics of λ-abstraction — whether it be call-by-name or call-by-value semantics — defines the semantics of both definition and parameter binding. An example like:

```
(1)       const k = 0, alias x = location1
          in procedure p(y:int) = x:= @x+y
             in x:= k; p(k)
```

is desugared into:

((λ`k`:*int*.λ`x`:*intloc*.(λ`p`:*int*→*comm*.`x := k; (p k)`)(λ`y`:*int*.`x := @x+y`)0)`location1`

We work with statically typed languages. For simplicity, we use `alias` definitions here rather than `var` declarations. A `var` declaration is a binding of an identifier to a location with the side effect of allocating the location in storage. One might desugar `var x:intloc in e` by `new`(λ`x` : *intloc*. `e`), where `new` is a storage allocation operator [4, 13, 14, 19]. Also, the `@` symbol denotes dereferencing. Notice that compound declarations, like `const k = 0, alias x = location1`, are desugared into curried bindings.

Since the desugaring pattern is regular and simple, Tennent [23] derived an ***abstraction principle***, stating that a definition construction (*e.g.*, constant, alias, function, procedure, module, ...) can be introduced for each of a language's syntax domains (*e.g.*, numerals, locations, expressions, commands, declarations, ...). Each definition construct is desugared into a λ-abstraction. Similarly, Schmidt [22] proposed a ***parameterization principle***, stating that parameter constructions (*e.g.*, numeral parameters, location parameters, expression parameters, command parameters, declaration parameters, ...) can be introduced for each of a language's syntax domains. Again, each formal parameter construct is desugared into a λ-abstraction. Properly applied, the two principles extend systematically a core programming language into a language for programming in-the-large [1, 21, 24, 26].

2.1 The need for records

Desugaring both definitions and parameters into purely λ-abstractions confuses definitions with parameters, which is problematic when a Pascal-like language is studied. Indeed, for the correspondence principle to be of value as a language design criterion, it must be possible for it to *fail*. More importantly, the desugaring of Ada/Standard ML-style modules, which are sets of bindings, as in:

```
(2)     begin module m = (const k = 0, alias x = location1)
        in begin use m, function f(a:integer) = a + 2
           in x := k + f(@x) end end
```

can not be modelled easily by simply-typed λ-abstractions if at all. The problem is that modules are "packages of bindings," which are not λ-abstraction-like. For these reasons, we will maintain the integrity of definitions and parameters by *desugaring them into records and λ-abstractions, respectively.* The metalanguage we use is defined in Figure 1. The metalanguage is reminiscent of the one in Lambek and Scott [7] in its extension of the simply-typed λ-calculus by a product construction, but here we use records rather than tuples. Unlike Lambek and Scott's construction, however, the records are motivated by the pragmatic reasons stated above, not by an explicit desire to discover cartesian closedness.

τ	$\in$	Type-expression
ι	$\in$	Primitive-type (for core language constructs)
π	$\in$	Type-assignment
e	$\in$	Expression
i	$\in$	Identifier
op	$\in$	Core-language-operator

$\tau ::= \iota \mid \tau_1 \rightarrow \tau_2 \mid \pi$

$\pi ::= \{i : \tau_i\}_{i \in I}$, where I is a finite set of distinct identifiers

$e ::= i \mid \lambda i : \tau . e \mid (e_1\ e_2) \mid op(e_1, \ldots, e_n) \mid \{i = e\} \mid \textit{with}\ e_1\ \textit{do}\ e_2 \mid e_1, e_2$

$$\pi \vdash i : \tau \text{ where } (i : \tau) \in \pi \qquad \frac{\pi + (i : \tau_1) \vdash e : \tau_2}{\pi \vdash \lambda i : \tau_1 . e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\pi \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \pi \vdash e_2 : \tau_1}{\pi \vdash (e_1\ e_2) : \tau_2}$$

$$\frac{\pi \vdash e_1 : \iota_1 \ldots \pi \vdash e_n : \iota_n}{\pi \vdash op(e_1, \ldots, e_n) : \iota} \qquad \frac{\pi \vdash e : \tau}{\pi \vdash \{i = e\} : \{i : \tau\}}$$

$$\frac{\pi \vdash e_1 : \{i : \tau_i\}_{i \in I} \qquad \pi + (i_1 : \tau_1) + \ldots + (i_n : \tau_n) \vdash e_2 : \tau}{\pi \vdash \textit{with}\ e_1\ \textit{do}\ e_2 : \tau} \text{ where } I = \{i_1, \ldots, i_n\}$$

$$\frac{\pi \vdash e_1 : \{i : \tau_i\}_{i \in I} \qquad \pi \vdash e_2 : \{j : \tau_j\}_{j \in J}}{\pi \vdash e_1, e_2 : \{k : \tau_k\}_{k \in I \cup J}} \text{ where } I \cap J = \emptyset$$

Note: $\pi + (i : \tau) = (\pi - \{(i : \tau') | (i : \tau') \in \pi\}) \cup \{i : \tau\}$

$\{i_1 = e_1, i_2 = e_2, \ldots, i_n = e_n\}$ abbreviates

$\{i_1 = e_1\}, \{i_2 = e_2\}, \ldots, \{i_n = e_n\}$

Figure 1: Meta-language

By convention, we refer to that part of the programming language consisting of primitive arithmetic, logical, and storage operators as the language's *core*; in Figure 1, $op(e_1, \ldots, e_n)$ represents those operators. Definitions and parameters are extensions to the core. Definitions are modelled by records, which are sets of identifier-expression bindings [2]: $\{i = e\}$ is a one field record, e_1, e_2 is record append, and *with* e_1 *do* e_2 makes visible to e_2 the bindings in e_1. The example programs (1) and (2) stated above are desugared respectively into:

```
with {k = 0, x = location1} do
     with {p = λy : int . x := @x+y} do
          x := k; (p k)
```

and

```
with {m = {k = 0, x = location1}} do
        with m, {f = λ a : int . a + 2} do
                x := k + (f (@ x))
```

This makes clear that binding by definitions, as modelled by records, is inherently different from binding by parameters, as handled by λ-abstraction. In particular, modules are revealed to be records, and importing a module is achieved by a *with* expression.

Records are product-like, and λ-abstractions are exponentiation-like. This leads us to explore their categorical relationship.

3 The Correspondence Theorem

3.1 Computation Rules

Say that a programming language is desugared into the language of Figure 1. The computation rules for the core language constructs are preserved, but the computation rules for definition and parameter binding must desugar into the computation rules for application and *with*, respectively. In a call-by-name calculus, the computation rules for application and *with* have strikingly similar forms:

Definition 1 (Call-by-name reduction)
β-name: $((\lambda\, i : \tau\, .\, e_1)\, e_2) \;\rhd\; [e_2/i]e_1$
ρ-name: *with* $\{i_1 = e_1, ..., i_n = e_n\}$ *do* $e \;\rhd\; [e_1/i_1, ..., e_n/i_n]e$
where $[e_1/i_1, ..., e_n/i_n]e$ *is parallel substitution.*[1]

Both treat binding as substitution. Indeed, the semantics of the two forms of binding *correspond* in Landin's sense. In a call-by-value language, the computation rules correspond again:[2]

Definition 2 (Call-by-value reduction)
β-value: $((\lambda\, i : \tau\, .\, e_1)\, e_2) \;\rhd\; [e_2/i]e_1$*, where e_2 is a value*
ρ-value: *with* $\{i_1 = e_1, ..., i_n = e_n\}$ *do* $e \;\rhd\; [e_1/i_1, ..., e_n/i_n]e$,
where $e_1, ..., e_n$ are values and the notion of value is predefined, e.g., [17].

But say that the language uses a β-name rule and a ρ-value rule—correspondence fails.

[1]The definition of parallel substitution is the usual one, but note that:

$$[e_i/i]_{i\in I}(\textit{with } e_1 \textit{ do } e_2) \;=\; \textit{with } [e_i/i]_{i\in I}e_1 \textit{ do } ([e_i/i]_{i\in I-J})e_2$$

where $\pi \vdash e_1 : \{j : \tau_j\}_{j\in J}$, and $[e_i/i]_{i\in I-J}$ is $[e_i/i]_{i\in I}$ less those substitutions e_j/j, where $j \in J$. This demands that we work only with well-typed programs and computation rules that preserve typing.

[2]Rather than view call-by-value reduction as the application of the call-by-name $\beta\rho$-rules with a fixed reduction strategy, we follow [17] and restrict the $\beta\rho$-rules. This leads to a pleasant equational theory.

Correspondence is aesthetically pleasing, but there is also theoretical justification for it: when it holds, regardless of the binding strategy employed, the category freely generated from the programming language and its computation rules is a symmetric monoidal weakly closed category, that is, the category has an associative, commutative tensor product (with unit) and it has weak exponentiation. These categorical properties formally ensure that records behave correctly (order of construction of subrecords and order of fields are unimportant) and λ-abstractions behave correctly (parameter binding is the same as definition binding).

3.2 The Value set

Let us now summarize the proof of the "correspondence theorem" stated in the previous paragraph. We begin by assuming that the correspondence principle is characterized by the *$\beta\rho$-value* computation rules of Definition 2, where the notion of "value" can be varied, depending on the desired binding strategy. For example, in the call-by-name calculus, *all* expressions are values. In a call-by-value calculus, only some proper subset of the expressions are values.

The set of expressions termed as "values" must satisfy the following conditions:

Definition 3 (Value) *Let Value $\subseteq$ Expression; Value is* well-defined *if the following conditions hold:*

(i) *for all $e \in$ Expression, $\lambda i : \tau . e \in$ Value;*

(ii) *for all $e_i \in$ Expression, $1 \leq i \leq n$*
if all $e_i \in$ *Value*, then $\{i_1 = e_1, ..., i_n = e_n\} \in$ *Value;*

(iii) *if $e \in$ Value, then for all visible subexpressions, e' within e, $e' \in$ Value. (A subexpression, e', within e, is* visible *if it is not contained inside e_0 of some $(\lambda i : \tau . e_0)$ and it is not contained inside e_2 of some (with e_1 do e_2).)*

The intuition behind Definition 3 goes as follows. All call-by-value calculi treat λ-abstractions as closures, hence Clause (i). If an expression, e, is a value, then packaging it inside a record, $\{i = e\}$, should preserve its value-ness, hence Clause (ii). Clause (iii) is the converse, generalized, of Clause (ii).

We say that an expression, e, *is a value* if $e \in$ *Value*; e *has a value* if $e \equiv e'$ and e' is a value; else e *has no value.* ($e \equiv e'$ means that e is convertible to e' by the computation rules, $\rhd$.) Also, we assume that the computation rules for the core language operators have the form: $op(e_1, ..., e_n) \rhd e$, *where $e_1, ..., e_n$ are values.* Given the above terminology, we add this last clause to Definition 3:

(iv) if $\pi \vdash e : \{i : \tau_i\}_{i \in I}$ and e has a value, then, for all $i \in I$, $e \downarrow i$ has a value. ($e \downarrow i$ abbreviates (*with* e *do* i)).

This clause, a minor addition, is needed to gain the main result.

For (closed) expressions e_1 and e_2, such that $\vdash e_1 : \tau$ and $\vdash e_2 : \tau$, $e_1 \approx e_2$ iff $e_1 \approx_\tau e_2$, where: $e_1 \approx_\tau e_2$ if e_1 and e_2 have no value, or else:

$e_1 \approx_\iota e_2$ if e_1 and e_2 have values, and $e_1 \equiv e_2$

$e_1 \approx_{\tau_1 \rightarrow \tau_2} e_2$ if e_1 and e_2 have values, and for all a and b such that $a \approx_{\tau_1} b$, $(e_1\ a) \approx_{\tau_2} (e_2\ b)$

$e_1 \approx_{\{i:\tau_i\}_{i \in I}} e_2$ if e_1 and e_2 have values, and for all $i \in I$, $e_1 \downarrow i \approx_{\tau_i} e_2 \downarrow i$,

Figure 2: Equivalence relation

When all the well-defined *Value* sets for *Expression* are ordered by subset inclusion, they form a complete lattice, where the bottom element is formed by the inductive closure over clauses **(i)** and **(ii)** (this is the usual call-by-value calculus, except that no core language expressions are values), and the top element is *Expression* itself (this is the usual call-by-name calculus). From here on, we work with only those λ-calculi whose *Value* sets are well-defined.

3.3 The equivalence relation and category

Our intention is to freely generate a category, **C**, from the language in Figure 1. (We assume acquaintance with elementary category theory [16].) The technique resembles that of Lambek and Scott [7]: objects of **C** are the elements of Type-expression; morphisms in $hom(\tau_1, \tau_2)$ are equivalence classes of (closed) λ-abstractions, $\lambda\, i : \tau_1 \,.\, e$, where $\vdash \lambda\, i : \tau_1 \,.\, e : \tau_1 \rightarrow \tau_2$ holds, with respect to the equivalence relation, $\approx$, defined in Figure 2.

For open terms, $\pi \vdash e_1 : \tau$ and $\pi \vdash e_2 : \tau$, $e_1 \approx_\tau^\pi e_2$ iff for all $(i : \tau_i) \in \pi$, for all $\vdash a_i : \tau_i$ and $\vdash b_i : \tau_i$ such that $a_i \approx_{\tau_i} b_i$, $[a_i/i]e_1 \approx_\tau [b_i/i]e_2$, that is, substitution of equivalent closed terms for free variables yields equivalent closed terms.

Proposition 1

For $\pi \vdash e_1 : \tau$ and $\pi \vdash e_2 : \tau$,

(i) *$\approx$ is an equivalence relation;*

(ii) *$e_1 \equiv e_2$ implies $e_1 \approx_\tau^\pi e_2$;*

(iii) *$a \approx_{\tau'}^{\pi'} b$ and $e_1 \approx_\tau^\pi e_2$ imply $[a/x]e_1 \approx_\tau^{\pi'} [b/x]e_2$, where $\pi' = \pi - \{x : \tau'\}$;*

(iv) *$\lambda\, x : \tau' \,.\, e_1 \approx_\tau^{\pi'} \lambda\, y : \tau' \,.\, [y/x]e_1$, where $\pi' = \pi - \{x : \tau'\}$;*

(v) *if e has a value, then so do all its visible subexpressions;*

The proofs are routine, although clauses **(i)** and **(ii)** must be proved simultaneously: $e \approx_\tau^\pi e$ with ($e_1 \equiv e_2$ *implies* $e_1 \approx_\tau^\pi e_2$). Proposition 1 implies:

$$((\lambda\, i : \tau \,.\, e)\; e') \approx_\tau^\pi \textit{with}\ \{i = e'\}\ \textit{do}\ e$$

which is the traditional statement of correspondence.

For convenience, we write a morphism, $[\lambda\, i : \tau_1 \,.\, e]_\approx \in \textit{hom}(\tau_1, \tau_2)$, as just $\lambda\, i : \tau_1 \,.\, e$. For an object, τ, the identity morphism, $\textit{id}_\tau$, is $\lambda\, i : \tau \,.\, i$. For morphisms $f = (\lambda\, i_1 : \tau_1 \,.\, e_1) \in \textit{hom}(\tau_1, \tau_2)$ and $g = (\lambda\, i_2 : \tau_2 \,.\, e_2) \in \textit{hom}(\tau_2, \tau_3)$, their composition, $g \circ f \in \tau_1 \rightarrow \tau_3$, is $\lambda\, i_1 : \tau_1 \,.\, ((\lambda\, i_2 : \tau_2 \,.\, e_2)\; e_1)$. With Proposition 1 in hand, we can prove that these definitions give a category.

3.4 Correspondence implies exponentiation

The next, natural, step is to try to show that the category has a categorical product, $\tau_1 \times \tau_2 = \{\textit{fst} : \tau_1,\ \textit{snd} : \tau_2\}$; where $\pi_j = \lambda\, i : \{\textit{fst} : \tau_1,\ \textit{snd} : \tau_2\} \,.\, i \downarrow j$, for $j \in \{\textit{fst}, \textit{snd}\}$; and $\langle f, g\rangle = \lambda\, i : \tau \,.\, \{\textit{fst} = (f\ i),\ \textit{snd} = (g\ i)\}$. But the projection laws fail in the case when there is a phrase in the language that has no value. For example, let $f = \lambda\, x : \textit{int} \,.\, 0$, $g = \lambda\, y : \textit{int} \,.\, \Omega$, and say that 0 is a value but Ω has no value. Then, $\pi_{\textit{fst}} \circ \langle f, g\rangle \not\approx f$.[3] Instead, we define a tensor product:

$$\begin{aligned} \tau_1 \otimes \tau_2 &= \{\textit{fst} : \tau_1,\ \textit{snd} : \tau_2\} \\ f \otimes g &= \lambda\, i : \tau_1 \otimes \tau_3 \,.\, \{\textit{fst} = f(i \downarrow \textit{fst}), \textit{snd} = g(i \downarrow \textit{snd})\} \end{aligned}$$

for $f \in \tau_1 \rightarrow \tau_2$, $g \in \tau_3 \rightarrow \tau_4$

Proposition 2 $\otimes$ *is a bifunctor on* **C**.

Next, we define the families of functions:

$$a_{\tau_1\tau_2\tau_3} : (\tau_1 \otimes (\tau_2 \otimes \tau_3)) \rightarrow ((\tau_1 \otimes \tau_2) \otimes \tau_3)$$

$$c_{\tau_1\tau_2} : (\tau_1 \otimes \tau_2) \rightarrow (\tau_2 \otimes \tau_1)$$

for all τ_1, τ_2, τ_3 and show that they are natural isomorphisms. If desired, a new type expression, $\{\}$, can be added to the language, and $\pi \vdash () : \{\}$ can be stated as a new axiom. The natural isomorphism:

$$i_\tau : (\{\} \otimes \tau) \rightarrow \tau$$

completes the collection: $\otimes$ is associative, commutative, has a unit, and satisfies the MacLane-Kelly coherence conditions [10], hence **C** is a symmetric monoidal category.

[3] Recall that all phrases - even Ω - in a call-by-name calculus are values, so $\{\textit{fst} : \tau_1,\ \textit{snd} : \tau_2\}$ is categorical product in this case.

The final step is the validation that **C** is closed, that is, it has categorical exponentiation. With the obvious definitions:

$$\begin{array}{lcl} \tau_1 \Rightarrow \tau_2 & = & \tau_1 \rightarrow \tau_2 \\ \textit{apply} & = & \lambda\, i : \{\textit{fst} : \tau_1 \Rightarrow \tau_2,\ \textit{snd} : \tau_1\} \,.\, ((i \downarrow \textit{fst})\ (i \downarrow \textit{snd})) \\ \textit{closure}(f) & = & \lambda\, i_1 : \tau_1 \,.\, \lambda\, i_2 : \tau_2 \,.\, f\{\textit{fst} = i_1,\ \textit{snd} = i_2\}, \\ & & \forall\, f : \{\textit{fst} : \tau_1,\ \textit{snd} : \tau_2\} \rightarrow \tau_3 \end{array}$$

we can show the commutativity of the exponentiation diagram:

$$f \approx \textit{apply} \circ (\textit{closure}(f) \otimes \textit{id}) \tag{1}$$

for all $f \in \textit{hom}(\tau_1, \tau_2)$. This gives us weak exponentiation.

Theorem 1 *The category freely generated by Figures 1 and 2 is a symmetric monoidal weakly closed category.*

We use the term "freely generated" in the theorem, because the extensionality conditions in Figure 2 are for all practical purposes necessary to make morphism composition associative and $\otimes$ a functor. (Indeed, we could formalize the previous remark by a suitable proof of initiality, but that is secondary to our goals here).

We would like to show that the exponentiation is "strong," that is, the choice of *closure*(f) is unique. But this can fail when there are phrases of function type that have no value. For example, say that there is a family of phrases, Ω_τ, for all types τ, and no Ω_τ has a value. Then, *closure*$(\lambda\, r : \{\textit{fst} : \tau_{11},\ \textit{snd} : \tau_{12}\} \,.\, \Omega_{\tau_2})$ could be either $(\lambda\, a : \tau_{11} \,.\, \lambda\, b : \tau_{12} \,.\, \Omega_{\tau_2})$ or $(\lambda\, a : \tau_{11} \,.\, \Omega_{\tau_{12} \rightarrow \tau_2})$, but $(\lambda\, b : \tau_{12} \,.\, \Omega_{\tau_2}) \not\approx \Omega_{\tau_{12} \rightarrow \tau_2}$, since the former has a value and the latter does not. If all phrases of function type have values, however, strong exponentiation holds. This is the case in the call-by-name calculus, and also in those call-by-value calculi that possess a lifting type, $(\tau)_\perp$, where $\perp$ represents those phrases that have no value. Then, $\vdash \Omega_{\tau_1 \rightarrow \tau_2} : (\tau_1 \rightarrow \tau_2)_\perp$

3.5 Exponentiation implies correspondence

Under restrictive conditions, the existence of weak exponentiation in a language's semantics definition implies that correspondence holds. Say that the category underlying the semantics definition has the weak exponentiation property as in Figure 3. That is, $e_1 \approx \textit{apply} \circ (\textit{closure}(e_1) \otimes \textit{id})$. The usual reading of the property is that e_1 can be given one of its inputs (the one of type π) and then be packaged into a closure. When the closure is unpackaged and applied to its other input, the result is exactly the same as e_1 applied to both its inputs. Now suppose that a programming language is written in desugared form and its semantics matches the pattern in Figure 4. Then the reading that is appropriate to Figure 3 goes as follows: e_1 is a code fragment that requires definitions of π- and τ_0-typed values. (π denotes the type of the nonlocal definitions; τ_0 is the type of the local definition.) Just the π-definitions can be supplied, giving a code fragment, *closure*(e_1), which

$$\begin{tikzcd}
\pi \otimes \pi \arrow[r, "id \otimes e_0"] & \pi \otimes \tau_0 \arrow[r, "closure(e_1) \otimes id"] \arrow[dr, "e_1"] & (\tau_0 \Rightarrow \tau_1) \otimes \tau_0 \arrow[d, "apply"] \\
& & \tau_1
\end{tikzcd}$$

Figure 3: Weak Exponentiation

Assume that ρ is an environment compatible with π [25].

$$[\![\pi \vdash \lambda\, i : \tau_1 \,.\, e : \tau_1 \rightarrow \tau_2]\!]\rho = closure[\![\pi + (i : \tau_1) \vdash e : \tau_2]\!]\rho$$
$$[\![\pi \vdash (e_1\ e_0) : \tau_2]\!]\rho = apply\{fst = [\![\pi \vdash e_1 : \tau_1 \rightarrow \tau_2]\!]\rho\ ,\ snd = [\![\pi \vdash e_0 : \tau_1]\!]\rho\}$$
$$[\![\pi \vdash \{i = e\} : \{i : \tau\}]\!]\rho = \{i = [\![\pi \vdash e : \tau]\!]\rho\}$$
$$[\![\pi \vdash \textit{with}\ e_0\ \textit{do}\ e_1 : \tau_1]\!]\rho = [\![\pi + (i : \tau_0) \vdash e_1 : \tau_1]\!](\rho \oplus ([\![\pi \vdash e_0 : \{i : \tau_0\}]\!]\rho)),$$
where $(\rho \oplus \{i = v\}) = \{fst = \rho\ ,\ snd = v\}$
$$[\![\pi \vdash i : \tau]\!]\rho = \rho(\text{deBruijn-index-of}(i))$$

Figure 4: Environment Semantics

requires a τ_0-typed parameter. When the parameter is supplied, the result is the same as e_1 with all its definitions for π and τ_0. Figure 3 tells us: $[\![\pi \vdash \textit{with}\ \{i = e_0\}\ \textit{do}\ e_1 : \tau_2]\!]\rho = [\![\pi \vdash ((\lambda\, a : \tau_1 \,.\, e_1)\ e_0) : \tau_2]\!]\rho$. Since the semantics in Figure 4 preserves meaning under substitution: $[\![\pi \vdash [e_1/i]e_2 : \tau_2]\!]\rho = [\![\pi + (i : \tau_1) \vdash e_2 : \tau_2]\!](\rho \oplus (i = [\![\pi \vdash e_1 : \tau_1]\!]\rho))$, for $e_1 \in Value$, then we immediately derive the soundness of the β-val and the ρ-val reduction rules – correspondence holds. The format in Figure 4 matches that used for lazy imperative languages and functional languages. Of course, a programming language can have correspondence even if its semantics does not match the format in the figure.

3.6 Extensions

Say there is a phrase, Ω_ι, of primitive type that has no value. This implies that phrases without value exist for all types τ: Ω_τ is $((\lambda\, a : \iota \,.\, e_0)\ \Omega_\iota)$, for a closed phrase, e_0, of type τ. This phenomenon prevents a proof of categorical product and strong exponentiation. On the positive side, it means that addition of higher order constants, like *fix*, do not impact the results already proved. We can add a family of binary *fix* operators, with the computation rule:

$$\textit{fix}\ (i : \tau)\ e \quad \rhd \quad [(\textit{fix}\ (i : \tau)\ e)/i]e$$

Although our proof does not require it, in practice, the definition of *Value* should be "monotonic" with respect to substitution and computation in the sense that: (i) if e_1, $e_2 \in Value$, then $[e_1/i]e_2 \in Value$; and (ii) if $e \in Value$ and $e \rhd e'$, then $e' \in Value$ as well. A consequence is that, if the computation rules are orthogonal [6], the rule set possesses the closure property and is confluent [5, 6, 15].

Finally, the computation rules we use can be restricted so that they perform ***weak reduction***, that is, an expression is a redex it if is visible and it matches the left-hand side of a computation rule. The proofs of the previous results carry through unaltered, since the reasoning in the proofs is extensional in nature. (But note that the previous remarks regarding confluence do not hold [3].)

4 Applications to Compiling

The obvious impact of correspondence on an implementation of a programming language is that the same implementation of binding can be used for both definitions and parameters. But the framework used to produce the results in Section 3 is of significance in itself because Definition 3 and the $\beta\rho$-value rules provide a natural style of compilation of a program. The idea is simple but important, because virtually all compilers exploit it: the binding of an identifier, i, to an expression, $e \in Value$, can be performed at compile-time. Indeed, this activity might be considered the essence of compiling [20, 27]. Here is an initial, significant example: regardless of a language's binding strategy, a collection of declarations of parameterized subroutines can be processed at compile-time because they form a record of λ-abstractions, which *must* be a value, by Definition 3. For example, regardless of binding strategy, the code segment:

```
begin module m = { procedure p(a:int) = x := a }
in begin use m, function f(b:int) = b + 2
   in call p(f(@x)) end end
```

which desugars to:
$with\ \{m = \{p = \lambda a : int \,.\, x := a\}\}\ do$
$\quad with\ m,\ \{f = \lambda b : int \,.\, b + 2\}\ do\ (p\ (f\ (@\ x)))$
can be evaluated by a compiler to: $((\lambda a : int \,.\, x := a)\ ((\lambda b : int \,.\, b + 2)\ (@\ x)))$. This matches the usual compile-time processing. (Of course, a compiler copies addresses to the code for `p` and `f`, rather than the code itself. If a compiler is given additional information, *e.g.*, that numerals are values, then definitions like `const a = 2` can also be evaluated at compile-time.

4.1 Commands as values

If commands are also values, then unparameterized procedures can be evaluated at compile-time, as in the following example:

```
begin const a = 2
in begin procedure p = x := f + a
   in call p; call p end end
```

This desugars to: *with* $\{a = 2\}$ *do* *with* $\{p = x := f + a\}$ *do* $p; p$ and evaluates at compile-time to: $x := f + 2$; $x := f + 2$. A corresponding example with a command parameter would read:

```
begin const a = 2  in
   begin procedure p = (x := f + a) , procedure q(r:comm) = (r; r)
      in call q(call p)   end end
```

This desugars to:

$$\textit{with } \{a = 2\} \textit{ do with } \{p = (x := f + a), q = (\lambda r : comm.r; r)\} \textit{ do } (q\ p)$$

and evaluates at compile-time to $x := f + 2$; $x := f + 2$ as well. Commands, like `x:= f+a`, are values in imperative-style call-by-name languages like Algol-60 as well as in imperative-style call-by-value languages like Pascal. The latter point is often overlooked by Pascal programmers, but a Pascal compiler is well aware of it.

Functional-style call-by-value languages like Scheme and SML usually disallow commands as values. An SML-like program fragment such as:

```
begin function f(a:comm) = 2  in  x:= 0;  f(x:=@x+1)  end
```

is desugared to *with* $\{f = \lambda a : comm.\ 2\}$ *do* $x := 0; f(x := @x + 1)$ and is compile-time evaluated to $x := 0; (\lambda a : comm.\ 2)(x := @x + 1)$, but the actual parameter, which is not a value, can not be bound to the formal parameter until run-time. Only when the run-time storage vector is available can the command be evaluated to a value. (In SML, the resulting "value" is ().) Thus, the compiler must generate object code for making the run-time binding.

4.2 Expressions as values

A related situation arises with arithmetic expressions. In a call-by-name language, all expressions are values, and the examples:

```
begin function f = @x + 1   in  x := f ; x := f  end

begin procedure p(f:int) = (x := f ; x := f)  in  call p(@x+1)  end
```

both compile to $x := @x + 1$; $x := @x + 1$. This is an example of the classic Algol-60 copy rule in action. In contrast, in a typical call-by-value language, numerals are values, but compound arithmetic expressions, like $@x + 1$, are not. The best a compiler can do with the previous examples is: *with* $\{f = @x + 1\}$ *do* $x := f$; $x := f$ and $(\lambda f : int.\ x := f;\ x := f)(@x + 1)$, respectively. The compiler can not copy the body of f for its invocations. The reason should be clear: the evaluation of `@x+1` to a numeral fixes `f`'s value for all its subsequent uses. Since `@x+1` requires the run-time store for its evaluation, the compiler must generate object code to evaluate the expression and bind its result into (a cell for) `f`.

The above example should not be read as suggesting necessarily that call-by-name is inherently better than call-by-value, but it *is* different. One should note from the example, however, that confusion easily arises when the semantics of expression definitions (*i.e.*, functions) does not correspond to the semantics of expression parameters.

A compiler that folds constants can evaluate arithmetic expressions like 1+2 to values like 3 and copy the results at compile-time.

4.3 Declarations as values

Modules are declarations that are records, and Clauses (ii) and (iii) of Definition 3 ensure that a module is a value exactly when all its components are. The consequences are straightforward. But in imperative languages, variable declarations in modules can affect sharing. Say that a declaration, **var x**, is desugared to **alias x=** *allocate*, where *allocate* needs the run-time store to evaluate. The sharing of variable x by modules n and p in:

```
begin module m = { var x } ,
  module n(a:{x:intloc}) = begin use a in
                            { procedure p = x:=0 }
                           end ,
  module p(b:{x:intloc}) = begin use b in
                            { procedure q = x:=@x+1 }
                           end
in  use n(m) , use p(m)
end
```

is directly dependent upon whether or not the *allocate* operation is a value. This example makes clear why compilers typically "evaluate" *allocate* to a relative address.

5 Conclusion

We have shown the importance of the correspondence principle to modular, higher-order programming languages, and we have validated correspondence by proving it is weak exponentiation in a symmetric monodial closed category. A simple perspective to the main result, the correspondence theorem (Theorem 1), is that the theorem is the natural generalization of the cartesian closedness property for the call-by-name simply-typed λ-calculus [7] to monoidal closedness for call-by-value λ-calculi. But it is only because of the correspondence between product and function space that the generalization is possible. The variety of correspondence (*i.e.*, which variant of call-by-value) is unimportant—what *is* important is the correspondence principle itself.

Acknowledgements

Olivier Danvy, Pascal Fradet and the referees contributed many helpful suggestions. Thanks are also due to Andrzej Filinski for perceptive comments which led to a generalization of the main result. Peter O'Hearn's comments benefitted

Section 3. Naz Karjian's help in making slides out of Figure 1 for the MFPS presentation is acknowledged. The categorical diagram of Section 3 was typeset using John Reynolds' category-theory macros.

References

[1] Anindya Banerjee, Olivier Danvy, and David A. Schmidt. A programming language workbench. Technical Report CIS-92-4, Kansas State University, Manhattan, Kansas, November 1991.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17:471–522, 1985.

[3] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. Rapports de Recherche 1617, I.N.R.I.A., February 1992.

[4] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[5] Jan W. Klop. *Combinatory Reduction Systems*. Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[6] Jan W. Klop. Term rewriting systems. Technical Report CS-R9073, CWI: Center for Mathematics and Computer Science, Amsterdam, 1990.

[7] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1986.

[8] Peter J. Landin. A correspondence between ALGOL60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[9] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[10] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[11] David B. MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, Austin, Texas, August 1984.

[12] David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, January 1986.

[13] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional programming languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[14] Ian A. Mason and Carolyn L. Talcott. Inferring the equivalence of functional programs that mutate data. *Theoretical Computer Science*, 105(2):167–215, 1992.

[15] Michael O'Donnell. *Computing in Systems Described by Equations*. Number 58 in Lecture Notes in Computer Science. Springer-Verlag, 1977.

[16] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Foundation of Computing Series. The MIT Press, 1991.

[17] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[18] John C. Reynolds. GEDANKEN — a simple typeless language based on the principles of completeness and the reference concept. *Communications of the ACM*, 13:308–319, 1970.

[19] John C. Reynolds. The essence of Algol. In van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, 1982. North-Holland.

[20] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1988.

[21] David A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, Cambridge, MA. To appear.

[22] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[23] Robert D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[24] Robert D. Tennent. *Principles of Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[25] Robert D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.

[26] David Watt. *Programming Languages Concepts and Paradigms*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[27] Stephen Weeks and Matthias Felleisen. On the orthogonality of assignments and procedures in algol. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 20–35, Charleston, South Carolina, January 1993.

An operational semantics for TOOPLE: A statically-typed object-oriented programming language *

Kim B. Bruce   Jonathan Crabtree[†]   Gerald Kanapathy[‡]

Department of Computer Science
Williams College
Williamstown, MA 01267
kim@cs.williams.edu

Abstract

In this paper we present an operational semantics for the language TOOPLE, a statically-typed functional object-oriented programming language which has a number of desirable properties. The operational semantics, given in the form of a natural semantics, is significantly simpler than the previous denotational semantics for the language. A "subject reduction" theorem for the natural semantics provides a proof that the language is type-safe. We also show that the natural semantics is consistent with the denotational semantics of the language.
Computing Review categories: D3.2 Object Oriented Languages, F3.2 Operational Semantics, F3.3 Type structure.

1 Introduction

Object-oriented languages promise to provide support for reusability and modularity of program code. Reusability is achieved by inheritance, which allows subclasses to be created easily from classes, and by subtyping, which allows elements of a subtype to be used in contexts which expect elements of the supertype. Modularity is achieved by the encapsulation of methods, which gives programs independence from the implementation details of the classes they use.

Static typing has clear advantages for programming languages as long as it does not interfere with expressibility in the language. Unfortunately, most extant statically-typed object-oriented languages are either type-unsafe or are unduly restrictive in the programs accepted by the type checker. For example, the holes in the type system for Eiffel are well-known, while C++, Object Pascal, and Modula-3 are unduly restrictive in not allowing changes in the types of methods in subclasses (derived classes). The design of the language TOOPLE[1] (see [Bru93a, Bru93b]) represents progress in solving both of these problems

*This research was partially supported by NSF grant CCR - 9121778.
[†]Current address: Department of Computer and Information Science, University of Pennsylvania.
[‡]Current address: Department of Computer Science, Rice University.

[1]TOOPLE is a minor modification of TOOPL (Typed Object-Oriented Programming Language) in which class terms are required to include slightly more typing information as part of their syntax. This change was necessary in order to provide an algorithm for type-checking terms and ensure that each term has a minimal type.

by ensuring type-safety while providing greater expressibility than other statically-typed object-oriented languages which are type-safe. The introduction to [Bru93b] includes an extensive comparison with other statically-typed object-oriented programming languages. We simply remark here that a key to the combination of safety and greater expressibility in TOOPLE is the separation of the subclass and subtype hierarchies, as suggested in [CHC90].

An important design goal of TOOPLE was to provide a modular type-checking system for the language. In most current object-oriented languages, the inheritance mechanism creates problems for the type checker. While a program that merely uses a class can be written and checked independently of the class code (assuming that an interface and corresponding type specification are given), to type check a class which inherits from another class, one often needs to go back and repeat the process of type checking the bodies of inherited methods from the superclass. This is necessary in order to ensure that overriding other methods from the superclass (in particular, changing their types) does not affect the types of inherited methods.

In an ideal object-oriented language, the inheritance mechanism itself will be modular. That is, one should be able to write and check programs, and find errors, looking only at the types of the superclass and the code of the modifications. For instance, a type-checking mechanism with these properties will be necessary if vendors are to be able to distribute libraries in compiled form only. The type-checking rules for TOOPLE do provide this modularity. The user need only know the type of a class in order to define and type check any subclass of that class.

We can summarize these important properties of TOOPLE as follows:

- **Type safety:** If a term has a type, τ, then the result of evaluating that term will be an element of type τ. In particular, no error messages of the form "message not understood" will arise during the evaluation of a well-typed term.
- **Modularity of type-checking:** If a class has a type, then methods inherited in a subclass continue to have the types specified in the superclass. Moreover, in order to type-check a subclass we need not have access to the bodies of methods inherited from the superclass. Only the types of the inherited methods are needed from the superclass.

Earlier papers on TOOPL provided a denotational semantics of the language. Building on earlier work of Cook *et al.* ([CHC90]) and Mitchell ([Mit90]) on the semantics of inheritance in typed programming languages, the denotational semantics is based on a higher-order extension of $F_{\leq}$, the bounded second-order lambda calculus. The denotational semantics involves fixed points at both the element and type level, making the semantics more complex than might be desired.

For some purposes, such as implementation or program verification, a natural semantics is more useful. Thus, in this paper, we present an operational semantics for the language. This semantics is significantly less complex than the denotational semantics, as it involves no higher order concepts and no fixed points. The operational semantics is given in the form of a natural semantics (see [NN92] or [Gun92] for more details on natural semantics).

In [Bru93a, Bru93b], the first author showed that the language is type-safe by showing that the meaning of a term is included in the set of values corresponding to its type. Here we provide an alternative proof of this fact by proving a "subject reduction" theorem for the operational semantics. This theorem states that if a term has a type, then the term which results from fully reducing (evaluating) the original term also has the same type.

Of course when one provides a different style of semantics for a language, one is in danger of creating a semantics which is no longer consistent with the original one. Thus we prove that the operational semantics is consistent with the earlier denotational semantics. In particular if a term e reduces to a term v in the operational semantics, we show that e and v have the same denotational semantics.

In section 2 of this paper, we provide a very brief description of the syntax of TOOPLE, along with a few simple sample programs. In section 3, we present the operational and denotational semantics of TOOPLE as well as some preliminary lemmas which are necessary for the results in the rest of the paper. This includes a statement of the minimal typing theorem for TOOPLE, which was proved in [BCD+93]. In section 4, the subject reduction theorem is proved. As noted above, this leads to an alternative proof of the type safety of the language. In section 5, the natural semantics of TOOPLE is shown to be consistent with the denotational semantics of the language. Finally, in the last two sections we provide a brief comparison with other attempts at modeling object-oriented programming languages, and conclude with a discussion of other results on TOOPLE.

Because of space restrictions, this version of the paper only discusses the restriction of TOOPLE to a language in which classes have no instance variables.

2 A brief introduction to TOOPLE

TOOPLE is a statically-typed functional object-oriented programming language. It offers full support for object-oriented features including objects, classes, methods, hidden instance variables, dynamic method invocation, subclasses, and subtypes. Moreover, TOOPLE provides mechanisms to allow the programmer to refer to the current object (*self*), its type (*MyType*), and the record of methods of its superclass (*super*). We presume the reader is familiar with the fundamental concepts of object-oriented languages, though they are described briefly below.

Objects consist of a collection of *instance variables*, representing the state of the object, and a collection of *methods*, which are routines for manipulating the object. When a *message* is sent to an object, the corresponding method of the object is executed. *Classes* are extensible templates for creating objects. In particular, classes contain initial values for instance variables and the bodies for methods. All objects generated from the same class share the same methods, but may contain different values for their instance variables. A *subclass* may be defined from a class by either adding to or modifying the methods and instance variables of the original class. (Restrictions on the modification of the types of methods and instance variables in subclasses are necessary in order to preserve type-safety.) Methods which are not modified in subclasses are said to be *inherited* from their superclass.

All terms of the language, including both classes and objects, have associated types. We say type T is a *subtype* of U if a value of type T can be used in any context in which a value of type U is expected. Note that subtyping depends only on the type of values, while subclasses and inheritance depends upon their implementations. It was pointed out in [CHC90] that if one class is a subclass of another, the type of the objects generated by the subclass need not be a subtype of the type of the objects generated by the original class.

A bound variable, usually written as *self*, may be used in methods as a name for the current object. Since our language is statically typed, it will be necessary to assign a type to all occurrences of *self*. Because the meaning of *self* will change when methods are inherited in subclasses, its type will change as well. Thus we will use a bound variable, usually written as *MyType*, as the type of *self*.

Finally, when new definitions are given to methods in a subclass, it is useful to be able to refer to the methods of the superclass. For instance, one often wishes to apply the method body from the superclass and then perform a few more operations before returning from the redefined method. We provide a bound variable, usually written as *super*, to refer to the record of methods of the superclass.

We note that instance variables are omitted in this conference paper to keep the language as simple as possible. The addition of instance variables raises no serious complications in the development of the technical results.

The types for TOOPLE are defined as follows:

Definition 2.1 *Let $\mathcal{V}^{Tp}$ be an infinite collection of type variables, $\mathcal{L}$ be an infinite collection of labels, and C^{Tp} be a collection of type constants which includes at least the type constants Bool and Num. The type expressions with respect to $\mathcal{V}^{Tp}$ and C^{Tp} are defined as follows:*

1. *If $t \in \mathcal{V}^{Tp} \cup C^{Tp}$ then t is a type expression.*
2. *If σ and τ are type expressions, then so is $\sigma \rightarrow \tau$.*
3. *If $m_1, \ldots, m_n \in \mathcal{L}$ and $\tau_1, \ldots, \tau_n$ are type expressions , then $\{m_1 : \tau_1; \ldots; m_n : \tau_n\}$ is a (record) type expression.*
4. *If τ is a record type expression and MyType $\in \mathcal{V}^{Tp}$, then ObjectType(MyType)τ and ClassType(MyType)τ are type expressions. MyType is considered to be a bound variable in these two type expressions, and binds all free occurrences of MyType in τ.*

Types of the form $\sigma \rightarrow \tau$ represent function spaces. Object types are written in the form *ObjectType*(*MyType*)τ, where τ is the type of the record of methods of the object. Similarly, class types are of the form *ClassType*(*MyType*)τ.

Definition 2.2 *The pre-terms of TOOPLE are as follows:*

$$\begin{aligned} M ::= \;& x \mid \textit{if } B \textit{ then } M \textit{ else } N \mid \textit{fun}(v : \sigma)\, M \mid M\ N \mid M = N \mid e.m_i \mid \\ & \{m_1 = e_1, \ldots, m_n = e_n\} \mid \textit{class}(\textit{self} : \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau)e \mid \\ & \textit{new } c \mid o \Leftarrow m \mid \textit{obj}(\textit{self} : \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau)e \mid \\ & \textit{update } c \textit{ by}(\textit{self} : \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau'; \textit{super})\{m_1 = e_1'\} \mid \\ & \textit{extend } c \textit{ with}(\textit{self} : \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau'; \textit{super})\{m_{n+1} = e_{n+1}\}. \end{aligned}$$

In the above grammar, B, M, N, c, o, e, and the various e_i refer to pre-terms.

Most of the pre-terms should be self-explanatory. A pre-term of the form *class*(*self* : *MyType* $\leq_{meth}$ *ObjectType*(*MyType*)τ)e represents a class whose method bodies are contained in the record e with type τ.[2] As discussed earlier, *self* can be used in the body of a method to refer to the object executing the method. *MyType* represents the type of *self*.

A pre-term of the form *obj*(*self* : *MyType* $\leq_{meth}$ *ObjectType*(*MyType*)τ)e represents an object with method bodies in e.[3] If c is a class then *new* c represents an object

[2]The addition of $\leq_{meth}$ *ObjectType*(*MyType*)τ after *MyType* in class definitions is necessary in order to obtain a minimal type for all terms of TOOPLE. This addition for class, update, and extend terms is the only difference between the language TOOPL described in previous papers and the language presented here.

[3]Obj terms are not actually in the source language. However they are used as an intermediate form of a term in the natural semantics. As a result, we include them here.

generated from c. "*Update*" and "*extend*" pre-terms provide ways of modifying or adding new methods to a class. A pre-term of the form $o \Leftarrow m$ represents sending the message m to object o. Sample TOOPLE code is given at the end of this section.

Aside from the subtyping relation ($\leq$) discussed above, we need another ordering on object types which is related to types obtained via subclasses. This ordering, $\leq_{meth}$, is a pointwise ordering on method types. If o is an object of type $ObjectType(MyType)\tau$, generated from class c, and $ObjectType(MyType)\tau'$ is the type of an object generated from a subclass of c, then $ObjectType(MyType)\tau' \leq_{meth} ObjectType(MyType)\tau$. The axioms and rules for $\leq$ and $\leq_{meth}$ are given in Appendix A. The subtyping rules and axioms are given with respect to a collection, C, of simple type constraints of the form $t \leq \tau$ and $t \leq_{meth} \tau$. See [Bru93b] or [Bru93a] for further explanation. Note that $C \vdash ObjectType(MyType)\tau' \leq_{meth} ObjectType(MyType)\tau$ iff $C \vdash \tau' \leq \tau$.

Most rules should be familiar with the possible exception of the subtyping rule for object types. This rule arises from the fact that object types are defined recursively (in order for *MyType* to stand for the type of the object in its type definition).

The actual terms of TOOPLE are those which can be type checked with respect to a collection, C, of simple type constraints, and an assignment, E, of types to variables. The type-checking rules for TOOPLE can be found in [Bru93b] or [Bru93a], where they are explained in some detail. Note that it is possible to redefine a method in a subclass in such a way that the type of the new method is a subtype of the type in the superclass.

The following restrictions on type constraint systems will allow us to show that each term has a minimum type.[4]

Definition 2.3 *Let C be a type constraint system. We say that C is* manageable *if the following conditions hold, where s and t range over type variables:*

1. *If $(t \leq_{meth} ObjectType(MyType)\tau) \in C$, then there is no term of the form $(s \leq t) \in C$.*
2. *There are no terms of the form $(t \leq ObjectType(MyType)\tau) \in C$.*

These two rules essentially disallow introducing a type variable which is a subtype of an object type.

A collection of type-checking rules for computing the minimal types of terms of TOOPLE can be found in Appendix B. These rules represent an algorithm for computing the minimal type of a term of TOOPLE as long as C is a manageable type constraint system. Note that there are two rules for each of function application (*MAppl* and *MAppl'*), record field extraction (*MProj* and *MProj'*), and message passing (*MMsg* and *MMsg'*). These are necessary since these operations may be applied to items whose minimum types are type variables. The most important place where this arises is when a message is sent to *self*, whose type is *MyType*.

The following theorem from [BCD+93] gives the relation between the type-checking rules given in [Bru93a, Bru93b] and the minimal typing rules given in Appendix B.

Theorem 2.4 *Suppose that C is manageable. Then $C, E \vdash e : \tau$ according to the type-checking rules in [Bru93b] or [Bru93a] iff there is a type τ' such that $C, E \vdash_M e : \tau'$ and $C \vdash \tau' \leq \tau$.*

As a result, it will be sufficient to use the rules for deriving minimum types given in Appendix B. This will be useful later in the paper after we have introduced the natural

[4]In [BCD+93] we restrict type constraints systems even further. However the definition of manageable type constraints given here is sufficient to prove minimum types for terms.

(operational) semantics of TOOPLE. We will show that the minimum types of terms generated in the evaluation of a TOOPLE term are all subtypes of the minimum type of the original term.

We end this brief introduction with the inclusion of a few examples of terms and their types from [Bru93b].

$$\begin{aligned}&\mathit{PointClass} = \\ &\quad \mathit{class}(\mathit{self} : \mathit{MyType} \le_{meth} \mathit{ObjectType}(\mathit{MyType})\{x, y : \mathit{Int};\ eq : \mathit{MyType} \to \mathit{Bool}\}) \\ &\quad \{x = 0,\ y = 0,\ eq = \mathit{fun}(p : \mathit{MyType})((\mathit{self} \Leftarrow x) = (p \Leftarrow x))\&((\mathit{self} \Leftarrow y) = (p \Leftarrow y))\}\end{aligned}$$

has type $\mathit{PointClassType} = \mathit{ClassType}(\mathit{MyType})\{x, y : \mathit{Int};\ eq : \mathit{MyType} \to \mathit{Bool}\}$, and represents points with x, y, and eq methods.

$\mathit{PtObj} = \mathit{new}\ \mathit{PointClass}$ is an object generated from *PointClass*. Its type is

$$\mathit{PointType} = \mathit{ObjectType}(\mathit{MyType})\{x, y : \mathit{Int};\ eq : \mathit{MyType} \to \mathit{Bool}\}.$$

Let

$$\mathit{ColorPointType} = \mathit{ObjectType}(\mathit{MyType})\{x, y : \mathit{Int};\ c : \mathit{ColorType};\ eq : \mathit{MyType} \to \mathit{Bool}\}.$$

We can add a color method to *PointClass* using the "extend" term:

$$\begin{aligned}&\mathit{ColorPointClass} = \\ &\quad \mathit{extend}\,\mathit{PointClass}\ \mathit{by}(\mathit{self}{:}\mathit{MyType} \le_{meth} \mathit{ColorPointType};\ \mathit{super})\ \{c = \mathit{Red}\}\end{aligned}$$

If we wish to change the method eq so that it now also checks the color components of two records, we define

$$\begin{aligned}\mathit{NuColorPtClass} = {}& \mathit{update}\ \mathit{ColorPointClass}\ \mathit{with}\ (\mathit{self}{:}\mathit{MyType} \le_{meth} \mathit{ColorPointType}; \\ & \mathit{super})\ \{eq = \mathit{fun}(p{:}\mathit{MyType})\ \mathit{super}.eq(p)\ \&\ ((\mathit{self} \Leftarrow c) = (p \Leftarrow c))\ \}.\end{aligned}$$

Notice the use of *super* in the updated eq method to perform the old eq body, before testing the equality of colors.

Finally we note that by rule (*MMsg*) the term $\mathit{PtObject} \Leftarrow eq$ has type $\mathit{PointType} \to \mathit{Bool}$. On the other hand, if *ColorPtObject* is an object generated from *NuColorPointClass* having type *ColorPointType*, then $\mathit{ColorPtObject} \Leftarrow eq$ has type $\mathit{ColorPointType} \to \mathit{Bool}$. This illustrates the flexibility obtained by the use of *self* and its type *MyType*.

3 Semantic definitions and preliminary lemmas

In this section we present some fundamental definitions and lemmas which will be useful in the proofs in the following sections. We also present the natural and denotational semantics for TOOPLE. We begin with a description of some of our notation.

Definition 3.1 *We write $a \equiv b$ to denote that a and b are syntactically identical, up to renaming of bound variables.*

Definition 3.2 *We write $e[a/x]$ to denote the substitution of a for x in e, where we first rename bound variables as necessary to avoid capture of free variables.*

The natural semantics and denotational semantics for terms of TOOPLE can be found in Appendices C and D. In the natural semantics we will use $C, E \vdash e : \tau \downarrow v$ as an abbreviation for $C, E \vdash e : \tau$, and $e \downarrow v$. We read this as e is a term with type τ which reduces to v, which is an irreducible term.

The irreductible terms are constants, function abstractions, records, classes, and objects. Most rules for non-object-oriented features should be familiar. By *RRecord*, *RClass*, *RAppl*, and *RNew*, the evaluation strategy is "lazy." That is, subterms are not evaluated until necessary.

The most interesting rule is *RMsg*, for sending a message to an object. When a term of the form $o \Leftarrow m$ is evaluated, o is reduced to a term of the form, $o' = obj(self : MyType \leq_{meth} \gamma')e$, where e is its record of methods. Then *self* is replaced by o' and *MyType* by γ' in e, which is then evaluated to a record term. Finally, the record component corresponding to m is evaluated and returned as the final answer.

While *self* is an irreducible value, it should be noted that when a message is sent to an object, all occurrences of *self* are replaced by the object, so *self* no longer occurs in the method body when it is actually evaluated.

The subject reduction theorem, presented in section 4, will show that the reduced term, v, will have a minimal type that is a subtype of the type of the original term e. It follows that, in the original typing system, if $C, E \vdash e : \tau \downarrow v$, then $C, E \vdash v : \tau$.

The substitution lemma for types is necessary to prove the subject reduction theorem. It ensures that substitution is a well-behaved operation with respect to the subtyping relation. More formally:

Lemma 3.3 *Let C be manageable and let x be a variable. Assume that there exist terms and types such that $C, E \vdash_M e : \tau$, $C, E \vdash_M x : \sigma$, $C, E \vdash_M a : \sigma'$, and $C \vdash \sigma' \leq \sigma$. Then there exists some τ' such that $C, E \vdash_M e[a/x] : \tau'$ and $C \vdash \tau' \leq \tau$. Furthermore, if τ is a class type, then $\tau' \equiv \tau$.*

Proof. The proof is by induction on the proof of minimum typing. The base case is straightforward. We present only a few of the inductive cases.

Inductive assumption. For all C, E, if $C, E \vdash_M e' : \rho$ in fewer than n steps and x and a are as described above, then there exists some ρ' such that $C, E \vdash_M e'[a/x] : \rho'$, where $C \vdash \rho' \leq \rho$. Furthermore, if ρ is a *ClassType* type, then $\rho' = \rho$.

MClass. $C, E \vdash_M class(self : MyType \leq_{meth} ObjectType(MyType)\tau)e :$
$ClassType(MyType)\tau$.

Case 1: $x = self$. In this case the expression is unchanged by the substitution as *self* is a bound variable of the term. As a result the type of the term is unchanged by the substitution.

Case 2: $x \neq self$. Thus, $(class(self : MyType \leq_{meth} ObjectType\ (MyType)\tau)e)[a/x] \equiv class(self : MyType \leq_{meth} ObjectType(MyType)\tau)e'$, where $e' \equiv e[a/x]$. By induction, $C, E \vdash_M e' : \tau'$, where $C \vdash \tau' \leq \tau$. Since classes have unique types, the class typing rule allows us to prove that $C, E \vdash_M (class(self : MyType \leq_{meth} ObjectType(MyType)\tau)e)[a/x] : ClassType(MyType)\tau$.

MMsg. $C, E \vdash_M o \Leftarrow m_i : \tau_i[\gamma/MyType]$.

We may assume that $C, E \vdash_M o : \gamma$, where $\gamma = ObjectType(MyType)\{\ldots; m_i : \tau_i; \ldots\}$, by the typing rules. By induction, $C, E \vdash_M o[a/x] : \gamma'$, where $\gamma' = ObjectType(MyType)\{\ldots; m_i : \tau_i'; \ldots\}$, and $C \vdash \gamma' \leq \gamma$. Note that the minimum type of $o[a/x]$ must be an *ObjectType* type, since, by the second part of the definition of manageability, there can be no expressions of the form $(t \leq ObjectType(MyType)\tau) \in C$. By the object subtyping rule, $C \cup \{s \leq t\} \vdash \{\ldots; m_i : \tau_i'; \ldots\}[s/MyType] \leq \{\ldots; m_i : \tau_i; \ldots\}[t/MyType]$, and it follows that $C \cup \{s \leq t\} \vdash \tau_i'[s/MyType] \leq \tau_i[t/MyType]$. Finally, because s and t do not occur in C, substitution of the types of the γ' and γ for s and t gives us that $C \vdash \tau_i'[\gamma'/MyType] \leq \tau_i[\gamma/MyType]$.

MMsg'. $C, E \vdash_M o \Leftarrow m_i : \tau_i[t/MyType]$.

$C, E \vdash_M o : t$, where t is a type variable. Thus $(t \leq_{meth} ObjectType(MyType)\{\ldots; m_i : \tau_i; \ldots\}) \in C$. Note that $(o \Leftarrow m_i)[a/x] \equiv o[a/x] \Leftarrow m_i$. Now, by induction, $C, E \vdash_M o[a/x] : t'$, where $C \vdash t' \leq t$. But the first part of the definition of C being manageable asserts that there can be nothing of the form $(r \leq t) \in C$, and, since t is a variable, no other proof can exist which shows that t' is a subtype of t. Therefore, $t = t'$, and so $C, E \vdash_M o[a/x] \Leftarrow m_i : \tau_i[t/MyType]$.

MUpdate. $C, E \vdash_M$ *update c by* $(self : MyType \leq_{meth} ObjectType(MyType)\gamma'; super)$ $\{m_1 = e_1'\} : ClassType(MyType)\ \gamma'$.
where $\gamma' = \{m_1 : \tau_1'; m_2 : \tau_2; \ldots; m_n : \tau_n\}$.

By the observations in the (MClass) case, we may assume that $x \neq self$, as no change will take place otherwise. Now note that

$$\begin{aligned}&(update\ c\ by\ (self : MyType \leq_{meth} ObjectType(MyType)\ \gamma'; super)\{m_1 = e_1'\})[a/x] \\ &\quad \equiv update\ c[a/x]\ by\ (self : MyType \leq_{meth} ObjectType(MyType)\ \gamma''; super) \\ &\qquad\qquad \{m_1 = e_1'[a/x]\}).\end{aligned}$$

By the typing rules, c has some *ClassType* type, and so by induction, $c[a/x]$ must have the same type. Thus $C, E \vdash_M c[a/x] : ClassType(MyType)\{m_1 : \tau_1; \ldots; m_n : \tau_n\}$. Also, by induction, $C, E \vdash_M e_1'[a/x] : \tau_1''$, where $C \vdash \tau_1'' \leq \tau_1'$. Thus we can use (MUpdate) to prove that $C, E \vdash_M$ *update* $c[a/x]$ *by* $(self : MyType \leq_{meth} ObjectType(MyType)\gamma'; super)\{m_1 = e_1'[a/x]\}) : ClassType\ \gamma'$. Thus the update expression after performing substitution has exactly the same type as it did before the substitution. ■

4 Subject reduction theorem

The subject reduction theorem shows that types are preserved under the reductions of our natural semantics. This can be used to show that TOOPLE is type-safe, since no computation on a well-typed term can ever result in a term which is ill-typed.

Theorem 4.1 Subject Reduction Theorem *Assume that $C, E \vdash_M e : \tau \downarrow e'$. Then there is a type τ' such that $C, E \vdash_M e' : \tau'$, where $C \vdash \tau' \leq \tau$. Furthermore, if τ is a class type then $\tau' \equiv \tau$.*

Proof. The proof is by induction on the number of steps used in the reduction.

Base case. If any of the rules *RAbs*, *RConst*, *RRecord*, *RClass*, or *RObj* apply, then $e' \equiv e$, so the theorem holds trivially.

Inductive cases. Suppose that for all C, E, and for all e' such that $e' \downarrow e''$ in fewer than n steps, if $C, E \vdash_M e' : \tau' \downarrow e''$ then there is some τ'' such that $C, E \vdash_M e'' : \tau''$ and $C \vdash \tau'' \leq \tau'$. Furthermore, if τ' is a class type then $\tau' \equiv \tau''$.

The proof proceeds by cases on the last semantic rule applied. In each case we assume that e is typable and that there is some v such that $e \downarrow v$, in n steps. We begin each case by specifying the form of e and the minimum type of e.

We include only a few interesting cases in this conference paper. We note that Lemma 3.3 is needed in the (omitted) case for function application.

RNew. $C, E \vdash_M$ *new* c : $\mathit{ObjectType}(\mathit{MyType})\tau$.

By the typing rules, $C, E \vdash_M c : \mathit{ClassType}(\mathit{MyType})\tau$, and by the semantic rule, $c \downarrow \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e'$. By induction, $C, E \vdash_M \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e' : \mathit{ClassType}(\mathit{MyType})\tau$, as the types of classes are invariant. By the semantic rule $v \equiv \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e'$, and we can prove that $C, E \vdash_M \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e' : \mathit{ObjectType}(\mathit{MyType})\tau$.

RMsg. $C, E \vdash_M o \Leftarrow m_i : \tau_i[\gamma/\mathit{MyType}]$.

It is clear that o is typable, as the entire expression is typable, and we proceed by cases on the last rule of the proof of minimum type for $o \Leftarrow m_i$.

MMsg. $C, E \vdash_M o : \mathit{ObjectType}(\mathit{MyType})\{\ldots; m_i : \tau_i; \ldots\}$.

Note that $\gamma = \mathit{ObjectType}(\mathit{MyType})\{\ldots; m_i : \tau_i; \ldots\}$. Since $o \downarrow \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \gamma')e$ for $\gamma' = \mathit{ObjectType}\{\ldots; m_i : \tau_i'; \ldots\}$, then, by induction,

$$C, E \vdash_M \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \gamma')e : \mathit{ObjectType}(\mathit{MyType})\gamma', \tag{1}$$

$$\text{where } C \vdash \gamma' \leq \gamma. \tag{2}$$

By the subtyping rules, the only way (2) could hold is if $C \cup \{s \leq t\} \vdash \tau_i'[s/\mathit{MyType}] \leq \tau_i[t/\mathit{MyType}]$. It follows that

$$C \vdash \tau_i'[\gamma'/\mathit{MyType}] \leq \tau_i[\gamma/\mathit{MyType}]. \tag{3}$$

By (1) and our typing rule for *obj* terms,

$$C \cup \{\mathit{MyType} \leq_{meth} \gamma'\}, E \cup \{\mathit{self} : \mathit{MyType}\} \vdash_M e : \{\ldots; m_i : \tau_i''; \ldots\}, \tag{4}$$

where

$$C \cup \{\mathit{MyType} \leq_{meth} \gamma'\} \vdash \tau_i'' \leq \tau_i'. \tag{5}$$

Let $e' = e[\mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \gamma')e/\mathit{self}, \gamma'/\mathit{MyType}]$. By Lemma 3.3, it follows that

$$C, E \vdash_M e' : \{\ldots; m_i : \tau_i'''; \ldots\} \tag{6}$$

where $C \vdash \tau_i''' \leq \tau_i''[\gamma'/\mathit{MyType}]$.

By (5), it follows that

$$C \vdash \tau_i''[\gamma'/\mathit{MyType}] \leq \tau_i'[\gamma'/\mathit{MyType}]. \tag{7}$$

Furthermore, since $e' \downarrow \{\ldots, m_i = e_i, \ldots\}$, it follows by induction that $C, E \vdash_M e_i : \rho$, where

$$C \vdash \rho \leq \tau_i'''. \tag{8}$$

Since $e_i \downarrow v$, by induction,

$$C, E \vdash_M v : \rho', \text{ where } C \vdash \rho' \leq \rho \leq \tau_i''' \leq \tau_i''[\gamma'/\mathit{MyType}]. \tag{9}$$

Thus, by (3), (7), (9), and transitivity,

$$C \vdash \rho' \leq \tau_i[\gamma/\mathit{MyType}].$$

In summary, if $(e \Leftarrow m_i) \downarrow v$, where $C, E \vdash_M e \Leftarrow m_i : \tau_i[\gamma/\mathit{MyType}]$, then $C, E \vdash_M v : \rho'$, with $C \vdash \rho' \leq \tau_i[\gamma/\mathit{MyType}]$, as desired.

MMsg'. $C, E \vdash_M o : t$, where t is a type variable.

Note that $\gamma = t$. The other hypothesis of the (MMsg') rule must be

$$(t \leq_{meth} ObjectType(MyType)\{\ldots; m_i : \tau_i; \ldots\}) \in C.$$

By the semantic rule (RMsg), $o \downarrow obj(self : MyType \leq_{meth} \{\ldots; m_i : \tau_i'; \ldots\})e$. Thus, by induction, $C, E \vdash_M obj(self : MyType \leq_{meth} \{\ldots; m_i : \tau_i'; \ldots\})e : t'$, where $C \vdash t' \leq t$. However, any proof of minimum type for an *obj* expression must end in the (MObj) typing rule. Thus t' is of the form $ObjectType(MyType)\tau$, and so $C \vdash ObjectType(MyType)\tau \leq t$. But this implies something of the form $(r \leq t) \in C$, which is impossible due to the first part of the definition of manageability. Thus it cannot be the case that $C, E \vdash_M e : t$, where t is a type variable. ■

The subject reduction theorem ensures that our programming language is type-safe, by showing that all intermediate terms in a computation can be typed (with a subtype of the type of the original term). In particular, this implies that no term of the form $f(e)$ occurs in the computation of a well-typed term if the types of f and e do not match. Similarly no object will be sent a message that it cannot handle in a computation on a well-typed term (since such a subterm would be ill-typed).

Thus, if we begin with a term with no free variables, a computation will proceed in a type-safe way to a value or may loop. However, it will never become stuck at a non-value since each well-typed term corresponds to a computation rule in the natural semantics.

5 Consistency of the natural semantics with the denotational semantics

The theoretical results about TOOPL in [Bru93a] are given in terms of the denotational semantics presented in Appendix D. Of course, we would like to be able to claim that any results using the natural semantics presented here actually refer to the same language as the one Bruce described.

We will prove that the natural semantics presented in appendix C is sound with respect to the denotational semantics. To do this, we must show that, whenever a term M reduces to a term v in the natural semantics, their meanings are the same according to the denotational semantics.

We first need the following substitution lemma for terms.

Lemma 5.1 *Suppose s is not free in C or E, $C \cup \{s \leq \tau\}, E \cup \{x : s\} \vdash M : \gamma$, $C, E \vdash a : \sigma$, and $C \vdash \sigma \leq \tau$. Then*

1. $C, E \vdash M[a/x, \sigma/s] : \gamma[\sigma/s]$, *and*
2. $[\![C \cup \{s \leq \tau\}, E \cup \{x : s\} \vdash M : \gamma]\!]\rho[[\![C, E \vdash a : \sigma]\!]\rho/x, [\![\sigma]\!]\rho/s] =$
$[\![C, E \vdash M[a/x, \sigma/s] : \gamma[\sigma/s]]\!]\rho.$

We now prove the correctness of the natural semantics with respect to the denotational semantics.

Theorem 5.2 *If $C, E \vdash M : \tau \downarrow v$, then $[\![C, E \vdash M : \tau]\!]\rho = [\![C, E \vdash v : \tau]\!]\rho$*

Proof. We will prove the natural semantics correct by induction on the number of steps in the reduction.

Base Cases. For terms that can be reduced by one of the rules *RAbs*, *RConst*, *RRecord*, *RClass*, or *RObj*, the theorem is clearly true, since each of these rules states that $M \downarrow M$.

Inductive Cases. We prove the consistency of the natural semantics for all terms whose reduction is of length n, where $n > 1$, assuming that the semantics are correct for all terms whose reductions are of length less than n. We provide only a few of the more interesting cases.

Function application. The natural semantics rule for function application is *RAppl*, which gives us, by the induction hypothesis, that

$$\begin{aligned}[\![C, E \vdash e : \sigma \to \tau]\!]\rho &= [\![C, E \vdash \lambda x : \sigma.M : \sigma \to \tau]\!]\rho \\ &= \lambda d \in \mathcal{A}^\sigma.[\![C, E \cup \{x : \sigma\} \vdash M : \tau]\!]\rho[d/x].\end{aligned}$$

Then

$$\begin{aligned}[\![C, E \vdash e\ e' : \tau]\!]\rho &= ([\![C, E \vdash e : \sigma \to \tau]\!]\rho)([\![C, E \vdash e' : \sigma]\!]\rho) \\ &= (\lambda d \in \mathcal{A}^\sigma.[\![C, E \cup \{x : \sigma\} \vdash M : \tau]\!]\rho[d/x])([\![C, E \vdash e' : \sigma]\!]\rho) \\ &= [\![C, E \cup \{x : \sigma\} \vdash M : \tau]\!]\rho[[\![C, E \vdash e' : \sigma]\!]\rho/x] \\ &= [\![C, E \vdash M[e'/x] : \tau]\!]\rho\end{aligned}$$

where the final step follows from Lemma 5.1, part 2.

Because $C, E \vdash M[e'/x] : \tau \downarrow v$ by induction, we obtain

$$[\![C, E \vdash M[e'/x] : \tau]\!]\rho = [\![C, E \vdash v : \tau]\!]\rho.$$

Thus

$$[\![C, E \vdash e\ e' : \tau]\!]\rho = [\![C, E \vdash v : \tau]\!]\rho.$$

Objects. The rule for creation of an object from a class is *RNew*. The corresponding denotational rule is:

$$\begin{aligned}&[\![C, E \vdash new\ c : ObjectType(MyType)\tau]\!]\rho = \\ &\qquad FIX(([\![C, E \vdash c : ClassType(MyType)\tau]\!]\rho)([\![ObjectType(MyType)\tau]\!]\rho)).\end{aligned}$$

Now, by the induction hypothesis,

$$\begin{aligned}&[\![C, E \vdash c : ClassType(MyType)\tau]\!]\rho = \\ &\qquad [\![C, E \vdash class(self : MyType)e : ClassType(MyType)\tau]\!]\rho.\end{aligned}$$

Substituting and using the semantics of objects we get:

$$\begin{aligned}&[\![C, E \vdash new\ c : ObjectType(MyType)\tau]\!]\rho \\ &\quad = FIX(([\![C, E \vdash class(self : MyType)e : ClassType(MyType)\tau]\!]\rho) \\ &\qquad\qquad ([\![ObjectType(MyType)\tau]\!]\rho)) \\ &\quad = [\![C, E \vdash obj(self : MyType)e : ObjectType(MyType)\tau]\!]\rho.\end{aligned}$$

Message passing. The next case to consider is an expression that sends a message to an object. The natural semantics rule for this is *RMsg*. Without loss of generality we presume that $C, E \vdash_M o \Leftarrow m_i : \tau_i[\gamma/MyType]$. We proceed by cases on the last typing rule applied:

<u>Case i.</u> $C, E \vdash_M o : \gamma$ for $\gamma = ObjectType(MyType)\{m_1 : \tau_1; \ldots; m_n : \tau_n\}$.

The denotational definition for message passing is *Msg*:

$$\begin{aligned}&[\![C, E \vdash o \Leftarrow m_i : \tau_i[\gamma/MyType]]\!]\rho = \\ &\qquad (\mathbf{convert}[[\![\{m_i : \tau_i\}]\!]\rho[[\![\gamma]\!]\rho/MyType]][[\![\gamma]\!]\rho][\![C, E \vdash o : \gamma]\!]\rho)(m_i).\end{aligned}$$

By the **convert** rules for records in [BL90], if $\sigma \leq_{\mathcal{A}} [\![\{s_1 : \sigma_1; \ldots, s_k : \sigma_k; \ldots; s_n : \sigma_n\}]\!]\rho$, and $r \in \mathcal{A}^{[\![\sigma]\!]\rho}$, then

$$(\mathbf{convert}[[\![\{s_1 : \sigma_1; \ldots; s_n : \sigma_n\}]\!]\rho][\sigma]r)(s_k) = (\mathbf{convert}[[\![\{s_k : \sigma_k\}]\!]\rho][\sigma]r)(s_k).$$

By the semantics of object types,

$$[\![\gamma]\!]\rho = [\![\{m_1 : \tau_1; \ldots; m_n : \tau_n\}]\!]\rho[[\![\gamma]\!]\rho/MyType].$$

Thus,

$$\begin{aligned}
&(\mathbf{convert}[[\![\{m_i : \tau_i\}]\!]\rho[[\![\gamma]\!]\rho/MyType]][[\![\gamma]\!]\rho][\![C, E \vdash o : \gamma]\!]\rho)\,(m_i)\\
&= (\mathbf{convert}[[\![\{m_1 : \tau_1; \ldots; m_n : \tau_n\}]\!]\rho[[\![\gamma]\!]\rho/MyType]][[\![\gamma]\!]\rho][\![C, E \vdash o : \gamma]\!]\rho)\,(m_i)\\
&= (\mathbf{convert}[[\![\gamma]\!]\rho][[\![\gamma]\!]\rho][\![C, E \vdash o : \gamma]\!]\rho)\,(m_i)\\
&= [\![C, E \vdash o : \gamma]\!]\rho(m_i).
\end{aligned}$$

Hence $[\![C, E \vdash o \Leftarrow m_i : \tau_i[\gamma/MyType]]\!]\rho = [\![C, E \vdash o : \gamma]\!]\rho(m_i)$.

Since $o \Leftarrow m_i \downarrow v$, if follows that

$$o \downarrow obj(self : MyType \leq_{meth} \gamma')e \tag{10}$$

for some $\gamma' = ObjectType(MyType)\tau'$, and some record e such that

$$e' = e[obj(self : MyType \leq_{meth} \gamma')e/self, \gamma'/MyType] \downarrow \{m_1 = e_1, \ldots, m_k = e_k\}, \tag{11}$$

and

$$e_i \downarrow v. \tag{12}$$

By the subject-reduction theorem and (10), $C \vdash \gamma' \leq \gamma$.

Let $C' = C \cup \{MyType \leq_{meth} ObjectType(MyType)\tau'\}$ and $E' = E \cup \{self : MyType\}$. By the denotational rule for objects, *Obj*,

$$\begin{aligned}
&[\![C, E \vdash obj(self : MyType)e : \gamma']\!]\rho\\
&= FIX(([\![C, E \vdash class(self : MyType)e : ClassType(MyType)\tau']\!]\rho)([\![\gamma']\!]\rho)).\\
&= FIX((\lambda\xi \leq [\![\tau']\!]\rho[\xi/MyType].\lambda o \in \mathcal{A}^{\xi}.[\![C', E' \vdash e : \tau']\!]\rho[\xi/MyType, o/self])([\![\gamma']\!]\rho))\\
&= FIX(\lambda o \in \mathcal{A}^{[\![\gamma']\!]\rho}.[\![C', E' \vdash e : \tau']\!]\rho[[\![\gamma']\!]\rho/MyType, o/self]).
\end{aligned}$$

Therefore,

$$\begin{aligned}
&[\![C, E \vdash obj(self : MyType)e : \gamma']\!]\rho\\
&= [\![C', E' \vdash e : \tau']\!]\rho[[\![\gamma']\!]\rho/MyType, [\![C, E \vdash obj(self : MyType)e : \gamma']\!]\rho/self],\\
&= [\![C, E \vdash e' : \tau'[\gamma'/MyType]]\!]\rho, \text{ by Lemma 5.1.}\\
&= [\![C, E \vdash \{m_1 = e_1, \ldots, m_i = e_i, \ldots, m_n = e_n\} : \tau'[\gamma'/MyType]]\!]\rho, \text{ by induction.}
\end{aligned}$$

Because $C \vdash \gamma' \leq \gamma$ and both are object types, it follows that $C \vdash \tau'[\gamma'/MyType] \leq \{m_1 : \tau_1; \ldots; m_n : \tau_n\}[\gamma/MyType]$. By induction, (10) and the above expansion of $[\![obj(self : MyType)e]\!]\rho$,

$$[\![C, E \vdash o : \gamma]\!]\rho = [\![C, E \vdash \{m_1 = e_1, \ldots, m_n = e_n\} : \tau[\gamma/MyType]]\!]\rho,$$

so

$$([\![C, E \vdash o : \gamma]\!]\rho)(m_i) = [\![C, E \vdash e_i : \tau_i[\gamma/MyType]]\!]\rho = [\![C, E \vdash v : \tau_i[\gamma/MyType]]\!]\rho$$

by the *Rec* rule and the induction hypothesis, respectively. We conclude, then, that

$$[\![C, E \vdash o \Leftarrow m_i : \tau_i[\gamma/MyType]]\!]\rho = [\![C, E \vdash v : \tau_i[\gamma/MyType]]\!]\rho.$$

Case ii. $C, E \vdash_M o : \gamma$ for t a type variable.

By assumption, $o \downarrow obj(self : MyType \leq_{meth} \gamma')e$ where γ' (which is also the type of the object expression) is an object type. By the subject reduction theorem, $C \vdash \gamma' \leq t$. However, by the definition of manageable type constraint system, a type variable may not be shown to be a subtype of an object type. Thus this case will never arise!

Class update. Classes are updated according to the rule *RUpdate*.

Let $\tau = \{m_1 : \tau_1; \ldots; m_n : \tau_n\}$ and $\tau' = \{m_1 : \tau_1'; m_2 : \tau_2; \ldots; m_n : \tau_n\}$. The denotational rule, *Update*, is:

$$[\![C, E \vdash update\ c\ by\ (self : MyType \leq_{meth} ObjectType(MyType)\tau'; super)\{m_1 = e_1'\} : \\ ClassType(MyType)\tau']\!]\rho = \lambda\xi \leq [\![\tau']\!]\rho[\xi/MyType].\lambda o \in \mathcal{A}^{\xi}.f,$$

where

$$\begin{array}{lcl} C' & = & C \cup \{MyType \leq_{meth} ObjectType(MyType)\tau'\}, \\ E' & = & E \cup \{self : MyType\}, \\ dom(f) & = & \{m_1, \ldots, m_n\}, \\ f(m_1) & = & [\![C', E' \vdash e_1' : \tau_1']\!]\rho[\xi/MyType, o/self, s/super], \\ f(m_j) & = & s(m_j), \forall j : 2 \leq j \leq n, \\ s & = & [\![C, E \vdash c : ClassType(MyType)\tau]\!]\rho(\xi)(o). \end{array}$$

Since

$$C, E \vdash c : ClassType(MyType)\tau \downarrow class(self : MyType \leq_{meth} ObjectType(MyType)\tau)e,$$

it follows by induction and the semantics of classes that

$$s = [\![C', E' \vdash e : \tau]\!]\rho[\xi/MyType, o/self].$$

Let $e' = \{m_1 = e_1'', m_2 = e.m_2, \ldots, m_n = e.m_n\}$, where $e_1'' = e_1'[e/super]$. By *Class*:

$$[\![C, E \vdash class(self : MyType \leq_{meth} ClassType(MyType)\tau'; super)e' : ClassType(MyType)\tau']\!]\rho \\ = \lambda\xi \leq [\![\tau']\!]\rho[\xi/MyType].\lambda o \in \mathcal{A}^{\xi}.[\![C', E' \vdash e' : \tau']\!]\rho[\xi/MyType, o/self]$$

To complete the proof we must show that

$$f(m_i) = ([\![C', E' \vdash e' : \tau']\!]\rho[\xi/MyType. o/self])(m_i)$$

for all $\xi \leq [\![\tau']\!]\rho[\xi/MyType], o \in \mathcal{A}^{\xi}$, and i from 1 to n.

Case: $i = 1$. We can say, by the denotational rule for records, *Proj*, that

$$\begin{array}{l} ([\![C', E' \vdash e' : \tau']\!]\rho[\xi/MyType, o/self])(m_1) \\ \quad = [\![C', E' \vdash e_1'' : \tau_1']\!]\rho[\xi/MyType, o/self] \\ \quad = [\![C', E' \vdash e_1' : \tau_1']\!]\rho[\xi/MyType, o/self, s/super] \\ \quad = f(m_1). \end{array}$$

Case: $i \geq 2$. By the definition of f above,

$$\begin{array}{ll} f(m_i) = & [\![C, E \vdash c : ClassType(MyType)\tau]\!]\rho(\xi)(o)(m_i) \\ = & ([\![C, E \vdash class(self : MyType \leq_{meth} ObjectType(MyType)\tau)e : \\ & \qquad ClassType(MyType)\tau]\!]\rho)(\xi)(o)(m_i) \text{ by induction} \\ = & ([\![C^+, E' \vdash e : \tau]\!]\rho[\xi/MyType, o/self])(m_i) \text{ by } Class \\ = & ([\![C', E' \vdash e' : \tau']\!]\rho[\xi/MyType, o/self])(m_i) \text{ by } Rec. \end{array}$$

where $C^+ = C \cup \{MyType \leq_{meth} ObjectType(MyType)\tau\}$.

Thus, the two functions are equal for all m_i in either domain. Since the functions were equal to the denotational meanings of the respective expressions, we can say that the two are equivalent.

The case for class extensions is similar. ■

6 Comparison with previous work

As indicated earlier, the work on TOOPLE grew out of work in [CHC90] and [Mit90] on the semantics of inheritance in typed object-oriented languages. We have also been greatly influenced by the work of Luca Cardelli.

The most interesting comparable work to ours is that of Cardelli and of Benjamin Pierce. In their papers, both alone and with collaborators (see [Car88a, CW85, Car88b, CL91, CM90, Car92] and [PT93, PT92, PH92]), both authors have been striving to find a core language which can be used to model all of the common features of object-oriented programming languages. Each uses extensions of $F_{\leq}$, the bounded second-order lambda calculus. While we have preferred to use the F-bounded second-order lambda calculus (see [CCH+89]) as a basis for the denotational semantics of our language, they have preferred a different variant which involves taking fixed points of higher order functions from types to types. As indicated by [Aba92], these extensions of $F_{\leq}$ are essentially identical.

A major difference between both Cardelli and Pierce and our work is that they adopt a mainly syntactic point of view of translating object-oriented features into extensions of the second-order lambda calculus. We have adopted a more semantic approach, originally giving the denotational semantics of TOOPLE in a model of the second-order lambda calculus. We have continued this approach here, treating our object-oriented programming language features as primitive and providing an operational semantics in terms of these constructs.

Pierce's approach eliminates the need for fixed points at the type level in his language (though they are still required at the element level to model objects). The price to be paid for this is not being able to express classes with binary methods like *eq* which take parameters of type *MyType*, as in *PointClass* in section 2. While our denotational semantics for TOOPLE requires fixed points at both the term and type levels, the natural semantics provided here is significantly simpler.

Castagna, Ghelli, and Longo (see [CGL92]) have proposed an interesting new approach to providing the features of object-oriented programming languages. They propose replacing inheritance by a disciplined use of overloading of operations. When combined with subtyping, the resulting language has many interesting features, including a mechanism for dealing with *multi-methods*, methods whose execution depends on the types of several parameters rather than just the type of the receiver of the message, as in most object-oriented programming languages.

Each of these approaches to modeling object-oriented programming languages has its strengths and weaknesses, many of which will be apparent only with time and experience. It is already clear that each of these approaches represents significant progress toward the ultimate understanding of object-oriented programming languages.

7 Summary and further work on TOOPLE

In this paper we presented a natural semantics (a form of operational semantics) for the statically-typed, functional, object-oriented language, TOOPLE. The natural semantics has the advantage of being easier to understand than the denotational semantics, since the denotational semantics requires fixed points at both the element and type levels. The major results in the paper were proofs of a subject reduction theorem for the natural semantics, and a consistency theorem for the natural semantics relative to the denotational semantics.

The language presented in this conference paper does not include instance variables. The language with only methods is extremely limited in expressibility, but we decided that it was much easier to present this simpler language in the limited space available here. We urge the reader to see [Bru93a] for a full discussion of TOOPLE with instance variables. The natural semantics presented here can be extended easily to the full language, and the theorems and proofs carry over fairly directly to this more complex language.

The subject reduction theorem shows that the natural semantics preserves the type system for the language. Thus a well-typed term can never go "wrong". In particular, it shows that a well-typed term will never result in a computation in which a message is sent to an object which cannot handle it. The earlier papers [Bru93b] and [Bru93a] included other results with respect to a denotational semantics which indicated that the language was type-safe.

The proofs of the subject reduction theorem and the type safety of the denotational semantics helped us discover and eliminate errors in the type-checking rules that might have remained had we not built the language on this theoretical base. It is our hope that this deeper understanding of object-oriented programming languages will provide the basis for a careful analysis of the pros and cons of each of their individual features. This should lead to the design of safe languages which are easy to reason about, and whose expressiveness is similar to that found in today's popular object-oriented languages.

Since earlier papers presented the semantics of the language as a denotational semantics, we showed here that the natural semantics is consistent with those earlier semantics.

The paper, [BCD+93], presents further results on TOOPLE. It shows that type checking TOOPLE is decidable and that every term of TOOPLE has a minimal type. The decidability of type checking was in some doubt since Pierce [Pie92] showed that type checking $F_{\leq}$ was undecidable and the denotational semantics of TOOPLE is expressed in an extension of $F_{\leq}$.

R. van Gent at Williams College has built a TOOPLE type checker and interpreter which is based on the type-checking algorithm presented in [BCD+93] and the natural semantics presented in this paper. More recently, Bruce and van Gent [vG93, BvG93] have designed an imperative language, TOIL, with a type system extending that of TOOPLE. Similar results about the safety and decidability of type-checking hold for TOIL. An interpreter has been written which is being used to further investigate the language. Work is currently proceeding on extending the type system of TOIL to include explicit polymorphism. We are also investigating the development of proof axioms and rules for reasoning about TOOPLE and TOIL programs.

While TOOPLE is missing many of the important features necessary to provide a truly useful language, we believe that TOOPLE can serve as the core of a statically-typed object-oriented language which combines type-safety with expressiveness approaching or even exceeding that of languages which fail to be strongly typed.

As this paper was going to press, we learned of the development of the language Strongtalk [BG93], which has adopted essentially the typing rules for TOOPLE (along with a few extensions) in order to type check a subset of Smalltalk. We look forward to learning of the efficacy of this typing system in large programming projects.

Acknowledgements

Thanks to the following for helpful comments on TOOPL and this entire project: Luca Cardelli, Giuseppe Longo, John Mitchell, Benjamin Pierce, the Harvard University Object-Oriented Languages Seminar (including Peter Wegner, Andrew Black, Allyn Dimock, and Robert Muller), and the other members of the Williams College TOOPL research group: Tom Murtagh and Robert van Gent. Special thanks go to Jon Riecke, who originally suggested that a structured operational semantics might be significantly simpler than the original denotational semantics, and introduced us to modern treatments of operational semantics.

References

[Aba92] M Abadi. Doing without F-bounded quantification. Message to Types electronic mail list, February, 1992.

[BCD+93] K. Bruce, J. Crabtree, A. Dimock, R. Muller, T. Murtagh, and R. van Gent. Safe and decidable type checking in an object-oriented language. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 29–46, 1993.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 215–230, 1993.

[BL90] K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990.

[Bru93a] K. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. Technical Report CS-92-01, revised, Williams College, 1993. To appear in Journal of Functional Programming.

[Bru93b] K. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 285–298, 1993.

[BvG93] Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. to appear, 1993.

[Car88a] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types*, Sophia-Antipolis (France), 1984.

[Car88b] L. Cardelli. Structural subtyping and the notion of powertype. In *Proc 15th ACM Symp. Principles of Programming Languages*, pages 70–79, 1988.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Technical Report 81, DEC Systems Research Center, 1992.

[CCH+89] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object-oriented programming. In *Functional Prog. and Computer Architecture*, pages 273–280, 1989.

[CG92] P.L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[CGL92] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 182–192, 1992.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, January 1990.

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[CM90] L. Cardelli and J.C. Mitchell. Operations on records. In *Math. Foundations of Prog. Lang. Semantics*, pages 22–52. Springer LNCS 442, 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: A formal introduction*. John Wiley & Sons, 1992.

[PH92] Benjamin C. Pierce and Martin Hoffman. An abstract view of objects and subtyping (preliminary report). Technical Report ECS-LFCS-92-226, University of Edinburgh, 1992.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proc 19th ACM Symp. Principles of Programming Languages*, pages 305–315, 1992.

[PT92] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. Technical report, University of Edinburgh, 1992.

[PT93] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Proc 20th ACM Symp. Principles of Programming Languages*, pages 299–312, 1993.

[vG93] Robert van Gent. *TOIL: An imperative type-safe object-oriented language*. Williams College Senior Honors Thesis, 1993.

A Subtyping Rules for TOOPLE

SRefl

$$C \vdash \tau \leq \tau$$

SVar

$$C \cup \{t \leq \tau\} \vdash t \leq \tau$$

STrans

$$\frac{\begin{array}{c} C \vdash \gamma \leq \sigma, \\ C \vdash \sigma \leq \tau \end{array}}{C \vdash \gamma \leq \tau}$$

SAbs

$$\frac{C \vdash \sigma' \leq \sigma, \quad C \vdash \tau \leq \tau'}{C \vdash \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'}$$

SRec

$$\frac{C \vdash \sigma_j \leq \tau_j, \forall j : 1 \leq j \leq k \leq n}{C \vdash \{m_1 : \sigma; \ldots ; m_k : \sigma_k; \ldots ; m_n : \sigma_n\} \leq \{m_1 : \tau_1; \ldots ; m_k : \tau_k\}}$$

SObj

$$\frac{C \cup s \leq t \vdash \tau[s/MyType] \leq \tau'[t/MyType]}{C \vdash ObjectType(MyType)\tau \leq ObjectType(MyType)\tau'}$$

IRefl

$$C \vdash ObjectType(MyType)\tau \leq_{meth} ObjectType(MyType)\tau$$

IVar

$$C \cup \{t \leq_{meth} \tau\} \vdash t \leq_{meth} \tau$$

ITrans

$$\frac{C \vdash \gamma \leq_{meth} ObjectType(MyType)\tau, \quad C \vdash \tau \leq \tau'}{C \vdash \gamma \leq_{meth} ObjectType(MyType)\tau'}$$

B Minimum Typing Rules for TOOPLE

Definition B.1 *The following are used in the minimum typing rules and axioms below.*

1. *(Fzrom [CG92]) We write $C \vdash t \ll \tau$, if t is a type variable, and $C \vdash t \leq \tau$ is provable using only (SVar) and (STrans).*
2. *The type $lub(\tau, \tau')$ is the least upper bound of τ and τ' according to the subtyping ordering. The least upper bound of two types exists as long as they have any upper bound. See [BCD+93] for details.*

MVar

$$C, E \vdash_M x : \tau, \text{ if } E(x) = \tau$$

Mcond

$$\frac{C, E \vdash_M B : \rho, \quad C \vdash \rho \leq Bool, \quad C, E \vdash_M M : \tau', \quad C, E \vdash_M N : \tau''}{C, E \vdash_M \mathit{if}\ B\ \mathit{then}\ M\ \mathit{else}\ N : lub(\tau', \tau'')}$$

MAbs

$$\frac{C, E \cup \{v : \sigma\} \vdash_M M : \tau}{C, E \vdash_M fun(v : \sigma)\ M : \sigma \rightarrow \tau}$$

MAppl

$$\frac{C, E \vdash_M M : \sigma \to \tau, \quad C, E \vdash_M N : \sigma', \quad C \vdash \sigma' \leq \sigma}{C, E \vdash_M M\ N : \tau}$$

MAppl'

$$\frac{C, E \vdash_M M : t, \quad C, E \vdash_M N : \sigma', \quad C \vdash t \ll \sigma \to \tau, \quad C \vdash \sigma' \leq \sigma}{C, E \vdash_M M\ N : \tau}$$

MEq?

$$\frac{C, E \vdash_M M : \tau, \quad C, E \vdash_M N : \tau', \quad C \vdash \tau \leq \mathit{Num}, \quad C \vdash \tau' \leq \mathit{Num}}{C, E \vdash_M M = N : \mathit{Bool}}$$

MRec

$$\frac{C, E \vdash_M e_i : \tau_i, \forall i : 1 \leq i \leq n}{C, E \vdash_M \{m_1 = e_1, \ldots, m_n = e_n\} : \{m_1 : \tau_1; \ldots; m_n : \tau_n\}}$$

MProj

$$\frac{C, E \vdash_M e : \{m_1 : \tau_1; \ldots; m_n : \tau_n\}}{C, E \vdash_M e.m_i : \tau_i, \forall i : 1 \leq i \leq n}$$

MProj'

$$\frac{C, E \vdash_M e : t, \quad C \vdash t \ll \{m_1 : \tau_1; \ldots; m_n : \tau_n\}}{C, E \vdash_M e.m_i : \tau_i, \forall i : 1 \leq i \leq n}$$

MClass

$$\frac{C \cup \{\mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau\}, E \cup \{\mathit{self} : \mathit{MyType}\} \vdash_M e : \tau', \quad C \cup \{\mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau\} \vdash \tau' \leq \tau}{C, E \vdash_M \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e : \mathit{ClassType}(\mathit{MyType})\tau}$$

MObj

$$\frac{C \cup \{\mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau\}, E \cup \{\mathit{self} : \mathit{MyType}\}\} \vdash_M e : \tau', \quad C \cup \{\mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau\} \vdash \tau' \leq \tau}{C, E \vdash_M \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e : \mathit{ObjectType}(\mathit{MyType})\tau}$$

MNew

$$\frac{C, E \vdash_M c : ClassType(MyType)\tau}{C, E \vdash_M new\ c : ObjectType(MyType)\tau}$$

MMsg

$$\frac{C, E \vdash_M o : ObjectType(MyType)\{m_1 : \tau_1; \ldots; m_n : \tau_n\}}{C, E \vdash_M o \Leftarrow m_i : \tau_i[ObjectType(MyType)\{m_1 : \tau_1; \ldots; m_n : \tau_n\}/MyType]}$$

MMsg'

$$\frac{\begin{array}{c} C, E \vdash_M o : t, \\ (t \leq_{meth} ObjectType(MyType)\{m_1 : \tau_1; \ldots; m_n : \tau_n\}) \in C \end{array}}{C, E \vdash_M o \Leftarrow m_i : \tau_i[t/MyType]}$$

MUpdate

$$\frac{\begin{array}{c} C, E \vdash_M c : ClassType(MyType)\tau, \\ C \vdash \tau_1' \leq \tau_1, \\ C \cup \{MyType \leq_{meth} ObjectType(MyType)\tau'\}, E \cup \{self : MyType, super : \tau\} \vdash_M e_1' : \tau_1'', \\ C \cup \{MyType \leq_{meth} ObjectType(MyType)\tau'\} \vdash \tau_1'' \leq \tau_1' \end{array}}{\begin{array}{r} C, E \vdash_M update\ c\ by\ (self : MyType \leq_{meth} ObjectType(MyType)\tau', super)\{m_1 = e_1'\} : \\ ClassType(MyType)\tau' \end{array}}$$

where $\tau = \{m_1 : \tau_1; \ldots; m_n : \tau_n\}$ and $\tau' = \{m_1 : \tau_1'; \ldots; m_n : \tau_n\}$.

MExtend

$$\frac{\begin{array}{c} C, E \vdash_M c : ClassType(MyType)\tau, \\ C \cup \{MyType \leq_{meth} ObjectType(MyType)\tau'\}, \\ E \cup \{self : MyType, super : \tau\} \vdash_M e_{n+1} : \tau_{n+1}', \\ C \cup \{MyType \leq_{meth} ObjectType(MyType)\tau'\} \vdash \tau_{n+1}' \leq \tau_{n+1} \end{array}}{\begin{array}{r} C, E \vdash_M extend\ c\ with\ (self : MyType \leq_{meth} ObjectType(MyType)\tau', super) \\ \{m_{n+1} = e_{n+1}\} : ClassType(MyType)\tau' \end{array}}$$

where $\tau = \{m_1 : \tau_1; \ldots; m_n : \tau_n\}$ and $\tau' = \{m_1 : \tau_1; \ldots; m_{n+1} : \tau_{n+1}\}$.

C Natural Semantics for TOOPLE

RAbs

$$C, E \vdash fun(x : \sigma)\ M : \sigma \rightarrow \tau \downarrow fun(x : \sigma)\ M$$

RConst

$$\begin{array}{c} C, E \vdash true : Bool \downarrow true, \\ C, E \vdash false : Bool \downarrow false, \\ C, E \vdash n : Num \downarrow n, \text{if } n \text{ is a constant of type } Num \\ C, E \vdash self : MyType \downarrow self \end{array}$$

RRecord

$$C, E \vdash \{r_1 = e_1, \ldots, r_n = e_n\} : \{r_1 : \tau_1; \ldots; r_n : \tau_n\} \downarrow \{r_1 = e_1, \ldots, r_n = e_n\}$$

RClass

$$\begin{array}{c} C, E \vdash \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e : \mathit{ClassType}(\mathit{MyType})\tau \downarrow \\ \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e \end{array}$$

RObj

$$\begin{array}{c} C, E \vdash \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e : \mathit{ObjectType}(\mathit{MyType})\tau \downarrow \\ \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e \end{array}$$

REq

$$\frac{\begin{array}{c} C, E \vdash e_1 : \mathit{Num} \downarrow v, \\ C, E \vdash e_2 : \mathit{Num} \downarrow v \end{array}}{C, E \vdash e_1 = e_2 : \mathit{Bool} \downarrow \mathit{true}}$$

RNeq

$$\frac{\begin{array}{c} C, E \vdash e_1 : \mathit{Num} \downarrow v_1, \\ C, E \vdash e_2 : \mathit{Num} \downarrow v_2, \\ v_1 \not\equiv v_2 \end{array}}{C, E \vdash e_1 = e_2 : \mathit{Bool} \downarrow \mathit{false}}$$

RTrue

$$\frac{\begin{array}{c} C, E \vdash B : \mathit{Bool} \downarrow \mathit{true}, \\ C, E \vdash e_1 : \tau \downarrow v, \\ C, E \vdash e_2 : \tau \end{array}}{C, E \vdash \mathit{if}\ B\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : \tau \downarrow v}$$

RFalse

$$\frac{\begin{array}{c} C, E \vdash B : \mathit{Bool} \downarrow \mathit{false}, \\ C, E \vdash e_2 : \tau \downarrow v, \\ C, E \vdash e_1 : \tau \end{array}}{C, E \vdash \mathit{if}\ B\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : \tau \downarrow v}$$

RAppl

$$\frac{\begin{array}{c} C, E \vdash e : \sigma \rightarrow \tau \downarrow \mathit{fun}(x : \sigma)\ M, \\ C, E \vdash M[e'/x] : \tau \downarrow v \end{array}}{C, E \vdash e\ e' : \tau \downarrow v}$$

RProj

$$\frac{\begin{array}{c} C, E \vdash e : \tau \downarrow \{r_1 = e_1, \ldots, r_i = e_i, \ldots, r_n = e_n\}, \\ C, E \vdash e_i : \tau_i \downarrow v \end{array}}{C, E \vdash e.r_i : \tau_i \downarrow v}$$

RNew

$$\frac{C, E \vdash c : \mathit{ClassType}(\mathit{MyType})\tau \downarrow \quad \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e}{C, E \vdash \mathit{new}\ c : \mathit{ObjectType}(\mathit{MyType})\tau \downarrow \quad \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e}$$

RMsg

$$\frac{\begin{array}{c} C, E \vdash o : \gamma \downarrow \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \gamma')e, \\ C, E \vdash (e[\mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e/\mathit{self}, \gamma'/\mathit{MyType}] : \\ \{m_1 : \tau_1; \ldots; m_n : \tau_n\})[\gamma'/\mathit{MyType}] \downarrow \{m_1 = e_1, \ldots, m_n = e_n\} \\ C, E \vdash e_i : \tau_i[\gamma'/\mathit{MyType}] \downarrow v \end{array}}{C, E \vdash o \Leftarrow m_i : \tau_i[\gamma/\mathit{MyType}] \downarrow v}$$

where $1 \leq i \leq n$, and $\gamma' = \mathit{ObjectType}(\mathit{MyType})\{m_1 : \tau_1; \ldots; m_n : \tau_n\}$

RUpdate

$$\frac{C, E \vdash c : \mathit{ClassType}(\mathit{MyType})\tau \downarrow \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e}{\begin{array}{c} C, E \vdash \mathit{update}\ c\ \mathit{by}\ (\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau', \mathit{super})\{m_1 = e_1'\} : \\ \mathit{ClassType}(\mathit{MyType})\tau' \downarrow \\ \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau')\{m_1 = e_1'', m_2 = e.m_2, \ldots, m_n = e.m_n\} \end{array}}$$

where $e_1'' = e_1'[e/\mathit{super}], \tau = \{m_1 : \tau_1; \ldots; m_n : \tau_n\}$, and $\tau' = \{m_1 : \tau_1'; \ldots; m_n : \tau_n\}$.

RExtend

$$\frac{C, E \vdash c : \mathit{ClassType}(\mathit{MyType})\tau \downarrow \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau)e}{\begin{array}{c} C, E \vdash \mathit{update}\ c\ \mathit{by}\ (\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau', \mathit{super})\{m_{n+1} = e_{n+1}\} : \\ \mathit{ClassType}(\mathit{MyType})\tau' \downarrow \\ \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{meth} \mathit{ObjectType}(\mathit{MyType})\tau') \\ \{m_1 = e.m_1, \ldots, m_n = e.m_n, m_{n+1} = e_{n+1}'\} \end{array}}$$

where $e_{n+1}' = e_{n+1}[e/\mathit{super}], \tau = \{m_1 : \tau_1; \ldots; m_n : \tau_n\}$,
and $\tau' = \{m_1 : \tau_1; \ldots; m_{n+1} : \tau_{n+1}\}$.

D Denotational Semantics for TOOPL

ObjectType

$$[\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho = FIX(\lambda\xi.[\![\tau]\!]\rho[\xi \mathit{MyType}]$$

ClassType

$$[\![\mathit{ClassType}(\mathit{MyType})\tau]\!]\rho = \prod_{\xi \leq_A [\![\tau]\!]\rho[\xi/\mathit{MyType}]} (\xi \rightarrow [\![\tau]\!]\rho[\xi/\mathit{MyType}])$$

Var

$$[\![C, E \vdash x : \tau]\!]\rho = \rho(x)$$

Cond

$$[\![C, E \vdash \mathit{if}\ B\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : \tau]\!]\rho = \begin{cases} [\![C, E \vdash e_1 : \tau]\!]\rho, \text{if } [\![C, E \vdash B : \mathit{Bool}]\!]\rho = \mathit{true} \\ [\![C, E \vdash e_2 : \tau]\!]\rho, \text{if } [\![C, E \vdash B : \mathit{Bool}]\!]\rho = \mathit{false} \\ \bot, \text{otherwise} \end{cases}$$

Abs

$$[\![C, E \vdash \mathit{fun}(x : \sigma)\ M : \sigma \to \tau]\!]\rho = \lambda d \in \mathcal{A}^{\sigma}.[\![C, E \vdash M : \tau]\!]\rho[d/x]$$

Appl

$$[\![C, E \vdash e\ e' : \tau]\!]\rho = ([\![C, E \vdash e : \sigma \to \tau]\!]\rho)([\![C, E \vdash e' : \sigma]\!]\rho)$$

Eq?

$$[\![C, E \vdash e_1 = e_2 : \mathit{Bool}]\!]\rho = \begin{cases} \mathit{true}, \text{if } [\![C, E \vdash e_1 : \tau]\!]\rho = [\![C, E \vdash e_2 : \tau]\!]\rho \\ \mathit{false}, \text{otherwise} \end{cases}$$

Rec

$$[\![C, E \vdash \{r_1 = e_1, \ldots, r_n = e_n\} : \tau]\!]\rho = f,$$

where

$$\begin{aligned} \mathit{dom}(f) &= \{r_1, \ldots, r_n\} \\ \forall i : 1 \leq i \leq n, f(r_i) &= [\![C, E \vdash e_i : \tau_i]\!]\rho \end{aligned}$$

Proj

$$[\![C, E \vdash e.r_i : \tau_i]\!]\rho = ([\![C, E \vdash e : \tau]\!]\rho)(r_i)$$

Class

$$\begin{aligned} &[\![C, E \vdash \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{\mathit{meth}} \mathit{ObjectType}(\mathit{MyType})\tau)e : \mathit{ClassType}(\mathit{MyType})\tau]\!]\rho = \\ &\quad \lambda\xi \leq [\![\tau]\!]\rho[\xi/\mathit{MyType}].\lambda o \in \mathcal{A}^{\xi}.[\![C \cup \{\mathit{MyType} \leq_{\mathit{meth}} \mathit{ObjectType}(\mathit{MyType})\tau\}, \\ &\qquad E \cup \{\mathit{self} : \mathit{MyType}\} \vdash e : \tau]\!]\rho[\xi/\mathit{MyType}, o/\mathit{self}] \end{aligned}$$

New

$$\begin{aligned} &[\![C, E \vdash \mathit{new}\ c : \mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho = \\ &\quad \mathit{FIX}(([\![C, E \vdash c : \mathit{ClassType}(\mathit{MyType})\tau]\!]\rho)([\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho)) \end{aligned}$$

Obj

$$\begin{aligned} &[\![C, E \vdash \mathit{obj}(\mathit{self} : \mathit{MyType} \leq_{\mathit{meth}} \mathit{ObjectType}(\mathit{MyType})\tau)e : \mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho = \\ &\quad \mathit{FIX}(([\![C, E \vdash \mathit{class}(\mathit{self} : \mathit{MyType} \leq_{\mathit{meth}} \mathit{ObjectType}(\mathit{MyType})\tau)e : \\ &\qquad \mathit{ClassType}(\mathit{MyType})\tau]\!]\rho)([\![\mathit{ObjectType}(\mathit{MyType})\tau]\!]\rho)) \end{aligned}$$

Msg

$$\begin{aligned} &[\![C, E \vdash o \Leftarrow m_i : \tau_i[\gamma/\mathit{MyType}]]\!]\rho \\ &\quad = (\mathbf{convert}[[\![\{m_i : \tau_i\}]\!]\rho[[\![\gamma]\!]\rho/\mathit{MyType}]][[\![\gamma]\!]\rho][\![C, E \vdash o : \gamma]\!]\rho)\,(m_i) \end{aligned}$$

Update

$$[\![C, E \vdash \textit{update } c \textit{ by } (\textit{self} : \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau, \textit{super})\{m_1 = e_1'\} : \\ \textit{ClassType}(\textit{MyType})\tau']\!]\rho = \lambda\xi \leq [\![\tau']\!]\rho[\xi/\textit{MyType}].\lambda o \in \mathcal{A}^{\xi}.f,$$

where

$$\begin{aligned} \textit{dom}(f) &= \{m_1, \ldots, m_n\}, \\ f(m_1) &= [\![C \cup \{\textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau'\}, \\ &\qquad E \cup \{\textit{self} : \textit{MyType}\} \vdash e_1' : \tau_1']\!]\rho[\xi/\textit{MyType}, o/\textit{self}, s/\textit{super}], \\ f(m_j) &= s(m_j), \forall j : 2 \leq j \leq n, \\ s &= ([\![C, E \vdash c : \textit{ClassType}(\textit{MyType})\{m_1 : \tau_1; \ldots; m_n : tau_n\}]\!]\rho)(\xi)(o). \end{aligned}$$

Extend

$$[\![C, E \vdash \textit{extend } c \textit{ with } (\textit{self} : \textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau, \textit{super}) \\ \{m_{n+1} = e_{n+1}\} : \textit{ClassType}(\textit{MyType})\tau']\!]\rho = \lambda\xi \leq [\![\tau']\!]\rho[\xi/\textit{MyType}].\lambda o \in \mathcal{A}^{\xi}.f,$$

where

$$\begin{aligned} \textit{dom}(f) &= \{m_1, \ldots, m_{n+1}\}, \\ f(m_{n+1}) &= [\![C \cup \{\textit{MyType} \leq_{meth} \textit{ObjectType}(\textit{MyType})\tau'\}, \\ &\qquad E \cup \{\textit{self} : \textit{MyType}\} \vdash e_{n+1} : \tau_{n+1}]\!]\rho[\xi/\textit{MyType}, o/\textit{self}, s/\textit{super}], \\ f(m_j) &= s(m_j), \forall j : 1 \leq j \leq n, \\ s &= ([\![C, E \vdash c : \textit{ClassType}(\textit{MyType})\{m_1 : \tau_1; \ldots; m_n : tau_n\}]\!]\rho)(\xi)(o). \end{aligned}$$

On the Transformation Between Direct and Continuation Semantics *

Olivier Danvy and John Hatcliff

Aarhus University ** and Kansas State University ***

Abstract. Proving the congruence between a direct semantics and a continuation semantics is often surprisingly complicated considering that direct-style λ-terms can be transformed into continuation style automatically. However, transforming the representation of a direct-style semantics into continuation style usually does *not* yield the expected representation of a continuation-style semantics (*i.e.*, one written by hand).

The goal of our work is to automate the transformation between textual representations of direct semantics and of continuation semantics. Essentially, we identify properties of a direct-style representation (*e.g.*, totality), and we generalize the transformation into continuation style accordingly. As a result, we can produce the expected representation of a continuation semantics, automatically.

It is important to understand the transformation between representations of direct and of continuation semantics because it is these representations that get processed in any kind of semantics-based program manipulation (*e.g.*, compiling, compiler generation, and partial evaluation). A tool producing a variety of continuation-style representations is a valuable new one in a programming-language workbench.

* 9th Conference on Mathematical Foundations of Programming Semantics. New Orleans, Louisiana, April 1993.

** Department of Computer Science, Ny Munkegade, 8000 Aarhus C, Denmark. E-mail: danvy@daimi.aau.dk — This work was initiated at Kansas State University, continued at Carnegie Mellon University, and completed at Aarhus University. It was partly supported by NSF under grant CCR-9102625.

*** Department of Computing and Information Sciences, Manhattan, Kansas 66506, USA. E-mail: hatcliff@cis.ksu.edu

1 Introduction

Proving the congruence between a denotational-semantics specification in direct style and a denotational-semantics specification in continuation style is not trivial [26, 28, 31]. Yet,

- both direct-style and continuation-style specifications can be represented as typed λ-terms: semantic domains are represented with types, and valuation functions with λ-terms [22, 28];
- typed λ-terms can be transformed into continuation style ***automatically*** using Plotkin's continuation-passing-style (CPS) transformation [9, 15, 24].

We have transformed the representation of several direct-style specifications into continuation style. Since the meta-language of denotational semantics obeys normal order [28], we have used the call-by-name CPS transformation. The result is ***not*** the expected representation of a continuation-style semantics (*i.e.*, one written by hand).

1.1 An example

It is sufficient to look at types to see where a mismatch occurs.

$$\mathcal{C}_d[\![\cdot]\!] : Env_d \to Com_d \quad \text{where} \quad Com_d = Store \to Store$$

$$\mathcal{C}_c[\![\cdot]\!] : Env_c \to Com_c \quad \text{where} \quad Com_c = Store \to (Store \to Ans) \to Ans$$

Fig. 1. Types of valuation functions for a simple imperative language

Figure 1 gives the types of two valuation functions for a simple imperative language. $\mathcal{C}_d[\![\cdot]\!]$ is a direct-style valuation function and $\mathcal{C}_c[\![\cdot]\!]$ is a continuation-style valuation function.

Figure 2 displays Plotkin's call-by-name CPS transformation $\mathcal{C}_{\mathrm{n}}$ for typed terms [9, 24]. ι represents a base type.

Transforming the types of the direct-style valuation function $\mathcal{C}_d[\![\cdot]\!]$ does not yield the types of the continuation-style valuation function $\mathcal{C}_c[\![\cdot]\!]$. For example, the transformation of the function space $Env_d \to Com_d$ yields

$$((\mathcal{C}_{\mathrm{n}}\langle Env_d\rangle \to Ans) \to Ans) \to (\mathcal{C}_{\mathrm{n}}\langle Com_d\rangle \to Ans) \to Ans$$

which does not match the type of the corresponding function space

$$Env_c \to Com_c$$

in $\mathcal{C}_c[\![\cdot]\!]$. Essentially, $\mathcal{C}_{\mathrm{n}}$ introduces too many continuations.

$$
\begin{aligned}
\mathcal{C}_n\llbracket i\rrbracket &= i\\
\mathcal{C}_n\llbracket \lambda i : t.\epsilon\rrbracket &= \lambda\kappa.\kappa\,(\lambda i : \mathcal{C}_n\llbracket t\rrbracket.\mathcal{C}_n\llbracket \epsilon\rrbracket)\\
\mathcal{C}_n\llbracket \epsilon_0\,\epsilon_1\rrbracket &= \lambda\kappa.\mathcal{C}_n\llbracket \epsilon_0\rrbracket\,(\lambda v_0.(v_0\,\mathcal{C}_n\llbracket \epsilon_1\rrbracket)\,\kappa)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_n\langle \iota\rangle &= \iota\\
\mathcal{C}_n\langle t_0 \rightarrow t_1\rangle &= \mathcal{C}_n\llbracket t_0\rrbracket \rightarrow \mathcal{C}_n\llbracket t_1\rrbracket\\
\mathcal{C}_n\llbracket t\rrbracket &= (\mathcal{C}_n\langle t\rangle \rightarrow Ans) \rightarrow Ans
\end{aligned}
$$

Fig. 2. Transformation of call-by-name λ-terms into continuation style

This mismatch is significant because it shows that a continuation semantics is not just a direct semantics with continuations. For another example, in a continuation semantics, environments (represented as functions) usually are expressed in direct style, *i.e.*, they are not passed any continuation [28, 31].

1.2 A choice

At this point we have a choice:

- We could establish the relationship between the result of CPS-transforming [the representation of] a direct-style semantics and [the representation of] a continuation-style semantics that one would write by hand, and maybe map one into the other.
- We could devise a new CPS transformation that would transform [the representation of] a direct-style semantics into [the representation of] a realistic continuation-style semantics. By a "realistic" continuation-style semantics, we mean "one that a professional denotational-semanticist would write".

We choose the latter option.

1.3 On the transformation between direct and continuation semantics

The goal of our work is to automate the transformation between textual representations of direct semantics and of continuation semantics. Essentially, we identify properties of a direct-style representation (*e.g.*, totality), and we generalize the call-by-name CPS transformation accordingly. As a result, we can produce the expected representation of a realistic continuation semantics, automatically.

It is important to understand the transformation between representations of direct and of continuation semantics for at least three reasons.

1. It is these representations that get processed in any kind of semantics-based program manipulation (*e.g.*, compiling, compiler generation, and partial evaluation).

2. The properties of [the representation of] the direct-style semantics should give precious insights to establishing the congruence relation between the direct semantics and the continuation semantics.
3. The properties of the transformation should give guidelines for proving the congruence between the direct semantics and the continuation semantics.

1.4 Issues

The tools used in this paper are interesting in their own right.

1. The generalized call-by-name CPS transformation is based on a system of annotations capturing reduction properties such as partiality and totality. Using these annotations, we extend Reynolds's classification of trivial and serious *λ-terms*[4] to serious and trivial *functions*. The extended classification gives a finer scheme for describing termination properties of terms — unlike Reynolds's original scheme, it allows us to state that *e.g.*, some applications are actually trivial.
2. The annotations are obtained by an automatic control-flow analysis that extends Mycroft's $\flat$ termination analysis to higher-order programs [20, 21]. This tool has applications in other areas such as compiling and partial evaluation.
3. Retaining Reynolds's method of introducing continuations in serious terms yields a transformation that introduces continuations only when necessary to achieve evaluation-order independence. Thus, this new transformation generalizes the call-by-name CPS transformation (should all functions be serious) and the identity transformation (should all functions be trivial). In an earlier work, we reported a CPS transformation after strictness analysis that generalizes the call-by-value CPS transformation (should all constructs be strict) and the call-by-name CPS transformation (should all constructs be non-strict) [5]. Tools producing a variety of continuation-style representations are valuable new ones in a programming-language workbench.

1.5 Organization

The rest of this paper is organized as follows. Section 2 presents an example language and two semantic definitions, one in direct style and one in continuation style. Section 3 describes how these semantic definitions can be represented as typed λ-terms. In Section 4, we generalize Reynolds's notion of trivial and serious terms. In Section 5, we extend the transformation into continuation style to handle terms with annotations describing trivial and serious properties. In Section 6, we examine the properties of the representation of the direct-style semantics of Section 2 and we annotate this representation. In Section 7, we

[4] Reducing a trivial λ-term always terminates whereas reducing a serious λ-term may not terminate [25]. Reynolds's notion of trivial λ-term coincides with Plotkin's notion of value [24].

$z \in$ Program	$l \in$ Location
$c \in$ Command	$m \in$ Ident[num]
$e \in$ Expression	$p \in$ Ident[proc]
$n \in$ Numeral	$f \in$ Ident[fun]

z ::= proc $p\,(m) = c$ in z | fun $f\,(m) = e$ in z | c.
c ::= skip | c_1 ; c_2 | l:=e | if e then c_1 else c_2 | while e do c | call $p\,(e)$
e ::= n | m | succ e | pred e | deref l | apply $f\,(e)$

Fig. 3. Abstract syntax of the simple imperative language

transform this annotated representation into continuation style and we obtain the expected representation of a continuation semantics. Finally, Section 8 concludes and puts this work into perspective.

2 Example Denotational Definitions

Figure 3 presents the abstract syntax of a simple imperative language with global and non-recursive first-order procedures. Figures 4 and 5 give a direct semantics and a continuation semantics for the simple imperative language. The functionality of the semantic algebras for stores, environments, and natural numbers are the usual ones and the specifications are omitted.

Proposition 1. *The semantics of Figures 4 and 5 define the same language, that is, they are congruent [31, page 340].* □

3 Representing Denotational Definitions as Typed λ-terms

Denotational definitions are usually implemented by *treating the semantic notation as a "machine language"* [28, Section 10.1]. A common notation of denotational semantics is the λ-calculus. Thus, domains are mapped into types and domain constructors into type constructors, valuation functions are mapped into λ-expressions, and semantic-algebra operations into δ-rules. Figure 6 presents the syntax of an extended λ-calculus used to represent denotational definitions as typed terms. The typing rules are the usual ones and are omitted. When e has type t under type assumptions π we write $\pi \vdash e : t$. Each element of the set of type assumptions π is of the form $i : t$. To simplify substitution, and without loss of generality, we assume that all identifiers are unique.

Let us summarize how the example denotational definitions of Figure 4 and 5 are represented by the typed terms of Figure 6.

Valuation Functions:		*Semantic Domains:*
$\mathcal{Z}[\![\text{Program}]\!] : Env \rightarrow Com$		$Com = Store \rightarrow Store_\perp$
$\mathcal{C}[\![\text{Command}]\!] : Env \rightarrow Com$		$Exp = Store \rightarrow Nat$
$\mathcal{E}[\![\text{Expression}]\!] : Env \rightarrow Exp$		$Proc = Nat \rightarrow Com$
$\mathcal{N}[\![\text{Numeral}]\!] : Nat$		$Fun = Nat \rightarrow Exp$
$\mathcal{L}[\![\text{Location}]\!] : Loc$		

Programs:

$$\mathcal{Z}[\![\text{proc}\; p\; (m) = c \;\text{in}\; z]\!] = \lambda\rho.\lambda\sigma.\mathcal{Z}[\![z]\!](\text{ext}\;\rho\; p\; (\lambda i.\mathcal{C}[\![c]\!](\text{ext}\;\rho\; m\; i)))\;\sigma$$
$$\mathcal{Z}[\![\text{fun}\; f\; (m) = e \;\text{in}\; p]\!] = \lambda\rho.\lambda\sigma.\mathcal{Z}[\![z]\!](\text{ext}\;\rho\; f\; (\lambda i.\mathcal{E}[\![e]\!](\text{ext}\;\rho\; m\; i)))\;\sigma$$
$$\mathcal{Z}[\![c.]\!] = \lambda\rho.\lambda\sigma.\mathcal{C}[\![c]\!]\,\rho\,\sigma$$

Commands:

$$\mathcal{C}[\![\text{skip}]\!] = \lambda\rho.\lambda\sigma.\sigma$$
$$\mathcal{C}[\![c_1\; ;\; c_2]\!] = \lambda\rho.\lambda\sigma.\mathit{let}\;\sigma' = \mathcal{C}[\![c_1]\!]\,\rho\,\sigma\;\mathit{in}\;\mathcal{C}[\![c_2]\!]\,\rho\,\sigma'$$
$$\mathcal{C}[\![l := e]\!] = \lambda\rho.\lambda\sigma.\text{upd}\;\sigma\;\mathcal{L}[\![l]\!]\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma)$$
$$\mathcal{C}[\![\text{if}\; e\;\text{then}\; c_1\;\text{else}\; c_2]\!] = \lambda\rho.\lambda\sigma.\mathit{if}\;\text{iszero?}\;(\mathcal{E}[\![e]\!]\,\rho\,\sigma)\;\mathit{then}\;(\mathcal{C}[\![c_1]\!]\,\rho\,\sigma)\;\mathit{else}\;(\mathcal{C}[\![c_2]\!]\,\rho\,\sigma)$$
$$\begin{aligned}\mathcal{C}[\![\text{while}\; e\;\text{do}\; c]\!] = \lambda\rho.\lambda\sigma.\;\mathit{letrec}\; w = \lambda\sigma.\;\mathit{if}\;\; & \text{iszero?}\;(\mathcal{E}[\![e]\!]\,\rho\,\sigma)\\ & \mathit{then\;let}\;\sigma' = \mathcal{C}[\![c]\!]\,\rho\,\sigma\;\mathit{in}\; w\,\sigma'\\ & \mathit{else}\;\;\sigma\\ \mathit{in}\; w\,\sigma\qquad &\end{aligned}$$
$$\mathcal{C}[\![\text{call}\; p\; (e)]\!] = \lambda\rho.\lambda\sigma.(\text{lookup}\;\rho\; p)\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma)\;\sigma$$

Expressions:

$$\mathcal{E}[\![n]\!] = \lambda\rho.\lambda\sigma.\mathcal{N}[\![n]\!]$$
$$\mathcal{E}[\![m]\!] = \lambda\rho.\lambda\sigma.\text{lookup}\;\rho\; m$$
$$\mathcal{E}[\![\text{succ}\; e]\!] = \lambda\rho.\lambda\sigma.\text{succ}\;(\mathcal{E}[\![e]\!]\,\rho\,\sigma)$$
$$\mathcal{E}[\![\text{pred}\; e]\!] = \lambda\rho.\lambda\sigma.\text{pred}\;(\mathcal{E}[\![e]\!]\,\rho\,\sigma)$$
$$\mathcal{E}[\![\text{deref}\; l]\!] = \lambda\rho.\lambda\sigma.\text{fetch}\;\sigma\;\mathcal{L}[\![l]\!]$$
$$\mathcal{E}[\![\text{apply}\; f\; (e)]\!] = \lambda\rho.\lambda\sigma.(\text{lookup}\;\rho\; f)\,(\mathcal{E}[\![e]\!]\,\rho\,\sigma)\;\sigma$$

Fig. 4. Direct semantics of the simple imperative language

Valuation Functions:

$$\begin{aligned}
\mathcal{Z}\llbracket\text{Program}\rrbracket &: \mathit{Env} \rightarrow \mathit{Com}\\
\mathcal{C}\llbracket\text{Command}\rrbracket &: \mathit{Env} \rightarrow \mathit{Com}\\
\mathcal{E}\llbracket\text{Expression}\rrbracket &: \mathit{Env} \rightarrow \mathit{Exp}\\
\mathcal{N}\llbracket\text{Numeral}\rrbracket &: \mathit{Nat}\\
\mathcal{L}\llbracket\text{Location}\rrbracket &: \mathit{Loc}
\end{aligned}$$

Semantic Domains:

$$\begin{aligned}
\mathit{Com} &= \mathit{Store} \rightarrow (\mathit{Store} \rightarrow \mathit{Ans}) \rightarrow \mathit{Ans}\\
\mathit{Exp} &= \mathit{Store} \rightarrow \mathit{Nat}\\
\mathit{Proc} &= \mathit{Nat} \rightarrow \mathit{Com}\\
\mathit{Fun} &= \mathit{Nat} \rightarrow \mathit{Exp}
\end{aligned}$$

Programs:

$$\begin{aligned}
\mathcal{Z}\llbracket\text{proc}\, p\,(m) = c \text{ in } z\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.\mathcal{Z}\llbracket z\rrbracket(\text{ext}\,\rho\, p\,(\lambda i.\mathcal{C}\llbracket c\rrbracket(\text{ext}\,\rho\, m\, i)))\,\sigma\,\kappa\\
\mathcal{Z}\llbracket\text{fun}\, f\,(m) = e \text{ in } z\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.\mathcal{Z}\llbracket z\rrbracket(\text{ext}\,\rho\, f\,(\lambda i.\mathcal{E}\llbracket e\rrbracket(\text{ext}\,\rho\, m\, i)))\,\sigma\,\kappa\\
\mathcal{Z}\llbracket c.\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.\mathcal{C}\llbracket c\rrbracket\,\rho\,\sigma\,\kappa
\end{aligned}$$

Commands:

$$\begin{aligned}
\mathcal{C}\llbracket\text{skip}\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.\kappa\,\sigma\\
\mathcal{C}\llbracket c_1\,;\, c_2\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.\mathcal{C}\llbracket c_1\rrbracket\,\rho\,\sigma\,(\lambda\sigma'.\mathcal{C}\llbracket c_2\rrbracket\,\rho\,\sigma'\,\kappa)\\
\mathcal{C}\llbracket l := e\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.\kappa\,(\text{upd}\,\sigma\,\mathcal{L}\llbracket l\rrbracket\,(\mathcal{E}\llbracket e\rrbracket\,\rho\,\sigma))\\
\mathcal{C}\llbracket\text{if } e \text{ then } c_1 \text{ else } c_2\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.\, \mathit{if}\ \text{iszero?}\ (\mathcal{E}\llbracket e\rrbracket\,\rho\,\sigma)\ \mathit{then}\ (\mathcal{C}\llbracket c_1\rrbracket\,\rho\,\sigma\,\kappa)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{else}\ (\mathcal{C}\llbracket c_2\rrbracket\,\rho\,\sigma\,\kappa)\\
\mathcal{C}\llbracket\text{while } e \text{ do } c\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.\ \mathit{letrec}\ w = \lambda\sigma.\lambda\kappa'.\ \mathit{if}\ \ \text{iszero?}\ (\mathcal{E}\llbracket e\rrbracket\,\rho\,\sigma)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{then}\ \mathcal{C}\llbracket c\rrbracket\,\rho\,\sigma\,(\lambda\sigma'.w\,\sigma'\,\kappa')\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{else}\ \ \kappa'\,\sigma\\
&\qquad\qquad \mathit{in}\ w\,\sigma\,\kappa\\
\mathcal{C}\llbracket\text{call}\, p\,(e)\rrbracket &= \lambda\rho.\lambda\sigma.\lambda\kappa.(\text{lookup}\,\rho\, p)\,(\mathcal{E}\llbracket e\rrbracket\,\rho\,\sigma)\,\sigma\,\kappa
\end{aligned}$$

Expressions:

$$\begin{aligned}
\mathcal{E}\llbracket n\rrbracket &= \lambda\rho.\lambda\sigma.\mathcal{N}\llbracket n\rrbracket\\
\mathcal{E}\llbracket m\rrbracket &= \lambda\rho.\lambda\sigma.\text{lookup}\,\rho\, m\\
\mathcal{E}\llbracket\text{succ}\, e\rrbracket &= \lambda\rho.\lambda\sigma.\text{succ}\,(\mathcal{E}\llbracket e\rrbracket\,\rho\,\sigma)\\
\mathcal{E}\llbracket\text{pred}\, e\rrbracket &= \lambda\rho.\lambda\sigma.\text{pred}\,(\mathcal{E}\llbracket e\rrbracket\,\rho\,\sigma)\\
\mathcal{E}\llbracket\text{deref}\, l\rrbracket &= \lambda\rho.\lambda\sigma.\text{fetch}\,\sigma\,\mathcal{L}\llbracket l\rrbracket\\
\mathcal{E}\llbracket\text{apply}\, f\,(e)\rrbracket &= \lambda\rho.\lambda\sigma.(\text{lookup}\,\rho\, f)\,(\mathcal{E}\llbracket e\rrbracket\,\rho\,\sigma)\,\sigma
\end{aligned}$$

Fig. 5. Continuation semantics of the simple imperative language

$\epsilon \in \lambda\text{-}exp \qquad a \in \lambda\text{-}abs \qquad r \in \delta\text{-}rule$

$\epsilon ::= i \mid a \mid \epsilon_0\,\epsilon_1 \mid \mathit{if}\ \epsilon_1\ \mathit{then}\ \epsilon_2\ \mathit{else}\ \epsilon_3 \mid \mathit{let}\ i = \epsilon_0\ \mathit{in}\ \epsilon_1 \mid \mathit{letrec}\ i = a\ \mathit{in}\ \epsilon \mid r$
$a ::= \lambda i : t.\epsilon$
$r ::= \mathsf{op}_0 \mid \mathsf{op}_1\,\epsilon \mid \mathsf{op}_2\,\epsilon_1\,\epsilon_2 \mid \dots$

$t \in \mathit{Type}$
$t ::= \iota \mid t_0 \rightarrow t_1$

Fig. 6. Abstract syntax for an extended typed λ-calculus

The primitive domains *Store*, *Env*, and *Ide* form the base types and the semantic-algebra operations such as **upd** and **fetch** become δ-rules.

The valuation functions become typed λ-terms. A key point in this step is that operational notions such as non-termination and recursion (represented explicitly in the denotational semantics by the special element $\perp$ and least fixed-point operations over *cpo*'s) must be captured implicitly in the reduction properties of the λ-terms.

Following Schmidt [28], *let* expressions used in the direct semantics of Figure 4 include a strictness check over some lifted domain $A_\perp$. They are defined as follows.

$$\mathit{let}\ i = e_0\ \mathit{in}\ e_1 \;=\; \begin{cases} \perp & \mathit{if}\ e_0 = \perp \\ (\lambda i.e_1)\,e_0 & \mathit{otherwise} \end{cases}$$

So each *let* expression is represented with an eager binding construct. The operational behavior of the binding construct (*i.e.*, call-by-value) captures the appropriate termination properties [24].

Similarly, *letrec* is defined by the usual desugaring into the fixed-point operator. So each *letrec* expression is represented with the usual recursive binding construct. Its operational behavior approximates the computation of the least fixed-point of a function.

4 Analyzing the Representation of a Direct-Style Definition

As pointed out in Section 1.1, transforming the λ-representation of a direct semantics into continuation style using a call-by-name transformation does *not* yield the λ-representation of a realistic continuation semantics. Essentially, the transformation introduces more continuations than are needed.[5] In this section,

[5] For example, in Figure 5, $\mathcal{E}$ is expressed in direct style even though it is part of a continuation semantics. We aim to clarify why $\mathcal{E}$ does not need any continuation, and to establish conditions that allow one to transform the text of Figure 4 into the text of Figure 5, automatically.

we go back to the source [25] and investigate where continuations are really necessary.

4.1 Reynolds's notion of trivial and serious terms

Originally, Reynolds distinguished between "trivial" terms (whose evaluation never diverges) and "serious" terms (whose evaluation might diverge) [25]. Trivial terms correspond to Plotkin's notion of "value" [24]. Since introducing continuations aims at obtaining evaluation-order independence, only serious terms need to be transformed into continuation style. As an approximation, Reynolds decided that all applications are serious terms and thus they all need a continuation — forcing each function to be passed a continuation.

Considering the particular case of denotational semantics, this approximation often is too coarse. For example, valuation functions are usually curried. Most of the time, the result of applying a valuation function to an abstract-syntax tree is a λ-abstraction. In fact, this is the case for $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{E}$ in Figure 4. Since a λ-abstraction is a trivial term, applying a valuation function does not yield a serious term. Thus it is too conservative to approximate all applications as serious terms.[6]

4.2 Trivial and serious functions

In a denotational-semantics specification, a function is defined textually as a λ-abstraction.

- If the body of this λ-abstraction is trivial, the function is obviously *total*. Since evaluating the body does not require a continuation, the function does not need a continuation either.
- Conversely, if the body of a λ-abstraction is serious, the corresponding function may be *partial*. Since evaluating the body requires a continuation, the function needs to be passed this continuation.

We refer to such λ-abstractions as "trivial functions" and "serious functions", respectively.

Let us now turn to the arguments of these functions. Denotational specifications are customarily higher-order, so it is not obvious which expression occurs as the argument of which λ-abstraction. However following Reynolds again [25], we can enumerate the λ-abstractions that may occur in each higher-order application. This enumeration is achieved by control-flow analysis (a.k.a. closure analysis) [29, 30].

[6] This is probably why Reynolds's definitional interpreters are uncurried [25].

4.3 Call-by-value and call-by-name functions

A λ-abstraction can be applied to a trivial argument or to a serious one. Again, trivial arguments do not need to be computed with a continuation. Conversely, serious arguments need to be computed with a continuation. We approximate this situation by stating that if a λ-abstraction is always applied to trivial arguments, we can pass the arguments as they are, and that if a function may be applied to a serious argument, then all arguments are computed with a continuation. By analogy with the fact that evaluating a trivial expression must yield a value, we refer to the former λ-abstractions as "call-by-value functions" and to the latter as "call-by-name functions". So let us consider the four cases of λ-abstractions:

1. trivial call-by-value functions (*i.e.*, λ-abstractions whose bodies are trivial and that are applied to trivial arguments);
2. trivial call-by-name functions (*i.e.*, λ-abstractions whose bodies are trivial and that are applied to serious arguments);
3. serious call-by-value functions (*i.e.*, λ-abstractions whose bodies are serious and that are applied to trivial arguments);
4. serious call-by-name functions (*i.e.*, λ-abstraction whose bodies are serious and that are applied to serious arguments).

Correspondingly, a variable declared in a call-by-value (resp. call-by-name) function is a trivial (resp. serious) expression.

In the following section, we describe how to annotate λ-abstractions and applications to account for their triviality and their seriousness, and for their mode of parameter passing.

5 Annotating the Representation of a Direct-Style Definition

Figure 7 presents the syntax of the annotated λ-calculus. Essentially, we introduce the explicit infix notation "@" in applications, and we tag the constructs and types that depart from standard call-by-name.[7]

Constructs and types associated with trivial functions are annotated with "t". Constructs and types associated with call-by-value are annotated with "v". Constructs and types associated with both are annotated with "tv". For example, $\lambda_{tv} x.e$ denotes a call-by-value trivial λ-abstraction. y is a variable declared in a call-by-name λ-abstraction. z_v is a variable declared in a call-by-value λ-abstraction. $e_0 \, @ \, e_1$ denotes the application of a call-by-name serious function to an argument. $e'_0 \, @_t \, e'_1$ denotes the application of a call-by-name trivial function to an argument.

We also use the annotations *trivial* and *serious* to tag trivial and serious expressions. The annotation tags form a partially ordered set ($AnnTag$, $\sqsubseteq$) where

[7] This follows the spirit of the diacritical convention: only the terms whose meaning is farthest to the original meaning (*i.e.*, call-by-name) are annotated [16, 31].

$$\epsilon \in \textit{Ann-}\lambda\textit{-exp} \qquad a \in \textit{Ann-}\lambda\textit{-abs} \qquad r \in \delta\textit{-rule}$$

$$\begin{aligned}
\epsilon &::= i_v \mid i \mid a \mid \epsilon_0 \,@_w\, \epsilon_1 \mid \epsilon_0 \,@_v\, \epsilon_1 \mid \epsilon_0 \,@_t\, \epsilon_1 \mid \epsilon_0 \,@\, \epsilon_1 \\
&\quad \mid \textit{if } \epsilon_1 \textit{ then } \epsilon_2 \textit{ else } \epsilon_3 \mid \textit{let } i = \epsilon_0 \textit{ in } \epsilon_1 \mid \textit{letrec } i = a \textit{ in } \epsilon \mid r \\
a &::= \lambda_w i : \tau.\epsilon \mid \lambda_v i : \tau.\epsilon \mid \lambda_t i : \tau.\epsilon \mid \lambda i : \tau.\epsilon \\
r &::= \mathsf{op}_0 \mid \mathsf{op}_1\, \epsilon \mid \mathsf{op}_2\, \epsilon_1\, \epsilon_2 \mid \ldots
\end{aligned}$$

$$\begin{aligned}
&\tau \in \textit{AnnType} \\
&\tau ::= \iota \mid \tau_0 \rightarrow_w \tau_1 \mid \tau_0 \rightarrow_v \tau_1 \mid \tau_0 \rightarrow_t \tau_1 \mid \tau_0 \rightarrow \tau_1
\end{aligned}$$

$$\begin{aligned}
&\alpha \in \textit{AnnTag} \\
&\alpha ::= \textit{trivial} \mid \textit{serious}
\end{aligned}$$

Fig. 7. Abstract syntax for the annotated typed λ-calculus

trivial $\sqsubseteq$ *serious*. They will contribute to characterizing reduction properties of individual terms.

Figure 8 presents type-annotation rules for the annotated λ-calculus. Each term is associated with a pair (τ, α). The first component $\tau \in \textit{AnnType}$ is an annotated type. The second component $\alpha \in \textit{AnnTag}$ indicates whether the term is trivial or serious. Γ is a set of type assumptions where each element is of the form $i : \tau$. For simplicity, we assume that all identifier names are unique, and that the algebraic operators cannot diverge.

The other binding constructs also warrant explanation. In the *let* construct, the actual parameter e_1 may be either trivial or serious. However, due to the eager evaluation of e_1, i always binds to a value and thus is annotated as trivial. Of course, binding may not occur at all due to the diverging evaluation of a serious e_1. This is captured by the fact that a serious e_1 causes the entire construct to be classified as serious. In the *letrec* construct, the declared identifier f always binds to a λ-abstraction and is thus annotated as trivial.

Note that there is redundancy in the given annotation scheme. In particular, annotation pairs (τ, α) are sufficient for our purposes.[8] Annotations on terms have been included to simplify the presentation of the transformation into continuation style in Section 5.2.

5.1 Correct assignment of annotations

To formalize the correctness of the annotation rules, let us introduce the following notation. $\Downarrow_n$ and $\Downarrow_v$ respectively denote the relations defined by a call-by-name

[8] The annotation scheme can also be phrased more elegantly in terms of Moggi's computational metalanguage [17] — *serious* terms are typed as *computations*, *trivial* terms are typed as *values*. This point is developed elsewhere [10, 11].

Identifiers:

$$\Gamma \cup \{i : \tau\} \vdash_a i : (\tau\,,\ \mathit{serious}) \qquad \Gamma \cup \{i_v : \tau\} \vdash_a i_v : (\tau\,,\ \mathit{trivial})$$

Primitive Operators:

$$\Gamma \vdash_a \mathsf{op}_0 : (\iota\,,\ \mathit{trivial}) \qquad \frac{\Gamma \vdash_a e_1 : (\iota\,,\ \alpha)}{\Gamma \vdash_a \mathsf{op}_1\, e_1 : (\iota\,,\ \alpha)} \qquad \frac{\Gamma \vdash_a e_1 : (\iota\,,\ \alpha_1) \quad \Gamma \vdash_a e_2 : (\iota\,,\ \alpha_2)}{\Gamma \vdash_a \mathsf{op}_2\, e_1\, e_2 : (\iota\,,\ \alpha_1 \sqcup \alpha_2)}$$

Conditional:

$$\frac{\Gamma \vdash_a e_1 : (\iota\,,\ \alpha_1) \quad \Gamma \vdash_a e_2 : (\tau\,,\ \alpha_2) \quad \Gamma \vdash_a e_3 : (\tau\,,\ \alpha_3)}{\Gamma \vdash_a \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : (\tau\,,\ \alpha_1 \sqcup \alpha_2 \sqcup \alpha_3)}$$

Eager Binding:

$$\frac{\Gamma \vdash_a e_0 : (\tau_0\,,\ \alpha_0) \quad \Gamma \cup \{i_v : \tau_0\} \vdash_a e_1 : (\tau_1\,,\ \alpha_1)}{\Gamma \vdash_a \mathit{let}\ i = e_0\ \mathit{in}\ e_1 : (\tau_1\,,\ \alpha_0 \sqcup \alpha_1)}$$

Recursive Binding:

$$\frac{\Gamma \cup \{i_v : \tau_0\} \vdash_a a : (\tau_0\,,\ \mathit{trivial}) \quad \Gamma \cup \{i_v : \tau_0\} \vdash_a e : (\tau_1\,,\ \alpha)}{\Gamma \vdash_a \mathit{letrec}\ i = a\ \mathit{in}\ e : (\tau_1\,,\ \alpha)}$$

Abstractions:

$$\frac{\Gamma \cup \{i : \tau_0\} \vdash_a e : (\tau_1\,,\ \mathit{trivial})}{\Gamma \vdash_a \lambda_t i : \tau_0.e : (\tau_0 \rightarrow_t \tau_1\,,\ \mathit{trivial})} \qquad \frac{\Gamma \cup \{i : \tau_0\} \vdash_a e : (\tau_1\,,\ \mathit{serious})}{\Gamma \vdash_a \lambda i : \tau_0.e : (\tau_0 \rightarrow \tau_1\,,\ \mathit{trivial})}$$

$$\frac{\Gamma \cup \{i_v : \tau_0\} \vdash_a e : (\tau_1\,,\ \mathit{trivial})}{\Gamma \vdash_a \lambda_w i : \tau_0.e : (\tau_0 \rightarrow_w \tau_1\,,\ \mathit{trivial})} \qquad \frac{\Gamma \cup \{i_v : \tau_0\} \vdash_a e : (\tau_1\,,\ \mathit{serious})}{\Gamma \vdash_a \lambda_v i : \tau_0.e : (\tau_0 \rightarrow_v \tau_1\,,\ \mathit{trivial})}$$

Applications:

$$\frac{\Gamma \vdash_a e_0 : (\tau_0 \rightarrow_w \tau_1\,,\ \alpha) \quad \Gamma \vdash_a e_1 : (\tau_0\,,\ \mathit{trivial})}{\Gamma \vdash_a e_0 \mathbin{@_w} e_1 : (\tau_1\,,\ \alpha)}$$

$$\frac{\Gamma \vdash_a e_0 : (\tau_0 \rightarrow_t \tau_1\,,\ \alpha) \quad \Gamma \vdash_a e_1 : (\tau_0\,,\ \mathit{serious})}{\Gamma \vdash_a e_0 \mathbin{@_t} e_1 : (\tau_1\,,\ \alpha)}$$

$$\frac{\Gamma \vdash_a e_0 : (\tau_0 \rightarrow_v \tau_1\,,\ \alpha) \quad \Gamma \vdash_a e_1 : (\tau_0\,,\ \mathit{trivial})}{\Gamma \vdash_a e_0 \mathbin{@_v} e_1 : (\tau_1\,,\ \mathit{serious})}$$

$$\frac{\Gamma \vdash_a e_0 : (\tau_0 \rightarrow \tau_1\,,\ \alpha) \quad \Gamma \vdash_a e_1 : (\tau_0\,,\ \mathit{serious})}{\Gamma \vdash_a e_0 \mathbin{@} e_1 : (\tau_1\,,\ \mathit{serious})}$$

Generalization:

$$\frac{\Gamma \vdash_a e : (\tau\,,\ \mathit{trivial})}{\Gamma \vdash_a e : (\tau\,,\ \mathit{serious})}$$

Fig. 8. Type-checking rules for the annotated λ-calculus

and call-by-value operational semantics for the non-annotated λ-calculus (λ-*exp*) of Figure 6. For either reduction relation, $e \Downarrow v$ (read "e halts at value v") denotes the reduction of some type-correct closed term $e \in \lambda\text{-}exp$ to a value v (*i.e.*, a constant or a λ-abstraction). Similarly, $e \Downarrow$ (read "e halts") denotes the reduction of some type-correct closed term e to an unspecified value.

Let $\mathcal{A} : \lambda\text{-}exp \longrightarrow Ann\text{-}\lambda\text{-}exp$ denote an annotation-assigning function. $\mathcal{A}$ is cor.,idered to be correct if and only if it satisfies both of the following properties.

Property 1 (Soundness) *An annotation function* $\mathcal{A}$ *is* sound *iff for all type-correct closed terms* $e \in \lambda\text{-}exp$, $\mathcal{A}[\![e]\!] = e' : (\tau \,, \, trivial)$ *implies* $e \Downarrow_n$ *— that is, the evaluation of e terminates under call-by-name reduction.*

Property 2 (Consistency) *An annotation function* $\mathcal{A}$ *is* consistent *iff for all type-correct closed terms* $e \in \lambda\text{-}exp$, $\mathcal{A}[\![e]\!] = e' : (\tau \,, \, \alpha)$ *implies* $\vdash_a e' : (\tau \,, \, \alpha)$.

The process of assigning annotations can be automated using the techniques of abstract interpretation or of type inference. The abstract interpretation approach is summarized as follows. As a first step, the application sites of each abstraction are enumerated using a control-flow analysis [30]. The enumeration of application sites allows a straightforward generalization of Mycroft's b termination analysis to our higher-order language [20]. The correctness of the termination analysis establishes the required soundness property (Property 1). Based on the results of the termination analysis, terms are assigned annotations via our type-annotation rules of Figure 8. At this step, the *Generalization* rule of Figure 8 needs to be used to establish the consistency requirement (Property 2). For example, if an abstraction is applied to both trivial and serious arguments, all trivial arguments are generalized to serious terms.

Note that the *Generalization* rule may lead to more than one correct assignment of annotations to a particular term. However, no semantic ambiguity results since the transformation $\mathcal{C}_a$ is correct for all correct annotation assignments $\mathcal{A}$ (see Proposition 3).

5.2 A transformation for the annotated λ-calculus

Figure 9 displays the extend·d transformation into continuation style. The transformation $\mathcal{C}_a[\![\cdot]\!]$ is used over both serious and trivial terms and dispatches to either $\mathcal{C}_a\langle\cdot\rangle$ or $\mathcal{C}_a\{\!|\cdot|\!\}$.

- $\mathcal{C}_a\langle\cdot\rangle$ transforms trivial terms. No continuations are introduced in the transformed terms.
- $\mathcal{C}_a\{\!|\cdot|\!\}$ transforms serious terms. Continuations are introduced in each transformed term.

Figure 9 displays the transformation $\mathcal{C}_a$ on types as well. $\mathcal{C}_a$ is extended to type assumptions by defining

$$\mathcal{C}_a[\![\{\ldots, i : \tau, \ldots, i'_v : \tau', \ldots\}]\!] \;=\; \{\ldots, i : \mathcal{C}_a\{\!|\tau|\!\}, \ldots, i' : \mathcal{C}_a\langle\tau'\rangle, \ldots\}$$

The following proposition states the relationship between the types of annotated terms and the types of terms in the image of $\mathcal{C}_a$.

General Transformation:

$$\begin{aligned}
\mathcal{C}_a\llbracket e : (\tau\,,\, \alpha)\rrbracket \;&:\; \mathcal{C}_a\llbracket \tau\rrbracket \\
\mathcal{C}_a\llbracket e : (\tau\,,\, \mathit{trivial})\rrbracket &= \lambda\kappa.\kappa\, \mathcal{C}_a\langle e\rangle \\
\mathcal{C}_a\llbracket e : (\tau\,,\, \mathit{serious})\rrbracket &= \mathcal{C}_a\llparenthesis e\rrparenthesis
\end{aligned}$$

Trivial Terms:

$$\begin{aligned}
\mathcal{C}_a\langle e : (\tau\,,\, \mathit{trivial})\rangle \;&:\; \mathcal{C}_a\langle \tau\rangle \\
\mathcal{C}_a\langle i_v\rangle &= i \\
\mathcal{C}_a\langle \mathsf{op}_0\rangle &= \mathsf{op}_0 \\
\mathcal{C}_a\langle \mathsf{op}_1\, e_1\rangle &= \mathsf{op}_1\, \mathcal{C}_a\langle e_1\rangle \\
\mathcal{C}_a\langle \mathsf{op}_2\, e_1\, e_2\rangle &= \mathsf{op}_2\, \mathcal{C}_a\langle e_1\rangle\, \mathcal{C}_a\langle e_2\rangle \\
\mathcal{C}_a\langle \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2\rangle &= \mathit{if}\ \mathcal{C}_a\langle e_0\rangle\ \mathit{then}\ \mathcal{C}_a\langle e_1\rangle\ \mathit{else}\ \mathcal{C}_a\langle e_2\rangle \\
\mathcal{C}_a\langle \mathit{let}\ i = e_0\ \mathit{in}\ e_1\rangle &= \mathit{let}\ i = \mathcal{C}_a\langle e_0\rangle\ \mathit{in}\ \mathcal{C}_a\langle e_1\rangle \\
\mathcal{C}_a\langle \mathit{letrec}\ i = a\ \mathit{in}\ e\rangle &= \mathit{letrec}\ i = \mathcal{C}_a\langle a\rangle\ \mathit{in}\ \mathcal{C}_a\langle e\rangle \\
\mathcal{C}_a\langle \lambda_{tv} i : \tau.e\rangle &= \lambda i : \mathcal{C}_a\langle \tau\rangle.\mathcal{C}_a\langle e\rangle \\
\mathcal{C}_a\langle \lambda_{t} i : \tau.e\rangle &= \lambda i : \mathcal{C}_a\llparenthesis \tau\rrparenthesis.\mathcal{C}_a\langle e\rangle \\
\mathcal{C}_a\langle \lambda_{v} i : \tau.e\rangle &= \lambda i : \mathcal{C}_a\langle \tau\rangle.\mathcal{C}_a\llparenthesis e\rrparenthesis \\
\mathcal{C}_a\langle \lambda i : \tau.e\rangle &= \lambda i : \mathcal{C}_a\llparenthesis \tau\rrparenthesis.\mathcal{C}_a\llparenthesis e\rrparenthesis \\
\mathcal{C}_a\langle e_0 \mathbin{@_{tv}} e_1\rangle &= \mathcal{C}_a\langle e_0\rangle\, \mathcal{C}_a\langle e_1\rangle \\
\mathcal{C}_a\langle e_0 \mathbin{@_{t}} e_1\rangle &= \mathcal{C}_a\langle e_0\rangle\, \mathcal{C}_a\llparenthesis e_1\rrparenthesis
\end{aligned}$$

Serious Terms:

$$\begin{aligned}
\mathcal{C}_a\llparenthesis e : (\tau\,,\, \mathit{serious})\rrparenthesis \;&:\; \mathcal{C}_a\llparenthesis \tau\rrparenthesis \\
\mathcal{C}_a\llparenthesis i\rrparenthesis &= i \\
\mathcal{C}_a\llparenthesis \mathsf{op}_1\, e_1\rrparenthesis &= \lambda\kappa.\mathcal{C}_a\llbracket e_1\rrbracket\, (\lambda v_1.\kappa\, (\mathsf{op}_1\, v_1)) \\
\mathcal{C}_a\llparenthesis \mathsf{op}_2\, e_1\, e_2\rrparenthesis &= \lambda\kappa.\mathcal{C}_a\llbracket e_1\rrbracket\, (\lambda v_1.\mathcal{C}_a\llbracket e_2\rrbracket\, (\lambda v_2.\kappa\, (\mathsf{op}_2\, v_1\, v_2))) \\
\mathcal{C}_a\llparenthesis \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2\rrparenthesis &= \lambda\kappa.\mathcal{C}_a\llbracket e_0\rrbracket\, (\lambda v_0.\mathit{if}\ v_0\ \mathit{then}\ \mathcal{C}_a\llbracket e_1\rrbracket\, \kappa\ \mathit{else}\ \mathcal{C}_a\llbracket e_2\rrbracket\, \kappa) \\
\mathcal{C}_a\llparenthesis \mathit{let}\ i = e_0\ \mathit{in}\ e_1\rrparenthesis &= \lambda\kappa.\mathcal{C}_a\llbracket e_0\rrbracket\, (\lambda i.\mathcal{C}_a\llbracket e_1\rrbracket\, \kappa) \\
\mathcal{C}_a\llparenthesis \mathit{letrec}\ i = a\ \mathit{in}\ e\rrparenthesis &= \lambda\kappa.\mathit{letrec}\ i = \mathcal{C}_a\langle a\rangle\ \mathit{in}\ \mathcal{C}_a\llbracket e\rrbracket\kappa \\
\mathcal{C}_a\llparenthesis e_0 \mathbin{@_{tv}} e_1\rrparenthesis &= \lambda\kappa.\mathcal{C}_a\llbracket e_0\rrbracket\, (\lambda v_0.\kappa\, (v_0\, \mathcal{C}_a\langle e_1\rangle)) \\
\mathcal{C}_a\llparenthesis e_0 \mathbin{@_{t}} e_1\rrparenthesis &= \lambda\kappa.\mathcal{C}_a\llbracket e_0\rrbracket\, (\lambda v_0.\kappa\, (v_0\, \mathcal{C}_a\llparenthesis e_1\rrparenthesis)) \\
\mathcal{C}_a\llparenthesis e_0 \mathbin{@_{v}} e_1\rrparenthesis &= \lambda\kappa.\mathcal{C}_a\llbracket e_0\rrbracket\, (\lambda v_0.(v_0\, \mathcal{C}_a\langle e_1\rangle)\, \kappa) \\
\mathcal{C}_a\llparenthesis e_0 \mathbin{@} e_1\rrparenthesis &= \lambda\kappa.\mathcal{C}_a\llbracket e_0\rrbracket\, (\lambda v_0.(v_0\, \mathcal{C}_a\llparenthesis e_1\rrparenthesis)\, \kappa)
\end{aligned}$$

Types:

$$\begin{aligned}
\mathcal{C}_a\llbracket \tau\rrbracket &= \mathcal{C}_a\llparenthesis \tau\rrparenthesis \\
\mathcal{C}_a\llparenthesis \tau\rrparenthesis &= (\mathcal{C}_a\langle \tau\rangle \to \mathit{Ans}) \to \mathit{Ans} \\
\mathcal{C}_a\langle \iota\rangle &= \iota
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}_a\langle \tau_0 \to_{tv} \tau_1\rangle &= \mathcal{C}_a\langle \tau_0\rangle \to \mathcal{C}_a\langle \tau_1\rangle \\
\mathcal{C}_a\langle \tau_0 \to_{v} \tau_1\rangle &= \mathcal{C}_a\langle \tau_0\rangle \to \mathcal{C}_a\llparenthesis \tau_1\rrparenthesis \\
\mathcal{C}_a\langle \tau_0 \to_{t} \tau_1\rangle &= \mathcal{C}_a\llparenthesis \tau_0\rrparenthesis \to \mathcal{C}_a\langle \tau_1\rangle \\
\mathcal{C}_a\langle \tau_0 \to \tau_1\rangle &= \mathcal{C}_a\llparenthesis \tau_0\rrparenthesis \to \mathcal{C}_a\llparenthesis \tau_1\rrparenthesis
\end{aligned}$$

Fig. 9. Transformation of annotated λ-terms into continuation style

Proposition 2.

- *If* $\Gamma \vdash_a e : (\tau, \alpha)$ *then* $\mathcal{C}_a[\Gamma] \vdash \mathcal{C}_a[e] : \mathcal{C}_a[\tau]$.
- *If* $\Gamma \vdash_a e : (\tau, serious)$ *then* $\mathcal{C}_a[\Gamma] \vdash \mathcal{C}_a\langle\!\langle e \rangle\!\rangle : \mathcal{C}_a\langle\!\langle \tau \rangle\!\rangle$.
- *If* $\Gamma \vdash_a e : (\tau, trivial)$ *then* $\mathcal{C}_a[\Gamma] \vdash \mathcal{C}_a\langle e \rangle : \mathcal{C}_a\langle \tau \rangle$.

The correctness of $\mathcal{C}_a$ is stated as follows. (The notation "$e \Downarrow_n r$" is defined in Section 5.1.)

Proposition 3. *For all type-correct closed terms e of base type and for all correct annotation-assigning functions* $\mathcal{A}$,

$$e \Downarrow_n r \iff (\mathcal{C}_a \circ \mathcal{A}[\![e]\!])(\lambda i : \iota.i) \Downarrow_n r \iff (\mathcal{C}_a \circ \mathcal{A}[\![e]\!])(\lambda i : \iota.i) \Downarrow_v r$$

Proof. See [10].

5.3 Assessment

Restricting the new transformation $\mathcal{C}_a$ (see Figure 9) to call-by-name serious λ-terms yields the call-by-name transformation into continuation style (see Figure 2). Conversely, restricting $\mathcal{C}_a$ to call-by-value trivial λ-terms yields the identity transformation — no continuations are needed at all (*e.g.*, the denotational semantics of a strongly-normalizing language or of the language of Figure 3 without the "while" statement). Thus, $\mathcal{C}_a$ generalizes both the call-by-name transformation into continuation style and the identity transformation.

6 Some Properties of the Direct-Style Definition of the Simple Imperative Language (Figure 4)

Property 3 $\mathcal{Z}[\![\text{Program}]\!]$, $\mathcal{C}[\![\text{Command}]\!]$, *and* $\mathcal{E}[\![\text{Expression}]\!]$ *are trivial and call-by-value.*

Proof. Each is call-by-value because it is not possible for an argument expression of type *Env* to diverge. Each is trivial because a λ-abstraction (a value) is always returned.

Property 4 *The function type Com is call-by-value and serious.*

Proof. Due to the eager binding of the *let* construct, each command is passed a reduced store value. Therefore, the function type can be classified as call-by-value. Commands are serious because looping may occur in the *while* construct (this was accounted for by the lifting of the codomain of *Com*).

Property 5 *The function type Exp is call-by-value and trivial.*

Proof. Due to call-by-value property of commands, each expression is passed a reduced store value. Therefore, the function type can be classified as call-by-value. No expression contains components which may loop. Therefore expressions are trivial.

Property 6 *Proc and Fun are call-by-value and trivial.*

Proof. The arguments to procedures and functions originate from the evaluation of expressions which can never loop. Therefore, the function spaces are classified as call-by-value. They are trivial because they both return λ-abstractions.

Figure 10 presents an annotated representation of the direct semantics of Figure 4.[9] Any reasonable implementation of $\mathcal{A}$ as outlined in Section 5.1 would assign such annotations automatically. Let us now transform the annotated terms into continuation style.

7 Transforming the Representation of a Direct-Style Definition

Fact 1 *Transforming the annotated λ-representation of the direct semantics in Figure 10 into continuation style does yield the λ-representation of the continuation semantics in Figure 5, after administrative reductions.*

Further, we now have the ability to specify *any kind* of continuation semantics. For example, we could classify *Exp* to be serious (this would happen if recursive functions were allowed in the simple imperative language). This classification suffices to construct a continuation semantics where the valuation functions for both commands and expressions are in continuation style, automatically.

8 Conclusion, Issues, and Future Work

We have tried to contribute to the study of the relation between direct and continuation semantics [26] by connecting it to the transformation of λ-terms into continuation style. To this end, we have described how to construct the representation of a realistic continuation semantics automatically, given the representation of a direct semantics. (Again, by "realistic", we mean "that could have been written by [illegible] The situation is summarized by the diagram below.

$$\begin{tikzcd}
\text{DS semantics} \arrow[r, phantom, "\cong"] \arrow[d] & \text{CS semantics} \arrow[d] \\
\text{DS } \lambda\text{-terms} \arrow[r, "\mathcal{C}_a \circ \mathcal{A}"] & \text{CS } \lambda\text{-terms}
\end{tikzcd}$$

[9] Note that non-termination — represented explicitly via lifting (*e.g.*, $Store_\perp$) in the semantics definition of Figure 4 — is captured implicitly in the reduction properties of terms in Figure 10. This was accounted for in Property 4: *Com* has a serious function type.

Valuation Functions:

$$\begin{aligned}
\mathcal{Z}\llbracket\text{Program}\rrbracket &: \mathit{Env} \to_{tw} \mathit{Com} \\
\mathcal{C}\llbracket\text{Command}\rrbracket &: \mathit{Env} \to_{tw} \mathit{Com} \\
\mathcal{E}\llbracket\text{Expression}\rrbracket &: \mathit{Env} \to_{tw} \mathit{Exp} \\
\mathcal{N}\llbracket\text{Numeral}\rrbracket &: \mathit{Nat} \\
\mathcal{L}\llbracket\text{Location}\rrbracket &: \mathit{Loc}
\end{aligned}$$

Semantic Domains:

$$\begin{aligned}
\mathit{Com} &= \mathit{Store} \to_{v} \mathit{Store} \\
\mathit{Exp} &= \mathit{Store} \to_{tw} \mathit{Nat} \\
\mathit{Proc} &= \mathit{Nat} \to_{tw} \mathit{Com} \\
\mathit{Fun} &= \mathit{Nat} \to_{tw} \mathit{Exp}
\end{aligned}$$

Programs:

$$\begin{aligned}
\mathcal{Z}\llbracket\text{proc}\, p\,(m) = c \text{ in } z\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.\mathcal{Z}\llbracket z\rrbracket @_{tw} (\text{ext}\ \rho_v\ p\ (\lambda_{tw} i.\mathcal{C}\llbracket c\rrbracket @_{tw} (\text{ext}\ \rho_v\ m\ i_v)))\ @_v\ \sigma_v \\
\mathcal{Z}\llbracket\text{fun}\, f\,(m) = e \text{ in } z\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.\mathcal{Z}\llbracket z\rrbracket @_{tw} (\text{ext}\ \rho_v\ f\ (\lambda_{tw} i.\mathcal{E}\llbracket e\rrbracket @_{tw} (\text{ext}\ \rho_v\ m\ i_v)))\ @_v\ \sigma_v \\
\mathcal{Z}\llbracket c.\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.\mathcal{C}\llbracket c\rrbracket @_{tw}\ \rho_v\ @_v\ \sigma_v
\end{aligned}$$

Commands:

$$\begin{aligned}
\mathcal{C}\llbracket\text{skip}\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.\sigma_v \\
\mathcal{C}\llbracket c_1\,;\,c_2\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.\mathit{let}_v\ \sigma' = \mathcal{C}\llbracket c_1\rrbracket @_{tw}\ \rho_v\ @_v\ \sigma_v\ \mathit{in}\ \mathcal{C}\llbracket c_2\rrbracket @_{tw}\ \rho_v\ @_v\ \sigma'_v \\
\mathcal{C}\llbracket l := e\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.\text{upd}\ \sigma_v\ \mathcal{L}\llbracket l\rrbracket\ (\mathcal{E}\llbracket e\rrbracket @_{tw}\ \rho_v\ @_{tw}\ \sigma_v) \\
\mathcal{C}\llbracket\text{if } e \text{ then } c_1 \text{ else } c_2\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.\ \mathit{if}\quad \text{iszero?}\ (\mathcal{E}\llbracket e\rrbracket @_{tw}\ \rho_v\ @_{tw}\ \sigma_v) \\
&\qquad\qquad\quad \mathit{then}\ (\mathcal{C}\llbracket c_1\rrbracket @_{tw}\ \rho_v\ @_v\ \sigma_v) \\
&\qquad\qquad\quad \mathit{else}\ \ (\mathcal{C}\llbracket c_2\rrbracket @_{tw}\ \rho_v\ @_v\ \sigma_v) \\
\mathcal{C}\llbracket\text{while } e \text{ do } c\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.\ \mathit{letrec}_v\ w = \lambda_v\sigma.\ \mathit{if}\quad \text{iszero?}\ (\mathcal{E}\llbracket e\rrbracket @_{tw}\ \rho_v\ @_{tw}\ \sigma_v) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{then}\ \mathit{let}_v\ \sigma' = \mathcal{C}\llbracket c\rrbracket @_{tw}\ \rho_v\ @_v\ \sigma_v \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{in}\ w_v\ @_v\ \sigma'_v \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ \ \sigma_v \\
&\qquad\qquad\quad \mathit{in}\ w_v\ @_v\ \sigma_v \\
\mathcal{C}\llbracket\text{call } p\,(e)\rrbracket &= \lambda_{tw}\rho.\lambda_v\sigma.(\text{lookup}\ \rho_v\ p)\ @_{tw}\ (\mathcal{E}\llbracket e\rrbracket @_{tw}\ \rho_v\ @_{tw}\ \sigma_v)\ @_v\ \sigma_v
\end{aligned}$$

Expressions:

$$\begin{aligned}
\mathcal{E}\llbracket n\rrbracket &= \lambda_{tw}\rho.\lambda_{tw}\sigma.\mathcal{N}\llbracket n\rrbracket \\
\mathcal{E}\llbracket m\rrbracket &= \lambda_{tw}\rho.\lambda_{tw}\sigma.\text{lookup}\ \rho_v\ m \\
\mathcal{E}\llbracket\text{succ}\ e\rrbracket &= \lambda_{tw}\rho.\lambda_{tw}\sigma.\text{succ}\ (\mathcal{E}\llbracket e\rrbracket @_{tw}\ \rho_v\ @_{tw}\ \sigma_v) \\
\mathcal{E}\llbracket\text{pred}\ e\rrbracket &= \lambda_{tw}\rho.\lambda_{tw}\sigma.\text{pred}\ (\mathcal{E}\llbracket e\rrbracket @_{tw}\ \rho_v\ @_{tw}\ \sigma_v) \\
\mathcal{E}\llbracket\text{deref}\ l\rrbracket &= \lambda_{tw}\rho.\lambda_{tw}\sigma.\text{fetch}\ \sigma_v\ \mathcal{L}\llbracket l\rrbracket \\
\mathcal{E}\llbracket\text{apply}\ f\,(e)\rrbracket &= \lambda_{tw}\rho.\lambda_{tw}\sigma.(\text{lookup}\ \rho_v\ f)\ @_{tw}\ (\mathcal{E}\llbracket e\rrbracket @_{tw}\ \rho_v\ @_{tw}\ \sigma_v)\ @_{tw}\ \sigma_v
\end{aligned}$$

Fig. 10. Annotated representation of the direct semantics in Figure 4

Based on the annotations produced by $\mathcal{A}$, the transformation $\mathcal{C}_a$ introduces just enough continuations to preserve call-by-name meaning under both call-by-name and call-by-value reduction. $\mathcal{C}_a$ generalizes both the call-by-name continuation transformation (should all terms be serious) and the identity transformation (should all terms be trivial).

8.1 A shortcoming?

One might criticize one shortcoming of this approach: it only produces the representation of a continuation semantics, not the continuation semantics itself. One answer to this criticism goes as follows.

Why would one want a continuation semantics when one already has a brave and honest direct semantics?[10] Not for the love of mathematics alone, but for implementation purposes [16, 18]! But then one does not need the continuation semantics, but its representation — which is precisely what our new transformation produces automatically. Therefore our approach enables the language developer to stay with one mathematical model — the direct semantics — and to derive the continuation semantics as part of the implementation work.

8.2 An alternative?

One could transform the direct semantics into continuation style and then simplify the result into a manageable continuation semantics. We believe that our approach is more natural since most of the work operates over the original direct semantics and the rest is automatic. (A worthwhile property considering how counter-intuitive continuation-style specifications may look.)

8.3 Applications

Semantics-directed compiler-generation systems often work on continuation semantics [13, 14, 16, 18], thus forcing one to write a continuation semantics and to prove its congruence with the direct one. Our new transformation allows one to produce the representation of a continuation semantics automatically.

Partial evaluators work better on continuation-passing programs, but again not all continuations are always necessary [2, 3]. Our extended transformation into continuation style makes it possible to reduce the occurrences of continuations in a source program. In addition, it also enables partial evaluation of call-by-name programs (after evaluation-order analysis — be it for strictness or termination) with a regular partial evaluator for call-by-value programs.

[10] Of course, the situation is different if the source language includes some form of jump. But then one has no direct semantics and thus *starts* with a continuation semantics.

8.4 Variations

Denotational definitions are written in various fashions. We briefly mention how the present work can be adapted to other fashions.

Partial functions are often used in place of total functions and lifted domains when modeling non-terminating computations [23, 34]. Our explanation of totality and partiality in terms of trivial and serious functions naturally applies to denotational specifications based on partial functions.

Strict functions are often used to model the strictness properties associated with eager (*i.e.*, call-by-value) functions [23, 28]. For simplicity of presentation, we have expressed strictness properties using *let* constructs only. Just as *let* expressions are represented using eager binding constructs, strict functions are represented using eager applications. Thus, a transformation of an annotated language including both eager and normal-order application generalizes both the call-by-value and the call-by-name transformation into continuation style. We have presented a formalization of such a mixed transformation elsewhere [5].

Continuation semantics of imperative languages often express the meaning of commands as "continuation transformers" [28, 34]. Specifically, the functionality of *Com* is given as

$$(\mathit{Store} \rightarrow \mathit{Ans}) \rightarrow \mathit{Store} \rightarrow \mathit{Ans}.$$

It is very simple to specify a transformation into continuation style that "puts continuations first", as in Fischer's original transformation [7, 27]. Such a transformation would naturally yield the functionality above.

Finally, our work has relied on denotational definitions being stated using a simply-typed meta-language. This meta-language is sufficient for defining simple imperative languages and simply-typed languages such as Algol 60, Pascal, and PCF. We are currently investigating how the results presented here can be extended to a meta-language with recursive types. This would be necessary for defining untyped languages such as Scheme.

8.5 Generalization

The work presented here can be generalized to other styles than continuation style. Alternatively, one could define a core meta-language and parameterize it with the style of the interpretation. This approach is reminiscent of Mosses and Watt's Action Semantics [19, 35], of the Nielsons's two-level meta-language [23], and of Moggi's computational λ-calculus [17]. We investigate it elsewhere [11].

8.6 Transforming the representation of a continuation semantics into direct style

The transformation from continuation style to direct style has been investigated recently [4, 6, 12, 27], and enables one to transform the representation of a continuation semantics into direct style. Of course, we can only produce the representation of a direct semantics from a continuation semantics where continuations are second-class [31] — for example, we could not produce a direct semantics for a language with jumps [32], not without adding some kind of control operator to the λ-calculus [6]. Syntactic conditions over a continuation-passing λ-term to ensure that continuations are second-class can be found elsewhere [4]. Overall we leave this transformation for future work.

8.7 Continuation style and evaluation-order independence

It is interesting to compare the structure of the terms produced by our transformation $\mathcal{C}_a$ with the structure of the terms produced by *e.g.*, Plotkin's continuation-style transformations [24]. In addition to satisfying his *Simulation*, *Indifference*, and *Translation* theorems, Plotkin's continuation-style terms have two additional properties that are often utilized for implementation purposes:

- all function calls are in "tail" position;[11]
- all intermediate values are given names.

In contrast, $\mathcal{C}_a$ inserts just enough continuations to preserve call-by-name meaning under both call-by-name and call-by-value reduction. Thus, $\mathcal{C}_a$ satisfies Plotkin's *Simulation*, *Indifference*, and *Translation* theorems [10], but the two additional properties above are lost because some applications may be trivial.

- Trivial applications may occur as function arguments — not a "tail" position.
- Trivial applications yield intermediate values that are not named — since trivial functions are not passed any continuation.

In other words, our transformation $\mathcal{C}_a$ does not produce "Continuation-Passing Style" terms (!) but it does produce terms that are independent of the evaluation order.

Acknowledgements

We are grateful to Andrzej Filinski, Julia Lawall, Karoline Malmkjær, David Schmidt, and the referees for their comments, to Frank Pfenning and Kirsten Solberg for their interest and time, and to Hanne Riis Nielson for a key comment about our style of annotation. Thanks are also due to Patrick Cousot for connecting our termination analysis with Alan Mycroft's $\sharp$ and $\flat$ analyses. The diagram of Section 8 was drawn using Kristoffer Rose's XY-pic package.

References

1. William Clinger, editor. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, San Francisco, California, June 1992. ACM Press.

[11] As characterized by Meyer and Wand [15], "[Continuation-style] terms are tail-recursive: no argument is an application."

2. Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 496–519, Cambridge, Massachusetts, August 1991.
3. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Graham [8], pages 493–501.
4. Olivier Danvy. Back to direct style. In Bernd Krieg-Brückner, editor, *Proceedings of the Fourth European Symposium on Programming*, number 582 in Lecture Notes in Computer Science, pages 130–150, Rennes, France, February 1992. Extended version to appear in Science of Computer Programming.
5. Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.
6. Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [1], pages 299–310.
7. Michael J. Fischer. Lambda calculus schemata. In Talcott [33]. An earlier version appeared in an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.
8. Susan L. Graham, editor. *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.
9. Bob Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Talcott [33].
10. John Hatcliff. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, USA, March 1994. Forthcoming.
11. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press. To appear.
12. Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Graham [8], pages 124–136.
13. Peter Lee. *Realistic Compiler Generation*. MIT Press, 1989.
14. Peter Lee and Uwe Pleban. On the use of LISP in implementing denotational semantics. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 233–248, Cambridge, Massachusetts, August 1986.
15. Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985.
16. Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and John Wiley, New York, 1976.
17. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
18. Margaret Montenyohl and Mitchell Wand. Correct flow analysis in continuation semantics. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 204–218, San Diego, California, January 1988.
19. Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

20. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *Proceedings of the Fourth International Symposium on Programming*, number 83 in Lecture Notes in Computer Science, pages 269–281, Paris, France, April 1980.
21. Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1981.
22. Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988. Special issue on ESOP'86, the First European Symposium on Programming, Saarbrücken, March 17-19, 1986.
23. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
24. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
25. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.
26. John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974.
27. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Clinger [1], pages 288–298.
28. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
29. Peter Sestoft. Replacing function parameters by global variables. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 39–53, London, England, September 1989. ACM Press.
30. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, CMU, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.
31. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
32. Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.
33. Carolyn L. Talcott, editor. *Special issue on continuations*. LISP and Symbolic Computation, Vol. 6, Nos. 3/4. Kluwer Academic Publishers, 1993.
34. Robert D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.
35. David Watt. *Programming Languages Concepts and Paradigms*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.