N00014-89-J-3155
Final Technical Report

8

Proof Theoretic Concepts for the Semantics of Types and Concurrency

MS-CIS-95-19
LOGIC & COMPUTATION LAB 90

Edited by Carl Gunter

LEGES SINE MORIBUS VANAE

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

April 1995

INTRODUCTION

This is a collection of five papers that concern applications of ideas from proof theory to problems in the semantics of types and concurrency. In the order they are arranged, the articles are

1. *Inheritance as implicit coercion*, Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. **Information and Computation,** vol. 93 (1991), pp. 172–221 by invitation from the papers presented at the 1989 **Symposium on Logic in Computer Science.**
2. *Computing with coercions*, Val Breazu-Tannen, Carl A. Gunter, and Andre Scedrov. In: **Conference on Lisp and Functional Programming,** edited by M. Wand, Nice, France, July 1990, pp. 44–60.
3. *Nets as tensor theories (preliminary report)*, Carl A. Gunter and Vijay Gehlot. In: **Conference on Application and Theory of Petri Nets,** edited by G. De Michelis, Bonn, F.R.G., June 1989, pp. 174-191.
4. *Normal process representatives*, Vijay Gehlot and Carl A. Gunter. In: **Symposium on Logic in Computer Science,** edited by J. Mitchell, IEEE Computer Society Press, Philadelphia, Pennsylvania, June 1990, pp. 200–207.
5. *Reference counting as a computational interpretation of linear logic*, Jawahar Chirimar, Carl A. Gunter, and Jon Riecke. To appear in: **Journal of Functional Programming.**

For further reading on the coherence issues studied in the first two articles, one may consult the MIT Press collection **Theoretical Aspects of Object Oriented Programming Languages: Types, Semantics, and Language Design.** edited by John C. Mitchell and Carl A. Gunter. Further details about the third and fourth topics can be found in Vijay Gehlot's 1992 University of Pennsylvania Ph.D. dissertation, **A Proof-Theoretic Approach to Semantics of Concurrency.**

INHERITANCE AS IMPLICIT COERCION [1]

Val Breazu-Tannen *Thierry Coquand* *Carl A. Gunter* *Andre Scedrov*[2]

Abstract. We present a method for providing semantic interpretations for languages with a type system featuring *inheritance* polymorphism. Our approach is illustrated on an extension of the language Fun of Cardelli and Wegner, which we interpret via a translation into an extended polymorphic lambda calculus. Our goal is to interpret inheritances in Fun via *coercion functions* which are definable in the target of the translation. Existing techniques in the theory of semantic domains can be then used to interpret the extended polymorphic lambda calculus, thus providing many models for the original language. This technique makes it possible to model a rich type discipline which includes parametric polymorphism and recursive types as well as inheritance.

A central difficulty in providing interpretations for explicit type disciplines featuring inheritance in the sense discussed in this paper arises from the fact that programs can type-check in more than one way. Since interpretations follow the type-checking derivations, *coherence* theorems are required: that is, one must prove that the meaning of a program does not depend on the way it was type-checked. The proof of such theorems for our proposed interpretation are the basic technical results of this paper. Interestingly, proving coherence in the presence of recursive types, variants, and abstract types forced us to reexamine fundamental equational properties that arise in proof theory (in the form of commutative reductions) and domain theory (in the form of strict *vs.* non-strict functions).

1 Introduction

In this paper we will discuss an approach to the semantics of a particular form of inheritance which has been promoted by John Reynolds and Luca Cardelli. This inheritance system is based on the idea that one may axiomatize a relation $\leq$ between type expressions in such a way that whenever the *inheritance judgement* $s \leq t$ is provable for type expressions s and t, then an expression of type s can be "considered as" an expression of type t. This property is expressed by the *inheritance* rule (sometimes also called the *subsumption* rule), which states that if an expression e is of type s and $s \leq t$, then e also has type t. The consequences from a semantic point of view of the inclusion of this form of typing rule are significant. It is our goal in this paper to look carefully at what we consider to be a robust and intuitive approach to systems which have this form of inheritance and examine in some detail the semantic implications of the inclusion of inheritance judgements and the inheritance rule in a type discipline.

Several attempts have been made recently to express some of the distinctive features of object-oriented programming, principally *inheritance*, in the framework of a rich type discipline which can accommodate strong static type-checking. This endeavor searches for a language that offers some of the flexibility of object-oriented programming [GR83] while maintaining the reliability, and sometimes increased efficiency of programs which type-check at compile-time (see [BBG88] for a related comparison).

[1]Appears in **Information and Computation** vol. 93 (1991), pp. 172–221.

[2]Author's addresses. Breazu-Tannen and Gunter: Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia PA 19104, USA. Coquand: INRIA, Domaine de Voluceau, 78150 Rocquencourt, France. Scedrov: Department of Mathematics, University of Pennsylvania, Philadelphia PA 19104, USA.

A type system of Reynolds introduced in [Rey80] captured some basic intuitions about inheritance relations between familiar type expressions built from records, variants (sums) and higher types. A language which exploited this form of type discipline was developed by Cardelli in [Car84, Car88a] where the first attempt was made to describe a rigorous form of mathematical semantics for such a system. His approach uses ideals and it is shown that the type discipline is consistent with the semantics in the sense that type-checking is shown to "prevent type errors". Subsequent work has aimed at combining inheritance with richer type disciplines, in particular featuring *parametric polymorphism.* One direction of research [Wan87, JM88, OB88, Sta88], has investigated expressing inheritance and type inference mechanisms, similarly to the way in which parametric polymorphism is expressed in ML-like languages. Another direction of research investigates expressing inheritance through explicit subtyping mechanisms which are part of the type-checking systems, such as in Cardelli and Wegner's language Fun [CW85] and further work [Car88b, Car89a, CM89]. Cardelli and Wegner sketch a model for Fun based on ideals. An extensional model for Fun was subsequently described by Bruce and Longo [BL88]. Their model interprets inheritances as identity relations between partial equivalence relations (PER's). Another model of Fun, using the interval interpretation of Cartwright [Car85] has been given by Martini [Mar88]. In Martini's semantics, inheritance is interpreted as a form of inclusion between intervals. This model also includes a general recursion operator for functions (but not types).

In this paper we present a novel approach to the problem of developing a simple mathematical semantics for languages which feature inheritance in the sense of Reynolds and Cardelli. The form of semantics that we propose will take a significant departure from the characteristic shared by the semantics mentioned above. We will not attempt to model inheritance as a binary relation on a family of types. In particular, our interpretation will not use anything like an inclusion relation between types. Instead, we interpret the inheritance relation between type *expressions* as indicating a certain coercion which remains implicit in instances in which the inheritance is used in type-checking. We show how these coercions can be made explicit using *definable* terms of a calculus without inheritance, and thus depart from the "relational" interpretation of the inheritance concept. Using this idea, we are able to show how many of the models of polymorphism and recursive types which have no relevant concept of type inclusion, can nevertheless be seen as models for a calculus with inheritance.

We illustrate our approach on the language Fun of Cardelli and Wegner extended with recursive types but, the kind of results we obtain are non-trivial for any calculus that combines inheritance, parametric polymorphism, and recursive types. The method we propose proceeds first with a translation of Fun into an extended polymorphic lambda calculus with recursive types. As we mentioned above, this translation interprets inheritances in Fun as *coercion functions* already *definable* in the extended polymorphic lambda calculus. Then, we can use existing techniques for modeling polymorphism and recursion (such as those described in [ABL86, Gir86, CGW87, CGW89]) to interpret the extended polymorphic lambda calculus, thus providing models for the original language with inheritance. This method achieves simultaneous modeling of parametric polymorphism, recursive types, and inheritance. In the process, the paradigm "inheritance as definable coercion" proves itself remarkably robust, which makes us confident that it will apply to a large class of rich type disciplines with inheritance.

The paper is divided into seven sections. Following this introduction, the second section provides some general examples and motivation to prepare the reader for the technical details in the subsequent sections. The third section discusses how our semantics applies to a calculus **SOURCE** which has inheritance, exponentials, records, generics and recursive types. We show how this is translated into a calculus **TARGET** without inheritance and state our results about the coherence of the translation. We hope that the results in this simpler setting will help the reader get an idea of what our program is before we proceed to a more interesting calculus in the remainder of the paper. The fourth section is devoted to developing a translation for an expanded calculus which adds variants. Fundamental equational properties of variants lead us to develop a target language which has a *type of coercions.* The fifth section, which contains the difficult technical results of the paper, shows that our translation is coherent. In the sixth section we discuss mathematical models for the full calculus. Since most of the work has already been done, we are able to produce many models using standard domain-theoretic techniques. The concluding section makes some remarks about what we feel has been achieved and what new challenges still need to be confronted.

2 Inheritance as implicit coercion.

A simple analogy will help explain our translation-based technique. Consider how the ordinary *untyped* λ-calculus is interpreted semantically in such sources as [Sco80, Mey82, Koy82, Bar84]. One begins by postulating the existence of a semantic domain D and a pair of arrows $\Phi: D \to (D \to D)$ and $\Psi: (D \to D) \to D$ such that $\Phi \circ \Psi$ is the identity on $D \to D$. Certain conditions are required of $D \to D$ to insure that "enough" functions are present. To interpret an untyped λ-term, one defines a translation $M \mapsto M^*$ on terms which takes an untyped term M and creates a typed term M^*. This operation is defined by induction:

- for a variable, $x^* \equiv x{:}\,D$,
- for an application, $M(N)^* \equiv \Phi(M^*)(N^*)$ and,
- for an abstraction, $(\lambda x.\ M)^* \equiv \Psi(\lambda x{:}\,D.\ M^*)$

(where we use $\equiv$ for syntactic equality of expressions). For example, the familiar term

$$\lambda f.\ (\lambda x.\ f(xx))(\lambda x.\ f(xx))$$

translates to

$$\Psi(\lambda f{:}\,D.\ \Phi(\Psi(\lambda x{:}\,D.\ \Phi(f)(\Phi(x)(x))))(\Psi(\lambda x{:}\,D.\ \Phi(f)(\Phi(x)(x))))).$$

The fact that the latter term is unreadable is perhaps an indication of why we use the former term *in which the semantic coercions are implicit.* Nevertheless, this translation provides us with the desired semantics for the untyped term since we have converted that term into a term in a calculus which we know how to interpret. Of course, this assumes that we really do know how to provide a semantics for the typed calculus supplemented with triples such as D, Φ, Ψ. Moreover, there are some equations we must check to show that the translation is sound. But, at the end of the day, we have a simple, intuitive explanation of the interpretation of untyped λ-terms based on our understanding of a certain simply typed λ-theory. In this paper we show how a similar technique may be used to provide an intuitive interpretation for inheritance, even in the presence of parametric polymorphism and type recursion. As mentioned earlier, our interpretation is carried out by translating the full calculus into a calculus without inheritance (the *target* calculus) whose semantics we already understand. However, our idea differs significantly from the interpretation of the untyped λ-calculus as described above in at least one important respect: typically, the coercions (such as Φ and Ψ above) which we introduce will be *definable* in the target calculus. Hence our target calculus needs to be an extension of the ordinary polymorphic λ-calculus with records, variants, abstract types, and recursive types. But it need not have any inheritance.

From this lead, we may now propose a way to explain the semantics of an expression in a language with inheritance. Our semantics interprets typing judgements, *i.e.* assertions $\Gamma \vdash e{:}\, s$ that expression e has type s in context Γ. Ordinarily such a judgement is assigned a semantics inductively in the proof of the judgement using the typing rules. However, the system we are considering may also include instances of the *inheritance rule* which says that if e has type s and s is a subtype of t, then e has type t. How are we to relate the interpretation of the type expressions s and t so that the meaning of e can be viewed as living in both places? Our proposal: the proof that s is a subtype of t generates a *coercion* P from s into t. The inheritance (subsumption) rule is interpreted by the application of the coercion P to the interpretation of e as an element of s. It will be seen below that this technique can be made to work very smoothly since the language we are interpreting may have a familiar *inheritance-free* fragment in which coercions such as P can be defined. In effect, we can therefore "project" the language onto an inheritance-free fragment of itself.

For further illustration, let us now look at an example which combines parametric polymorphism and inheritance. In the polymorphic λ-calculus, it is possible to form expressions in which there are abstractions over *type variables.* For example, the term $e \equiv \Lambda a.\ \lambda x{:}\, a.\ x$ is an operator which takes a type s as an argument and returns the identity function $\lambda x{:}\, s.\ x$ on that type as a value. The type of e is indicated by the expression $\forall a.\ a \to a$. Semantically, one may think of the meaning of this expression as an indexed product where a ranges over all types. Although this explanation is a bit too simple as it stands, it does help with the basic intuition. If one wishes to make an abstraction over the *subtypes* of a given type, one may use the concept of a *bounded quantification* [CW85]. Consider, for example, the term

$$e' \equiv \Lambda a \leq \{l{:}\, s\}.\ \lambda x{:}\, a.\ (x.l)$$

where $\{l{:}\, s\}$ is a *record* expression which has one field, labelled l, with type s. The expression e' denotes an operator which takes a subtype t of $\{l{:}\, s\}$ (we write $t \leq \{l{:}\, s\}$) and returns as value a function from t to s. (The reader should not confuse a, a type variable, with t, a type expression.) Intuitively, a subtype of $\{l{:}\, s\}$ is a record which has an l field whose type is a subtype of s. The type of e' is indicated by the expression $u' \equiv \forall a \leq \{l{:}\, s\}.\ a \to s$. How should we think of this type semantically? Taking an analogy with the intuitive semantics of polymorphic quantification, we want to think of the meaning of u' as some kind of indexed product. But indexed over what? In this paper we argue that one may get an intuitive semantics of bounded quantification by thinking of a type expression such as u' as a family of types *indexed over coercions (i.e. certain functions) from a type a into the type s.*

To support this intuition we must explain the meaning of the application $e'(t)$ of the expression e' to a type expression t which is a subtype of $\{l{:}\, s\}$. The key fact is this: given type expressions v and w and a proof that v is a subtype of w, there is a canonical coercion from v into w. Hence, the application $e'(t)$ has, as its meaning, the element of $t \to s$ obtained by applying the meaning of e'—which is an element of an indexed product—to the canonical coercion from t to $\{l{:}\, s\}$. This leads us to consider u' as the type

$$\forall a.\ (a \circ\!\!\to \{l{:}\, s\}) \to a \to s$$

where $a \circ\!\!\to \{l{:}\, s\}$ is a "type of coercions". In category-theoretic jargon: the meaning of a bounded quantification with bound v will be an adjoint to a fibration over the *slice category* over v. This follows the analogy with models of polymorphism which are based on adjoints to fibrations over the category of all domains (as in [CGW89] for example).

Although we believe that the translation just illustrated is intuitive, we need to show that it is *coherent.* In other words, we must show that the semantic function is well defined. The need for coherence comes from the fact that a typing judgement may have many different derivations. In general, it is customary to present the semantics of typed lambda calculi as a map defined inductively on type-checking derivations. Such a method would therefore assign a meaning to each derivation tree. We do believe though, that the *language* consists of the derivable typing judgements, rather than of the derivation trees. For many calculi, such as the simply typed or the polymorphic lambda calculus, there is at most one derivation for any typing judgement. Therefore, in such calculi, giving meaning to derivations is the same as giving meaning to derivable judgements. But for other calculi, such as Martin-Löf's Intuitionistic Type Theory (ITT) [Mar84] (see [Sal88]), and the Calculus of Constructions [CH88] (see [Str88]), and—of immediate concern to us—Cardelli and Wegner's Fun, this is not so, and one must prove that derivations yielding the same judgement are given the same meaning. This idea has also appeared in the context of category theory and our use of the term "coherence" is partially inspired by its use there, where it means the uniqueness of certain canonical morphisms (see *e.g.* [KL71] and [LP85]). Although we have not attempted a rigorous connection in this paper, the possibility of unifying coherence results for a variety of different calculi offers an interesting direction of investigation. In the case of Fun, we show the coherence of our semantic approach by proving that *translations of any two derivations of the same typing judgement are equated in the target calculus.*

Hence, the coherence of a given translation is a property of the equational theory of the target calculus. When the target calculus is the polymorphic lambda calculus extended with records and recursive types, the standard axiomatization of its equational theory is sufficient for the coherence theorem. But when we add variants, the standard axiomatization of these features, while sufficient for coherence, clashes with the standard axiomatization of recursive types, yielding an inconsistent theory (see [Law69, HP89a] for variants, that is, coproducts). The solution lies in two observations: (1) the (too) strong axioms are only needed for "coercion terms", and (2) in the various models we examined these coercion terms have special interpretations (such as *strict*, or *linear* maps), so special in fact, that they satisfy the corresponding restrictions of the strong axioms! Correspondingly, one has to restrict the domains over which "coercion variables" can range, which leads naturally to the type of coercions mentioned above.

3 Translation for a fragment of the calculus

For pedagogical reasons, we begin by considering a language whose type structure features function spaces (exponentials), record types, bounded generic types (an inheritance-generalized form of universal polymorphism), recursive types, and, of course, inheritance. In the next section we will enrich this calculus by the addition of variants. As we have mentioned before, this leads to some (interesting) complications which we avoid by restricting ourselves to the simpler calculus of this section. Since the calculus in the next section is stronger, we omit details for the proofs of results in this section. They resemble the proofs for the calculus with variants, but the calculations are simpler. Rather than generate four different names for the calculi which we shall consider in this section and the next we simply refer to the calculus with inheritance as **SOURCE** and the inheritance-free calculus into which it is translated as **TARGET**. The fragment of the calculus which we consider in this section is fully described in the appendices to the paper.

We provide semantics to **SOURCE** via a *translation* into a language for which several well-understood semantics already exist. This "target" language, which we shall call **TARGET**, is an extension with record and recursive types of the Girard-Reynolds polymorphic lambda calculus (see [CGW87] for the semantics of **TARGET**). Therefore, **SOURCE** extends with inheritance and bounded generics **TARGET**, which is at its turn an extension of what Girard calls **System F** in [Gir86]. Our translation takes derivations of inheritance and typing judgements in **SOURCE** into derivations of typing judgements in **TARGET**. We translate the inheritance judgements of **SOURCE** into definable terms of **TARGET** which can be thought of as *canonical explicit coercions.* Bounded generics translate into usual generics, but of "higher" type, which take an additional argument which can be thought of as an *arbitrary coercion.*

In arguing that this translation yields a semantics for **SOURCE**, we encounter, as mentioned in the introduction, an important complication: as we shall see, in **SOURCE** as well as in Fun, there may be *several* distinct derivations of the *same* typing judgement (or inheritance judgement, for that matter). We consider, however, the *language* to consists of the derivable typing judgements, rather than of the derivation trees. This distinction can be ignored in System **F** or **TARGET**, where there is at most one derivation for any typing judgements, so giving meaning to derivations is the same as giving meaning to derivable judgements. But for **SOURCE** and Fun, this is not so, and one must show that derivations yielding the same judgement are given the same meaning. This meaning is then defined to be the meaning of the judgement. This crucial problem was overlooked by publications on the semantics of inheritance prior to [BCGS89].

We solve the problem as follows. It turns out that our translation takes syntactically distinct derivations of the same **SOURCE** judgement into syntactically distinct derivations in **TARGET**. But we give an *equational axiomatization* as an integral part of **TARGET**, and we show that our translation takes derivations of the same **SOURCE** judgement into derivations of *provably equal* judgements in **TARGET**. By this *coherence* result, *any* model of **TARGET**, being also a model of its equational theory, will provide a well-defined semantics for the derivable judgements of **SOURCE**.

The source calculus. For notation, we will follow the spirit of Fun [CW85] making precise only the differences. The type expressions include type variables a and a distinguished constant *Top*. If s and t are type expressions, then $s \rightarrow t$ is the type of functions from s to t. If $s_1, \ldots, s_n$ are type expressions, and $l_1, \ldots, l_n$ is a collection of distinct *labels,* then $\{l_1 : s_1, \ldots, l_n : s_n\}$ is a *record* type expression. We make the syntactic assumption that the order of the labels is irrelevant. If s and t are type expressions then $\forall a \leq s.\, t$ is a *bounded quantification* which binds free occurrences of the variable a in the type expression t (but not in s). Similarly, $\mu a.\, t$ is a *recursive* type expression in which the type variable a is bound in the type expression t. Intuitively, $\mu a.\, t$ is the solution of the equation $a = t$. We will use $[s/a]t$ for substitution. The *raw* terms of the language include (term) variables x, applications $d(e)$ and lambda abstractions $\lambda x{:}t.\, e$. An expression $\{l_1 = e_1, \ldots, l_n = e_n\}$ is called a record with *fields* $l_1, \ldots, l_n$ and the expression $e.l$ is the *selection* of the field l. Again, we assume that the order of the fields of a record is irrelevant, but the labels must all be distinct. We also have bounded type abstraction $\Lambda a \leq t.\, e$ and the corresponding application $e(t)$. To form terms of recursive type $\mu a.t$ we have *intro* expressions $\mathsf{intro}[\mu a.t]e$ and they are eliminated from the recursion by *elim* expressions $\mathsf{elim}\ e$. See Appendix A to find a grammar for the type expressions and raw terms of the fragment.

Raw terms are type-checked by deriving *typing judgements*, of the form $\Gamma \vdash e : t$. where Γ is a context. *Contexts* are defined recursively as follows: $\emptyset$ is a context; if Γ is a context which does not declare a, and the free variables of t are declared in Γ, then $\Gamma, a \leq t$ is a context; if Γ is a context which does not declare x, and the free variables of t are declared in Γ, then $\Gamma, x{:}t$ is a context. The proof system for deriving typing judgments is the relevant fragment of the corresponding proof system for Fun (see [CW85] on pages 519–520) enriched with two type-checking rules for the introduction and elimination of recursive types [CGW87]. A complete list of these proof rules is in

Appendix A under the heading **Fragment.**

Among these proof rules, the following two illustrate the effect of inheritance on type-checking:

[INH] $$\frac{\Gamma \vdash e : s \qquad \widehat{\Gamma} \vdash s \leq t}{\Gamma \vdash e : t}$$

[B-SPEC] $$\frac{\Gamma \vdash e : \forall a {\leq} s.\, t \qquad \widehat{\Gamma} \vdash r \leq s}{\Gamma \vdash e(r) : [r/a]t}$$

They make use of *inheritance judgements* which have the form $C \vdash s \leq t$ where C is an inheritance context. *Inheritance contexts* are contexts in which only declarations of the form $a \leq t$ appear. If Γ is a context, we denote-by $\widehat{\Gamma}$ teh inheritance context obtained from Γ by erasing the declarations of the form $x{:}t$. The proof system for deriving inheritance judgments is, with the exception of one rule, the same as the relevant fragment of the corresponding proof system for Fun (see [CW85], on page 519). In this paper we do not attempt to enrich it with any rule deriving inheritances *between* recursive types. A discussion of this issue appears in our conclusions. The Appendix contains a complete list of these proof rules too.

In comparison with Fun, we would like to strengthen the rule deriving inheritances between bounded generics, and we are able to do so for some of our results. Where Fun had just

(W-FORALL) $$\frac{C, a{\leq}t \vdash u \leq v}{C \vdash \forall a{\leq}t.\, u \leq \forall a{\leq}t.\, v}$$

we will consider

(FORALL) $$\frac{C \vdash s \leq t \qquad C, a{\leq}s \vdash u \leq v}{C \vdash \forall a{\leq}t.\, u \leq \forall a{\leq}s.\, v}$$

This makes the system strictly stronger, allowing more inheritances to be derived, and thus more terms to type-check.

Originally, we believed that coherence could be proved for a system that includes variants and the stronger rule (FORALL) [BCGS89]. In dealing with the case construct for variant types, however, our coherence proof uses an order-theoretic property (see Lemma 11) which fails for the stronger system for deriving inheritances that uses (FORALL) (for a counterexample, see Giorgio Gelli's dissertation [Ghe90]). Thus, we prove the coherence of the translation of variants (Theorem 13) only for the weaker system with (W-FORALL). Note, however, that we prove coherence in the presence of (FORALL) for the system without variants (Theorem 4) and for the system for deriving inheritances between types, including variant types (Lemma 9).

Remark. Decidability of type-checking in the stronger system is a non-trivial question. The question whether an algorithm of Luca Cardelli will decide the provability of judgements in this calculus has only recently been settled by Ghelli [Ghe90].

The salient feature of bringing inheritance into a type system is that (in given contexts) terms will *not* have a unique type any more. For example, due to the rule

(TOP) $$C \vdash t \leq Top$$

where the free variables of t are declared in C, by [INH], all terms that type-check with some type will also type-check with type *Top*. This makes it possible to define ordinary generics as syntactic sugar: $\forall a.\, t \overset{\text{def}}{=} \forall a{\leq} Top.\, t$.

The proof system for **SOURCE**, while quite intuitive, allows for the following complication: there may be more than one derivation of the same typing judgement. In fact, we only need record types, (RECD), [VAR], [SEL] and [INH] (see Appendix) to provide such an example: in the context $x\colon\{l_1\colon Top, l_2\colon Top\}$, we can either directly derive by [SEL] $x.l_1 : Top$, or we can derive by [VAR] $x : \{l_1\colon Top, l_2\colon Top\}$, then by (RECD) and [INH] $x : \{l_1\colon Top\}$, and finally by [SEL] $x.l_1 : Top$. In view of this, for any semantics given by "induction on the rules", one needs to prove that derivations of the same judgement have the same meaning.

The target calculus. As mentioned before, **TARGET** is the Girard-Reynolds polymorphic lambda calculus, enriched with record and recursive types [CGW87, BC88, CGW89]. Here, we present it as a simplification of **SOURCE**. Types are given by

$$a \mid s \to t \mid \forall a.\, t \mid \{l_1\colon s_1, \ldots, l_n\colon s_n\} \mid \mu a.\, t$$

and terms by

$$x \mid M(N) \mid \lambda x\colon t.\, M \mid \Lambda a.\, M \mid M(t) \mid \{l_1 = M_1, \ldots, l_n = M_n\} \mid M.l \mid \mathsf{intro}[\mu a.\, t]M \mid \mathsf{elim}\ M$$

For $n = 0$ we get the the *empty record type* $1 \overset{\text{def}}{=} \{\}$ and the *empty record*, for which we will keep the notation $\{\}$. *Typing contexts* are the obvious simplification of contexts in which only typing judgements occur (there is no inheritance relation in **TARGET**). The rules for deriving typing judgements in the fragment of **TARGET** discussed in this section can be found in Appendix B. The following is a well-known fact:

Proposition 1 *In* **TARGET**, *derivations of typing judgements are unique.*

Proof: All the "elimination" rules, [APPL], [SEL], [SPEC], and [R-ELIM] are "cut" rules, in the sense that there is information in the premisses that does not appear in the conclusion. Consequently, they should in principle cause problems for the uniqueness of derivations. However, the lost information is always in the type part, and types "should" be unique. This suggests the strengthening of the induction hypothesis, which then passes trivially through these "cut" rules.

One proves therefore that for any two derivations Δ_1 and Δ_2, if Δ_1 ends in $\Upsilon \vdash M : t_1$ and Δ_2 ends in $\Upsilon \vdash M : t_2$ then $\Delta_1 \equiv \Delta_2$ (in particular, $t_1 \equiv t_2$).

The proof can be done straightforwardly, either by induction on the maximum of the heights of Δ_1 and Δ_2, or on the sum of those heights, or even on the structure of M (with a bit of reformulation). ▮

A technical point: it turns out that type decorations are unnecessary on "elimination" constructs, but they are in fact necessary on some "introduction" constructs, such as lambda abstraction and the recursive type construct intro[]. Later on, with the addition of variants in section 4, we will find that we need to differ with [CW85], and decorate with types the constructs that "inject" into variant types (see Appendix B).

Equations are derived by a proof system (see [CGW87, BC88, CGW89]) which contains rules like reflexivity, symmetry, transitivity, congruence with respect to function application, closure under functional abstraction (ξ), congruence with respect to application to types, closure with respect to type abstraction (type ξ). There are also the {BETA} and {ETA} rules for both functional and type abstraction, rules saying that intro[ ] and elim are inverse to each other, as well as

{RECD-BETA} $\qquad \{l_1 = M_1, \ldots, l_n = M_n\}.l_i \;=\; M_i$

where $n \geq 1$, and

{RECD-ETA}. $$\{l_1 = M.l_1, \ldots, l_n = M.l_n\} \; = \; M$$

where $M : \{l_1 : s_1, \ldots, l_n : s_n\}$ .The last rule gives, for $n = 0$, the equation $\{\} = M$ which makes 1 into a terminator. Under our interpretation, the type *Top* will be nothing like a "universal domain" which can be used to interpret *Type:Type* [CGW89, GJ90]. On the contrary, it will be interpreted as a one point domain in the models we list below!

The translation. For any **SOURCE** `item` we will denote by `item`* its translation into **TARGET**. We begin with the types. Note the translation of bounded generics and of *Top*.

$$\begin{array}{rcl@{\qquad}rcl}
a^* & \overset{\text{def}}{=} & a & \{l_1 : s_1, \ldots, l_n : s_n\}^* & \overset{\text{def}}{=} & \{l_1 : s_1^*, \ldots, l_n : s_n^*\} \\
Top^* & \overset{\text{def}}{=} & \mathbf{1} & (\forall a \leq s.\, t)^* & \overset{\text{def}}{=} & \forall a.\, (a \to s^*) \to t^* \\
(s \to t)^* & \overset{\text{def}}{=} & s^* \to t^* & (\mu a.\, t)^* & \overset{\text{def}}{=} & \mu a.\, t^*
\end{array}$$

One shows immediately that $([s/a]t)^* \equiv [s^*/a]t^*$. We extend this to contexts and inheritance contexts, which translate into just typing contexts in **TARGET**.

$$\begin{array}{rcl@{\qquad}rcl}
\emptyset^* & \overset{\text{def}}{=} & \emptyset & \emptyset^* & \overset{\text{def}}{=} & \emptyset \\
(\Gamma, a \leq t)^* & \overset{\text{def}}{=} & \Gamma^*, a, f : a \to t^* & (C, a \leq t)^* & \overset{\text{def}}{=} & C^*, a, f : a \to t^* \\
 & & & (\Gamma, x : t)^* & \overset{\text{def}}{=} & \Gamma^*, x : t^*
\end{array}$$

where f is a *fresh* variable for each a.

Next we will describe how we translate the derivations of judgments of **SOURCE**. The translation is defined by recursion on the structure of the derivation trees. Since these are freely generated by the derivation rules, it is sufficient to provide for each derivation rule of **SOURCE** a corresponding rule on trees of **TARGET** judgments. It will be a lemma (Lemma 2 to be precise) that these corresponding rules are *directly derivable* in **TARGET**, therefore the translation takes derivations in **SOURCE** into derivations in **TARGET**.

A **SOURCE** derivation yielding an inheritance judgment $C \vdash s \leq t$ is translated as a tree of **TARGET** judgments yielding $C^* \vdash P : s^* \to t^*$. We present three of the rules here; the full list for the fragment appears in Appendix C. The coercion into *Top* is simply the constant map:

(TOP)* $$C^* \vdash \lambda x : t^*.\, \{\} : t^* \to \mathbf{1}$$

To see how coercion works on types, assume that we are given a coercion $P : s \to t$ from s into t and a coercion $Q : u \to v$ from u into v. Then it is possible to coerce a function $f : t \to u$ into a function from s to v as follows. Given an argument of type s, coerce it (using P) into an argument of type t. Apply the function f to get a value of type u. Now coerce this value in u into a value in v by applying Q. This describes a function of the desired type. More formally, we translate the (ARROW) rule by

(ARROW)* $$\frac{C^* \vdash P : s^* \to t^* \qquad C^* \vdash Q : u^* \to v^*}{C^* \vdash R : (t^* \to u^*) \to (s^* \to v^*)}$$

where $R \overset{\text{def}}{=} \lambda z : t^* \to u^*.\, P; z; Q$. (We use ; as shorthand for *composition*. For example, $P; z; Q$ above stands for $\lambda x : s^*.\, Q(z(P(x)))$ where x is fresh.) Now, to translate the rule (FORALL) which describes the inheritance relation for the bounded quantification we view the quantification as ranging over a type together with a coercion from that type into the bound:

$$\text{(FORALL)}^* \qquad \frac{C^* \vdash P : s^* \to t^* \qquad C, a, f{:}\, a \to s^* \vdash Q : u^* \to v^*}{C^* \vdash R : (\forall a.\,(a \to t^*) \to u^*) \to (\forall a.\,(a \to s^*) \to v^*)}$$

where $R \stackrel{\text{def}}{=} \lambda z{:}\,(\forall a.\,(a \to t^*) \to u^*).\,\Lambda a.\,\lambda f{:}\,a \to s^*.\,Q(z(a)(f;P))$

Now, a **SOURCE** derivation yielding an typing judgment $\Gamma \vdash e : t$ is translated as a tree of **TARGET** judgments yielding $\Gamma^* \vdash M : t^*$. For example, the inheritance rule is translated by simply making the inheritance coercion "explicit":

$$[\text{INH}]^* \qquad \frac{\Gamma^* \vdash M : s^* \qquad \widehat{\Gamma}^* \vdash P : s^* \to t^*}{\Gamma^* \vdash P(M) : t^*}$$

The specialization of a bounded quantification is more subtle. The variable is instantiated by substituting the type expression to which the abstraction is applied, but then the coercion from the argument type to the bound type must be passed as an argument to the resulting function:

$$[\text{B-SPEC}]^* \qquad \frac{\Gamma^* \vdash M : \forall a.\,(a \to s^*) \to t^* \qquad \widehat{\Gamma}^* \vdash P : r^* \to s^*}{\Gamma^* \vdash M(r^*)(P) : [r^*/a]t^*}$$

The remaining rules for translating the fragment are given in Appendix C. It is possible to check that the translated rules are derivable in the target language:

Lemma 2 *The rules* (TOP)* – (TRANS)* *and* [VAR]* – [INH]* *are directly derivable in* **TARGET.** ▌

Coherence of the translation. For any derivation Δ in **SOURCE**, let Δ^* be the **TARGET** derivation into which it is translated. The central result about *inheritance* judgements says that, given a judgement $s \leq t$ and a pair of proofs Δ_1 and Δ_2 of this judgement, the coercions induced by these two proofs are provably equal in the equational theory of **TARGET**. More formally, we have the following:

Lemma 3 (Coherence of the translation of inheritance) *Let Δ_1 and Δ_2 be two* **SOURCE** *derivations of the same inheritance judgement, $C \vdash s \leq t$. Let Δ_1^*, Δ_2^* yield (coercion) terms P_1, P_2. Then, $P_1 = P_2$ is provable in* **TARGET**.

The central result about *typing* judgements says that, given a judgement $e{:}\,t$ and a pair of proofs Δ_1 and Δ_2 of this judgement, the translations of these proofs end in sequents (translations of $e{:}\,t$) which are provably equal in the equational theory of **TARGET**, *i.e.* we have:

Theorem 4 (Coherence) *Let Δ_1 and Δ_2 be two* **SOURCE** *derivations yielding the same typing judgement, $\Gamma \vdash e : t$. Let Δ_1^*, Δ_2^* yield terms M_1, M_2. Then, $M_1 = M_2$ is provable in* **TARGET**.

The proofs of the lemma and theorem are almost as difficult as the ones we shall give for the corresponding results in the full language. Since the proofs of these results for the fragment follow similar lines to the proofs for the full language we omit the proofs of Lemma 3 and Theorem 4 in favor of the proofs of Lemma 9 and Theorem 13 below.

4 Between incoherence and inconsistency: adding variants

The calculus described so far does not deal with a crucial type constructor: variants. In particular, it is very useful to have a combination of variant types with recursive types. On the other hand, the combination of these operators in the same calculus is also problematic, especially for the equational theory. The situation is familiar from both domain theory and proof theory. In this section we propose an approach which will suffice to prove the coherence theorem which we need to show that our semantic function is well-defined.

We extend the type formation rules of **SOURCE** by adding *variant* type expressions: $[l_1{:}\,t_1,\ldots,l_n{:}\,t_n]$ where $n \geq 1$. We also extend the term formation rule by the formation of variant terms $[l_1{:}\,t_1,\ldots,l_i{=}e,\ldots,l_n{:}\,t_n]$ and the *case statement:*

$$\mathsf{case}\ e\ \mathsf{of}\ l_1 \Rightarrow f_1,\ldots,l_n \Rightarrow f_n$$

The inheritance judgement derivation rules are extended correspondingly with the rule:

$$\text{(VART)} \qquad \frac{C \vdash s_1 \leq t_1 \quad \cdots \quad C \vdash s_p \leq t_p}{C \vdash [l_1{:}\,s_1,\ldots,l_p{:}\,s_p] \leq [l_1{:}\,t_1,\ldots,l_p{:}\,t_p,\ldots,l_q{:}\,t_q]}$$

Note the "duality" between this rule and the inheritance rule (RECD) for records (see Appendix A). While a record subtype has more fields, a variant subtype has fewer variations (summands).

Like before, we intend to translate this calculus into a calculus without inheritance and, naturally, we extend **TARGET** with variants (see Appendix B). Note how the syntax of variant injections differs from [CW85]. This is in order for the resulting system to enjoy the property of having unique type derivations: the proof of Proposition 1 extends immediately to the variant constructs. Most importantly, we must extend the equational theory of **TARGET** in a manner that insures the coherence of our translation. It is here that we encounter an interesting problem which readers who know domain theory will find familiar. The following two axioms hold in a variety of models:

$$\{\text{VART-BETA}\} \qquad \mathsf{case}\ \mathsf{inj}_{l_i}(M_i)\ \mathsf{of}\ l_1 \Rightarrow F_1,\ldots,l_n \Rightarrow F_n \ = \ F_i(M_i)$$

where $F_1 : t_1 \rightarrow t,\ldots,F_n : t_n \rightarrow t,\ M_i : t_i$ and inj_{l_i} is shorthand for $\lambda x{:}\,t_i.\,[l_1{:}\,t_1,\ldots,l_i{=}x,\ldots,l_n{:}\,t_n]$.

$$\{\text{VART-ETA}\} \qquad \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow \mathsf{inj}_{l_1},\ldots,l_n \Rightarrow \mathsf{inj}_{l_n} \ = \ M$$

where $M : [l_1{:}\,t_1,\ldots,l_n{:}\,t_n]$. Unfortunately, these two axioms do not suffice to prove all the identifications required by the coherence of our translation!

To see the problem, we start with an example. In **SOURCE**, suppose that $t \leq s$ is derivable in the context $\widehat{\Gamma}$, and that we have a derivation Δ of $\Gamma \vdash e : [l_1{:}\,t_1, l_2{:}\,t_2]$ and derivations Δ_i of $\Gamma \vdash f_i : t_i \rightarrow t,\ i = 1,2$. Consider then the following two **SOURCE** derivations of the typing judgement $\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ l_1 \Rightarrow f_1, l_2 \Rightarrow f_2 : s$.

1. by Δ, Δ_1, Δ_2 and the rule [CASE], one deduces $\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ l_1 \Rightarrow f_1, l_2 \Rightarrow f_2 : t$. Since $\widehat{\Gamma} \vdash t \leq s$ by hypothesis, one infers by inheritance $\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ l_1 \Rightarrow f_1, l_2 \Rightarrow f_2 : s$.

2. from $\widehat{\Gamma} \vdash t \leq s$ we can deduce $\widehat{\Gamma} \vdash (t_i \rightarrow t) \leq (t_i \rightarrow s)$. Hence, by inheritance from Δ_i, one deduces $\Gamma \vdash f_i : t_i \rightarrow s$. Then, from Δ and by the rule [CASE], one deduces $\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ l_1 \Rightarrow f_1, l_2 \Rightarrow f_2 : s$.

The coherence property requires that these two derivations have provably equal translations. With the obvious translation for the variant type constructor and the rules [VART] and [CASE] (see Appendix C) and with the translation of the rules [INH], (ARROW) and (REFL) as in Section 3, this comes down to the following identity

$$P(\mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1, l_2 \Rightarrow F_2) \ = \ \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow (F_1; P), l_2 \Rightarrow (F_2; P)$$

where $P : t^* \rightarrow s^*$ is a "coercion term", $M : [l_1{:}\,t_1^*, l_2{:}\,t_2^*]$, $F_i : t_i^* \rightarrow t^*$, $i = 1, 2$. Thus, we are tempted to postulate

$$\{\text{VART-CRN?}\} \qquad P(\mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) \ = \ \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1; P, \ldots, l_n \Rightarrow F_n; P$$

where $M{:}\,[l_1{:}\,t_1, \ldots, l_n{:}\,t_n]$, $F_1{:}\,t_1 \rightarrow t, \ldots, F_n{:}\,t_n \rightarrow t$, $P{:}\,t \rightarrow s$. This equation follows from the equation that axiomatizes variants analogously to coproducts:

$$\{\text{VART-COP?}\} \qquad Q(M) \ = \ \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow (\mathsf{inj}_{l_1}; Q), \ldots, l_n \Rightarrow (\mathsf{inj}_{l_n}; Q)$$

where $M{:}\,[l_1{:}\,t_1, \ldots, l_n{:}\,t_n]$, $Q{:}\,[l_1{:}\,t_1, \ldots, l_n{:}\,t_n] \rightarrow t$. More precisely, it is possible to check that the system {VART-BETA}+{VART-COP} is equivalent to {VART-BETA}+{VART-CRN}+{VART-ETA}. However, it is known [Law69, HP89a] that {VART-BETA}+{VART-COP} is inconsistent with the existence of fixed-points. In fact, this may be refined:

Proposition 5 *The system* {VART-BETA}+{VART-CRN} *is (equationally) inconsistent with the existence of fixed-points.*

Proof: The "categorical" equation { VART-COP } may be thought of as an "induction" principle on a sum: it reduces the proof of an equation $P(M) = Q(M)$, $M{:}\,[l_1{:}\,t_1, l_2{:}\,t_2]$, to the proofs of $P(\mathsf{inj}_{l_1}(x)) = Q(\mathsf{inj}_{l_1}(x))$, for $x{:}\,t_1$ and $P(\mathsf{inj}_{l_2}(x)) = Q(\mathsf{inj}_{l_2}(x))$, for $x{:}\,t_2$. Indeed, we have $P(M) = \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow \lambda x.\ P(\mathsf{inj}_{l_1}(x)), l_2 \Rightarrow \lambda x.\ P(\mathsf{inj}_{l_2}(x))$ and $Q(M) = \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow \lambda x.\ Q(\mathsf{inj}_{l_1}(x)), l_2 \Rightarrow \lambda x.\ Q(\mathsf{inj}_{l_2}(x))$. Given a type t, it is possible to define a "negation-like" operation on $[l_1{:}\,t, l_2{:}\,t]$ by $\mathsf{neg}(M) = \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow \lambda x.\mathsf{inj}_{l_2}(x), l_2 \Rightarrow \lambda x.\mathsf{inj}_{l_1}(x)$. Given $x, y{:}\,t$, it is easy enough to define an operation $f(M,N){:}\,t$, for $M, N{:}\,[l_1{:}\,t, l_2{:}\,t]$ in such a way that $f(\mathsf{inj}_{l_1}(u), \mathsf{inj}_{l_1}(u)) = f(\mathsf{inj}_{l_2}(v), \mathsf{inj}_{l_2}(v)) = x$, and $f(\mathsf{inj}_{l_1}(u), \mathsf{inj}_{l_2}(v)) = f(\mathsf{inj}_{l_2}(v), \mathsf{inj}_{l_1}(u)) = y$. We deduce then from the "induction principle" that $f(M, M) = x$, and $f(M, \mathsf{neg}(M)) = y$, identically for $M{:}\,[l_1{:}\,t, l_2{:}\,t]$, hence the (equational) inconsistency when we have a fixed-point combinator.

The fact that we can use instead of {VART-COP?} + {VART-BETA} the weaker system {VART-BETA} + {VART-CRN?} comes simply from the fact that we can "relativise" this reasoning to the elements of $[l_1{:}\,t, l_2{:}\,t]$ of the form $\mathsf{case}\ M\ \mathsf{of}\ \mathsf{inj}_{l_1} \mathsf{inj}_{l_2}$, elements that satisfy the equation { VART-ETA }. ▮

Thus, a naive approach gives us an unattractive choice between incoherence and inconsistency! We are saved from this by the observation that, at least in the example above, we do not seem to need the "full" usage of {VART-CRN} but only those instances in which P is a term coming out of a translation of an inheritance judgement, *i.e.*, a "coercion term". Such terms are much simpler than general terms. In particular, we note that in models based on continuous maps, such terms denote *strict* maps, and in models based on stable maps, they denote *linear* maps. Appropriate constructions for interpreting variants can be given in both cases, such that {VART-CRN} is sound, as long as P ranges only over strict (or linear) maps.

Maintaining the same philosophy to our approach as in Section 3 we will try to *abstractly* embody in **TARGET** a sufficient amount of formalism to insure the provable coherence of our translation. Thus, the previous discussion of variants leads us to introduce a new type constructor $s \multimap t$, the type of "coercions" from s to t. Consequently, the coercion assumptions $a \leq t$ that occur in inheritance contexts must translate to variables ranging over types of coercions $f\colon a \multimap t^*$. As a consequence, the translation of bounded quantification must change:

$$(\forall a \leq s.\, t)^* \stackrel{\text{def}}{=} \forall a.\, ((a \multimap s^*) \to t^*)$$

In order to express the correct versions of {VART-CRN}, we introduce a family of constants in **TARGET**

$$\iota_{s,t} : (s \multimap t) \to (s \to t)$$

called *coercion-coercion combinators.* With this, we have

$$\{\text{VART-CRN}\}\quad \iota(P)(\mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) \;=\; \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1; \iota(P), \ldots, l_n \Rightarrow F_n; \iota(P)$$

$$\text{where}\quad M\colon [l_1\colon t_1, \ldots, l_n\colon t_n],\ F_1\colon t_1 \to t, \ldots, F_n\colon t_n \to t,\ P\colon t \multimap s\,.$$

(the complete list is in Appendix B).

In order to translate all inheritance judgements into coercion terms, we add a special set of constants (coercion combinators) that "compute" the translations of the rules for deriving inheritance judgements. To prove coherence, we axiomatize the behavior of the ι-images of these combinators. For example, the coercion combinator for the rule (ARROW) takes a pair of coercions as arguments and yields a new coercion as value:

$$\mathsf{arrow}[s,t,u,v] : (s \multimap t) \to (u \multimap v) \to ((t \to u) \multimap (s \to v))$$

Since (ARROW) is a rule *scheme*, we naturally have a *family* of such combinators, indexed by types. To simplify the notation, these types will be omitted whenever possible. The equational property of the arrow combinator is given in terms of the coercion coercer:

$$\iota(\mathsf{arrow}(P)(Q)) \;=\; \lambda z\colon t \to u.\, (\iota(P)); z; (\iota(Q))$$

where $P\colon s \multimap t,\ Q\colon u \multimap v$. For the rule (TRANS), we introduce

$$\mathsf{trans}[r,s,t] : (r \multimap s) \to (s \multimap t) \to (r \multimap t)$$

which, of course, behaves like composition, modulo the coercion coercer:

$$\iota(\mathsf{trans}(P)(Q)) \;=\; \iota(P); \iota(Q)$$

where $P\colon r \multimap s,\ Q\colon s \multimap t$. The combinator for the rule (FORALL) is the most involved:

$$\mathsf{forall}[s,t,a,u,v] : (s \multimap t) \to \forall a.\, ((a \multimap s) \to (u \multimap v)) \to (\forall a.\, ((a \multimap t) \to u) \multimap \forall a.\, ((a \multimap s) \to v))$$

with the equational axiomatization

$$\iota(\mathsf{forall}(P)(W)) \;=\; \lambda z\colon (\forall a.\, (a \multimap t) \to u).\, \Lambda a.\, \lambda f\colon a \multimap s.\, \iota(W(a)(f))(z(a)(\mathsf{trans}(f)(P)))$$

where $P\colon s \multimap t,\ W\colon \forall a.\, (a \multimap s) \to (u \multimap v)$. Of course, we have gone to the extra inconvenience of introducing the type of coercions in order to provide a satisfactory account of variants. These require a scheme of combinators having the types:

$$\mathsf{vart}[s_1, \ldots, s_p, t_1, \ldots, t_q] : (s_1 \multimap t_1) \to \cdots \to (s_p \multimap t_p) \to ([l_1\colon s_1, \ldots, l_p\colon s_p] \multimap [l_1\colon t_1, \ldots, l_p\colon t_p, \ldots, l_q\colon t_q])$$

And it is now possible to assert a consistent equation for these combinators:

$$\iota(\mathsf{vart}(R_1)\cdots(R_p)) \;=\; \lambda w\colon [l_1\colon s_1,\ldots,l_p\colon s_p].\ \mathsf{case}\ w\ \mathsf{of}\ l_1 \Rightarrow \iota(R_1);\mathsf{inj}_{l_1},\ldots,l_p \Rightarrow \iota(R_p);\mathsf{inj}_{l_p}$$

where $R_1\colon s_1 \multimap t_1,\ldots,R_p\colon s_p \multimap t_p$. In order to prove equalities between terms of coercion type one uses the following rule:

$$\{\text{IOTA-INJ}\} \qquad \frac{\iota(P) \;=\; \iota(Q)}{P \;=\; Q}$$

which asserts that ι is an injection. In fact, all of the models we give below will interpret ι as an inclusion. It is natural to ask whether the coercion coercer ι could have been omitted from the calculus in favor of a rule:

$$\frac{P\colon s \multimap t}{P\colon s \to t}.$$

This would have the unfortunate consequence that a typing judgement $e\colon s$ would no longer uniquely encode its proof and the coherence question would therefore arise again! The other combinators and their equational properties are described in Appendix B.

We are now ready to explain how to translate our full language **SOURCE** (complete with variants) into the language **TARGET** (with the coercion coercer and combinators). For starters, the inheritance judgement for the function space is simply translated using the arrow combinator:

$$(\text{ARROW})^* \qquad \frac{C^* \vdash P : s^* \multimap t^* \qquad C^* \vdash Q : u^* \multimap v^*}{C^* \vdash \mathsf{arrow}(P)(Q) : (t^* \to u^*) \multimap (s^* \to v^*)}$$

The translation of an inheritance between quantified types takes the induced coercion and a polymorphic function as its arguments:

$$(\text{FORALL})^* \qquad \frac{C^* \vdash P : s^* \multimap t^* \qquad C^*, a, f\colon a \multimap s^* \vdash Q : u^* \multimap v^*}{C^* \vdash \mathsf{forall}(P)(\Lambda a.\, \lambda f\colon a \multimap s^*.\, Q) : \forall a.\, ((a \multimap t^*) \to u^*) \multimap \forall a.\, ((a \multimap s^*) \to v^*)}$$

Other inheritance judgements are similarly translated. The real work is being done by equational properties of the combinators.

The proofs of typing judgements are translated in a manner quite similar to how they were translated in the fragment. For example,

$$[\text{B-SPEC}]^* \qquad \frac{\Gamma^* \vdash M : \forall a.\, ((a \multimap s^*) \to t^*) \qquad \widehat{\Gamma}^* \vdash P : r^* \multimap s^*}{\Gamma^* \vdash M(r^*)(P) : [r^*/a]t^*}$$

is affected only by indicating that the map into the bound must be a coercion. The inheritance rule is translated by

$$[\text{INH}]^* \qquad \frac{\Gamma^* \vdash M : s^* \qquad \widehat{\Gamma}^* \vdash P : s^* \multimap t^*}{\Gamma^* \vdash \iota(P)(M) : t^*}$$

since a coercion cannot be applied until it is made into a function by an application of the coercion coercer. The full description of the translation of the full language is given in Appendix C. We now turn to the proof of the central technical results of the paper.

5 Coherence of the translation for the full calculus

In this section we prove first the coherence of the translation of inheritance judgements. This result is then used to show the coherence of the translation of typing judgements.

The main cause for having distinct derivations of the same inheritance judgements is the rule (TRANS). Our strategy is to show that the usage of (TRANS) can be coherently postponed to the end of derivations (Lemma 6), and then to prove the coherence of the translation of (TRANS)-postponed derivations (Lemma 8).

We introduce some convenient notations for the rest of this section. For any derivation Δ in **SOURCE**, let Δ^* be the **TARGET** derivation into which it is translated. We will write $C \vdash r_0 \leq \cdots \leq r_n$ instead of $C \vdash r_0 \leq r_1 ,\ldots, C \vdash r_{n-1} \leq r_n$. The composition of coercions given by trans occurs so often that we will write $P \odot Q$ instead of $\mathsf{trans}(P)(Q)$. It is easy to see, making essential use of the rule {IOTA-INJ}, that $\odot$ is provably *associative.* We will take advantage of this to unclutter the notation. We will also write I instead of refl . Again it is easy to see that I is provably an identity for $\odot$, that is, $I \odot M = M \odot I = M$ is provable in **TARGET**.

Lemma 6 *For any* **SOURCE** *derivation Δ yielding the inheritance judgement $C \vdash s \leq t$, there exist types $r_0,\ldots,r_n$ such that $s \equiv r_0$, $r_n \equiv t$, and (TRANS)-free derivations $\Delta_1,\ldots,\Delta_n$ yielding respectively*

$$C \vdash r_0 \leq \cdots \leq r_n$$

Moreover, if the translations $\Delta^*, \Delta_1^*, \ldots, \Delta_n^*$ yield respectively the (coercion) terms $C^* \vdash P : s^* \multimap t^*$, $C^* \vdash P_1 : r_0^* \multimap r_1^* ,\ldots,$ $C^* \vdash P_n : r_{n-1}^* \multimap r_n^*$ then

$$C^* \vdash P = P_1 \odot \cdots \odot P_n$$

is provable in **TARGET**.

Proof: By induction on the height of the derivation Δ. The base is trivial since derivations consisting of instances of (TOP), (VAR), or (REFL) are already (TRANS)-free. We present the more interesting cases of the induction step.

Suppose Δ ends with an application of (ARROW). By induction hypothesis there are (TRANS)-free derivations for

$$s \equiv r_0 \leq \cdots \leq r_m \equiv t \quad \text{and} \quad u \equiv w_0 \leq \cdots \leq w_n \equiv v$$

(for simplicity, we omit the context). From these, using (REFL) and (ARROW) we get (TRANS)-free derivations for

$$t \to u \equiv r_m \to u \leq \cdots \leq r_0 \to u \equiv s \to w_0 \leq \cdots \leq s \to w_n \equiv s \to v \ .$$

(This is not most economical: one can get a derivation requiring only $\max(m,n)$, rather than $m+n$, steps of (TRANS) at the end.) Proving the equality of the corresponding translations uses the associativity of $\odot$ and the fact that I acts like an identity, as well as

$$\text{(1)} \qquad \mathsf{arrow}(P)(Q) \odot \mathsf{arrow}(R)(S) = \mathsf{arrow}(R \odot P)(Q \odot S)$$

which can be verified, in view of {IOTA-INJ}, by applying ι to both sides, resulting in a simple {BETA}-conversion.

Suppose Δ ends with an application of (FORALL). By induction hypothesis there are (TRANS)-free derivations for

$$C \vdash s \equiv r_0 \leq \cdots \leq r_m \equiv t \quad \text{and} \quad C, a \leq s \vdash u \equiv w_0 \leq \cdots \leq w_n \equiv v$$

From these, using (REFL) and (FORALL) we get (TRANS)-free derivations for

$$C \vdash \forall a \leq t.u \equiv \forall a \leq r_m.u \leq \cdots \leq \forall a \leq r_0.u \equiv \forall a \leq s.u \equiv \forall a \leq s.w_0 \leq \cdots \leq \forall a \leq s.w_n \equiv \forall a \leq s.v \ .$$

Proving the equality of the corresponding translations uses

(2)
$$\begin{aligned}\mathsf{forall}(P)(\Lambda a.\, \lambda f\colon a \multimap s.\, Q) \odot \mathsf{forall}(R)(\Lambda a.\, \lambda g\colon a \multimap t.\, S) &= \\ = \mathsf{forall}(R \odot P)(\Lambda a.\, \lambda g\colon a \multimap t.\, [g \odot R/f]Q \odot S)&\end{aligned}$$

and which can be verified by applying ι to both sides.

Suppose Δ ends with an application of (VART). By induction hypothesis there are (TRANS)-free derivations for

$$\begin{gathered} s_1 \equiv r_0^1 \leq \cdots \leq r_{n_1}^1 \equiv t_1 \\ \vdots \\ s_p \equiv r_0^p \leq \cdots \leq r_{n_p}^p \equiv t_p \end{gathered}$$

(for simplicity, we omit the context). From these, using (REFL) and (VART) we get (TRANS)-free derivations for

$$\begin{gathered}[l_1\colon s_1, \ldots, l_p\colon s_p] \equiv [l_1\colon r_0^1, \ldots, l_p\colon s_p] \leq \cdots \leq [l_1\colon r_{n_1}^1, \ldots, l_p\colon s_p] \leq \cdots \leq [l_1\colon r_{n_1}^1, \ldots, l_p\colon r_0^p] \leq \cdots \\ \cdots \leq [l_1\colon r_{n_1}^1, \ldots, l_p\colon r_{n_p}^p] \equiv [l_1\colon t_1, \ldots, l_p\colon t_p] \leq [l_1\colon t_1, \ldots, l_p\colon t_p, \ldots, l_q\colon t_q] \ .\end{gathered}$$

Proving the equality of the corresponding translations uses

(3)
$$\mathsf{vart}(P_1)\cdots(P_p) \odot \mathsf{vart}(Q_1)\cdots(Q_q) = \mathsf{vart}(P_1 \odot Q_1)\cdots(P_p \odot Q_p) \quad (p \leq q).$$

To verify this, let L be the left hand side of the equation, R the right hand side and let w be a fresh variable. By extensionality (or {ETA} and {XI}) and by {IOTA-INJ}, it is sufficient to show $\iota(L)(w) = \iota(R)(w)$. By {VART-COP}, this follows from

$$\mathsf{case}\ w\ \mathsf{of}\ l_1 \Rightarrow (\mathsf{inj}_{l_1}; \iota(L)), \ldots, l_p \Rightarrow (\mathsf{inj}_{l_p}; \iota(L)) = \mathsf{case}\ w\ \mathsf{of}\ l_1 \Rightarrow (\mathsf{inj}_{l_1}; \iota(R)), \ldots, l_p \Rightarrow (\mathsf{inj}_{l_p}; \iota(R))$$

which is readily verified.

When Δ ends with (TRANS), we just concatenate the chains of (TRANS)-free derivations and the equality of the translations is an immediate consequence of the associativity of $\odot$. ∎

The following is used to handle one of the cases in Lemma 8 below.

Lemma 7 *For any two derivations, Δ yielding $C \vdash s \leq t$ and Θ yielding $C, a \leq t \vdash u \leq v$, there exists a derivation Σ yielding $C, a \leq s \vdash u \leq v$ such that height$(\Sigma) =$ $\max($height(Δ), height$(\Theta))$. Moreover, if the translations $\Delta^*, \Theta^*, \Sigma^*$ yield respectively*

$$C^* \vdash P : s^* \multimap t^* \ , \ C^*, a, g\colon a \multimap t^* \vdash Q : u^* \multimap v^* \ , \ C^*, a, f\colon a \multimap s^* \vdash R : u^* \multimap v^*$$

then

$$C^*, a, f\colon a \multimap s^* \vdash R = [f \odot P/g]Q$$

is provable in TARGET.

Proof: By induction on the height of Θ. ▮

Lemma 8 *Let $\Delta_1, \ldots, \Delta_m$ be (TRANS)-free derivations in* **SOURCE** *yielding respectively $C \vdash s_0 \leq \cdots \leq s_m$ and $\Theta_1, \ldots, \Theta_n$ be (TRANS)-free derivations yielding respectively $C \vdash t_0 \leq \cdots \leq t_n$. Let the translations $\Delta_1^*, \ldots, \Delta_m^*, \Theta_1^*, \ldots, \Theta_n^*$ yield respectively the (coercion) terms*

$$C^* \vdash P_1 : s_0^* \multimap s_1^* , \ldots , C^* \vdash P_m : s_{m-1}^* \multimap s_m^* , C^* \vdash Q_1 : t_0^* \multimap t_1^* , \ldots , C^* \vdash Q_n : t_{n-1}^* \multimap t_n^* .$$

If $s_0 \equiv t_0$ and $s_m \equiv t_n$ then

$$C^* \vdash P_1 \odot \cdots \odot P_m = Q_1 \odot \cdots \odot Q_n$$

is provable in **TARGET**.

Proof: We begin with the following remarks:

- If one of $s_0, \ldots, s_m, t_0, \ldots, t_n$ is *Top* then the desired equality holds. Indeed, then $s_m \equiv Top \equiv t_n$ and the equality follows from the identity

  (4) $$P \leq \mathsf{top}$$

  which is verified by applying ι to both sides (recall that **1** is a terminator).

- Those derivations among $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$ which consist entirely of one application of (REFL) can be eliminated without loss of generality. Indeed, the corresponding coercion term is I which acts as an identity for $\odot$.

- If none of the derivations among $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$ consists of just (TOP), then those derivations which consist of just (VAR) can also be eliminated without loss of generality. Indeed, once we have eliminated the (REFL)'s, the (VAR)'s must form an initial segment of both $\Delta_1, \ldots, \Delta_m$ and $\Theta_1, \ldots, \Theta_n$ because whenever $s \leq a$ is derivable, s must also be a type variable. Let's say that $s_0 \equiv a_0, \ldots, s_p \equiv a_{p-1}$, $(p \leq m)$, where $\Delta_1, \ldots, \Delta_p$ are *all* the derivations consisting of just (VAR), and also that $t_0 \equiv b_0, \ldots, t_q \equiv b_{q-1}$, $(q \leq n)$, where $\Theta_1, \ldots, \Theta_q$ are all the derivations consisting of just of (VAR). Then, $a_0 \leq a_1, \ldots, a_{p-1} \leq s_p$ as well as $b_0 \leq b_1, \ldots, b_{q-1} \leq t_q$ must all occur in C. But $a_0 \equiv s_0 \equiv t_0 \equiv b_0$ so by the uniqueness of declarations in contexts, $a_1 \equiv b_1, \ldots$, *etc.* Suppose $p < q$. Then, $s_p \equiv b_p$ is a variable. Since Δ_{p+1} can't be just a (REFL) or a (TOP) is must be a (VAR) contradicting the maximality of p. Thus $p = q$ and $s_p \equiv t_q$ and the (VAR)'s can be eliminated.

We proceed to prove the lemma by induction on the maximum of the heights of the derivations $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$. The basis of the induction is an immediate consequence of the remarks above.

For the induction step, in the view of the remarks above, we can assume without loss of generality that none of the derivations is just a (TOP), (VAR), or (REFL). Consequently, $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$ must all end with the same rule, depending on the type construction used in $s_0 \equiv t_0$.

If all derivations end in (ARROW), the desired equality follows from the induction hypothesis, the associativity of $\odot$ and the equation (1). Similarly for (VART) using the equation (3). The desired equality in the case (FORALL) follows from the induction hypothesis using Lemma 7, from the associativity of $\odot$ and from the equation (2). The remaining cases are straight-forward. ▮

This gives us the coherence of the translation of inheritance judgements. To state it we need some terminology. We say that two **SOURCE** derivations which yield the same judgement are *congruent* if their translations in **TARGET** yield provably equal terms. We will write $\Delta_1 \cong \Delta_2$ for congruence of derivations. It is easy to check that $\cong$ *is* in fact a congruence with respect to the operations on derivations induced by the rules.

Lemma 9 (Coherence of the translation of inheritance) *If Δ_1 and Δ_2 are two* **SOURCE** *derivations yielding the same inheritance judgement then $\Delta_1 \cong \Delta_2$ (their translations yield provably equal terms in* **TARGET***).*

Proof: Immediate consequence of Lemmas 6 and 8 ▌

Before we turn to the coherence of the translation of typing judgements, we will note a few facts about inheritance judgements that follow from Lemma 6 and that will be invoked subsequently. These facts are closely related to the remarks opening the proof of Lemma 8.

Remark 10 *If $C \vdash s \leq t$ is derivable, $s \equiv a$, a type variable, and $t \not\equiv a$ then*

- *if $t \equiv b$, also a type variable, there must exist type variables $a_0, \ldots, a_n$, $n \geq 1$ such that $a \equiv a_0$, $b \equiv a_n$, and $a_{i-1} \leq a_i \in C$, $i = 1, \ldots, n$;*
- *if t is not a type variable, there must exist type variables $a_0, \ldots, a_n$, $n \geq 0$ and a type u such that $a \equiv a_0$, $a_{i-1} \leq a_i \in C$, $i = 1, \ldots, n$, $a_n \leq u \in C$, and $C \vdash u \leq t$ (of course, this is trivial when $t \equiv Top$);*

If $C \vdash s \leq t$ is derivable, and s is not a type variable variable, then t cannot be a type variable, and if moreover $t \not\equiv Top$, then s and t must both have the "same" outermost type constructor (as detailed exhaustively below) and

- *if $s \equiv s_1 \rightarrow s_2$ and $t \equiv t_1 \rightarrow t_2$ then $C \vdash t_1 \leq s_1$ and $C \vdash s_2 \leq t_2$;*
- *if $s \equiv \{l_1: s_1, \ldots, l_q: s_q\}$ and $t \equiv \{l_1: t_1, \ldots, l_p: t_p\}$ then $p \leq q$ and $C \vdash s_1 \leq t_1, \ldots, C \vdash s_p \leq t_p$;*
- *if $s \equiv \forall a \leq s_1 . s_2$ and $t \equiv \forall a \leq t_1 . t_2$ then $C \vdash t_1 \leq s_1$ and $C, a \leq t_1 \vdash s_2 \leq t_2$;*
- *if s and t are both recursive types then they must be identical;*
- *if $s \equiv [l_1: s_1, \ldots, l_p: s_p]$ and $t \equiv [l_1: t_1, \ldots, l_q: t_q]$ then $p \leq q$ and $C \vdash s_1 \leq t_1, \ldots, C \vdash s_p \leq t_p$.*

We turn now to the coherence of the translation of typing judgements, which is the central technical result of the paper. As explained in section 3, we weaken the system by replacing the rule (FORALL) with (W-FORALL) (see Appendix A). With this, we have the following order-theoretic property about the inheritance judgments, which fails in the presence of (FORALL). The property asserts the existence of conditional greatest lower bounds and of least upper bounds.

Lemma 11 *Replace (FORALL) with (W-FORALL). Let C be an inheritance context and let t_1, t_2 be types.*

1. *If there is an r with $C \vdash r \leq t_i$, $(i = 1, 2)$, then there exists a type $t_1 \sqcap t_2$ such that*

- *$C \vdash t_1 \sqcap t_2 \leq t_i$, $(i = 1, 2)$ and*
- *for any s such that $C \vdash s \leq t_i$, $(i = 1, 2)$ we have $C \vdash s \leq t_1 \sqcap t_2$.* ∎

2. There is a type $t_1 \sqcup t_2$ such that

- *$C \vdash t_i \leq t_1 \sqcup t_2$, $(i = 1, 2)$ and*
- *for any s such that $C \vdash t_i \leq s$, $(i = 1, 2)$ we have $C \vdash t_1 \sqcup t_2 \leq s$.* ∎

Proof: Because of the contravariance property of the first argument of the function space operator manifest in the rule (ARROW), we will prove items 1 and 2 simultaneously. In view of Lemma 6, it is sufficient to work with proofs where all instances of (TRANS) appear at the end. Since moreover any two types have a common upper bound, *Top*, the statement of the lemma is equivalent to the following formulation:

For any $\Delta_1, \ldots, \Delta_m$, (TRANS)-free derivations in **SOURCE** *yielding respectively $C \vdash u_0 \leq \cdots \leq u_m$ and any $\Theta_1, \ldots, \Theta_n$, (TRANS)-free derivations yielding respectively $C \vdash v_0 \leq \cdots \leq v_n$,*

1. *if $u_0 \equiv v_0$, and let $t_1 \equiv u_m$ and $t_2 \equiv v_n$, then there is a type $t_1 \sqcap t_2$ having the properties in item 1 of the lemma;*
2. *if $u_m \equiv v_n$, and let $t_1 \equiv u_0$ and $t_2 \equiv v_0$, then there is a type $t_1 \sqcup t_2$ having the properties in item 2 of the lemma.*

This is shown by induction on the maximum of m, n and of the heights of $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$. To be able to apply the induction hypothesis, a case analysis is performed, depending on the structure of t_1 and t_2. We will only look at a few illustrative cases. The facts listed in Remark 10 and the reasoning that produced these facts as well as the remarks opening the proof of Lemma 8 are used throughout.

For example, if t_1 is a type variable in item 1, then u_i is also a type variable for each i, and $u_{i-1} \leq u_i \in C$, $i = 1, \ldots, n$. Then, one of $C \vdash u_0 \leq \cdots \leq u_m$ or $C \vdash v_0 \leq \cdots \leq v_n$, must be an initial segment of the other, so t_1 and t_2 are comparable and $t_1 \sqcap t_2$ can be taken as the smaller among them. For item 2, if t_1 is a type variable, then $u_0 \leq u_1 \in C$ and, by induction hypothesis (m decreases), $t_1 \sqcup t_2$ can be taken to be $u_1 \sqcup t_2$.

As another example, suppose that in item 1 t_1 has the form $\forall a \leq s.\, r_1$. If $u_0 \equiv v_0$ is a type variable, then $u_0 \leq u_1 \in C$ and $v_0 \leq v_1 \in C$ hence $u_1 \equiv v_1$ and we can apply the induction hypothesis by eliminating Δ_1, Θ_1. Assume that $u_0 \equiv v_0$ is not a type variable. By Remark 10 (simplified to take into account the weakening of (FORALL)), it must have the form $\forall a \leq s.\, r$. Again by Remark 10 t_2 is either *Top* or has the form $\forall a \leq s.\, r_2$. If $t_2 \equiv Top$ then $t_1 \sqcap t_2$ can be taken to be t_1. Otherwise, there are (TRANS)-free derivations $\Delta'_1, \ldots, \Delta'_m$ yielding $C, a \leq s \vdash u'_0 \leq \cdots \leq u'_m$ and $\Theta'_1, \ldots, \Theta'_n$ yielding respectively $C, a \leq u \vdash v'_0 \leq \cdots \leq v'_n$ where $u'_0 \equiv v'_0$ and $u'_m \equiv r_1$ and $v'_n \equiv r_2$, and where each of these derivations has strictly smaller height than the corresponding one among $\Delta_1, \ldots, \Delta_m, \Theta_1, \ldots, \Theta_n$. By induction hypothesis we get a type $r_1 \sqcap r_2$, and we can then take $t_1 \sqcap t_2$ to be $\forall a \leq s.\, r_1 \sqcap r_2$. This calculation makes clear where our proof breaks down if we were to use the more general rule (FORALL) instead of (W-FORALL). Indeed, if the bounds on the type variables were allowed to differ, as in the more general case, we would be unable to apply the induction hypothesis since the two contexts would differ between the Θ's and the Δ's.

We omit the remaining cases, which use similar ideas. ∎

We will use this property in the proof of Lemma 12, which is a slightly stronger result than the actual coherence of the translation of typing judgements. Of course, the strengthening is exploited in a proof by induction. First we introduce a definition and more convenient notations. For derivations yielding typing judgements we define the *essential height* which is computed as the usual height, with the proviso that [INH] and the rules yielding inheritance judgements do *not* increase it. We will also use a special notation for describing "composition" of derivations via the rules. We explain this notation through two examples. If Σ yields $\Gamma \vdash e : s$ and Θ yields $\widehat{\Gamma} \vdash s \leq t$, then [INH]$\langle \Sigma , \Theta\rangle$ yields $\Gamma \vdash e : t$. If Δ yields $\Gamma, x{:}s \vdash e : t$ then [ABS]$\langle \Delta \rangle$ yields $\Gamma \vdash \lambda x{:}s.e : s{\rightarrow}t$.

In preparation for the proof of the next lemma, we have two remarks.

- We have the following congruence
$$[\mathrm{INH}]\langle [\mathrm{INH}]\langle \Sigma , \Theta_1\rangle , \Theta_2\rangle \cong [\mathrm{INH}]\langle \Sigma , (\mathrm{TRANS})\langle \Theta_1 , \Theta_2\rangle\rangle \ .$$
This follows from the fact that $\iota(Q)(\iota(P)(M)) \ = \ \iota(P \odot Q)(M)$ which is immediately verified.

- Any **SOURCE** derivation is congruent to a derivation of the form [INH]$\langle \Delta , \Theta\rangle$ where Δ does *not* end with an application of the [INH] rule. This follows from the previous remark and, in the case when the original derivation did not end in [INH], from
$$\Delta \cong [\mathrm{INH}]\langle \Delta , (\mathrm{REFL})\rangle$$
which in turn follows from $M \ = \ \iota(I)(M)$.

Lemma 12 *Replace (FORALL) with (W-FORALL). For any two* **SOURCE** *derivations,* Δ_i *yielding* $\Gamma \vdash e : t_i$, $(i = 1,2)$, *there exists a type s, a derivation* Σ *yielding* $\Gamma \vdash e : s$ *and two derivations* Θ_i *yielding* $\widehat{\Gamma} \vdash s \leq t_i$, $(i = 1,2)$ *such that*
$$\Delta_i \cong [\mathrm{INH}]\langle \Sigma , \Theta_i\rangle , \ \ (i = 1,2).$$

Proof: By induction on the maximum of the essential heights of Δ_1, Δ_2. In view of the previous remarks, it is sufficient to prove the statement of the lemma assuming that neither Δ_1 nor Δ_2 ends in [INH] (but we retain the actual statement of the lemma in the induction hypothesis). For such derivations, Δ_1 and Δ_2 must end with the same rule (which rule, depends on the structure of e). We do a case analysis according to this last rule, and we include here only the cases which we believe are important for the understanding of the result (even if their treatment is straightforward) as well as some cases which are particularly complex. We will call the type s, whose existence is the essence of the result, the *common type.*

Rule [VAR]. It must be the case that $t_1 \equiv t_2 \equiv r$ where $x{:}r$ occurs in Γ. Consequently, the treatment of this rule is trivial: take the common type to be r, $\Sigma \equiv$ [VAR], and $\Theta_1 \equiv \Theta_2 \equiv$ (REFL).

The introduction rules are quite simple and we illustrate them with the **rule** [ABS]. Suppose that $\Delta_i \equiv$ [ABS]$\langle \Delta_i' \rangle$ and that Δ_i yields $\Gamma \vdash \lambda x{:}s.e : s{\rightarrow}t_i$ (s is the same since it appears in the term), thus Δ_i' yields $\Gamma, x{:}s \vdash e : t_i$, $(i = 1,2)$. Apply the induction hypothesis to Δ_1', Δ_2' obtaining $r, \Sigma', \Theta_1', \Theta_2'$. Also by induction hypothesis,
$$\Delta_i \cong [\mathrm{ABS}]\langle [\mathrm{INH}]\langle \Sigma', \Theta_i'\rangle \rangle , \ \ (i = 1,2).$$
We claim that the right hand side is congruent to
$$[\mathrm{INH}]\langle [\mathrm{ABS}]\langle \Sigma' \rangle , (\mathrm{ARROW})\langle (\mathrm{REFL}) , \Theta_i'\rangle\rangle \ .$$

This implies that the statement of the lemma holds for Δ_1, Δ_2, with common type $s \rightarrow r$, with $\Sigma \equiv [\text{ABS}]\langle \Sigma' \rangle$, and with $\Theta_i \equiv (\text{ARROW})\langle(\text{REFL}), \Theta_i'\rangle$, $(i = 1,2)$. The congruence claim follows from

$$\lambda x{:}\, s.\, \iota(P)(M) \;=\; \iota(\mathsf{arrow}(I)(P)(\lambda x{:}\, s.\, M)$$

which is readily verified.

Rule[B-SPEC]. To simplify the notation, we omit the contexts. Suppose that $\Delta_i \equiv [\text{B}-\text{SPEC}]\langle \Delta_i', \Xi_i\rangle$ and that Δ_i yields $e(r) : [r/a]t_i$ (r is the same since it appears in the term and we can take the bound variable to be the same without loss of generality), thus Δ_i' yields $e : \forall a \leq s_i.\, t_i$ and Ξ_i yields $r \leq s_i$, $(i = 1,2)$. Apply the induction hypothesis to Δ_1', Δ_2' obtaining $w, \Sigma', \Theta_1', \Theta_2'$. Also by induction hypothesis,

(5) $$\Delta_i \cong [\text{B}-\text{SPEC}]\langle [\text{INH}]\langle \Sigma', \Theta_i'\rangle, \Xi_i\rangle\,, \quad (i = 1,2).$$

Since $w \leq \forall a \leq s_i.\, t_i$, $(i = 1,2)$ it follows from Remark 10 (simplified to take into account the weakening of (FORALL)) that there must exist types u, v such that $s_i \equiv u$, $a \leq s_i \vdash v \leq t_i$, $(i = 1,2)$ and $w \leq \forall a \leq u.\, v$ are derivable. It follows that $r \leq u$, and, by Lemma 7, that $a \leq r \vdash v \leq t_i$, $(i = 1,2)$ are derivable. Next, we will use the following sublemma:

> **Sublemma** For any derivation Δ yielding $C, a \leq r \vdash s \leq t$ there exists a derivation Σ yielding $C \vdash [r/a]s \leq [r/a]t$ such that, if the translations Δ^*, Σ^* yield respectively
>
> $$C^*,\, a,\, f{:}\, a \multimap r^* \vdash P : s^* \multimap t^*\;,\; C^* \vdash Q : [r^*/a]s^* \multimap [r^*/a]t^*$$
>
> then
>
> $$C^* \vdash Q \;=\; (\Lambda a.\, \lambda f{:}\, a \multimap r^*.\, P)(r^*)(I)$$
>
> is provable in **TARGET.** ▮

The sublemma is proved by induction on the height of Δ and is omitted. The sublemma allows us to obtain $[r/a]v \leq [r/a]t_i$ from $a \leq r \vdash v \leq t_i$, $(i = 1,2)$. Let Θ_i be some derivation of $[r/a]v \leq [r/a]t_i$, $(i = 1,2)$. Let Ξ be some derivation of $r \leq u$. Let Ω be some derivation of $w \leq \forall a \leq u.\, v$. One can readily verify that the right hand side of (5) is congruent to

$$[\text{INH}]\langle [\text{B}-\text{SPEC}]\langle [\text{INH}]\langle \Sigma', \Omega\rangle, \Xi\rangle, \Theta_i\rangle$$

This implies that the statement of the lemma holds for Δ_1, Δ_2, with common type $[r/a]v$, with $\Sigma \equiv [\text{B}-\text{SPEC}]\langle [\text{INH}]\langle \Sigma', \Omega\rangle, \Xi\rangle$, and with Θ_i being just Θ_i, $(i = 1,2)$. (**Note.** There is no difficulty in dealing with (FORALL) instead of (W-FORALL) here: $s_i \equiv u$ would be simply replaced by $s_i \leq u$.)

Rule[R-ELIM]. Suppose that $\Delta_i \equiv [\text{R}-\text{ELIM}]\langle \Delta_i'\rangle$ and that Δ_i yields $\Gamma \vdash \mathsf{elim}\; e : [\mu a_i.\, t_i/a_i]t_i$, thus Δ_i' yields $\Gamma \vdash e : \mu a_i.\, t_i$, $(i = 1,2)$;. Apply the induction hypothesis to Δ_1', Δ_2' obtaining $s', \Sigma', \Theta_1', \Theta_2'$. Also by induction hypothesis,

$$\Delta_i \cong [\text{R}-\text{ELIM}]\langle [\text{INH}]\langle \Sigma', \Theta_i'\rangle\rangle\,, \quad (i = 1,2).$$

Since $s' \leq \mu a_i.\, t_i$, $(i = 1,2)$ are derivable, it follows from Remark 10 that there must exist a, t such that $\mu a_i.\, t_i \equiv \mu a.\, t$, $(i = 1,2)$ and $s' \leq \mu a.\, t$ are derivable. Let Θ' be any derivation of $s' \leq \mu a.\, t$. Since by Lemma 9, $\Theta_1' \cong \Theta_2' \cong \Theta'$, the statement of the lemma holds with common type $[\mu a.\, t/a]t$, with $\Sigma \equiv [\text{R}-\text{ELIM}]\langle [\text{INH}]\langle \Sigma', \Theta'\rangle\rangle$, and with $\Theta_i \equiv (\text{REFL})$, $(i = 1,2)$.

Rule[CASE]. Again, to simplify the notation, we omit the contexts. Suppose that $\Delta_i \equiv$ [CASE]$\langle \Delta'_i, \Delta'_{1i}, \ldots, \Delta'_{ni}\rangle$ and that Δ_i yields $\mathsf{case}\ e\ \mathsf{of}\ l_1 \Rightarrow f_1, \ldots, l_n \Rightarrow f_n : t_i$, thus Δ'_i yields $e : [l_1\colon t_{1i}, \ldots, l_n\colon t_{ni}]$, and Δ'_{ji} yield $f_j : t_{ji} \to t_i$, $(j = 1, \ldots, n)$, $(i = 1,2)$. Apply the induction hypothesis to Δ'_1, Δ'_2 obtaining $s, \Sigma', \Theta'_1, \Theta'_2$. Also apply the induction hypothesis, to $\Delta'_{j1}, \Delta'_{j2}$ obtaining $s_j, \Sigma'_j, \Theta'_{j1}, \Theta'_{j2}$, $(j = 1, \ldots, n)$. By induction hypothesis,

(6) $\quad \Delta_i \cong$ [CASE]$\langle$[INH]$\langle \Sigma', \Theta'_i\rangle$, [INH]$\langle \Sigma'_1, \Theta'_{1i}\rangle, \ldots,$ [INH]$\langle \Sigma'_n, \Theta'_{ni}\rangle\rangle$, $(i = 1,2)$.

Since $s \leq [l_1\colon t_{1i}, \ldots, l_n\colon t_{ni}]$, $(i = 1,2)$ are derivable, it follows again from Remark 10 that there must exist $m \leq n$ and types $r_1, \ldots, r_m$ such that $r_1 \leq t_{1i}, \ldots, r_m \leq t_{mi}$, $(i = 1,2)$ and $s \leq [l_1\colon r_1, \ldots, l_m\colon r_m]$ are derivable. Again similarly, for each of $j = 1, \ldots, n,$, since $s_j \leq t_{ji} \to t_i$, $(i = 1,2)$ are derivable, there must exist u_j, v_j such that $t_{ji} \leq u_j$ and $v_j \leq t_i$, $(i = 1,2)$ as well as $s_j \leq u_j \to v_j$ are derivable. Thus, we can derive $r_j \leq t_{ji} \leq u_j$,$(j = 1, \ldots, n)$, $(i = 1,2)$. However, the fact that the v_j's may be distinct causes a problem when we want to apply [CASE]. This is resolved by Lemma 11. Since $n \geq 1$, there exists a common lower bound of t_1 and t_2 (say v_1) hence $v \equiv t_1 \sqcap t_2$ exists and we can derive $v_j \leq v \leq t_i$,$(j = 1, \ldots, n)$, $(i = 1,2)$. We conclude that there exists a derivation Θ'' of $s \leq [l_1\colon u_1, \ldots, l_n\colon u_n]$, that there exist derivations Θ''_j of $s_j \leq u_j \to v$, $(j = 1, \ldots, n)$ and that there exist derivations Θ_i of $v \leq t_i$, $(i = 1,2)$. With these, we claim that the right hand side of (6) is congruent to

$$\text{[INH]}\langle \text{[CASE]}\langle \text{[INH]}\langle \Sigma', \Theta''\rangle, \text{[INH]}\langle \Sigma'_1, \Theta''_1\rangle, \ldots, \text{[INH]}\langle \Sigma'_n, \Theta''_n\rangle\rangle, \Theta_i\rangle ,$$

This implies that the statement of the lemma holds for Δ_1, Δ_2, with common type v , with $\Sigma \equiv$ [CASE]$\langle$[INH]$\langle \Sigma', \Theta''\rangle$, [INH]$\langle \Sigma'_1, \Theta''_1\rangle, \ldots,$ [INH]$\langle \Sigma'_n, \Theta''_n\rangle\rangle$, and with Θ_i being just Θ_i, $(i = 1,2)$.

To prove the congruence claim we introduce notations for certain derivations of inheritance judgements whose existence we have established. For each $j = 1, \ldots, n$, $i = 1,2$, let Ξ_{ji} be some derivation for $t_{ji} \leq u_j$. Then, (ARROW)$\langle \Xi_{ji}, \Theta_i\rangle$ is a derivation for $u_j \to v \leq t_{ji} \to t_i$. By Lemma 9 we have

(7) $$\Theta'_{ji} \cong \text{(TRANS)}\langle \Theta''_j, \text{(ARROW)}\langle \Xi_{ji}, \Theta_i\rangle\rangle$$

Let Ξ be some derivation of $s \leq [l_1\colon r_1, \ldots, l_m\colon r_m]$. For each $j = 1, \ldots, m$, $i = 1,2$, let Ω_{ji} be some derivation for $r_j \leq t_{ji}$. By Lemma 9 we have

(8) $$\Theta'_i \cong \text{(TRANS)}\langle \Xi, \text{(VART)}\langle \Omega_{1i}, \ldots, \Omega_{mi}\rangle\rangle$$

and

(9) $$\Theta'' \cong \text{(TRANS)}\langle \Xi, \text{(TRANS)}\langle \text{(VART)}\langle \Omega_{1i}, \ldots, \Omega_{mi}\rangle, \text{(VART)}\langle \Xi_{1i}, \ldots, \Xi_{ni}\rangle\rangle\rangle .$$

With these, the congruence claim follows from

$$\mathsf{case}\ \iota(P\odot\mathsf{vart}(Q_1)\cdots(Q_m))(M)\ \mathsf{of}\ l_1 \Rightarrow \iota(R_1\odot\mathsf{arrow}(S_1)(T))(F_1), \ldots, l_n \Rightarrow \iota(R_n\odot\mathsf{arrow}(S_n)(T))(F_n) =$$

$$= \iota(T)(\mathsf{case}\ \iota(P\odot\mathsf{vart}(Q_1)\cdots(Q_m)\odot\mathsf{vart}(S_1)\cdots(S_n))(M)\ \mathsf{of}\ l_1 \Rightarrow \iota(R_1)(F_1), \ldots, l_n \Rightarrow \iota(R_n)(F_n)) .$$

By (3) and {VART-CRN} the right hand side equals

$$\mathsf{case}\ \iota(P \odot \mathsf{vart}(Q_1 \odot S_1)\cdots(Q_m \odot S_m))(M)\ \mathsf{of}\ l_1 \Rightarrow \iota(R_1)(F_1); \iota(T), \ldots, l_n \Rightarrow \iota(R_n)(F_n); \iota(T)$$

and the equality is readily verified. ▌

Theorem 13 (Coherence) *Replace (FORALL) with (W-FORALL). If Δ_1 and Δ_2 are two* **SOURCE** *derivations yielding the same typing judgement then $\Delta_1 \cong \Delta_2$ (their translations yield provably equal terms in* **TARGET***).*

Proof: Take $t_1 \equiv t_2$ in Lemma 12. By Lemma 9, $\Theta_1 \cong \Theta_2$. It follows that $\Delta_1 \cong \Delta_2$. ▮

6 Models

So far we have not actually given a model for the language **SOURCE**. In this section we correct this omission. However, it is a central point of this paper that there is *basically nothing new that we need to do in this section,* since calculi satisfying the equational theory of **TARGET** have been thoroughly studied in the literature on the semantics of type systems. Domain-theoretic semantics suggests natural candidates for a special class of maps with the properties needed to interpret the operators $\rightarrow$ and $\multimap$. Here we present list some of these semantic solutions; all of which apply to abstract types as well as to variants. A syntactic version could also be given by a syntactic translation into an extension of the target calculus of section 2, which expresses the properties mentioned above and the consistency of which is ensured by our semantic considerations.

The domain-theoretic interpretations that we have examined so far are summarized in the following table. The necessary properties for all but the last row can be found in [TT87, HP89b], [CGW89],[ABL86], [CGW87], and [Gir87] respectively. The properties needed for the last row can be checked in a manner similar to [Gir87].

TYPES	TERMS	COERCIONS	VARIANTS
Algebraic lattices		bistrict maps	sep sum of lattices
Scott domains	continuous maps	strict maps	
Finitary projections			separated sums
dI domains		strict stable maps	
coherent spaces	stable maps	linear maps	$!A \oplus !B$
dI domains			

By a bistrict map of lattices we mean a continuous map which preserves both bottom and top elements. A separated sum of lattices L and M is the disjoint sum of L and M together with new top and bottom elements. Note that the category of Scott domains (finitary projections, respectively) and strict maps does have finite coproducts, given by coalesced sums of domains, and this implies that the required equation

$$\{\text{VART-CRN?}\} \qquad P(\textsf{case}\ M\ \textsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) \ = \ \textsf{case}\ M\ \textsf{of}\ l_1 \Rightarrow F_1; P, \ldots, l_n \Rightarrow F_n; P$$

holds if P is a strict map (in fact, a separated sum of domains A and B is just the coalesced sum of the lifted domains $A_\perp$ and $B_\perp$). Furthermore, it may be checked that strictness is preserved by the formation of coercion maps from given ones according to the coercion rules given in section 3 and at the beginning of this section. This model satisfies also {VART-BETA}+{VART-ETA}. An important property used in the case of Scott domains (finitary projections, respectively) is that the continuous maps from C to D are in one-to-one correspondence with the strict maps from $C_\perp$ to D. Analogous remarks hold for stable maps and linear maps, with $!C$ instead of $C_\perp$ (see [Gir89], Chapter 8).

From a category-theoretic point of view, the main point is that we are dealing with *two categories,* one a reflective subcategory of the other, i.e. the inclusion functor has a left adjoint. The subcategory contains all objects of the larger category. While the larger category is cartesian closed, the reflective subcategory (in which our coercions live) does have coproducts.

From a proof-theoretic point of view, it is interesting to note that our solution is similar to the treatment of proof-theoretic commutation rules for disjunction (see [Tro73], 4.1.3, on page 279 for a presentation of commutation rules). The so-called commutation rules for sums in proof theory are closely related to the equations {VART-CRN?} where P is an "evaluation" map (see the Appendix B of [Gir88]).

7 Conclusions and directions for further investigation

The development of calculi for the representation of inheritance polymorphism and the semantics of such calculi is a growing and dynamic area of research investigation in programming languages. We expect that the calculi considered in this paper are only a small sample of what is yet to be developed. In this section we will speculate on a few of the most important directions for further development which will play a significant role in future work of the authors of this paper in particular and the research community in general.

Partial Equivalence Relations. Much of the research on the semantics of the system which we have considered has been based on the use of PER's as described by Bruce and Longo [BL88]. It is therefore worthwhile to compare the approach in this paper to this alternative approach. There is an evident means of carrying out a technical comparison: since the PER model interprets the calculus **TARGET**, it also interprets **SOURCE** via our translation. But the semantics in [BL88] gives the interpretation (without recursion) directly using PER's. Could these two interpretations be the same? For a certain fragment of **SOURCE** (including recursion but not bounded quantification), Cardone has recently answered the question in the affirmative for his form of semantics [Car89b] (where coherence is not an issue because the interpretation of a judgement $e{:}\,s$ is given as the equivalence class, in s, of the interpretation of the erasure of e—hence the meaning is not defined inductively on a derivation). For the full calculus the answer is still unknown as this paper is being written. Amadio's thesis contains some results about the relationship between explicit coercions and PER inclusion [Ama91].

Equational Theory. The reader has probably noted that we have never offered an equational theory for **SOURCE**, only one for **TARGET**. At the current time, the proper equational theory for **SOURCE** is still a subject of active research. However, our translation does suggest an equational theory. One can prove that two terms of **SOURCE** are equal by showing that their translations are equivalent in the equational theory for **TARGET**. Any of the models we have proposed will satisfy the resulting equational theory. (Whether this is also true of the interpretation of [BL88] may follow if this interpretation is the same as ours.) Since our translation is computable, it follows that this reflected equational theory for **SOURCE** is recursively enumerable; it is natural to ask for a reasonable axiomatization of this theory. Note, for example, if $e = e'{:}\,s$ holds in **SOURCE** and $s \le t$, then $e = e'{:}\,t$ also holds in the reflected theory. There are probably many similarly interesting derived equational rules.

Recursion Any attempt to provide a model for a calculus which combines inheritance and recursion must deal with the seemingly contradictory semantic characteristics of inheritance and recursion at higher types. Ordinarily, the rule for inheritance between exponentials (function spaces) is given as follows:

$$\frac{u \le s \qquad t \le v}{s \to t \;\le\; u \to v}$$

where s, t, u, v are type expressions and $\leq$ is the relation of inheritance (reading $s \leq t$ as "s inherits from t"). Note, in particular, the *contra*variance in the first argument of the $\rightarrow$ operator. In contrast, semantic domains which solve recursive domain equations such as $D = D \rightarrow D$ are generally constructed using a technique—adjoint pairs to be precise—which make it possible to "order" types using a concept of approximation based on the rule

$$\frac{\phi : s \rightarrow u \qquad \psi : t \rightarrow v}{\phi \rightarrow \psi : (s \rightarrow t) \rightarrow (u \rightarrow v)}$$

where $\phi = \langle \phi^L, \phi^R \rangle$ and $\psi = \langle \psi^L, \psi^R \rangle$ are adjoint pairs and $\phi \rightarrow \psi$ is the adjoint pair $\langle \lambda f.\ \psi^L \circ f \circ \phi^R,\ \lambda f.\ \psi^R \circ f \circ \phi^L \rangle$. Note, for this case, the *co*variance in the first argument of the $\rightarrow$ operator. Because of this difference, models such as the PER interpretation of Bruce and Longo [BL88], which provides a semantics for inheritance and parametric polymorphism, do not evidently extend to a semantics for recursive types. To provide for recursive types under this interpretation M. Coppo and M. Zacchi [Cop85, CZ86] utilize an appeal to the structure of the underlying universal domain, which is itself an inverse limit which solves a recursive equation. R. Amadio [Ama89, Ama90] and F. Cardone [Car89b] have explored this approach in considerable detail. There has also been progress on understanding the solution of recursive equations over domains internally to the PER model which should provide further insights [FMRS89, Fre89]. On the other hand, models such as those of Girard [Gir86] and Coquand, Gunter and Winskel [CGW87, CGW89], which handle parametric polymorphism and recursive types, do not provide an evident interpretation for inheritance. It has been the purpose of this paper to resolve this problem by an appeal to the paradigm of "inheritance and implicit coercion". However, this leaves open the question of how recursive types can be treated with this technique if one is to include a more powerful set of rules for deriving inheritance judgements between recursive types.

One complicating problem is to decide exactly what form of inheritance between recursive types is desired. For example, it seems very reasonable that if s is a subtype of t then the type of lists of s's should be a subtype of lists of t's. This is not actually derivable in the inheritance system described in this paper since there are no rules for inheritance between recursive types. But care must be taken: if s is a subtype if t then is the solution of the equations $a = a \rightarrow s$ be a subtype of the solution of $a = a \rightarrow t$? There are several possible approaches to answering this question. The PER interpretation provides a good guide: we can ask whether the solutions of these two equations have the desired relation in the PER model. Concerning the coercions approach we are forced to ask whether there is any intuitive coercion between these two types. If there is, we have not seen it! It is reasonable to conjecture that inheritance relations derived using the following rule will be acceptable:

$$\text{(REC)} \qquad \frac{C, a \leq Top \vdash s \leq t}{C \vdash \mu a.\, s \leq \mu a.\, t}$$

where types s and t have only *positive* occurrences of the variable a. Unfortunately, this misses many interesting inheritance relations that one would like to settle. Discussions of this problem will appear in several future publications on this subject. A rather satisfactory treatment using coercions has been described in [BGS89] by using the "Amber rule" of Cardelli [Car86].

Operational semantics. Despite its importance there is virtually no literature on theoretical issues concerning the operational semantics of languages with inheritance polymorphism. In particular, at the time we are writing there are no published discussions of the relationship (if any!) of the denotational models which have been studied to the intended operational semantics of a programming language based on the models. In fact, the operational semantics of no existing "practical" programming language is based on the kind of semantics discussed in this or any of the other papers on the semantics of Fun. This is because there is a divergence between the "traditional" style of semantics for the λ-calculus and the way the evaluation mechanisms of modern functional programming languages actually work. In particular, no functional programming language in common use evaluates past a lambda abstraction. Hence the identification of the constantly divergent function with the divergent element will cause the denotational semantics to fail to be computationally adequate with respect to the evaluation. Another related problem concerns the use of the β-rule and call-by-value evaluation. Many of the functional programming languages now in use evaluate all actual function parameters. This evaluation strategy immediately causes the full β-rule to fail. For example, the application of a constant function to a divergent argument will diverge in general. Semantically, this means that terms of higher type must be interpreted as *strict* functions. In a subsequent paper [BGS90], three of the authors of the current document have explored the operational semantics of inheritance with a coercion semantics in a call-by-value setting. The results there are intuitively pleasing, but there is much more that needs to be done. This direction of investigation offers several opportunities for practical applications of the specification and implementation of compilers and interpreters for new languages with inheritance.

Existentials. We have omitted discussion of existentials in this paper. We believe that the coherence results we have described will extend to a suitable interpretation of the existential types using the equational theory for weak sums, but did not choose to involve ourselves in additional cases that this would mean for our proofs.

Order-sorted algebra. The use of coercions in a first-order setting has been investigated in work of J. A. Goguen, J-P. Jouannaud and J. Meseguer on order-sorted algebras [GJM85, GM]. In particular, the implementation of OBJ2 utilized a form of "inheritance as implicit coercion" approach. Related work by Bruce and Wegner appears in [BW90].

Abstract coherence. Since there are many different calculi for which a coherence theorem is interesting, it is very useful to have a more abstract theory from which special instances of coherence can be derived, thus making coherence a more routine part of a semantic theory for an inheritance calculus such as the one we have discussed. We mentioned earlier that coherence was an issue in category theory and this might provide a framework for a more general theory. (Although, the results on coherence in the category theory literature are insufficient for the results of this paper so further extensions will be needed). Using rewriting techniques, Curien and Ghelli have developed a type-theoretic approach to the abstract coherence problem for $F_{\leq}$ which is a subsystem of **SOURCE** featuring only function and bounded generic types [CG90]. It would be interesting to see this technique extended to all of **SOURCE**, especially in view of the complications we encountered with variants.

Subtyping of bounded quantification. Our main coherence result was proved for a weaker version of the system, one that uses the rule (W-FORALL) instead of (FORALL) (see Appendix A). We believe that this is only a technical restriction that arose from our particular proof, and that coherence holds for the stronger system. A proof would however require a way to circumvent the usage of Lemma 11 in the treatment of the [CASE] rule in Lemma 12, since Lemma 11 fails when (FORALL) is postulated (for a counterexample, see Giorgio Gelli's dissertation [Ghe90]). Perhaps greatest lower bounds and least upper bounds can be replaced by some canonical choice of lower and upper bounds, a choice that may result from the derivation of the typing judgement itself.

Record update. For practical applications of calculi such as Fun, a particularly important problem concerns the semantics of "record update". The idea is this: given a function $f\colon s \to t$ and a record

e with a field l of type s, we would like to modify or update the l field of e by replacing $e.l$ by $f(e.l)$ *without losing or modifying any of the other fields of* e. The development of calculi which can deal with this form of polymorphism and the ways in which Fun and related languages can be used to represent similar techniques are an object of considerable current investigation. One recent effort in this direction is [CM89] but several other efforts are under way. Despite its importance we have not explored this issue in this paper since the discussion about it is very unsettled and it will merit independent treatment at a later date.

We believe that the "inheritance as implicit coercion" method is quite robust. For example, it easily extends to accommodate "constant" inheritances between base types, such as $int \leq real$, as long as coherence conditions similar to the ones arising in the proofs of the relevant lemmas in this paper hold between the the constant coercions which interpret these inheritances. Moreover, we expect that our methods will extend to the functional part of Quest [Car89a] and to the language described in [CM89], using the techniques of Coquand [Coq88] and Lamarche [Lam88]. Current work on inheritance and subtyping such as [CHC90] and [Mit90] will provide new challenges. We *do not claim* that every interesting aspect of inheritance can necessarily be handled in this way. However, our treatment, by showing that inheritance can be uniformly eliminated in favor of definable coercion, provides a challenge to formalisms which purport to introduce inheritance as a fundamentally new concept. Moreover, our basic approach to the semantics of inheritance should provide a useful contrast with other approaches.

8 Acknowledgements.

Breazu-Tannen's research was partially supported by ARO Grant DAAG29-84-K-0061 and ONR Grant NOOO14-88-K-0634. Many of the results of this paper were obtained during Coquand's visit to the University of Pennsylvania, partially sponsored by the Natural Sciences Association. Gunter's research was partially supported by ARO Grant DAAG29-84-K-0061 and ONR Grant NOOO14-88-K-0557. Scedrov's research was partially supported by NSF Grant CCR-87-05596, by ONR Grant NOOO14-88-K-0635, and by the 1987 Young Scientist Award from the Natural Sciences Association of the University of Pennsylvania.

Appendix A : The language SOURCE

Type expressions:
Fragment: $a \mid Top \mid s \to t \mid \{l_1: s_1, \ldots, l_m: s_m\} \mid \forall a \leq s.\, t \mid \mu a.\, t$
Variants: $\mid [l_1: t_1, \ldots, l_n: t_n]$
where a ranges over type variables, $m, n \geq 1$, and, in $\forall a \leq s.\, t$, a cannot be free in s. We will use $[s/a]t$ for substitution.

Raw terms:
Fragment:

$$x \mid d(e) \mid \lambda x{:}\, t.\, e \mid \{l_1 = e_1, \ldots, l_m = e_m\} \mid e.l \mid \Lambda a \leq t.\, e \mid e(t) \mid \mathsf{intro}[\mu a.\, t]e \mid \mathsf{elim}\ e$$

Variants:

$$\mid [l_1: t_1, \ldots, l_i = e, \ldots, l_n: t_n] \mid \mathsf{case}\ e\ \mathsf{of}\ l_1 \Rightarrow f_1, \ldots, l_n \Rightarrow f_n$$

where x ranges over (term) variables and $m, n \geq 1$. (Note the type decorations on variant "injections"; this is necessary for the uniqueness of type derivations in the inheritance-less system and it differs from [CW85].)

Raw terms are type-checked by deriving *typing judgements*, of the form $\Gamma \vdash e : t$. where Γ is a context. *Contexts* are defined recursively as follows: $\emptyset$ is a context; if Γ is a context which does not declare a, and the free variables of t are declared in Γ, then $\Gamma, a \leq t$ is a context; if Γ is a context which does not declare x, and the free variables of t are declared in Γ, then $\Gamma, x{:}\, t$ is a context. The proof system for deriving typing judgements makes use of *inheritance judgements* which have the form $C \vdash s \leq t$ where C is an inheritance context. *Inheritance contexts* are contexts in which only declarations of the form $a \leq t$ appear. If Γ is a context, we denoted by $\widehat{\Gamma}$ the inheritance context obtained from Γ by erasing the declarations of the form $x{:}\, t$.

Rules for deriving inheritance judgements:

Fragment:

(TOP)
$$C \vdash t \leq Top$$
where the free variables of t are declared in C

(VAR)
$$C_1, a \leq t, C_2 \vdash a \leq t$$

(ARROW)
$$\frac{C \vdash s \leq t \qquad C \vdash u \leq v}{C \vdash t \to u \leq s \to v}$$

(RECD)
$$\frac{C \vdash s_1 \leq t_1 \quad \cdots \quad C \vdash s_p \leq t_p}{C \vdash \{l_1: s_1, \ldots, l_p: s_p, \ldots, l_q: s_q\} \leq \{l_1: t_1, \ldots, l_p: t_p\}}$$

(FORALL) $$\frac{C \vdash s \leq t \qquad C, a{\leq}s \vdash u \leq v}{C \vdash \forall a{\leq}t.\, u \leq \forall a{\leq}s.\, v}$$

For Lemmas 11 and 12, and for Theorem 13 this is replaced with the weaker

(W-FORALL) $$\frac{C, a{\leq}t \vdash u \leq v}{C \vdash \forall a{\leq}t.\, u \leq \forall a{\leq}t.\, v}$$

(REFL) $$C \vdash t \leq t$$

where the free variables of t are declared in C

(TRANS) $$\frac{C \vdash r \leq s \qquad C \vdash s \leq t}{C \vdash r \leq t}$$

Variants:

(VART) $$\frac{C \vdash s_1 \leq t_1 \quad \cdots \quad C \vdash s_p \leq t_p}{C \vdash [l_1{:}\, s_1, \ldots, l_p{:}\, s_p] \leq [l_1{:}\, t_1, \ldots, l_p{:}\, t_p, \ldots, l_q{:}\, t_q]}$$

Rules for deriving typing judgements:

Fragment:

[VAR] $$\Gamma_1, x{:}t, \Gamma_2 \vdash x : t$$

[ABS] $$\frac{\Gamma, x{:}s \vdash e : t}{\Gamma \vdash \lambda x{:}s.\, e : s \to t}$$

[APPL] $$\frac{\Gamma \vdash d : s \to t \qquad \Gamma \vdash e : s}{\Gamma \vdash d(e) : t}$$

[RECD] $$\frac{\Gamma \vdash e_1 : t_1 \quad \cdots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash \{l_1 = e_1, \ldots, l_m = e_m\} : \{l_1{:}\, t_1, \ldots, l_m{:}\, t_m\}}$$

[SEL] $$\frac{\Gamma \vdash e : \{l_1{:}\,t_1, \ldots, l_m{:}\,t_m\}}{\Gamma \vdash e.l_i : t_i}$$

[B-GEN] $$\frac{\Gamma, a{\leq}s \vdash e : t}{\Gamma \vdash \Lambda a{\leq}s.\, e : \forall a{\leq}s.\, t}$$

[B-SPEC] $$\frac{\Gamma \vdash e : \forall a{\leq}s.\, t \qquad \widehat{\Gamma} \vdash r \leq s}{\Gamma \vdash e(r) : [r/a]t}$$

[R-INTRO] $$\frac{\Gamma \vdash e : [\mu a.\, t/a]t}{\Gamma \vdash \mathsf{intro}[\mu a.\, t]e : \mu a.\, t}$$

[R-ELIM] $$\frac{\Gamma \vdash e : \mu a.\, t}{\Gamma \vdash \mathsf{elim}\; e : [\mu a.\, t/a]t}$$

[INH] $$\frac{\Gamma \vdash e : s \qquad \widehat{\Gamma} \vdash s \leq t}{\Gamma \vdash e : t}$$

Variants:

[VART] $$\frac{\Gamma \vdash e : t_i}{\Gamma \vdash [l_1{:}\,t_1, \ldots, l_i{=}e, \ldots, l_n{:}\,t_n] : [l_1{:}\,t_1, \ldots, l_i{:}\,t_i, \ldots, l_n{:}\,t_n]}$$

[CASE] $$\frac{\Gamma \vdash e : [l_1{:}\,t_1, \ldots, l_n{:}\,t_n] \qquad \Gamma \vdash f_1 : t_1 \to t \quad \cdots \quad \Gamma \vdash f_n : t_n \to t}{\Gamma \vdash \mathsf{case}\; e\; \mathsf{of}\; l_1 \Rightarrow f_1, \ldots, l_n \Rightarrow f_n : t}$$

Appendix B: The language TARGET

Type expressions:
Fragment: $a \mid s \to t \mid \{l_1 : s_1, \ldots, l_m : s_m\} \mid \forall a.\, t \mid \mu a.\, t$
Variants: $\mid [l_1 : t_1, \ldots, l_n : t_n]$
Coercion space: $\mid s \circ\!\!\to t$
where a ranges over type variables and $n \geq 1$. For $m = 0$ we get the *empty record type* $1 \overset{\text{def}}{=} \{\}$.

Raw terms:
Fragment:

$$x \mid M(N) \mid \lambda x{:}t.\, M \mid \{l_1 = M_1, \ldots, l_m = M_m\} \mid M.l \mid \Lambda a.\, M \mid M(t) \mid \mathsf{intro}[\mu a.\, t]M \mid \mathsf{elim}\ M$$

Variants:

$$\mid [l_1 : t_1, \ldots, l_i = M, \ldots, l_n : t_n] \mid \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n$$

Coercion-coercion combinator:

$$\mid \iota_{s,t}$$

Coercion combinators:

$$\mid \mathsf{top}[t] \mid \mathsf{arrow}[s,t,u,v] \mid \mathsf{recd}[s_1, \ldots, s_q, t_1, \ldots, t_p] \mid \mathsf{forall}[s,t,a,u,v] \mid$$

$$\mathsf{vart}[s_1, \ldots, s_p, t_1, \ldots, t_q] \mid \mathsf{refl}[t] \mid \mathsf{trans}[r,s,t]$$

where x ranges over (term) variables and $n \geq 1$. For $m = 0$ we get the *empty record*, for which we will keep the notation $\{\}$. We will usually omit the cumbersome type tags on the coercion(-coercion) combinators. We use $[N/x]M$ for substitution.

Typing judgements, have the form $\Upsilon \vdash M : t$, where Υ is a typing context. *Typing contexts* are defined recursively as follows: $\emptyset$ is a context; if Υ is a context which does not declare a, then Υ, a is a typing context; if Υ is a context which does not declare x, and the free variables of t are declared in Υ, then $\Upsilon, x{:}t$ is a typing context.

Rules for deriving typing judgements:

Fragment:
Same as in Appendix A: [VAR] , [ABS] , [APPL] , [RECD] (in particular, for $n = 0$, $\Upsilon \vdash \{\} : 1$) , [SEL].

[GEN]
$$\frac{\Upsilon, a \vdash M : t}{\Upsilon \vdash \Lambda a.\, M : \forall a.\, t}$$

[SPEC]
$$\frac{\Upsilon \vdash M : \forall a.\, t}{\Upsilon \vdash M(s) : [s/a]t}$$

Same as in Appendix A: [R-INTRO] , [R-ELIM].

Variants:

Same as in Appendix A: [VART] , [CASE].

Coercion(-coercion) combinators:
We omit the typing contexts to simplify the notation.

$$\iota_{s,t} : (s \multimap t) \to (s \to t)$$

$$\mathsf{top}[t] : t \multimap \mathbf{1}$$

$$\mathsf{arrow}[s,t,u,v] : (s \multimap t) \to (u \multimap v) \to ((t \to u) \multimap (s \to v))$$

$$\mathsf{recd}[s_1,\ldots,s_q,t_1,\ldots,t_p] : (s_1 \multimap t_1) \to \cdots \to (s_p \multimap t_p) \to (\{l_1\colon s_1,\ldots,l_p\colon s_p,\ldots,l_q\colon s_q\} \multimap \{l_1\colon t_1,\ldots,l_p\colon t_p\})$$

$$\mathsf{forall}[s,t,a,u,v] : (s \multimap t) \to \forall a.\, ((a \multimap s) \to (u \multimap v)) \to (\forall a.\, ((a \multimap t) \to u) \multimap \forall a.\, ((a \multimap s) \to v))$$

$$\mathsf{vart}[s_1,\ldots,s_p,t_1,\ldots,t_q] : (s_1 \multimap t_1) \to \cdots \to (s_p \multimap t_p) \to ([l_1\colon s_1,\ldots,l_p\colon s_p] \multimap [l_1\colon t_1,\ldots,l_p\colon t_p,\ldots,l_q\colon t_q])$$

$$\mathsf{refl}[t] : t \multimap t$$

$$\mathsf{trans}[r,s,t] : (r \multimap s) \to (s \multimap t) \to (r \multimap t)$$

Equational theory:
Technically, equational judgements should all contain a typing context under which both terms in the equation typecheck with the same type [CGW87, BC88, CGW89]. To simplify the notation, we will in most cases omit these contexts.

Fragment:
We omit the simple rules for reflexivity, symmetry, transitivity, and congruence with respect to function application, record formation, field selection, application to types, recursive type introduction, and recursive type elimination.

{XI}
$$\frac{\Upsilon, x\colon s \vdash M = N}{\Upsilon \vdash \lambda x\colon s.\, M = \lambda x\colon s.\, N}$$

{TYPE-XI}
$$\frac{\Upsilon, a \vdash M = N}{\Upsilon \vdash \Lambda a.\, M = \Lambda a.\, N}$$

{BETA} $$(\lambda x{:}\, s.\, M)(N) \;=\; [N/x]M$$

where $N : s$.

{ETA} $$\lambda x{:}\, s.\, M(x) \;=\; M$$

where $M : s \rightarrow t$ and x not free in M.

{RECD-BETA} $$\{l_1 = M_1, \ldots, l_m = M_m\}.l_i \;=\; M_i$$

where $m \geq 1$, $M_1{:}\, t_1, \ldots, M_m{:}\, t_m$.

{RECD-ETA} $$\{l_1 = M.l_1, \ldots, l_m = M.l_m\} \;=\; M$$

where $M : \{l_1{:}\, t_1, \ldots, l_m{:}\, t_m\}$. For $m = 0$, this rule gives $\{\} \;=\; M$ which makes **1** into a terminator.

{FORALL-BETA} $$(\Lambda a.\, M)(r) \;=\; [r/a]M$$

{FORALL-ETA} $$\Lambda a.\, M(a) \;=\; M$$

where $M : \forall a.\, t$ and a not free in M.

{R-BETA} $$\mathsf{elim}\,(\mathsf{intro}[\mu a.\, t]M) \;=\; M$$

where $M : \mu a.\, t$.

{R-ETA} $$\mathsf{intro}[\mu a.\, t](\mathsf{elim}\; M) \;=\; M$$

where $M : [\mu a.\, t/a]t$.

Variants:

We omit the simple rules for congruence with respect to variant formation, and case analysis.

{VART-BETA} $\quad \mathsf{case}\ \mathsf{inj}_{l_i}(M_i)\ \mathsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n \ = \ F_i(M_i)$

where $F_1 : t_1 \to t, \ldots, F_n : t_n \to t,\ M_i : t_i$ and inj_{l_i} is shorthand for $\lambda x{:}\,t_i.\,[l_1{:}\,t_1, \ldots, l_i = x, \ldots, l_n{:}\,t_n]$.

{VART-ETA} $\quad \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow \mathsf{inj}_{l_1}, \ldots, l_n \Rightarrow \mathsf{inj}_{l_n} \ = \ M$

where $M{:}\,[l_1{:}\,t_1, \ldots, l_n{:}\,t_n]$.

{VART-CRN} $\quad \iota(P)(\mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) \ = \ \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow F_1; \iota(P), \ldots, l_n \Rightarrow F_n; \iota(P)$

where $M{:}\,[l_1{:}\,t_1, \ldots, l_n{:}\,t_n],\ F_1{:}\,t_1 \to t, \ldots, F_n{:}\,t_n \to t,\ P{:}\,t \multimap s$.

Alternatively, we could require instead of { VART-ETA } + { VART-CRN }:

{VART-COP} $\quad \iota(Q)(M) \ = \ \mathsf{case}\ M\ \mathsf{of}\ l_1 \Rightarrow (\mathsf{inj}_{l_1}; \iota(Q)), \ldots, l_n \Rightarrow (\mathsf{inj}_{l_n}; \iota(Q))$

where $M{:}\,[l_1{:}\,t_1, \ldots, l_n{:}\,t_n],\ Q{:}\,[l_1{:}\,t_1, \ldots, l_n{:}\,t_n] \multimap t$.

Coercion(-coercion) combinators:

$$\iota(\mathsf{top}) \ = \ \lambda x{:}\,t.\ \{\}$$

$$\iota(\mathsf{arrow}(P)(Q)) \ = \ \lambda z{:}\,t \to u.\,(\iota(P)); z; (\iota(Q))$$

where $P{:}\,s \multimap t,\ Q{:}\,u \multimap v$.

$$\iota(\mathsf{recd}(R_1)\cdots(R_p)) \ = \ \lambda w{:}\,\{l_1{:}\,s_1, \ldots, l_p{:}\,s_p, \ldots, l_q{:}\,s_q\}.\ \{l_1{:}\,\iota(R_1)(w.l_1), \ldots, l_p{:}\,\iota(R_p)(w.l_p)\}$$

where $R_1{:}\,s_1 \multimap t_1, \ldots, R_p{:}\,s_p \multimap t_p$.

$$\iota(\mathsf{forall}(P)(W)) \ = \ \lambda z{:}\,(\forall a.\,(a \multimap t) \to u).\ \Lambda a.\ \lambda f{:}\,a \multimap s.\ \iota(W(a)(f))(z(a)(\mathsf{trans}(f)(P)))$$

where $P{:}\,s \multimap t,\ W{:}\,\forall a.\,(a \multimap s) \to (u \multimap v)$.

$$\iota(\mathsf{vart}(R_1)\cdots(R_p)) \ = \ \lambda w{:}\,[l_1{:}\,s_1, \ldots, l_p{:}\,s_p].\ \mathsf{case}\ w\ \mathsf{of}\ l_1 \Rightarrow \iota(R_1); \mathsf{inj}_{l_1}, \ldots, l_p \Rightarrow \iota(R_p); \mathsf{inj}_{l_p}$$

where $R_1{:}\,s_1 \multimap t_1, \ldots, R_p{:}\,s_p \multimap t_p$.

$$\iota(\mathsf{refl}) \ = \ \lambda x{:}\,t.\,x$$

$$\iota(\mathsf{trans}(P)(Q)) \ = \ \iota(P); \iota(Q)$$

where $P{:}\,r \multimap s,\ Q{:}\,s \multimap t$.

{IOTA-INJ} $$\frac{\iota(P) \ = \ \iota(Q)}{P \ = \ Q}$$

Appendix C: The translation

We present first the remaining of the translation of the fragment discussed in section 3.

(VAR)*

$$C_1^*,\, a,\, f{:}\, a \to t^*,\, C_2^* \ \vdash\ f : a \to t^*$$

(RECD)*

$$\frac{C^* \ \vdash\ P_1 : s_1^* \to t_1^* \quad \cdots \quad C^* \ \vdash\ P_p : s_p^* \to t_p^*}{C^* \ \vdash\ R : \to \{l_1{:}\, s_1^*, \ldots, l_p{:}\, s_p^*, \ldots, l_q{:}\, s_q^*\}\{l_1{:}\, t_1^*, \ldots, l_p{:}\, t_p^*\}}$$

where $R \overset{\text{def}}{=} \lambda w{:}\, \{l_1{:}\, s_1^*, \ldots, l_p{:}\, s_p^*, \ldots, l_q{:}\, s_q^*\}.\ \{l_1{:}\, P_1(w.l_1), \ldots, l_p{:}\, P_p(w.l_p)\}$

(REFL)*

$$C^* \ \vdash\ \lambda x{:}\, t^*.\, x : t^* \to t^*$$

where the free variables of t^* are declared in C^*

(TRANS)*

$$\frac{C^* \ \vdash\ P : r^* \to s^* \qquad C^* \ \vdash\ Q : s^* \to t^*}{C^* \ \vdash\ P;Q : r^* \to t^*}$$

The rules [VAR] , [ABS] , [APPL] , [RECD] , [SEL] , [R-INTRO] , [R-ELIM] are translated straightforwardly, see below. Here is the translation of the only other rule left (the translations of the other rules appears in section 3).

[B-GEN]

$$\frac{\Gamma^*,\, a,\, f{:}\, a \to s^* \ \vdash\ M : t^*}{\Gamma^* \ \vdash\ \Lambda a.\, \lambda f{:}\, a \to s^*.\, M : \forall a.\, ((a \to s^*) \to t^*)}$$

In the following, we present the translation for the full calculus. As before, for any **SOURCE** `item` we will denote by `item*` its translation into **TARGET** . We begin with the types. Note the translation of bounded generics and of *Top*.

$$\begin{array}{rcl@{\qquad}rcl}
a^* & \overset{\text{def}}{=} & a & (\forall a \leq s.\, t)^* & \overset{\text{def}}{=} & \forall a.\, ((a \multimap s^*) \to t^*) \\
Top^* & \overset{\text{def}}{=} & \mathbf{1} & (\mu a.\, t)^* & \overset{\text{def}}{=} & \mu a.\, t^* \\
(s \to t)^* & \overset{\text{def}}{=} & s^* \to t^* & [l_1{:}\, s_1, \ldots, l_n{:}\, s_n]^* & \overset{\text{def}}{=} & [l_1{:}\, s_1^*, \ldots, l_n{:}\, s_n^*] \\
\{l_1{:}\, s_1, \ldots, l_m{:}\, s_m\}^* & \overset{\text{def}}{=} & \{l_1{:}\, s_1^*, \ldots, l_m{:}\, s_m^*\} & & &
\end{array}$$

where $s \times t \overset{\text{def}}{=} \{\mathit{left}{:}\, s, \mathit{right}{:}\, t\}$.

One shows immediately that $([s/a]t)^* \equiv [s^*/a]t^*$. We extend this to contexts and inheritance contexts, which translate into just typing contexts in **TARGET** .

$$\begin{array}{rcl@{\qquad}rcl}
\emptyset^* & \overset{\text{def}}{=} & \emptyset & \emptyset^* & \overset{\text{def}}{=} & \emptyset \\
(\Gamma,\, a \leq t)^* & \overset{\text{def}}{=} & \Gamma^*,\, a,\, f{:}\, a \multimap t^* & (C,\, a \leq t)^* & \overset{\text{def}}{=} & C^*,\, a,\, f{:}\, a \multimap t^* \\
(\Gamma,\, x{:}\, t)^* & \overset{\text{def}}{=} & \Gamma^*,\, x{:}\, t^* & & &
\end{array}$$

where f is a *fresh* variable for each (a, f).

Next we will describe how we translate the derivations of judgments of **SOURCE** . The translation is defined by recursion on the structure of the derivation trees. Since these are freely generated by the derivation rules, it is sufficient to provide for each derivation rule of **SOURCE** a corresponding rule on trees of **TARGET** judgments. One then checks that these corresponding rules are *directly derivable* in **TARGET** (Lemma 14 below), therefore the translation takes derivations in **SOURCE** into derivations in **TARGET** .

A **SOURCE** derivation yielding an inheritance judgment $C \vdash s \leq t$ is translated as a tree of **TARGET** judgments yielding $C^* \vdash P : s^* \multimap t^*$. Here are the **TARGET** rules that correspond to the rules for deriving inheritance judgements in **SOURCE**.

(TOP)*
$$C^* \vdash \mathsf{top} : t^* \multimap \mathbf{1}$$

(VAR)*
$$C_1^*, a, f{:}\, a \multimap t^*, C_2^* \vdash f : a \multimap t^*$$

(ARROW)*
$$\frac{C^* \vdash P : s^* \multimap t^* \qquad C^* \vdash Q : u^* \multimap v^*}{C^* \vdash \mathsf{arrow}(P)(Q) : (t^* \to u^*) \multimap (s^* \to v^*)}$$

(RECD)*
$$\frac{C^* \vdash P_1 : s_1^* \multimap t_1^* \quad \cdots \quad C^* \vdash P_p : s_p^* \multimap t_p^*}{C^* \vdash \mathsf{recd}(P_1)\cdots(P_p) : \{l_1{:}\, s_1^*, \ldots, l_p{:}\, s_p^*, \ldots, l_q{:}\, s_q^*\} \multimap \{l_1{:}\, t_1^*, \ldots, l_p{:}\, t_p^*\}}$$

(FORALL)*
$$\frac{C^* \vdash P : s^* \multimap t^* \qquad C^*, a, f{:}\, a \multimap s^* \vdash Q : u^* \multimap v^*}{C^* \vdash \mathsf{forall}(P)(\Lambda a.\, \lambda f{:}\, a \multimap s^*.\, Q) : \forall a.\, ((a \multimap t^*) \to u^*) \multimap \forall a.\, ((a \multimap s^*) \to v^*)}$$

(VART)*
$$\frac{C^* \vdash P_1 : s_1^* \multimap t_1^* \quad \cdots \quad C^* \vdash P_p : s_p^* \multimap t_p^*}{C^* \vdash \mathsf{vart}(P_1)\cdots(P_p) : [l_1{:}\, s_1^*, \ldots, l_p{:}\, s_p^*] \multimap [l_1{:}\, t_1^*, \ldots, l_p{:}\, t_p^*, \ldots, l_q{:}\, t_q^*]}$$

(REFL)*
$$C^* \vdash \mathsf{refl} : t^* \multimap t^*$$

where the free variables of t^* are declared in C^*

(TRANS)*
$$\frac{C^* \vdash P : r^* \multimap s^* \qquad C^* \vdash Q : s^* \multimap t^*}{C^* \vdash \mathsf{trans}(P)(Q) : r^* \multimap t^*}$$

A **SOURCE** derivation yielding an typing judgment $\Gamma \ \vdash \ e : t$ is translated as a tree of **TARGET** judgments yielding $\Gamma^* \ \vdash \ M : t^*$. Here are the **TARGET** rules that correspond to the rules for deriving typing judgements in **SOURCE**.

The rules [VAR] , [ABS] , [APPL] , [RECD] , [SEL] , [R-INTRO] , [R-ELIM] , [VART] , [CASE] all have direct correspondents in **TARGET** so their translation is straightforward. We ilustrate it with two examples.

[VAR]* $$\Gamma_1^*, x{:}t^*, \Gamma_2^* \vdash x : t^*$$

[ABS]* $$\frac{\Gamma^*, x{:}s^* \vdash M : t^*}{\Gamma^* \vdash \lambda x{:}s^*.\, M : s^* \to t^*}$$

Here is the translation of the other three rules.

[B-GEN] $$\frac{\Gamma^*, a, f{:}a \multimap s^* \vdash M : t^*}{\Gamma^* \vdash \Lambda a.\, \lambda f{:}a \multimap s^*.\, M : \forall a.\, ((a \multimap s^*) \to t^*)}$$

[B-SPEC]* $$\frac{\Gamma^* \vdash M : \forall a.\, ((a \multimap s^*) \to t^*) \qquad \widehat{\Gamma}^* \vdash P : r^* \multimap s^*}{\Gamma^* \vdash M(r^*)(P) : [r^*/a]t^*}$$

[INH]* $$\frac{\Gamma^* \vdash M : s^* \qquad \widehat{\Gamma}^* \vdash P : s^* \multimap t^*}{\Gamma^* \vdash \iota(P)(M) : t^*}$$

Lemma 14 *The rules* (TOP)* – (TRANS)* *and* [VAR]* – [INH]* *are directly derivable in* **TARGET** . ▮

References

[ABL86] R. Amadio, K. B. Bruce, and G. Longo. The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In A. Meyer, editor, *Logic in Computer Science*, pages 122–130, IEEE Computer Society Press, 1986.

[Ama89] R. Amadio. *Recursion over realizability structures.* Research Report TR 1/89, Universitá di Pisa, January 1989.

[Ama90] R. Amadio. Recursion over realizability structures. *Information and Computation*, 91:55–85, 1990. To appear.

[Ama91] R. Amadio. *Recursion and subtyping in lambda calculi.* PhD thesis, University of Pisa, 1991.

[Bar84] H. Barendregt. *The Lambda Calculus: Its syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, revised edition, 1984.

[BBG88] V. Breazu-Tannen, P. Buneman, and C. A. Gunter. Typed functional programming for the rapid development of reliable software. In J. E. Gaffney, editor, *Productivity: Progress, Prospects and Payoff*, pages 115–125, Association for Computing Machinery, June 1988.

[BC88] V. Breazu-Tannen and T. Coquand. Extensional models for polymorphism. *Theoretical Computer Science*, 59:85–114, 1988.

[BCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explict coercion (preliminary report). In R. Parikh, editor, *Logic in Computer Science*, pages 112–134, IEEE Computer Society, June 1989.

[BGS89] V. Breazu-Tannen, C. Gunter, and A. Scedrov. *Denotational Semantics for Subtyping between Recursive Types.* Research Report MS-CIS-89-63/Logic & Computation 12, Department of Computer and Information Science, University of Pennsylvania, 1989.

[BGS90] V. Breazu-Tannen, C. Gunter, and A. Scedrov. Computing with coercions. In M. Wand, editor, *Lisp and Functional Programming*, pages 44–60, ACM, 1990.

[BL88] K. B. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. In Y. Gurevich, editor, *Logic in Computer Science*, pages 38–50, IEEE Computer Society, July 1988.

[BW90] K. Bruce and P. Wegner. An algebraic model of subtype and inheritance. In F. Bancilhon and P. Buneman, editors, *Advances in database programming languages*, pages 75–96, ACM Press and Addison-Wesley, New York, 1990.

[Car84] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. D. Plotkin, editors, *Semantics of Data Types*, pages 51–67, *Lecture Notes in Computer Science vol. 173,* Springer, 1984.

[Car85] R. Cartwright. Types as intervals. In B. K. Reid, editor, *Symposium on Principles of Programming Languages*, pages 22–36, ACM, 1985.

[Car86] L. Cardelli. Amber. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*, pages 21–47, *Lecture Notes in Computer Science vol. 242,* Springer, 1986.

[Car88a] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car88b] L. Cardelli. Structural subtyping and the notion of power type. In J. Ferrante and P. Mager, editors, *Symposium on Principles of Programming Languages*, pages 70–79, ACM, 1988.

[Car89a] L. Cardelli. *Typeful programming.* Research Report 45, DEC Systems, Palo Alto, May 1989.

[Car89b] F. Cardone. Relational semantics for recursive types and bounded quantification. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *International Colloquium on Automata, Languages and Programs*, pages 164–178, *Lecture Notes in Computer Science vol. 372,* Springer, July 1989.

[CG90] P.-L. Curien and G. Ghelli. Coherence of subsumption. In *Proceedings CAAP'90, LNCS 431*, 1990. Full version to appear in *Mathematical Structures in Computer Science.*

[CGW87] T. Coquand, C. A. Gunter, and Glynn Winskel. DI-domains as a model of polymorphism. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, pages 344–363, *Lecture Notes in Computer Science vol. 298,* Springer, April 1987.

[CGW89] T. Coquand, C. A. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation.*, 81:123–167, 1989.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CHC90] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In P. Hudak, editor, *Principles of Programming Languages*, pages 125–135, ACM, 1990.

[CM89] L. Cardelli and J. Mitchell. Operations on records. In M. Mislove, editor, *Mathematical Foundations of Programming Semantics*, *Lecture Notes in Computer Science,* Springer, March 1989.

[Cop85] M. Coppo. A completeness theorem for recursively defined types. In W. Brauer, editor, *International Colloquium on Automata, Languages and Programs*, pages 120–129, *Lecture Notes in Computer Science vol. 194,* Springer, 1985.

[Coq88] T. Coquand. Categories of embeddings. In Y. Gurevich, editor, *Logic in Computer Science*, pages 256–263, IEEE Computer Society, July 1988.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[CZ86] M. Coppo and M. Zacchi. Type inference and logical relations. In A. Meyer, editor, *Symposium on Logic in Computer Science*, pages 218–226, ACM, 1986.

[FMRS89] P. Freyd, P. Mulry, G. Rosolini, and D.S. Scott. Domains in PER. 1989. Unpublished manuscript.

[Fre89] P. Freyd. Recursive types. 1989. Unpublished manuscript.

[Ghe90] G. Ghelli. *Proof-Theoretic studies about a minimal type system integrating inclusion and parametric polymorphism.* PhD thesis, University of Pisa, 1990.

[Gir86] J. Y. Girard. The system F of variable types: fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[Gir87] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir88] J. Y. Girard. Normal functors, power series, and λ-calculus. *Annals of Pure and Applied Logic*, 37:129–177, 1988.

[Gir89] J. Y. Girard. *Proofs and Types.* Cambridge University Press, 1989.

[GJ90] C. A. Gunter and A. Jung. Coherence and consistency in domains (extended outline). *Journal of Pure and Appled Algebra*, 63:49–66, 1990.

[GJM85] J. A. Goguen, J-P. Jouannaud, and J. Meseguer. Operational semantics for order-sorted algebra. In W. Brauer, editor, *International Colloquium on Automata, Languages and Programs*, pages 221–231, *Lecture Notes in Computer Science vol. 194*, Springer, July 1985.

[GM] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Unpublished manuscript.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, Reading, MA, 1983.

[HP89a] H. Huwig and A. Poigné. A note on inconsistencies caused by fixpoints in a cartesian closed category. *Theoretical Computer Science*, 1989. To appear.

[HP89b] J. M . E. Hyland and A. Pitts. The theory of constructions: categorical semantics and topos-theoretic models. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 137–199, ACM, 1989.

[JM88] L. Jategaonkar and J. C. Mitchell. ML with extended pattern matching and subtypes. In R. Cartwright, editor, *Symposium on LISP and Functional Programming*, pages 198–211, ACM, 1988.

[KL71] G. M. Kelly and S. Mac Lane. Coherence in closed categories. *J. Pure Appl. Algebra*, 1:97–140, 1971. Erratum ibid. 2(1971), p. 219.

[Koy82] C. Koymans. Models of the lambda calculus. *Information and Control*, 52:306–332, 1982.

[Lam88] F. Lamarche. *Modelling Polymorphism with Categories.* PhD thesis, McGill University, 1988.

[Law69] F. W. Lawvere. Diagonal arguments and cartesian closed categories. In *Category theory, homology theory, and their applications II*, pages 134–145, *Lecture Notes in Mathematics*, Vol. 92, Springer-Verlag, 1969.

[LP85] S. Mac Lane and R. Pare. Coherence for bicategories and indexed categories. *Journal of Pure and Appled Algebra*, 37:59–80, 1985.

[Mar84] P. Martin-Löf. *Intutionistic Type Theory. Studies in Proof Theory*, Bibliopolis, 1984.

[Mar88] S. Martini. Bounded quantifiers have interval models. In R. Cartwright, editor, *Symposium on LISP and Functional Programming*, pages 164–173, ACM, 1988.

[Mey82] A. R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

[Mit90] J. Mitchell. Toward a typed foundation for method specialization and inheritance. In P. Hudak, editor, *Principles of Programming Languages*, pages 109–124, ACM, 1990.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In R. Cartwright, editor, *Symposium on LISP and Functional Programming*, pages 174–183, ACM, New York, 1988.

[Rey80] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 211–258, *Lecture Notes in Computer Science vol. 94*, Springer, 1980.

[Sal88] A. Salvesen. Polymorphism and monomorphism in Martin-Löf's Type Theory. In *Logic Colloquium'88*, 1988.

[Sco80] D. S. Scott. Relating theories of the lambda calculus. In J. R. Hindley, editor, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 403–450, Academic Press, 1980.

[Sta88] R. Stansifer. Type inference with subtypes. In J. Ferrante and P. Mager, editors, *Symposium on Principles of Programming Languages*, pages 88–97, ACM, 1988.

[Str88] T. Streicher. *Correctness and completeness of a categorical semantics of the Calculus of Constructions.* PhD thesis, Passau University, 1988.

[Tro73] A. S. Troelstra. *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics vol. 344*, Springer, 1973.

[TT87] T. Coquand and T. Ehrhard. An equational presentation of higher-order logic. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, pages 40–56, *Lecture Notes in Computer Science vol. 283*, Springer, 1987.

[Wan87] M. Wand. Complete type inference for simple objects. In D. Gries, editor, *Symposium on Logic in Computer Science*, pages 37–46, IEEE Computer Society Press, Ithaca, New York, June 1987.

COMPUTING WITH COERCIONS[1]

(Extended Abstract)

V. Breazu-Tannen *C. A. Gunter* *A. Scedrov*

University of Pennsylvania

Abstract. This paper relates two views of the operational semantics of a language with multiple inheritance. It is shown that the introduction of explicit coercions as an interpretation for the implicit coercion of inheritance does not affect the evaluation of a program in an essential way. The result is proved by semantic means using a denotational model and a computational adequacy result to relate the operational and denotational semantics.

1 Introduction

There have been a number of efforts to understand the denotational semantics of inheritance polymorphism and a variety of mathematical models for languages with subtle semantic features have been discovered. However, as far as the authors of this paper know, no one has attempted to discuss what, if anything, these denotational models have to do with the intended execution of programs in the languages they model. For example, *all* of the published denotational models of the language Fun of Cardelli Wegner [CW85] (including the work of authors of this paper) model this language in way that corresponds to no reasonable interpretation of its operational semantics! No functional programming language in common use diverges when evaluating the program $\lambda x.\ e$, even when the expression e may diverge. Yet the models for Fun which have been studied identify the abstraction $\lambda x.\ \perp$ with the divergent program $\perp$. Besides this problem, all existing models satisfy the unrestricted β rule, which fails to be a legitimate transformation in call-by-value languages. Since call-by-value is the most common form of evaluation, one is led to ask whether this commitment to β was an important feature of the models concerned. In short, very little has been done to close the gap between denotational and operational theories of inheritance. We see two basic things as missing from the current theories: (1) a careful discussion of the structional operational semantics of languages with inheritance type systems and (2) any account of the relationship between the suggested models and a reasonable account of operational semantics.

Our goal in this paper is to attempt an account of problem (1) guided by an approach to (2). We carry out this study in a simple, familiar context by using an extension of Plotkin's illustrative language PCF [Plo77]. We develop a simple structural operational semantics for this language in the spirit of the evaluation mechanisms of languages such as LISP and ML in which functions call their parameters by value. Our extension, which we call PCF+, is obtained by adding record and variant types. This language is extended to a new language, PCF++, by permitting the use of a form of inheritance which allows more programs to be viewed as type correct. We then study the question of the proper operational interpretation of PCF++. One possible approach is simple to understand: after a PCF++ program is shown to be type correct, the type information in the term is erased and the resulting term (which lives in an extended untyped lambda-calculus) is

[1]Appears in **Conference on Lisp and Functional Programming**, edited by M. Wand, Nice, France, July 1990, pp. 44–60.

evaluated. However, in view of the form of semantics that we have studied in our work on Fun and its relatives [BCGS89, BCGS90] there is another view of the proper operational semantics of PCF++. Under this view, a term of PCF++ is translated into a PCF+ term by inserting explicit coercions which "explain" the inheritance in the original PCF++ program in an intuitive way. If this "explanation" really is intuitive and the first form of evaluation (which is a common from of implementation) is reasonable, then it seems that there must be some relationship between these two views of program evaluation! Moreover, this latter approach is also not uncommon as a form of evaluation, and therefore has independent interest. In this paper we will show that these two forms of evaluation are essentially the same for observable types.

To give the reader an idea of what translation we have in mind let us look at an example of how a simple program would be evaluated. Applying our semantic paradigm to PCF++, we translate its programs into PCF+ programs, essentially by inserting explicit coercion terms wherever inheritance is used in type-checking. In anticipation of an exact definition of this translation (section 2), here is an example. PCF++ type-checks the program $P \equiv G(F)$ where

$$\begin{aligned} G &\equiv \lambda f : \{l : \mathbf{num}\} \to \mathbf{num}.\ \{k_1 = f(\{l = \mathbf{0}, l_1 = \mathbf{1}\}), k_2 = f(\{l = \mathbf{2}, l_2 = \mathbf{false}\})\} \\ F &\equiv \lambda x : \{l : \mathbf{num}\}.\ x.l \end{aligned}$$

Note that G will not type-check in PCF+ because of the different types of the two arguments to which f is applied.

The translation to PCF+ depends on the way P is type-checked. One possible translation is $P' \equiv G'(F)$ where

$$G' \equiv \lambda f : \{l : \mathbf{num}\} \to \mathbf{num}.\ \{k_1 = f(\xi_1(\{l = \mathbf{0}, l_1 = \mathbf{1}\})), k_2 = f(\xi_2(\{l = \mathbf{2}, l_2 = \mathbf{false}\}))\}$$

where ξ_1 and ξ_2 are the following *coercion terms*

$$\xi_1 \equiv \lambda x_1 : \{l : \mathbf{num}, l_1 : \mathbf{num}\}.\ \{l = x_1.l\} \qquad \xi_2 \equiv \lambda x_2 : \{l : \mathbf{num}, l_2 : \mathbf{bool}\}.\ \{l = x_2.l\}\ .$$

Another possible translation is $P'' \equiv G''(F)$ where

$$G'' \equiv \lambda f : \{l : \mathbf{num}\} \to \mathbf{num}.\ \{k_1 = \zeta_1(f)(\{l = \mathbf{0}, l_1 = \mathbf{1}\}), k_2 = \zeta_2(f)(\{l = \mathbf{2}, l_2 = \mathbf{false}\})\}$$

where

$$\zeta_1 \equiv \lambda f_1 : \{l : \mathbf{num}\} \to \mathbf{num}.\ \lambda x_1 : \{l = \mathbf{num}, l_1 = \mathbf{num}\}.\ f_1(\xi_1(x_1))$$

and

$$\zeta_2 \equiv \lambda f_2 : \{l : \mathbf{num}\} \to \mathbf{num}.\ \lambda x_2 : \{l = \mathbf{num}, l_2 = \mathbf{bool}\}.\ f_2(\xi_2(x_2))$$

The fact that the translation (more generally, the meaning) depends on the type-checking derivation entails the need for denotational coherence results [BCGS89]. In this paper, however, we will examine the computational (operational) aspects of this translation. Notice that the "execution" of both P' and P'' yields the same result

$$\{k_1 = \mathbf{0}, k_2 = \mathbf{2}\}$$

More importantly, so does the direct execution of P. The "direct" operational semantics for PCF++ that we have in mind is just the same as that of PCF+. It is a simple but crucial observation that the same evaluation rules work on programs allowed by the more permissive type discipline of PCF++. Not surprisingly, this is the natural way to implement such languages (Cardelli, personal communication about Quest [Car89]). Although it is may not be useful to fully translate a term before executing it, it is reasonable to ask whether translation would affect the evaluation. Since coercions remove the "junk" in a term, they may play a useful role in efficient implementation. However, our primary interest is in the abstract specification of the language and not the details of its efficient implementation.

Our *main result* relates the direct execution of a PCF++ program phrase e to the execution of *any* of its PCF+ translations, e^*. We prove that

$$e \text{ terminates if and only if } e^* \text{ terminates.}$$

If both e and e^* terminate, what can we say about the relationship between the results of the two computations? Of course, we are able to show that if the type of e is *ground*, (integer or boolean) then the results are the exactly the same. In this language we are also interested in computing with more complex objects, such as records/variants of records/variants of ground data (this is particularly consistent with the way things are viewed in object-oriented database programming applications [OBB89] for example). We call the types of such data *observable types*. Now, the philosophy of PCF++ is that the type of program phrases is part of them, i.e., user-supplied in some sense. (This is in contrast with the approaches based on type inference; see for example [Wan89].) At observable types, we show that the results of the two computations have the same *components* in those record fields which appear in the prescribed type of the program phrase. This is the best we can hope for, since the introduction of coercions yields computations which may remove "junk" fields, namely the fields not occurring in the prescribed type. Moral: if you specify a type for your program, don't expect to observe more than what the type allows. Anyway, our conclusion is that coercions *make no essential difference* to the computation.

While this result only relates our translation to the operational semantics, it can be used for *transfer of computational adequacy*. Consider a denotational semantics $\mathcal{D}^+$ of PCF+ for which our translation is coherent. This yields a denotational semantics $\mathcal{D}^{++}$ for PCF++ where a term is interpreted by first translating it into PCF+ and then taking the $\mathcal{D}^+$-meaning of the translation. Under some reasonable assumptions about $\mathcal{D}^+$, our main result implies that if $\mathcal{D}^+$ is computationally adequate (*i.e.* the meaning of a term e is non-bottom iff the evaluation of e terminates) for the operational semantics of PCF+ then $\mathcal{D}^{++}$ is computationally adequate for the operational semantics of PCF++.

An interesting methodological twist is that our proof of the main result actually uses a specific denotational semantics $[\![\cdot]\!]^+$ which is computationally adequate for PCF+ and for which this transfer can be done! As it is, we show directly that $[\![\cdot]\!]^{++}$ is computationally adequate for PCF++ and we derive our main result from this. We regard this as a nice example of the use of a domain-theoretic semantics for obtaining an essentially syntactic result.

Another comment on methodology. We have chosen to focus on call-by-value operational semantics since this is the most common style of implementation for the languages we are studying and because it offers a change of pace from our earlier results [BCGS89] where we focused on models in which the unrestricted β axiom holds. We expect that results such as the ones we are proving in this paper could be formulated for a call-by-name operational semantics, although this would call for some changes in our concept of observability.

In section 2 we begin by introducing the syntax of PCF++ as an extension of PCF+. Then we describe the translation back, from PCF++ to PCF+. Finally we give the call-by-value operational semantics and state our main theorem. In section 3 we give a domain-theoretic denotational semantics of PCF+ for which our translation is coherent and for which the operational semantics of PCF+ is sound and computationally adequate. We prove that the operational semantics of PCF++ is sound and computationally adequate for the induced denotational semantics and then we show how to derive from this our main theorem. The paper ends with a section of conclusions and ideas for more work.

2 From PCF+ to PCF++ and back again.

In this section we introduce the two calculi on which the central result of the paper focuses.

2.1 Extending PCF+ to PCF++.

The following grammar defines the syntax of *type expressions* s and *raw terms* e of our calculi. We assume primitive syntax classes of variables and labels:

$$\begin{array}{rcl} x & \in & \mathbf{Variable} \\ l & \in & \mathbf{Label} \\ s & ::= & \mathbf{num} \mid \mathbf{bool} \mid s \to s \mid \{l_1 : s_1, \ldots, l_n : s_n\} \mid [l_1 : s_1, \ldots, l_n : s_n] \\ e & ::= & \mathbf{0} \mid \mathbf{Succ}(e) \mid \mathbf{Pred}(e) \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{IsZero}(e) \mid \\ & & x : s \mid \lambda x : s.\ e \mid e(e) \mid \mu x : s.\ e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \\ & & \{l_1 = e, \ldots, l_n = e\} \mid e.l \mid [l = e] \mid \mathbf{case}\ e\ \mathbf{of}\ l_1 \Rightarrow e, \ldots, l_n \Rightarrow e \end{array}$$

For *records* $\{l_1 = e_1, \ldots, l_n = e_n\}$ and *variants* $[l_1 = e_1, \ldots, l_n = e_n]$, it is assumed that the labels $l_1, \ldots, l_n$ are all distinct. We assume that the reader can infer from our notation what is meant by free and bound variables of raw terms. A raw term is said to be closed if it has no free variables.

A *type context* is a list $x_1 : s_1, \ldots, x_n : s_n$ of pairs of variables and types. We assume that the variables x_i in such a context are distinct. A *typing judgement* is a sequent of the form $H \vdash e : s$ where H is a typing context which includes all of the free variables of the raw term e. A typing judgement is said to be *derivable in PCF+* if it can be proved using the axioms and rules listed in Table 1. It is not hard to see that any derivable sequent has a *unique* derivation. This latter fact will not be true of the calculus PCF++ which we now define. PCF++ is the extension of PCF+ to a calculus with multiple inheritance. First of all, we define a binary relation $s < t$ of subtyping between type expressions s and t using the rules in Table 2. The reader can check that $<$ is a preorder on type expressions. This relation is now incorporated into the typing system of PCF++ by the addition of the *subsumption rule:*

$$\frac{H \vdash e : s \qquad s < t}{H \vdash e : t}$$

2.2 Translation from PCF++ into PCF+.

Definition: Given types s and t such that $s < t$ is provable, we define a PCF+ term $\mathsf{coerce}[s < t]$ of type $s \to t$ by induction on the proof of $s < t$ as follows:

- $\mathsf{coerce}[\mathbf{bool} < \mathbf{bool}] \equiv \lambda x : \mathbf{bool}.\ x$ and $\mathsf{coerce}[\mathbf{num} < \mathbf{num}] \equiv \lambda x : \mathbf{num}.\ x$.
- $\mathsf{coerce}[s \to t < s' \to t'] \equiv \lambda f : s \to t.\ \mathsf{coerce}[t < t'] \circ f \circ \mathsf{coerce}[s' < s]$
- Say $s \equiv \{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\}$ and $t \equiv \{l_1 : t_1, \ldots, l_n : t_n\}$ and $s < t$, then

$$\mathsf{coerce}[s < t] \equiv \lambda x : s.\ \{l_1 = \mathsf{coerce}[s_1 < t_1](x.l_1), \ldots l_n = \mathsf{coerce}[s_1 < t_n](x.l_n)\}$$

$$\mathbf{0} : \mathbf{num} \qquad \mathbf{false} : \mathbf{bool} \qquad \mathbf{true} : \mathbf{bool}$$

$$\frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{Pred}(e) : \mathbf{num}} \qquad \frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{Succ}(e) : \mathbf{num}} \qquad \frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{IsZero}(e) : \mathbf{bool}}$$

$$H,\ x : s,\ H' \vdash x : s \qquad \frac{H,\ x : s \vdash e : t}{H \vdash \lambda x : s.\ e : t} \qquad \frac{H \vdash e : s \to t \qquad H \vdash e' : s}{H \vdash e(e') : t}$$

$$\frac{H,\ x : s \vdash e : s}{H \vdash \mu x : s.\ e : s} \qquad \frac{H \vdash e : \mathbf{bool} \qquad H \vdash e' : s \qquad H \vdash e'' : s}{H \vdash \mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' : s}$$

$$\frac{H \vdash e_1 : s_1 \quad \cdots \quad H \vdash e_n : s_n}{H \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : s_1, \ldots, l_n = s_n\}} \qquad \frac{H \vdash e : \{l_1 : s_1, \ldots, l_n : s_n\}}{H \vdash e.l_i : s_i}$$

$$\frac{H \vdash e_i : s_i}{H \vdash [l_i = e_i] : [l_1 : s_1, \ldots, l_n = s_n]}$$

$$\frac{H \vdash e : [l_1 : s_1, \ldots, l_n = s_n] \qquad H \vdash f_1 : s_1 \to s \quad \cdots \quad H \vdash e_n : s_n \to s}{H \vdash \mathbf{case}\ e\ \mathbf{of}\ l_1 \Rightarrow f_1 \cdots l_n \Rightarrow f_n : s}$$

Table 1: Typing rules for PCF+.

$$\begin{array}{c} \mathbf{num} < \mathbf{num} \\ \mathbf{bool} < \mathbf{bool} \end{array} \qquad \frac{s' < s \qquad t < t'}{s \to t < s' \to t'}$$

$$\frac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{\{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\} < \{l_1 : t_1, \ldots, l_n : t_n\}}$$

$$\frac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{[l_1 : s_1, \ldots, l_n : s_n] < [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]}$$

Table 2: Inheritance rules.

- Say $s \equiv [l_1 : s_1, \ldots, l_n : s_n]$ and $t \equiv [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]$ and $s < t$, then

$$\mathsf{coerce}[s < t] \equiv \lambda x : s.\ \mathbf{case}\ x\ \mathbf{of}\ l_1 \Rightarrow f_1, \ldots, l_n \Rightarrow f_n$$

  where $f_i \equiv \lambda y : s_i.\ [l_i = \mathsf{coerce}[s_i < t_i](y)]$ for each $i = 1, \ldots, n$. ▮

Lemma 1 *If $s < t$ is derivable, then so is $\vdash \mathsf{coerce}[s < t] : s \to t$* ▮

We will now describe how we translate the derivations of typing judgments of PCF++ into derivations of PCF+. The translation is defined by recursion on the structure of the derivation trees. Since these are freely generated by the typing rules, it is sufficient to provide for each rule of PCF++ a corresponding rule on trees of PCF+ judgments. For the correspondence which we describe, it is possible to show that these corresponding rules are *directly derivable* in PCF+, therefore the translation takes derivations in PCF++ into derivations in PCF+.

A PCF++ derivation Δ yielding an inheritance judgment $H \vdash e : s$ is translated as a tree $T\Delta$ of PCF+ judgments yielding a *translation* $T^*\Delta$ of the form $H \vdash e^* : s$. All of the rules of PCF++ except the subsumption rule are translated "without change." For example, the axiom $\mathbf{0} : \mathbf{num}$ is translated as itself, whereas the rule

$$\frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{Succ}(e) : \mathbf{num}}$$

is translated as

$$\frac{H \vdash e^* : \mathbf{num}}{H \vdash \mathbf{Succ}(e^*) : \mathbf{num}}$$

where $H \vdash e^* : \mathbf{num}$ is the root of the translation of the derivation of $H \vdash e : \mathbf{num}$. Only the subsumption rule is altered by the translation. In particular, the rule

$$\frac{H \vdash e : s \qquad s < t}{H \vdash e : t}$$

is translated by the rule

$$\frac{H \vdash e^* : s \qquad \mathsf{coerce}[s < t] : s \to t}{H \vdash \mathsf{coerce}[s < t](e^*) : t}$$

which "makes the implicit coercion explicit."

It is not hard to see that a PCF++ typing judgement may have many different derivations. The reader may wish to look at different possible derivations for the term in the introduction to get a sense of why this is the case. This presents a problem for the translation: is there any sense in which two translations to PCF+ of a given PCF++ term are related? In particular, this paper's main theorem can be used to demonstrate a close relationship between the *operational* semantics of the two translations.

2.3 Operational semantics and Main Theorem.

The operational semantics of the closed raw terms of is given by the least relation $\Downarrow$ between raw terms and canonical forms which satisfies the rules and axioms in Table 3. Canonical froms are defined as follows: $\mathbf{0}$, $\mathbf{true}$, and $\mathbf{false}$ are canonical forms. For any expression e, $\lambda x : s.\ e$ is a canonical form. If $c_1, \ldots c_n$ are canonical forms, then $\{l_1 = c_1, \ldots, l_n = c_n\}$ is a canonical form. If c is a canonical form, then $\mathbf{Succ}(c)$ and $[l = c]$ are canonical forms. We may also write $e \Downarrow$ if there is a canonical form c such that $e \Downarrow c$.

$$\mathbf{0} \Downarrow \mathbf{0} \qquad \mathbf{true} \Downarrow \mathbf{true} \qquad \mathbf{false} \Downarrow \mathbf{false}$$

$$\frac{e \Downarrow \mathbf{Succ}(c)}{\mathbf{Pred}(e) \Downarrow c} \qquad \frac{e \Downarrow c}{\mathbf{Succ}(e) \Downarrow \mathbf{Succ}(c)} \qquad \frac{e \Downarrow \mathbf{0}}{\mathbf{IsZero}(e) \Downarrow \mathbf{true}} \qquad \frac{e \Downarrow \mathbf{Succ}(c)}{\mathbf{IsZero}(e) \Downarrow \mathbf{false}}$$

$$\lambda x : s.\ e \Downarrow \lambda x : s.\ e \qquad \frac{e \Downarrow \lambda x : s.\ e'' \qquad e' \Downarrow c' \qquad [c'/x]e'' \Downarrow c}{e(e') \Downarrow c}$$

$$\frac{e_1 \Downarrow \mathbf{true} \qquad e_2 \Downarrow c}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow c} \qquad \frac{e_1 \Downarrow \mathbf{false} \qquad e_3 \Downarrow c}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow c}$$

$$\frac{e_1 \Downarrow c_1 \quad \cdots \quad e_n \Downarrow c_n}{\{l_1 = e_1, \ldots, l_n = e_n\} \Downarrow \{l_1 = c_1, \ldots, l_n = c_n\}} \qquad \frac{e \Downarrow \{l_1 = c_1, \ldots, l_n = c_n\}}{e.l_i \Downarrow c_i}$$

$$\frac{e \Downarrow c}{[l = e] \Downarrow [l = c]} \qquad \frac{e \Downarrow [l_i = c'] \qquad f_i(c') \Downarrow c}{\mathbf{case}\ e\ \mathbf{of}\ l_1 \Rightarrow f_1, \ldots, l_i \Rightarrow f_i, \ldots, l_n \Rightarrow f_n \Downarrow c}$$

$$\frac{[\mu x.\ e/x]e \Downarrow c}{\mu x.\ e \Downarrow c}$$

Table 3: Call-by-value evaluation.

For raw terms e and e' we write $[e'/x]e$ for the result of substituting e' for x in e. We demand all of the usual assumptions about the renaming of bound variables in e to avoid capturing free variables of e'. We assume that the substitution operation associates to the right and we may write $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$ for the simultaneous substitution of $e_1, \ldots, e_n$ for $x_1, \ldots, x_n$ respectively in e. In the event that the terms e_i are closed, note that this is the same as $[e_1/x_1] \cdots [e_n/x_n]e$ and, indeed, the order of the substitutions does not matter.

It is not hard to see that if e is a closed raw term such that $e \Downarrow c$, then c is uniquely determined. This can be proved by showing that, for a given term e, there is at most one axiom or rule from Table 3 which applies to it. Hence the rules define a deterministic evaluation strategy. The evaluation of function application is call-by-value, since the argument to the application is evaluated before being substituted into the body of the applied procedure. There is no evaluation under a lambda-abstraction, but note that records are eagerly evaluated. For example, the evaluation of an expression $\{l = e, l' = e'\}.l$ will result in the evaluation of e' as well as e even though e' is "not needed" in the result. Putting aside efficiency issues, this is only significant if e' diverges since, in that case, the evaluation of $\{l = e, l' = e'\}.l$ will also diverge. Since evaluation is deterministic, we may define a partial function $\mathcal{E}$ on raw terms as follows

$$\mathcal{E}(e) \simeq \begin{cases} c & e \Downarrow c \text{ if there is such a } c \\ \textit{undefined} & \text{otherwise} \end{cases}$$

We use the symbol $\simeq$ between mathematical expressions to indicate that one of the expressions being related may be undefined. In general, for expressions E and E', $E \simeq E'$ means that if either E or E' is defined, then so is the other and the values are the same.

Let $\vdash e : s$ be a judgement which type-checks in PCF+ and suppose $e \Downarrow c$. It is easy to show that $\vdash c : s$ also type-checks in PCF+. This fact is less obvious for PCF++. We express it in the following:

Lemma 2 *Suppose $\vdash e : s$ is derivable in PCF++ and $e \Downarrow c$, then $\vdash c : s$.* ▌

This sort of result is closely related to the subject reduction theorems that appear in type theory research.

Let e be raw term such that $\vdash e : s$ is a derivable in PCF++ and suppose e^* is a translation of e into PCF+. Our central question is this: *what, if anything, is the relationship between $\mathcal{E}(e)$ and $\mathcal{E}(e^*)$?* Naturally, we might start by guessing that $\mathcal{E}(e) \simeq \mathcal{E}(e^*)$ in the sense that when one of them exists, then so does the other, and the results of evaluation are syntactically identical. However, it does not take much looking to see that the syntactic identity may fail in some cases. First of all, if $\mathcal{E}(e)$ is a record, then it may contain some fields which do not appear in the result $\mathcal{E}(e^*)$ of evaluating the coerced term since the latter evaluation will include coercions which may strip various fields in the course of the evaluation of e^*. Moreover, if $\mathcal{E}(e)$ is a lambda term, then $\mathcal{E}(e^*)$ may contain unexecuted coercions in its body which do not appear in $\mathcal{E}(e)$. Worse yet, it seems that even two different translations of $e : s$ may have different canonical forms! Hence we cannot expect a result as simple as the one just proposed and, indeed, we cannot expect a simple-minded statement of an operational coherence result. Nevertheless, there are some obvious counter-observations to the problems just mentioned. In the case of records, the extra fields which appear in $\mathcal{E}(e)$ may be "junk fields" which were not mentioned in the type s. One might argue that it is not even desirable that the result of the evaluation should have fields not included in the specified type s. Could it be that $\mathcal{E}(e)$ and $\mathcal{E}(e^*)$ share "essential" fields in common? Also, the problem with higher types (lambda-abstractions) misses a central point: the "appearance" of a term at non-observable type is not important. Since most interpreters do not display any description of a higher-order procedure, we are interested only in the applicative behavior of such terms in observable contexts. Our goal is therefore to define what we mean by an observable type and define a notion of essential observable equivalence for PCF++ judgements at these types.

Definition: Types **bool** and **num** are *ground* types. A type s is *observable* if

- s is a ground type, or
- $s \equiv \{l_1 : s_1, \ldots, l_n : s_n\}$ where $s_1, \ldots, s_n$ are observable types, or
- $s \equiv [l_1 : s_1, \ldots, l_n : s_n]$ where $s_1, \ldots, s_n$ are observable types. ▌

Definition: The relation $=_s$ between canonical forms of PCF++ observable type s is defined inductively as follows:

- If s is a ground type, then $c =_s c'$ iff $c \equiv c'$.
- Let $s = \{l_1 : s_1, \ldots, l_n : s_n\}$, then

$$\{l_1 = c_1, \ldots, l_n = c_n, \ldots, l_j = c_j\} =_s \{l_1 = c'_1, \ldots, l_n = c'_n, \ldots, l'_k = c'_k\}$$

iff $c_i =_{s_i} c'_i$ for $i = 1, \ldots, n$.

- Let $s = [l_1 : s_1, \ldots, l_n : s_n]$, then $[l_i = c_i] =_s [l_j = c'_j]$ iff $c_i =_{s_i} c'_j$. ∎

If E and E' are expressions that may be undefined, write $E \simeq_s E'$ to mean that if one expression exists, then so does the other and $E =_s E'$. We may now express the desired result:

Main Theorem: *Suppose* $\vdash e : s$ *is derivable in PCF++ and* e^* *is any PCF+ term which translates this sequent, then* $e \Downarrow$ iff $e^* \Downarrow$. *Moreover, if* s *is observable, then* $\mathcal{E}(e) \simeq_s \mathcal{E}(e')$. ∎

It seems difficult to prove this result directly because of the recursion case. This problem is resolved by appealing to denotational models for PCF+ and PCF++ which we now describe.

3 A computationally adequate denotational semantics.

For technical reasons we have found that it is useful to appeal to some results relating PCF+ and PCF++ to a specific denotational model which we will describe in this section. Although our goal is to prove a purely syntactic result (the Main Theorem at the end of the previous section), the semantic results which we will now establish are of independent interest.

We describe a domain-theoretic model for PCF+. The interpretation of types is as follows:

- $[\![\mathbf{bool}]\!]$ is the flat domain with three distinct elements *tt*, *ff* and least element $\bot$.
- $[\![\mathbf{num}]\!]$ is the flat domain consisting of the numbers $0, 1, 2, \ldots$ together with a least element $\bot$.
- $[\![s \to t]\!] = (s \multimap t)_\bot$, the lifted domain of strict (*i.e.* $\bot$-preserving) functions from $[\![s]\!]$ into $[\![t]\!]$.
- $[\![\ \{l_1 : s_1, \ldots, l_n : s_n\}\ ]\!]$ consists of a bottom element $\bot$, together with the set of tuples $\{l_1 = d_1, \ldots, l_n = d_n\}$ where each d_i is a non-bottom element of $[\![s_i]\!]$. The ordering is defined by
$$\{l_1 = d_1, \ldots, l_n = d_n\} \sqsubseteq \{l_1 = d'_1, \ldots, l_n = d'_n\}$$
iff $d_i \sqsubseteq d'_i$ for each $i = 1, \ldots, n$ and $\bot \sqsubseteq d$ for each record d.
- $[\![\ [l_1 : s_1, \ldots, l_n : s_n]\ ]\!]$ consists of a bottom element $\bot$, together with the set of pairs $[l_i = d_i]$ such that d_i is a non-bottom element of $[\![s_i]\!]$. For two such pairs, $[l_i = d] \sqsubseteq [l_j = d']$ iff $i = j$ and $d \sqsubseteq d'$.

Suppose $H = x_1 : s_1, \ldots x_n : s_n$ is a type context. An *H-environment* is a function which assigns to each variable x_i an element $\rho(x_i)$ of the domain $[\![s_i]\!]$. The PCF+ interpretation of a sequent $H \vdash e : s$ is a function which assigns to each H-environment ρ a value $[\![H \vdash e : s]\!]^{++}\rho$ in $[\![s]\!]$.

We will refrain from writing out all of the semantic equations for the sequents of PCF+. The rules for the introduction and elimination operators for the record and variant types are straightforward, holding in mind that the interpretation of a record with a field which is $\bot$ is itself equal to $\bot$. Recursion is defined in the usual way using least fixed points. The function space requires some explanation which we now provide.

The *lift* $D_\bot$ of a domain D is obtained by adding a new bottom element. There is a continuous function $\mathsf{up} : D \to D_\bot$ which sends elements of D to their images in the lifted domain. This function is not strict, since it sends the bottom of D to an element of $D_\bot$ which dominates the "new" bottom element. There is a unique continuous strict function $\mathsf{down} : D_\perp \to D$ such that $(\mathsf{down} \circ \mathsf{up})(x) = x$ for any x and $(\mathsf{up} \circ \mathsf{down})(y) = y$ for any $y \neq \perp$. This equational relationship between the two functions plays an essential role in the computational adequacy result which we will state later. Now, the meaning of a derivable typing judgement of the form $H \vdash \lambda x.\, e : s \to t$ is given as follows:

$$[\![H \vdash \lambda x.\, e : s \to t]\!]^+\rho = \mathsf{up}(\mathsf{strict}\lambda d \in [\![s]\!].\ [\![H, x : s \vdash e : t]\!]^+\rho[d/x])$$

where $\rho[d/x]$ is the $H, x : s$ environment which is the same as ρ except it sends x to d (we assume that x is a "fresh" variable which does not appear in H) and the second lambda abstraction is the "semantic" notation for a function which takes an argument $d \in [\![s]\!]$. Since the interpretation a function application to a program with value $\perp$ should have value $\perp$ to model call-by-value properly, one must apply the function strict defined as

$$\mathsf{strict}(f)(x) = \begin{cases} f(x) & \text{if } x \neq \perp \\ \perp & \text{if } x = \perp \end{cases}$$

The resulting strict function is lifted by the function up to insure that its value is non-bottom. Again, this will be important later when we prove a correspondence between operational divergence and having $\perp$ as a meaning. Under our intended operational semantics, no lambda-abstraction is a divergent program. The definition of application is given as follows:

$$[\![H \vdash e(e') : t]\!]^+\rho = \mathsf{down}([\![H \vdash e : s \to t]\!]^+\rho)([\![H \vdash e' : s]\!]\rho)$$

We may now show how our model for PCF+ can be used to construct a model for PCF++. Following ideas from [BCGS89] we use the following Semantic Coherence Theorem due to Rick Blute:

Theorem 3 *(Semantic Coherence) If Γ and Δ are PCF++ derivations of a sequent $H \vdash e : s$, then $[\![H \vdash T^*(\Gamma) : s]\!]^{++} = [\![H \vdash T^*(\Delta) : s]\!]^{++}$.* ▮

A similar result was a central objective of the work in [BCGS89] where the coherence is proved for a class of models of an equational theory. The model here differs from the ones considered there since the unrestricted β rule does not hold in the model we have described in the current paper. The semantic function for sequents of PCF++ is now defined as follows: $[\![H \vdash e : s]\!]^{++}\rho = [\![H \vdash e^* : s]\!]^+\rho$ where $H \vdash e^* : s$ is any translation of $H \vdash e : s$. If we note that any PCF+ derivation *is* a PCF++ derivation, then we get the following corollary:

Corollary 4 *If $H \vdash e : s$ is a derivable judgment of PCF+, then $[\![H \vdash e : s]\!]^+\rho = [\![H \vdash e : s]\!]^{++}\rho$ for any H-environment ρ.* ▮

As we shall see later, this corollary permits us to transfer some hard-earned results about PCF++ to results about PCF+. In light of the corollary, we may sometimes omit the tags on the semantic brackets for PCF+ derivable typing judgements.

We now wish to show that the semantics for PCF++ which we have just defined is closely related to its operational semantics. Here is our first crucial relationship:

Theorem 5 *(Soundness) If $e : s$ is derivable in PCF++, and $e \Downarrow c$, then $[\![e : s]\!]^{++} = [\![c : s]\!]^{++}$.* ▮

We have omitted the proof, which is straight-forward but tedious. We mention only the following facts which are needed:

Lemma 6 *1. If $r < s < t$, then $[\![\mathsf{coerce}[r < t] : r \to t]\!] = [\![\mathsf{coerce}[s < t] : s \to t]\!] \circ [\![\mathsf{coerce}[r < s] : r \to s]\!]$*

2. If $s < t$, then $[\![\mathsf{coerce}[s < t] : s \to t]\!](d) = \bot$ iff $d = \bot$. ▌

Lemma 7 *If $\vdash c : s$ is a derivable judgement of PCF++ and c is a canonical form, then $[\![c : s]\!]^{++} \neq \bot$.* ▌

Most of the rest of this section is devoted to a proof of a kind of converse to the Soundness Theorem which we will call *computational adequacy* (the term is suggested by Albert Meyer [Mey88], although his definition includes soundness). For PCF++, it can be stated as follows:

Theorem 8 *(Computational Adequacy.) Suppose $e : s$ is derivable in PCF++. If $[\![e : s]\!]^{++} \neq \bot$ then $e \Downarrow c$ for some canonical form c.*

We focus on explaining how the methods that one uses for results such as those above are applied to a calculus with multiple inheritance. We will look at the proof of adequacy in some detail. The proof requires a relation between program meanings and programs sometimes called an "inclusive predicate". We define this relationship as follows:

Definition: Define a relation $\lesssim_s$ between elements of $[\![s]\!]$ on the left and closed raw terms of type s on the right as follows. $d \lesssim_s e$ if $d = \bot$ or $e \Downarrow c$ for some c and $d \lesssim_s c$ where

- $f \lesssim_{s \to t} \lambda x : r.\ e$ iff for each $d \in [\![s]\!]$ and term c, $d \lesssim_s c$ implies $\mathsf{down}(f)(d) \lesssim_t [c/x]e$.
- $\{l_1 = d_1, \ldots, l_n = d_n\} \lesssim_{\{l_1 : s_1, \ldots, l_n : s_n\}} \{l_1 = e_1, \ldots, l_m = e_m\}$ iff $m \geq n$ and $d_i \lesssim_{s_i} c_i$ for $i = 1, \ldots n$.
- $[l_i = d] \lesssim_{[l_1 : s_1, \ldots, l_n : s_n]} [l_j = c]$ iff $i = j$ and $d \lesssim_{s_i} c$.
- $tt \lesssim_{\mathbf{bool}}$ **true** and $f\!f \lesssim_{\mathbf{bool}}$ **false**.
- $0 \lesssim_{\mathbf{num}} \mathbf{0}$ and if $n \lesssim_{\mathbf{num}} c$ for a number n, then $n + 1 \lesssim_{\mathbf{num}} \mathbf{Succ}(c)$. ▌

Some of the essential semantic properties of $\lesssim$ are given in the following:

Lemma 9 *1. If $a \sqsubseteq b \lesssim_s e$, then $a \lesssim_s e$.*

2. If $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots$ is an ascending chain and $a_n \lesssim_s e$ for each n, then $\bigsqcup_{n=0}^{\infty} a_n \lesssim_s e$. ▌

We are now ready to sketch the proof of the primary technical lemma which is needed for the proof of PCF++ adequacy.

Lemma 10 *Suppose $H = x_1 : s_1^\dagger \ldots x_n : s_n^\dagger$ and $H \vdash e^\dagger : s^\dagger$ is derivable. If $d_i \in [\![s_i^\dagger]\!]$ and $d_i \lesssim_{s_i^\dagger} e_i^\dagger$ for $i = 1, \ldots, k$, then $[\![H \vdash e^\dagger : s^\dagger]\!]^{++}[d_1, \ldots, d_k / x_1, \ldots, x_k] \lesssim_{s^\dagger} [e_1^\dagger, \ldots, e_k^\dagger / x_1, \ldots, x_k]e^\dagger$.*

Proof: Let ρ be the environment $[d_1, \ldots, d_n / x_1, \ldots x_n]$ and σ be the substitution $[e_1^\dagger, \ldots, e_n^\dagger / x_1, \ldots, x_n]$. Let Δ be a PCF++ derivation of the typing judgement $H \vdash e^\dagger : s^\dagger$. We prove that $[\![H \vdash e^\dagger : s^\dagger]\!]^{++}\rho \lesssim_{s^\dagger} \sigma e^\dagger$ by an induction on Δ. Assume that the Theorem is known for proofs of lesser height. There are eleven possibilities for the last step of Δ. Some of the more interesting cases (subsumption in particular) are written out fully below.

- *Base case:* $H \vdash x_i : s_i^\dagger$.
  Suppose the sequent $H \vdash e^\dagger : s^\dagger$ is an axiom of the form above (*i.e.* $e^\dagger \equiv x_i$). Then we have $[\![H \vdash x_i : s_i^\dagger]\!]^{++}\rho = d_i \lesssim_{s_i^\dagger} e_i \equiv \sigma x_i$ by assumption.

- *Lambda abstraction:* $\dfrac{H,\ x : s \vdash e : t}{H \vdash \lambda x : s.\ e : s \to t}$
  Suppose the last inference in Δ has the form above (in particular, $e^\dagger : s^\dagger \equiv \lambda x : s.\ e : s \to t$). Let Δ' be part of the proof Δ which proves $H,\ x : s \vdash e : t$ and suppose $H,\ x : s \vdash e^* : t$ is $T^*\Delta'$. Let $f = [\![H \vdash \lambda x : s.\ e : s \to t]\!]^{++}\rho = [\![H \vdash \lambda x : s.\ e^* : s \to t]\!]^{+}\rho$ and suppose $d \lesssim_s c$. We must show that
  $$d' = \mathsf{down}(f)(d) \lesssim_t (\sigma\lambda x : r.\ e)(c) \tag{1}$$
  If $d' = \bot$ then there is no problem. Suppose $d' \neq \bot$, then $\mathsf{down}(f) \neq \bot$, so
  $$\begin{aligned} d' &= \mathsf{down}(\mathsf{up}(\mathsf{strict}\lambda d'' \in [\![s]\!].\ [\![H, x : s \vdash e^* : t]\!]^{+}\rho[d''/x]))(d) \\ &= (\mathsf{strict}\lambda d'' \in [\![s]\!].\ [\![H, x : s \vdash e^* : t]\!]^{+}\rho[d''/x])(d) \end{aligned}$$
  and there are two cases. if $d = \bot$, then $d' = \bot \lesssim_t \sigma[c/x]e$ as desired. However, if $d \neq \bot$, then
  $$\begin{aligned} d' &= [\![H, x : s \vdash e^* : t]\!]^{+}\rho[d/x] \\ &= [\![H, x : s \vdash e : t]\!]^{++}\rho[d/x] \\ &\lesssim_t \sigma[c/x]e \end{aligned}$$
  by the induction hypothesis. Since $d' \neq \bot$, there is a canonical c' such that $\sigma[c/x]e \Downarrow c'$ and $d' \lesssim_t c'$. Since $[c/x]\sigma e \equiv \sigma[c/x]e$, we have $(\lambda x : s.\ \sigma e)(c) \equiv (\sigma\lambda x : s.\ e)(c)$ so it must be the case that $(\sigma\lambda x : s.\ e)(c) \Downarrow c'$ too, so 1 holds.

- *Application:* $\dfrac{H \vdash e_1 : s \to t \qquad H \vdash e_2 : s}{H \vdash e_1(e_2) : t}$
  Let $H \vdash e_1^* : s \to t$ and $H \vdash e_2^* : s$ be translations dictated by Δ. If $d' = [\![H \vdash e_1(e_2) : t]\!]^{++}\rho = [\![H \vdash e_1^*(e_2^*) : t]\!]^{+}\rho \neq \bot$, then $f = [\![H \vdash e_1 : s \to t]\!]^{++}\rho = [\![H \vdash e_1^* : s \to t]\!]^{+}\rho \neq \bot$ and $d = [\![H \vdash e_2 : s]\!]^{++}\rho = [\![H \vdash e_2^* : s]\!]^{+} \neq \bot$ (using the fact that all our functions are strict!). By the induction hypothesis, $f \lesssim_{s \to t} \sigma e_1 : s \to t$ and $d \lesssim_s \sigma e_2 : s$, so there is a term e_3 and a canonical form c such that
  $$\begin{aligned} &\sigma e_1 \Downarrow \lambda x : s.\ e_3 \text{ and } f \lesssim_{s \to t} \lambda x : s.\ e_3 \\ &\sigma e_2 \Downarrow c \text{ and } d \lesssim_s c \end{aligned}$$
  Now $d' = \mathsf{down}(f)(d) \lesssim_t [c/x]e_3$ by the definition of $\lesssim_{s \to t}$, so there is a canonical c' such that $[c/x]e_3 \Downarrow c'$ and $d' \lesssim_t c'$. But $[c/x]e_3 \Downarrow c'$ means $(\sigma e_1)(\sigma e_2) \Downarrow c'$. Since $(\sigma e_1)(\sigma e_2) \equiv \sigma(e_1(e_2))$, we have $d' \lesssim_t \sigma(e_1(e_2))$ as desired.

- *Recursion:* $\dfrac{H,\ x : s \vdash e : s}{H \vdash \mu x : s.\ e : s}$.
  Let $H,\ x : s \vdash e^* : s$ be the translation dictated by Δ. Let $d_0 = \bot$ and $d_{i+1} = [\![H, x : s \vdash e : s]\!]^{++}\rho[d_i/x] = [\![H, x : s \vdash e^* : s]\!]^{+}\rho[d_i/x]$. We show that $d_i \lesssim_s \sigma\mu x : s.\ e$ for each i. This is immediate for $d_0 = \bot$. Suppose $d_i \lesssim_s \sigma\mu x : s.\ e$. By induction hypothesis, $d_{i+1} = [\![H, x : s \vdash e : s]\!]^{++}\rho[d_i/x] \lesssim_s \sigma[\mu x : s.e/x]e$. If $d_{i+1} \neq \bot$, then $\sigma[\mu x : s.\ e/x]e \Downarrow c$ for some c such that $d_{i+1} \lesssim_s c$. Now $\sigma[\mu x : s.\ e/x]e \equiv [\sigma\mu x : s.\ e/x]\sigma e \equiv [\mu x : s.\ \sigma e/x]\sigma e$. Hence $\sigma\mu x : s.\ e \equiv \mu x : s.\ \sigma e \Downarrow c$ as well. Since $[\![H \vdash \mu x : s.\ e : s]\!]^{++}\rho = [\![H \vdash \mu x : s.\ e^* : s]\!]^{+}\rho = \bigsqcup_{i=0}^{\infty} d_i$, the desired result follows.

- *Subsumption rule:* $\dfrac{H \vdash e : s \qquad s < t}{H \vdash e : t}$.

  The proof for this case is by induction on the height of the proof that $s < t$. Assume that we know that the theorem holds for $H \vdash e : s$ and let $H \vdash e^* : s$ be any translation of this sequent to PCF+. There are four subcases:

  – *Base types:* These are both obvious since the coercion is the identity map.

  – *Functions:* $\dfrac{u' < u \qquad v < v'}{u \to v < u' \to v'}$.

  Suppose $s \equiv u \to v$ and $t \equiv u' \to v'$. Let $\xi_1 = \mathsf{down}[\![\mathsf{coerce}[u' < u]]\!]$ and $\xi_2 = \mathsf{down}[\![\mathsf{coerce}[v < v']]\!]$. Then $\xi = \mathsf{down}[\![\mathsf{coerce}[u \to v < u' \to v']]\!]$ satisfies $\xi(f) = \xi_2 \circ f \circ \xi_1$ for $f : [\![u]\!] \circ\!\!\to [\![v]\!]$. Set $f = \mathsf{down}[\![H \vdash e : s]\!]^{++}\rho$. If $d \lesssim_{u'} c$, then $\xi_2(d) \lesssim_u c$ by induction hypothesis on $u' < u$. Thus $f(\xi_2(d)) \lesssim_v (\sigma e)(c)$ by induction hypothesis on $H \vdash e : s$. We may now apply the induction hypothesis on $v < v'$ to conclude that $\xi(f) = \xi_1(f(\xi_2(d))) \lesssim_{v'} (\sigma e)(c)$. Since $\xi(f) = [\![H \vdash e : t]\!]^{++}\rho$ we conclude that $[\![H \vdash e : t]\!]^{++}\rho \lesssim_t \sigma e$.

  – *Records:* $\dfrac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{\{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\} < \{l_1 : t_1, \ldots, l_n : t_n\}}$.

  Let $\xi_i = \mathsf{down}[\![\mathsf{coerce}[s_i < t_i]]\!]$ for $i = 1, \ldots n$ and let $\xi = \mathsf{down}[\![\mathsf{coerce}[s < t]]\!]$. By induction hypothesis, we have $d = [\![H \vdash e : s]\!]^{++}\rho \lesssim_s \sigma e$. If $d = \bot$, then $\xi(d) = [\![H \vdash e : t]\!]^{++}\rho = \bot$ and we are done. If $d \neq \bot$, then $d = \{l_1 = d_1, \ldots, l_m = d_m\}$ where $d_1, \ldots, d_m \neq \bot$ and $\sigma e \Downarrow c$ for some canonical c of the form $c \equiv \{l_1 = c_1, \ldots, l_j = c_j\}$ such that $j \geq m$ and $d_i \lesssim_{s_i} c_i$ for $i = 1, \ldots m$. By the induction hypothesis on inheritance judgements, we must therefore have $\xi_i(d_i) \lesssim_{t_i} c_i$ for each $i = 1, \ldots, n$. Hence $\xi(d) = \{l_1 = \xi_1(d_1), \ldots, l_n = \xi_n(d_n)\} \lesssim_t \{l_1 = c_1, \ldots, l_j = c_j\}$ by the definition of $\lesssim_t$ and we are done.

  – *Variants:* $\dfrac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{[l_1 : s_1, \ldots, l_n : s_n] < [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]}$.

  Let $\xi_i = \mathsf{down}[\![\mathsf{coerce}[s_i < t_i]]\!]$ for $i = 1, \ldots n$ and let $\xi = \mathsf{down}[\![\mathsf{coerce}[s < t]]\!]$. By induction hypothesis, we have $d = [\![H \vdash e : s]\!]^{++}\rho \lesssim_s \sigma e$. If $d = \bot$, then $\xi(d) = [\![H \vdash e : t]\!]^{++}\rho = \bot$ and we are done. If $d \neq \bot$, then $d = [l_i = d_i]$ where $d_i \neq \bot$ and $\sigma e \Downarrow c$ where $\sigma e \Downarrow c$ and $d \lesssim_s c$. By the definition of $\lesssim_s$, the term c has the form $[l_i = c_i]$ and $d_i \lesssim_{s_i} c_i$. By induction hypothesis on $s_i < t_i$, we know that $\xi_i(d_i) \lesssim_{t_i} c_i$ so $\xi(d) = [l_i = \xi_i(d)] \lesssim_t [l_i = c_i]$. ▌

We may now express the desired proof of Computational Adequacy for PCF++.

Proof: (of Theorem 8) By Lemma 10 we know that $[\![e : s]\!]^{++} \lesssim_s e$. Since the value on the left is assumed to differ from $\bot$, the Theorem follows immediately from the definition of $\lesssim_s$. ▌

The following theorem follows immediately from Soundness and Computational Adequacy for PCF++ together with Corollary 4 of the Semantic Coherence Theorem for PCF++.

Theorem 11 *(Soundness and Adequacy for PCF+) If* $\vdash e : s$ *is derivable in PCF+, then*

1. *(Soundness)* $e \Downarrow c$ *implies* $[\![e : s]\!] = [\![c : s]\!]$.

2. *(Computational Adequacy)* $[\![e : s]\!]^{+} \neq \bot$ *implies* $e \Downarrow c$ *for some canonical form* c. ▌

The following lemma is needed for the proof of the Main Theorem:

Lemma 12 *Let c and c' be canonical forms such that $\vdash c : s$ and $\vdash c' : s$ are derivable in PCF++ for an observable type s. Then $[\![c : s]\!]^{++} = [\![c' : s]\!]^{++}$ iff $c =_s c'$.* ▮

Corollary 13 *Let e and e' be raw terms such that $\vdash e : s$ and $\vdash e' : s$ are derivable for an observable type s. Then $[\![e : s]\!]^{++} = [\![e' : s]\!]^{++}$ iff $\mathcal{E}(e) \simeq_s \mathcal{E}(e')$.*

Proof: This follows from Adequacy, Soundness and Lemma 12. ▮

Main Theorem: Suppose $\vdash e : s$ is derivable in PCF++ and e^* is any PCF+ term which translates this sequent, then $e \Downarrow$ iff $e^* \Downarrow$. Moreover, if s is observable, then $\mathcal{E}(e) \simeq_s \mathcal{E}(e')$. ▮

Proof: Suppose $e \Downarrow c$. Then $[\![e : s]\!]^{++} = [\![c : s]\!]^{++} \neq \bot$ by the Soundness Theorem for PCF++ and Lemma 7. Since $[\![e : s]\!]^{++} = [\![e^* : s]\!]^{+}$, we may conclude from Soundness and Adequacy for PCF+ that there is a canonical form c' such that $e^* \Downarrow c'$ and $[\![c' : s]\!]^{++} = [\![c : s]\!]^{++}$. If s is an observable type then $c =_s c'$ by Lemma 12.

Suppose conversely that $e^* \Downarrow c'$ for some canonical c'. By the Soundness for PCF+, $[\![e^* : s]\!]^{+} = [\![c' : s]\!]^{+} \neq \bot$. Hence $[\![e : s]\!]^{++} = [\![e^* : s]\!]^{+} \neq \bot$. By Adequacy and Soundness for PCF++, there is a canonical form c such that $e \Downarrow c$ and $[\![c : s]\!]^{++} = [\![c' : s]\!]^{++}$. Thus $c =_s c'$ by Lemma 12. ▮

4 Conclusions and directions for further research.

We have shown that inheritance-interpreted-as-definable-coercion semantic paradigm behaves well with respect to operational semantics. More specifically, we have shown that the coercion terms that we introduce in this interpretation, while possibly generating more computation, will only generate "harmless" computation, in particular that no unexpected divergence can be introduced, nor can expected divergence be lost. (In the process, we actually exhibited a nice domain-theoretic model which is sound and computationally adequate for PCF++'s straightforward operational semantics.)

There are at least two points where we can see improvements to our results. One problem is that we would like to strengthen the main theorem so as to say something interesting about the relationship between $\mathcal{E}(e)$ and $\mathcal{E}(e^*)$ when their type is not necessarily observable. The other problem is that the proof of Theorem 5 does not really use the particularities of the denotational semantics but rather the fact that certain identities between PCF+ terms hold in it. These two points are related and here is a conjectured improvement which would solve both problems.

Suppose $\vdash e : s$ type-checks in PCF++ and let e^* be any PCF+ translation of it. Further suppose that $e \Downarrow c$ for some canonical form c. By our main theorem, there is a canonical form c' such that $e^* \Downarrow c'$. We would like to relate c and c' as PCF+ terms, but $\vdash c : s$ may type-check in PCF++ only. So, let c^* be any PCF+ translation of it. What do we know about the relationship between c^* and c'? It is a consequence of the soundness results that the model we introduce in section 3 equates them. But equality in this model is Π^0_2-hard. Surely the relationship between c^* and c' is much simpler...

We believe that it is possible to formulate a reasonable logical theory about PCF+ terms, call it $\mathcal{T}$ in which c^* and c' can be shown to be *provably equal*. In fact, we believe that such a theory would be closely related to the call-by-value lambda-calculi studied by Plotkin and Moggi [Mog88]. This result would have the following pleasant corollaries. Let $\mathcal{D}^+$ be any denotational model of PCF+ in which the operational semantics and the axiomatization of $\mathcal{T}$ are sound (actually, we expect that the soundness of the later will imply that of the former). One immediately concludes that our translation is denotationally coherent with respect to $\mathcal{D}^+$, which induces a model $\mathcal{D}^{++}$ of PCF++, and that the operational semantics of PCF++ terms is sound in $\mathcal{D}^{++}$. Of course, by the main theorem of this paper, we can also get transfer of computational adequacy. Therefore, we would be able to neatly concentrate in the axiomatization of $\mathcal{T}$ all the conditions needed by a "good" model of PCF+ in order to become a model of PCF++ in accordance to our paradigm.

An intriguing question is whether $c^* = c'$ will turn out to be more than an r.e. statement, whether it is actually decidable? In other words, is full PCF+ computation required in order to systematically disentangle the coercions we introduce?

Finally, we should restate that we expect that the results of this paper generalize to more complicated type disciplines (Fun, Quest, *etc.*) and that analogs can be shown for call-by-name operational semantics.

5 Acknowledgements.

We thank Albert Meyer for inspiring us to examine operational issues surrounding calculi with multiple inheritance. We have had helpful discussions about our draft version of this paper with Hans Boehm, Luca Cardelli, Pierre-Louis Curien, Matthias Felleisen, Dave MacQueen, Jose Meseguer, John Mitchell and Mitch Wand.

Breazu-Tannen's research was partially supported by ONR Grant NOOO14-88-K-0634. Gunter's research was partially supported by ONR Grant NOOO14-88-K-0557. Breazu-Tannen and Gunter were also partially supported by ARO Grant DAAG29-84-K-0061. Scedrov's research was partially supported by NSF Grant CCR-87-05596, by ONR Grant NOOO14-88-K-0635, and by the 1987 Young Scientist Award from the Natural Sciences Association of the University of Pennsylvania.

References

[BCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explicit coercion (preliminary report). In R. Parikh, editor, *Logic in Computer Science*, pages 112–134, IEEE Computer Society, June 1989.

[BCGS90] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. University of Pennsylvania, Department of Computer and Information Science technical report number MS-CIS-89-01. Journal version of [BCGS89] submitted to *Information and Computation.*

[Car89] L. Cardelli. *Typeful programming.* Research Report 45, DEC Systems, Palo Alto, May 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Mey88] A. R. Meyer. Semantical paradigms: notes for an invited lecture. In Y. Gurevich, editor, *Logic in Computer Science*, pages 236–253, IEEE Computer Society, July 1988.

[Mog88] E. Moggi. *The Partial Lambda-Calculus.* PhD thesis, University of Edinburgh, 1988.

[OBB89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Datbase programming in Machiavelli—a polymorphic language with static type inference. In *SIGMOD Conference on the Management of Data*, pages 46–57, ACM, 1989.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Wan89] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the Symposium on Logic in Computer Science*, pages 92–97, IEEE, June 1989.

NETS AS TENSOR THEORIES[*]

Carl Gunter *Vijay Gehlot* [†]

University of Pennsylvania
Department of Computer and Information Sciences
Philadelphia, PA 19104 U.S.A.

October 1989

Abstract

This report is intended to describe and motivate a relationship between a class of nets and the fragment of linear logic built from the tensor connective. In this fragment of linear logic a net may be represented as a *theory* and a computation on a net as a *proof.* A rigorous translation is described and a soundness and completeness theorem is stated. The translation suggests connections between concepts from concurrency such as causal dependency and concepts from proof theory such as cut elimination. The main result of this report is a "cut reduction" theorem which establishes that any proof of a sequent can be transformed into another proof of the same sequent with the property that all cuts are "essential". A net-theoretic reading of this result tells that unnecessary dependencies from a computation can be eliminated resulting in a maximally concurrent computation. We note that it is possible to interpret proofs as arrows in the strictly symmetric strict monoidal category freely generated by a net and establish soundness of our proof reduction rules under this interpretation. Finally, we discuss how other linear connectives may be related to the concepts of internal and external choice.

1 Introduction

In this paper we explore the idea of describing the operational semantics of a net (the so-called "token game") in proof-theoretic terms. Under our approach, a net will correspond to a logical theory, and the token games on the net will be represented as proof trees in the "logic" of the net. This correspondence reveals an interesting relationship between concepts of proof theory (such as cut elimination) and fundamental concepts in concurrency (such as causal dependency) as they are

[*]This is an extended and revised version of the preliminary report that appeared in: **Application and Theory of Petri Nets**, edited by G. De Michelis, June 1989, pp. 174-191.

[†]Research of both authors is supported by Office of Naval Research Grant N00014-88-K-0557. Electronic mail addresses for the authors are `gunter@linc.cis.upenn.edu` and `gehlot@linc.cis.upenn.edu`

illustrated by net theory. Our proof-theoretic representation works for a certain class of nets in which events are uniquely determined by their pre and post conditions. Such nets are represented as sets of sequents in a fragment of linear logic based on the tensor connective.

Linear logic is a system introduced by J. Y. Girard based on the inspiration of his work on a class of mathematical domains called *coherence spaces* [7, 8]. One way of understanding propositional linear logic is to see it as a modification of propositional logic which takes seriously the concept of a *resource.* As such it is related to such systems as *relevance logic* which incorporate this concept as well (see [3] for a full discussion). Resources are also a familiar aspect of the theory of Petri nets. In what follows, we will attempt to convince the reader that the senses in which linear logic and Petri nets deal with resources have many things in common. Indeed, we will demonstrate a translation which characterizes the relationship exactly.

However, the way linear logic and nets represent resources is only a part of what we feel is a much more important common characteristic of the two theories: the way in which they illustrate *true concurrency.* It is well-known that nets provide an intuitive and pictorial way of seeing many fundamental ideas of concurrent computation. In what follows, we will show how this intuition may also be seen in the theories and proof trees of (a fragment of) linear logic.

Other researchers have independently looked at the relationship between Petri nets and linear logic. The work of Asperti [1, 2] follows much the same basic intuition that we discuss below for the tensor connective. Carolyn Brown at the University of Edinburgh has proven a result similar to our soundness and completeness theorem and studied a fragement of linear logic formulae with additional connectives [4]. Narciso Martì-Oliet and José Meseguer [11] have discussed the relationship between Petri nets and linear logic from the point of view of category theory. We would like to acknowledge the assistance of Jean Yves Girard, who provided much of the inspiration for this investigation. We also thank Dexter Kozen, Prakash Panangaden, and Andre Scedrov for ideas and encouragement and acknowledge helpful discussions with Eike Best, Ursula Goltz, Ugo Montanari, and Wolfgang Reisig.

Throughout the rest of the paper we will assume some familiarity with net theory and proof theory. Concepts and notations related to former can be found in [15] and [6]. For the latter, [18] and [17] are excellent references.

2 Relating Nets and Theories

In this section we outline the fragment of linear logic on which this paper will be concentrating. The theory will be given in the form of a Gentzen style sequent calculus.

A *tensor formula* is either a propositional atom or the tensor product $A \otimes B$ of tensor formulas A and B. A *tensor sequent* is a pair $\Gamma \vdash A$ where Γ is a list of tensor formulae. A *tensor theory* is a set of tensor sequents. Of course, any set of sequents T will generate a tensor theory $\mathsf{Th}(T)$ which is the least set of sequents containing T and closed under the rules in Figure 1. We say that

Structural Rules

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,(\text{Exchange}) \qquad \frac{}{A \vdash A}\,(\text{Identity}) \qquad \frac{\Gamma \vdash A \qquad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\,(\text{Cut})$$

Logical Rules

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,(\otimes\text{R}) \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\,(\otimes\text{L})$$

Figure 1: Structural and logical rules for a fragment of linear logic.

$\Gamma \vdash A$ is *provable in* T if $\Gamma \vdash A$ is in $\mathsf{Th}(T)$. We say that $\Gamma \vdash A$ is provable if it is in $\mathsf{Th}(\emptyset)$. Let us say that a pair $A \vdash\dashv B$ is provable if $A \vdash B$ and $B \vdash A$ are both provable. It is not hard to see from these axioms that the tensor connective is associative and commutative:

Proposition 1 *For any A, B, C, the sequents $A \otimes B \vdash\dashv B \otimes A$ and $(A \otimes B) \otimes C \vdash\dashv A \otimes (B \otimes C)$ are provable.* ▮

However, the tensor connective is *not* absorptive; for example, the sequent $A \otimes A \vdash A$ is not provable. It is therefore possible to think of a tensor formula as a *multi-set* (or "bag") of propositional atoms. Given a tensor formula A, let $\mathsf{M}(A)$ be the multi-set of propositional atoms determined by A. It follows from the proposition that tensor formulae A and B such that $\mathsf{M}(A) = \mathsf{M}(B)$ are equivalent, *i.e.* $A \vdash\dashv B$. Moreover, sequents $\Gamma \vdash A$ and $\Delta \vdash A$ are equivalent in the sense that each can be derived from the other if the lists Γ and Δ determine the same multi-set of propositions. For this reason, we will treat sequents as pairs $\Gamma \vdash A$ where Γ is a multi-set.

For the purposes of this paper, a *net* N is a set S_N of *places* together with a set T_N of pairs of multi-sets over S_N. A pair $t = ({}^{\bullet}t, t^{\bullet}) \in N$ is called a *transition* of the net with *pre-condition* ${}^{\bullet}t$ and *post-condition* $t^{\bullet}$. Of course, this is only one of the many flavors of nets that have been studied in the rich literature on such structures. Nets, as defined here, are similar to place/transition-systems as defined, for example, in [15]. However, our notion of net has less structure since there are no capacities and a transition is uniquely determined by its pre and post conditions. Moreover, a net in our sense does not have a specified initial marking. One of the appealing characteristics of nets is the way they lend themselves to pictorial representation. For example, the net N_0 consisting of the pairs $(\{A\}, \{B, B, C\})$ and $(\{B\}, \{A\})$ is pictured as a labelled graph in Figure 2.

Before we offer a technical definition of just how a net determines a theory, we will attempt to motivate the basic idea by means of examples. Consider the net N_1 pictured in Figure 3. In this net, if we are given a token on the condition A, then it is possible to fire the event r. Firing this event, exhausts the token on A but provides a token on B. Logically, let us read the event r as an axiom $A \vdash B$ meaning "from A it is possible to obtain B." Similar ideas apply to the events s, t and u which we may read as $B \vdash D$ and $A \vdash C$ and $C \vdash E$ respectively. Now, event v requires a

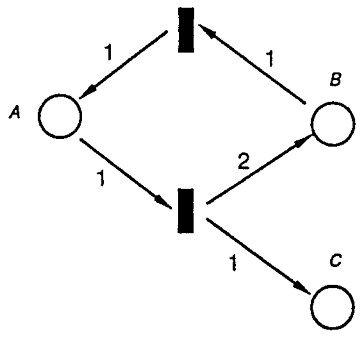


Figure 2: Net N_0.

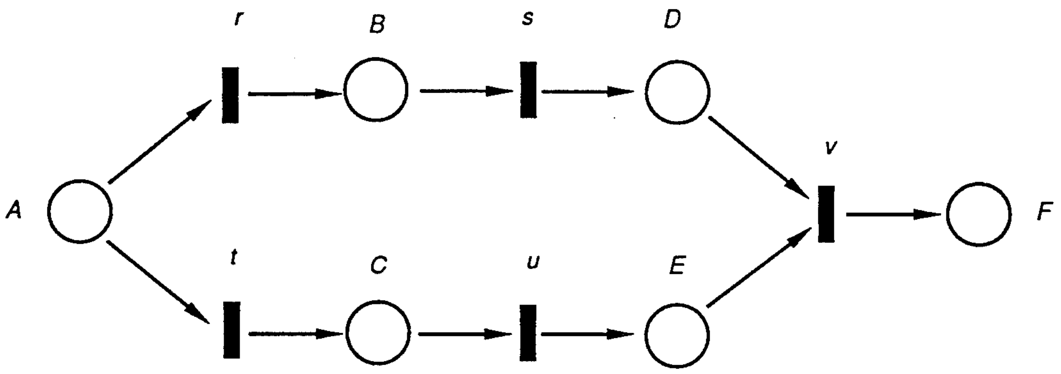


Figure 3: A net N_1 with concurrency and choice.

token on D and a token on E in order to fire and produce a token on F. We might therefore take $D, E \vdash F$ as the logical content of v. In summary, let T_1 be the set of axioms

$$\begin{array}{lll} A \vdash B & B \vdash D & \\ & & D, E \vdash F \\ A \vdash C & C \vdash E & \end{array}$$

Do these axioms somehow characterize the net "logically"? If one interprets the comma between the D and E in the way that one ordinarily does in logic, this tempts one to think of $D, E \vdash F$ as $D \wedge E \vdash F$. But something is now wrong with the proposed "logical interpretation" of the net. In particular, it is easy to check that $A \vdash F$ is provable from the axioms T_1. However, if one's interpretation of $A \vdash F$ is "from the resource A one is able to obtain the resource F," then the deduction is evidently incorrect. The problem lies in the fact that ordinary propositional logic does not support properly a concept of "proof resource." The culprit (in this case) is the rule from first order logic which gives us:

$$\frac{A \vdash D \qquad A \vdash E}{A \vdash D \wedge E}$$

This rule clearly does not reflect the desired intuition about resources. If I can use \$1 to buy a

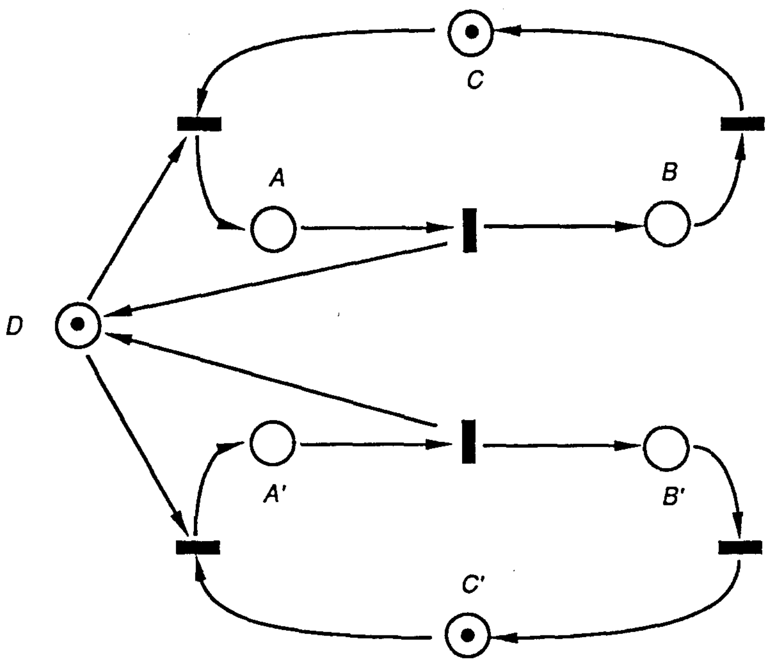


Figure 4: A net N_2 with a critical region.

pepsi and \$1 to buy a coke, then I can't expect to use \$1 to buy both a pepsi *and* a coke. Of course, one can also write the conjunction rule as

$$\frac{A \vdash D \qquad A \vdash E}{A, A \vdash D \wedge E}$$

but this only begs the issue, since some instance of the thinning rule:

$$\frac{\Gamma, X, X \vdash Y}{\Gamma, X \vdash Y}$$

would be used at a later step in the proof to remove the second copy of A and this rule is just as suspect as the earlier version of the conjunction rule. To deal with this problem, one needs a logic in which the thinning rule is omitted and the second of the conjunction rules is used for the "and" connective that we have in mind.

The proper rules are those given in Figure 1 for the linear logic tensor connective $\otimes$. These rules keep track of the resources as needed. In linear logic, the sequent $A \vdash F$ is *not* provable in T_1. However, it is possible to check that $A, A \vdash F$ *is* provable in T_1, as we expect it should be. There are, in fact, several proofs of $A, A \vdash F$ in T_1; three of these are listed in Figure 5 (on page 7). We will come back to these proofs later to discuss how they relate to the net token games that move a token from the marking A, A to the marking F.

To give a slightly larger example, which we hope will suffice in giving the reader the general idea, consider the net N_2 in Figure 4. This net corresponds to the tensor theory T_2 with the following six axioms:

$$\begin{array}{ccc} C \otimes D \vdash A & & B \vdash C \\ & A \vdash B \otimes D & \\ & A' \vdash B' \otimes D & \\ C' \otimes D \vdash A' & & B' \vdash C' \end{array}$$

As one might expect, it will never be the case that from starting marking C, C', D, the resource $A \otimes A'$ is obtained. More precisely, one can show that $C \otimes C' \otimes D \not\vdash A \otimes A' \otimes A$ for any choice of linear proposition A.

A formal definition may be now be expressed as follows. Let N be a net and let S be the set of places of N. These will be the propositional atoms over which we form a set of tensor sequents as follows:

$$\mathcal{L}(N) = \{A \vdash B \mid \mathsf{M}(A) = {}^{\cdot}t \text{ and } \mathsf{M}(B) = t^{\cdot} \text{ for some } t \in T_N\}$$

We will refer to $\mathcal{L}(N)$ as the *tensor theory determined by* N.

On the other hand, let T be a set of tensor sequents in a language with propositional atoms S. The theory T determines a net $\mathcal{N}(T)$ as the set $T_{\mathcal{N}(T)}$ of pairs $(\mathsf{M}(A), \mathsf{M}(B))$ such that $A \vdash B$ is in T. It is clear that $\mathcal{N}(\mathcal{L}(N)) = N$ for any net N. If $A' \vdash B'$ is an element of the set T whenever there is a sequent $A \vdash B$ in T such that $\mathsf{M}(A) = \mathsf{M}(A')$ and $\mathsf{M}(B) = \mathsf{M}(B')$, then it will also be the case that $\mathcal{L}(\mathcal{N}(T)) = T$. For example, the net N_1 in Figure 3 has $\mathcal{L}(N_1) = \{A \vdash B,\ B \vdash D,\ A \vdash C,\ C \vdash E,\ D \otimes E \vdash F,\ E \otimes D \vdash F\}$.

As the reader can guess from the examples, a marking M on a net N corresponds to a linear proposition A such that $\mathsf{M}(A) = M$. For example, the marking of the net N_1 in Figure 4 is represented by the proposition $C \otimes C' \otimes D$. In general, we have the following:

Theorem 2 (Soundness and Completeness) *Given a net N and markings M and M', the marking M' is in the forward marking set $[M\rangle$ of M if and only if the sequent $A \vdash A'$ is provable in the linear theory $\mathcal{L}(N)$ associated with N for any tensor formulae A and A' such that* $\mathsf{M}(A) = M$ *and* $\mathsf{M}(A') = M'$. ▮

We may apply the Soundness and Completeness Theorem to show how a non-trivial result from net theory leads to a result for a fragment of linear logic. Given a finite net N, it is decidable whether $M' \in [M\rangle$ for markings M and M'. This result is the culmination of a body of research which began with van Leeuwen [19] and has been worked on by a number of researchers [16, 9, 12, 10, 13]. Here is an immediate consequence:

Corollary 3 *Let N be a finite net and $\mathcal{L}(N)$ its associated linear theory. It is decidable whether $A \vdash B$ is provable in the theory $\mathcal{L}(N)$ for tensor formulae A and B.* ▮

Of course, the Corollary holds only for linear formulae in the small fragment of the system that we have discussed. Getting an assessment of how this result compares to known results about linear logic involves expanding our discussion to a larger fragment of the calculus. Since rules from $\mathcal{L}(N)$ may be used arbitrarily often, they must be represented as linear logic propositions using the "of course" operator, written $!A$. (Given a linear proposition A, the proposition $!A$ represents the "pure propositional content" of A. In the current context we may think of it as an unlimited resource of A's.) Linear propositional logic with the ! operator is not known to be decidable. The result above suggests that the decision procedure for this calculus, if it exists, will not be easy to find.

Proof 1.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A\vdash A \qquad A\vdash B}{A,A\vdash A\otimes B}{\scriptstyle\otimes\mathrm{R}} \qquad \dfrac{\dfrac{A\vdash C \qquad B\vdash B}{A,B\vdash C\otimes B}{\scriptstyle\otimes\mathrm{R}}}{A\otimes B\vdash C\otimes B}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash C\otimes B}{\scriptstyle Cut} \qquad \dfrac{\dfrac{B\vdash D \qquad C\vdash C}{B,C\vdash D\otimes C}{\scriptstyle\otimes\mathrm{R}}}{C\otimes B\vdash D\otimes C}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes C}{\scriptstyle Cut} \qquad \dfrac{\dfrac{D\vdash D \qquad C\vdash E}{D,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{D\otimes C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes E}{\scriptstyle Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

Proof 2.

$$\dfrac{\dfrac{\dfrac{A\vdash B \qquad A\vdash C}{A,A\vdash B\otimes C}{\scriptstyle\otimes\mathrm{R}} \qquad \dfrac{\dfrac{B\vdash D \qquad C\vdash E}{B,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{B\otimes C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes E}{\scriptstyle Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

Proof 3.

$$\dfrac{\dfrac{\dfrac{A\vdash B \qquad B\vdash D}{A\vdash D}{\scriptstyle Cut} \qquad \dfrac{A\vdash C \qquad C\vdash E}{A\vdash E}{\scriptstyle Cut}}{A,A\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

Figure 5: Three proofs that $A, A \vdash F$.

3 Proofs as Computations

Let us return now to our discussion of the net N_1 in Figure 3 (on page 4). This net displays some of the intuitive representations of concepts which have made nets an appealing model for both theoreticians and practitioners. The events r and t "compete" for the resource A and the events s and u are capable of running concurrently if they have the necessary resources B and C. There is a causal dependency between r and s: if r fires then s will be enabled. A similar dependency holds between t and u. If there is a line of computation which passes through r, s and another which passes through t, u, then these must "synchronize" before v is enabled. Most of these intuitions are represented in one form or another in the proof trees of the linear theory $\mathcal{L}(N_1)$. In particular, the *cut rule* corresponds to the concept of causal dependency or sequentialization. For example, to prove that $A \vdash D$, it is essential to use a cut. This relates to the fact that the event r must take place before the event s can be enabled. Basically, the only situation in which the cut rule is never needed for a proof is for a net whose theory is *trivial* since only in this case are there no causal dependencies! Hence, for the theory determined by a non-trivial net, we cannot expect that cut elimination is possible.

Given an initial marking $\{A, A\}$ on the net N_1, consider the following sequence of firings to produce F: first fire r, then fire t, then fire s, then fire u and then fire v. We can represent this by the following expression:

$$((((1_A \parallel r);(t \parallel 1_B));(s \parallel 1_C));(1_D \parallel u));v$$

where the semi-colon represents sequentialization, the parallel operator represents concurrency and an expression 1_X is the "idle event" on X. This computation is "maximally sequential" in the sense that it makes no real use of the possibility of doing two things "at the same time." This corresponds to a linear logic proof in which there are many applications of the cut rule. This proof is given as Proof 1 in Figure 5. But there are other ways the firing sequence from A, A to F could be carried out. For example: first fire r and t, then fire s and u, and after this, fire v. The following expression represents this firing sequence:

$$((r \parallel t);(s \parallel u));v.$$

This computation, which corresponds to Proof 2 in Figure 5, has still not made "maximal" use of concurrency, although it is better than the first firing sequence. Although r and t are not constrained to fire in any particular order, the event s, for example, is not permitted to fire until t has fired. This restriction is not really intrinsic to the causal dependencies of the net. On the other hand, it is clear that *no* firing sequence will allow v to be fired before both s and t have done so. The "best" or most concurrent firing sequence is therefore the following: fire r and then s while also firing t and then u, after this, fire v. This is represented by the expression:

$$((r;s) \parallel (t;u));v$$

which corresponds the Proof 3 in Figure 5.

Following these intuitions, it is desirable to provide a set of rewrite rules which will take proofs such as 1 and 2 and convert them to a "maximally concurrent" proof such as 3. This process resembles the cut elimination results from proof theory, but must differ in some ways since the cut elimination is being carried out in a theory in which cut *elimination* is impossible. A similar situation arises for cut elimination in a theory with equality where all but the cuts involving equational axioms can be eliminated. However, the "maximally concurrent" proof we desire cannot be obtained by a straight-forward translation of these ideas. Instead, it is necessary to rely on other intuitions about the correct forms.

4 Cut reduction.

In this section, we formalize the concepts intuitively discussed in the previous section. Our goal is to demonstrate a set of rewrite rules for transforming a given proof into a "maximally concurrent" proof of the same sequent. We begin by defining essential cuts and then state and prove the cut reduction theorem. The proof is based on giving a finite set of proof reduction rules which is shown to be strongly normalizing.

Definition 1 An instance of the cut rule in a proof is *trivial* if at least one of the premisses is an axiom of the form $A \vdash A$.

Definition 2 An instance of a cut rule in a proof is called *essential* if it is non-trivial and has the form

$$\frac{\Gamma \vdash A \qquad A \vdash B}{\Gamma \vdash B}\, Cut$$

where A is a netformula.

Theorem 4 (Cut-Reduction) *Given a net N and its associated deductive system $\mathcal{L}(N)$. If a sequent $\Gamma \vdash A$ is provable in $\mathcal{L}(N)$, then there is a proof of this sequent in $\mathcal{L}(N)$ such that all cuts are essential.*

Intuitively, essential cuts seem to capture dependencies exactly as dictated by the underlying net. A proof is *cut-reduced* if all instances of cuts in it are essential.

We will give a collection of rewrite rules for proofs and show the existence of a normalizing sequence. We will then strengthen this result by establishing that the set of reduction rules is strongly normalizing . The theorem above will immediately follow from the proposition that every normal proof is cut-reduced.

Remark: Prawitz [14] distinguishes "normal form theorem", "normalization theorem", and "strong normalization theorem". In his terminology then, our cut-reduction theorem is a normal form theorem, the second theorem will be a normalization theorem, and the last one will be a strong normalization theorem.

We begin by enumerating transformations on proofs. Assume that a proof $\mathcal{P}$ ends with an inessential cut, *i.e.* it has the following form:

$$\mathcal{P}:\quad \frac{\Gamma \vdash A \qquad A,\Delta \vdash B}{\Gamma,\Delta \vdash B}\,{\scriptstyle Cut:A}$$

We will refer to the left and right sub-proofs as $\mathcal{P}'$ and $\mathcal{P}''$, respectively. The various transformations, based on the form of $\mathcal{P}'$ and $\mathcal{P}''$, are:

1. *Axioms.* This case is applicable when at least one of the sub-proofs is an axiom.

1.1 $\mathcal{P}'$ is an axiom. We have the following transformation:

$$\frac{A \vdash A \qquad A,\Delta \vdash B}{A,\Delta \vdash B}\,{\scriptstyle Cut:A} \qquad \Rightarrow \qquad A,\Delta \vdash B$$

1.2 $\mathcal{P}''$ is an axiom. We transform $\mathcal{P}$ as follows:

$$\frac{\Gamma \vdash A \qquad A \vdash A}{\Gamma \vdash A}\,{\scriptstyle Cut:A} \qquad \Rightarrow \qquad \Gamma \vdash A$$

2. *Permutation.* This rule is applied when at least one of the sub-proofs $\mathcal{P}'$ and $\mathcal{P}''$ terminates with a logical rule with the main formula being different from the cut formula A or with an essential cut. Following are the various possibilities.

2.1 Endsequent of $\mathcal{P}'$ is obtained by an essential cut or a logical rule whose main formula is different from the cut formula. We distinguish the following cases.

2.1.1 The last rule is a $\otimes$L. We obtain the new proof as follows:

$$\frac{\dfrac{\Gamma,B,C \vdash A}{\Gamma,B\otimes C \vdash A}\,{\scriptstyle \otimes L} \qquad \Delta,A \vdash D}{\Gamma,B\otimes C,\Delta \vdash D}\,{\scriptstyle Cut:A} \qquad \Rightarrow \qquad \frac{\dfrac{\Gamma,B,C \vdash A \qquad \Delta,A \vdash D}{\Gamma,B,C,\Delta \vdash D}\,{\scriptstyle Cut:A}}{\Gamma,B\otimes C,\Delta \vdash D}\,{\scriptstyle \otimes L}$$

2.1.2 The last rule in $\mathcal{P}'$ is an essential cut. Note that since we allow at most one formula in the succedent of a sequent, the last rule of $\mathcal{P}'$ cannot be a $\otimes$R in 2.1.

$$\frac{\dfrac{\Gamma \vdash B \qquad B \vdash A}{\Gamma \vdash A}\,{\scriptstyle Cut:B} \qquad A \vdash C}{\Gamma \vdash C}\,{\scriptstyle Cut:A}$$

$$\Rightarrow \qquad \frac{\Gamma \vdash B \qquad \dfrac{B \vdash A \qquad A \vdash C}{B \vdash C}\,{\scriptstyle Cut:A}}{\Gamma \vdash C}\,{\scriptstyle Cut:B}$$

Note in above that B is a netformula.

2.2 Similar to 2.1 above but for the sub-proof $\mathcal{P}''$. We distinguish the following cases.

2.2.1 The last rule is a $\otimes$L.

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta,C,D,A\vdash B}{\Delta,C\otimes D,A\vdash A}\,{\scriptstyle\otimes L}}{\Gamma,\Delta,C\otimes D\vdash B}\,{\scriptstyle Cut:A} \quad\Rightarrow\quad \dfrac{\dfrac{\Gamma\vdash A \qquad \Delta,C,D,A\vdash B}{\Gamma,\Delta,C,D\vdash B}\,{\scriptstyle Cut:A}}{\Gamma,\Delta,C\otimes D\vdash B}\,{\scriptstyle\otimes L}$$

2.2.2 The last rule is a $\otimes$R. We distinguish following two cases.

2.2.2.1 Cut formula A in upper left sequent of the last rule of $\mathcal{P}''$.

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta',A\vdash B \qquad \Delta''\vdash C}{\Delta',\Delta'',A\vdash B\otimes C}\,{\scriptstyle\otimes R}}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,{\scriptstyle Cut:A} \quad\Rightarrow\quad \dfrac{\dfrac{\Gamma\vdash A \qquad \Delta',A\vdash B}{\Gamma,\Delta'\vdash B}\,{\scriptstyle Cut:A} \qquad \Delta''\vdash C}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,{\scriptstyle\otimes R}$$

2.2.2.2 Cut formula A in upper right sequent of the last rule of $\mathcal{P}''$.

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta'\vdash B \qquad \Delta'',A\vdash C}{\Delta',\Delta'',A\vdash B\otimes C}\,{\scriptstyle\otimes R}}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,{\scriptstyle Cut:A} \quad\Rightarrow\quad \dfrac{\Delta'\vdash B \qquad \dfrac{\Gamma\vdash A \qquad \Delta'',A\vdash C}{\Gamma,\Delta''\vdash C}\,{\scriptstyle Cut:A}}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,{\scriptstyle\otimes R}$$

2.2.3 The last rule of $\mathcal{P}''$ is an essential cut. In this case, the cut formula cannot come from the upper right sequent of the essential cut above. Thus we have only one case to consider.

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta',A\vdash B \qquad B\vdash C}{\Delta',A\vdash C}\,{\scriptstyle Cut:B}}{\Gamma,\Delta'\vdash C}\,{\scriptstyle Cut:A}$$

$$\Rightarrow\qquad \dfrac{\dfrac{\Gamma\vdash A \qquad \Delta',A\vdash B}{\Gamma,\Delta'\vdash B}\,{\scriptstyle Cut:A} \qquad B\vdash C}{\Gamma,\Delta'\vdash C}\,{\scriptstyle Cut:B}$$

Note once again that B belongs to some netaxiom in the two cases above.

3. *Logical.* This is the case where the cut formula is the main formula of a logical rule in both $\mathcal{P}'$ and $\mathcal{P}''$ and is introduced only by this instance of the rule. The transformation in this case depends on the outermost logical symbol of the cut formula and since we only have one logical connective, there is only one case to consider here.

$$\dfrac{\dfrac{\Gamma'\vdash A_1 \qquad \Gamma''\vdash A_2}{\Gamma',\Gamma''\vdash A_1\otimes A_2}\,{\scriptstyle\otimes R} \qquad \dfrac{A_1,A_2,\Delta\vdash B}{A_1\otimes A_2,\Delta\vdash B}\,{\scriptstyle\otimes L}}{\Gamma',\Gamma'',\Delta\vdash B}\,{\scriptstyle Cut:A_1\otimes A_2}$$

$$\Rightarrow\qquad \dfrac{\Gamma''\vdash A_2 \qquad \dfrac{\Gamma'\vdash A_1 \qquad A_1,A_2,\Delta\vdash B}{\Gamma',A_2,\Delta\vdash B}\,{\scriptstyle Cut:A_1}}{\Gamma'',\Gamma',\Delta\vdash B}\,{\scriptstyle Cut:A_2}$$

Remark: It may seem that the rule 2.1.2 does not appear in its most general form and one may be tempted to consider the following as its most general form:

$$\dfrac{\dfrac{\Gamma \vdash B \qquad B \vdash A}{\Gamma \vdash A}\,{\scriptstyle Cut:B} \qquad \Delta, A \vdash C}{\Gamma, \Delta \vdash C}\,{\scriptstyle Cut:A}$$

However, such a form is not only redundant but incorrect too. First, note that in the situation as above, the comma suggests that $\Delta, A \vdash C$ is obtained by a $\otimes$R or $\otimes$L, and hence 2.2.1 or 2.2.2 would be applicable. An attempt to give a reduction rule based on the form above by permuting the the two cuts will make the cut on B inessential (unless Δ is empty, in which case 2.1.2 applies), thus destroying an important invariance property of these transformations. Also, note that in the case 3 above, the transformation splits a cut into two cuts but with cut formulas with less number of logical symbols. The transformation as presented first performs a cut on A_1 and then on A_2. However, we could have done a cut on A_2 before A_1 giving us another transformation. But including one or the other or both does not affect our results.

The following lemma singles out an important property of the above transformations.

Lemma 5 *Let $\mathcal{P}$ be a proof and let $\mathcal{P}'$ be a proof obtained from $\mathcal{P}$ by the applications of the transformations above, then the number of sequents (nodes) in $\mathcal{P}'$ (viewed as a tree) is less than or equal to the number of nodes in $\mathcal{P}$,* i.e., *the number of nodes in a proof is never increased by the application of the transformations above.*

Proof: Immediate. ▌

Definition 3 A proof $\mathcal{P}$ is in *normal* form if there does not exist a proof $\mathcal{P}'$ such that $\mathcal{P} \Rightarrow \mathcal{P}'$ (one step reduction) by the transformations above.

Lemma 6 *A proof P is in normal form iff it is cut-reduced.* ▌

Proof: (if part) Clearly $\mathcal{P}$ is trivially normal if it does not contain any inessential cut.

(only if part) Assume on the contrary that $\mathcal{P}$ is normal and contain inessential cuts. In $\mathcal{P}$ choose an inessential cut above which there is no other inessential cut. Clearly then one of the reduction rules given above is applicable to this (sub) proof $\mathcal{P}'$ in $\mathcal{P}$ depending on how the premisses of the (only) inessential cut in $\mathcal{P}'$ are obtained. This contradicts the assumption that $\mathcal{P}$ is normal. Hence a normal proof is cut-reduced. ▌

The following lemma is our main lemma which shows the existence of a normalizing sequence of reduction.

Lemma 7 *If $\mathcal{P}$ is a proof of $\Gamma \vdash A$ which contains only one (inessential) cut occurring as the last inference, then $\Gamma \vdash A$ is provable with no inessential cut.*

The proof of the theorem then immediately follows from the above lemma by an easy induction on the number of inessential cuts appearing in a proof. In any proof consider an inessential cut above whose lower sequent no inessential cuts appear; thus satisfying the condition of the lemma. According to the lemma this (sub) proof can be transformed into another (equivalent) proof which does not contain this cut. In doing so, rest of the proof remains unchanged. We get a cut-reduced (equivalent) proof by repeating this process until all the inessential cuts have been eliminated.
Proof: (of the main lemma) Easy induction on the number of nodes in a proof satisfying the condition of the lemma. ▮

The following is now immediate.

Theorem 8 *Let $\mathcal{P}$ be a proof. Then there exists a sequence of reductions such that $\mathcal{P} \Rightarrow^* \mathcal{P}'$, and $\mathcal{P}'$ is in normal form.* ▮

The following definition will be used in the proof of our next theorem.

Definition 4 The *grade g* of a formula A is the number of $\otimes$ contained in A. The grade of an inessential cut is the grade of its cut formula.

Thus, by the definition above, grade of an essential cut is 0.

Theorem 9 (Strong Normalization) *There is no infinite reduction sequence beginning with any proof $\mathcal{P}'$.*

Proof: We define a measure on proofs and show that each one step transformation reduces this measure.

Let the *complexity* of a proof be a pair (a, b), where

- $a =$ sum of the grade g of cut formulas of all inessential cuts in the proof.
- $b =$ sum of the nodes above all inessential cuts (including the premisses and conclusion of the cut).

Clearly, a cut-reduced proof has complexity (0,0).

Now consider the three (main) classes of the transformations above. It is easy to see that application of these transformations in each case to a proof reduces its complexity.

Axiom: Both a and b are reduced.

Permutation: b is reduced keeping a the same.

Logical: a is reduced.

Thus, all reduction sequences terminate. ▮

In Appendix A we have written out how the rewriting works on Proof 1 and 2 in Figure 5.

5 Proofs as arrows.

A variety of publications have focused on the category-theoretic characteristics of Petri nets. In this section we hope to demonstrate that the proof transformations which we describe in the previous section are compatible with at least one elegant theory of nets as categories. To this end, we note how proofs can be interpreted as arrows in the category $\mathcal{T}[\mathcal{N}]$ of Degano, Meseguer, and Montanari [5] and then show that the proof reduction rules as presented above are sound with respect to this interpretation, *i.e.* they transform arrows to equal arrows.

Let N be a net and $\mathcal{L}(\mathcal{N})$ be the tensor theory determined by it. Also, let *Proofs*(N) denote the class of proofs in the theory $\mathcal{L}(\mathcal{N})$ and let *Mor*(N) denote the class of morphisms of the strictly symmetric strict monoidal category $\mathcal{T}[\mathcal{N}]$ freely generated by N. We define our interpretation $\mathbf{I} : \mathit{Proofs}(N) \longrightarrow \mathit{Mor}(N)$ as follows, where in writing

$$\begin{array}{c}\Pi\\ \Gamma \vdash A\end{array}$$

we mean a proof Π with conclusion $\Gamma \vdash A$.

1.
$$\mathbf{I}(A \vdash A) = i_A : A \to A$$

2.
$$\mathbf{I}(A_1 \otimes A_2 \cdots \otimes A_n \vdash B_1 \otimes B_2 \cdots \otimes B_m) = t : A_1 \otimes A_2 \cdots \otimes A_n \to B_1 \otimes B_2 \cdots \otimes B_m,$$

where t is in N.

3.
$$\mathbf{I}\left(\frac{\begin{array}{c}\Pi_1\\ \Gamma \vdash A\end{array} \qquad \begin{array}{c}\Pi_2\\ \Delta \vdash B\end{array}}{\Gamma, \Delta \vdash A \otimes B}\right) = f \otimes g : \Gamma \otimes \Delta \to A \otimes B,$$

where $f : \Gamma \to A = \mathbf{I}(\begin{array}{c}\Pi_1\\ \Gamma \vdash A\end{array})$ and $g : \Delta \to B = \mathbf{I}(\begin{array}{c}\Pi_2\\ \Delta \vdash B\end{array})$.

4.
$$\mathbf{I}\left(\frac{\begin{array}{c}\Pi\\ \Gamma, A, B \vdash C\end{array}}{\Gamma, A \otimes B \vdash C}\right) = \mathbf{I}(\begin{array}{c}\Pi\\ \Gamma, A, B \vdash C\end{array})$$

5.
$$\mathbf{I}\left(\frac{\begin{array}{c}\Pi_1\\ \Gamma \vdash A\end{array} \qquad \begin{array}{c}\Pi_2\\ A, \Delta \vdash B\end{array}}{\Gamma, \Delta \vdash B}\right) = (f \otimes i_\Delta) \circ g : \Gamma \otimes \Delta \to B,$$

$\Pi_1 \qquad\qquad\qquad\qquad \Pi_2$

where $f : \Gamma \to A = \mathbf{I}(\Gamma \vdash A)$ and $g : A \otimes \Delta \to B = \mathbf{I}(A, \Delta \vdash B)$.

Proposition 10 *The proof reduction rules are sound with respect to the interpretation above.*

Proof: We just consider an illustrative case here. Consider the reduction rule 2.2.2.1. The function $\mathbf{I}$ yields an arrow corresponding to the left hand side as follows. Let $f : \Gamma \to A$, $g : \Delta' \otimes A \to B$, and $h : \Delta'' \to C$. Then we have:

$$\begin{aligned}
&(f \otimes i_{\Delta' \otimes \Delta''}) \circ (g \otimes h) : \Gamma \otimes (\Delta' \otimes \Delta'') \to B \otimes C \\
&= (f \otimes (i_{\Delta'} \otimes i_{\Delta''})) \circ (g \otimes h) \\
&= ((f \otimes i_{\Delta'}) \otimes i_{\Delta''}) \circ (g \otimes h) \\
&= ((f \otimes i_{\Delta'}) \circ g) \otimes (i_{\Delta''} \circ h) \\
&= ((f \otimes i_{\Delta'}) \circ g) \otimes h \\
&= \mathbf{I}(\text{right hand side}) \ \blacksquare
\end{aligned}$$

In view of the above proposition and the strong normalization theorem, the following is immediate.

Corollary 11 *Every proof reduces to a unique normal form modulo the interpretation.* ▮

It has long been argued by proof theorists that a notion of equivalence of proofs based on mere provability is too extensional and inadequate. But the question of the right notion of equivalence of proofs still remains open. Prawitz [14], for the system of Natural Deduction and his set of reduction rules, conjectured that two derivations represent the same proof if and only if they reduce to the same normal form. Now in view of corollary 11 above we may say something similar for the identification of the derivations in a tensor theory. However, it seems that such an identification does not quite capture the intuitive sense of equivalence (based on processes) that we have in mind for net computations and is still too extensional. For example, proof 2 and proof 3 of section 3 would be identified as the corresponding arrows are equal because $\_\otimes\_$ is a bifunctor. However, the process interpretation that we have in mind should not result in such an identification. Thus the sense in which proof 3 is not equivalent to proof 2 (and in fact better) is lost in the denotational view that we have presented in this section. We are currently looking at how to attach such intensional interpretations to proofs in our setting. We have made some partial progress towards this, though mostly via some *ad hoc* means.

6 Choice Situations

We have so far restricted attention to a rather small fragment of linear logic because this fragment is already sufficient to illustrate several important concepts that suggest interesting relationships between concurrent computations and proofs. However, we believe that this is really only the beginning of the story. To give the reader a taste of how the theory can be further developed, we will give two simple examples that illustrate the potential role of the linear connectives known as "direct product" and "direct sum."

Given linear formulas, A and B, the expression $A \,\&\, B$ is a linear formula pronounced "A direct product B". The &R rule is

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$$

Intuitively, $A \,\&\, B$ can be obtained from the resources Γ provided these resources can be used to obtain A and can also be used to obtain B. Note carefully how this differs from the $\otimes$R rule where the resources must be divided in two parts—one part for proving A and the other for proving B. The &L rules are

$$\frac{\Gamma, A \vdash C}{\Gamma, A \,\&\, B \vdash C} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \,\&\, B \vdash C}$$

Intuitively, the resources C can be obtained from $A \,\&\, B$ provided they can be obtained *either* from A *or* from B.

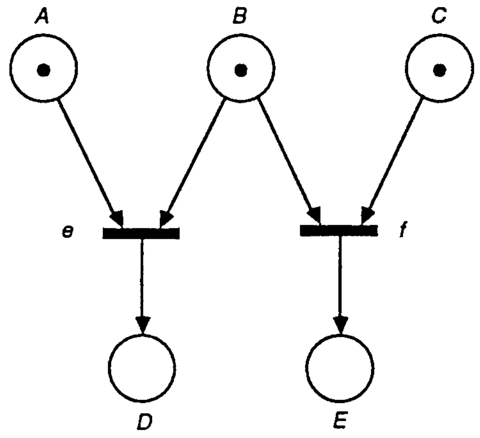


Figure 6: $A \otimes B \otimes C \vdash (D \otimes C) \,\&\, (A \otimes E)$

The intuitive explanations given above are meant to suggest to the reader the idea that the *direct product* operator represents a form of *choice.* To see a very simple example which we hope will be convincing enough to capture the reader's interest, consider the net pictured in Figure 6. Following the theory that we have developed in the preceding sections, this net is represented by the linear theory consisting of the sequents $A \otimes B \vdash D$ and $B \otimes C \vdash E$. Now, given a starting marking of one token on each of A, B and C, it is clear that a token can be moved to *at most one* of the conditions D and E. One might say that "$D \vee E$" is an obtainable marking, but "$D \wedge E$" is not. On the other hand, it seems that "$D \wedge E$" *is* obtainable in the sense that either of the conditions D and E *may* be fulfilled by some firing. In linear logic one may express this state of

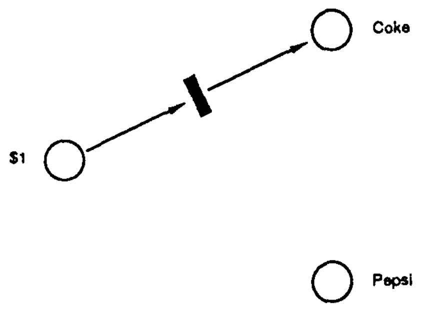


Figure 7: Coca Cola implements $\$1 \vdash \text{coke} \oplus \text{pepsi}$.

affairs with the formula $(D \otimes C) \& (A \otimes E)$. Here is a proof of the proper statment:

$$\dfrac{\dfrac{\dfrac{A \otimes B \vdash D \qquad C \vdash C}{A \otimes B, C \vdash D \otimes C}\otimes R}{A \otimes B \otimes C \vdash D \otimes C}\otimes L \qquad \dfrac{\dfrac{A \vdash A \qquad B \otimes C \vdash E}{A, B \otimes C \vdash A \otimes E}\otimes R}{A \otimes B \otimes C \vdash A \otimes E}\otimes L}{A \otimes B \otimes C \vdash (D \otimes C) \& (A \otimes E)}\& R$$

It is our feeling that the *direct product* operator captures a form of *external* choice. On the other hand, the *linear disjunction* captures a concept of *internal* choice. Given two linear propositions A and B, one proves the linear disjunction $A \oplus B$ of A and B from hypotheses Γ by using one of the following rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}$$

In other words, the resource $A \oplus B$ can be obtained from Γ just in case *either* A *or* B can be. On the other hand, if one wishes to obtain C from Γ and resource $A \oplus B$, then it must be shown that C can be obtained from *both* A *and* B. The rule is

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C}$$

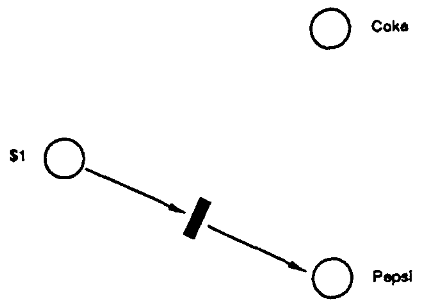


Figure 8: Pepsi Cola implements $\$1 \vdash \text{coke} \oplus \text{pepsi}$.

The internal/external distinction can be illustrated by a simple example which takes linear propositions as a specification language. Let us assume that we wish to contract a vendor to build

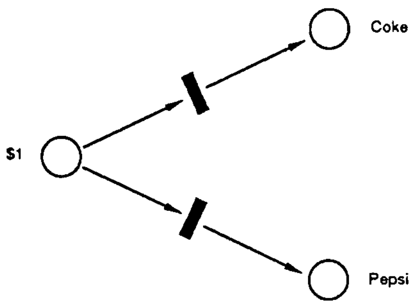


Figure 9: An implementation with choice: $1 ⊢ coke & pepsi.

us a machine for dispensing soft drinks for $1 and there are two possible flavors of drink that are available: coke and pepsi. If we do not actually care which of these is dispensed when a dollar is given to the machine, we may make the specification $1 ⊢ coke ⊕ pepsi. The choice of which beverage we are given in exchange for $1 will then be *internal* to the machine. For example, the machine which the Coca Cola company might design is pictured in Figure 7 and would dispense only coke. On the other hand, the Pepsi Cola company might design the machine in Figure 8 which dispenses only pepsi. In both cases, the tensor theories determined by the nets are strong enough that it is possible to prove the proposition $1 ⊢ coke ⊕ pepsi. Thus the machines have met the specification. If, on the other hand, we wish to insure that there is an *external* choice so that a user can pick the flavor of his preference, then we might have used the specification $1 ⊢ coke & pepsi. This specification is not met by either of the machines in Figures 7 and 8 since this sequent is not provable in either of the linear theories associated with these machines. On the other hand, the net in Figure 9 *does* meet the specification, since its associated theory is strong enough to prove the specifying sequent.

More research is needed to understand the possible significance of linear connectives in specification languages. While the example seems reasonable, it is worth keeping in mind that several other nets would meet the desired specification. For example, the net associated with the following theory

$$\begin{gathered}\$1 \vdash \text{coke}\\ \$1 \vdash \text{pepsi}\\ \text{pepsi} \vdash \$1\end{gathered}$$

which gives a $1 in exchange for a pepsi, also meets the specification! On the other hand, the net associated with the sequent $1 ⊢ coke ⊗ pepsi which dispenses both a coke *and* a pepsi for each $1 does *not* implement the specification.

There are several more linear connectives which we will not discuss. No account with which we are familiar has addressed the linear logic negation, which seems to represent the dual of a resource—a "debt" perhaps. A treatment of negation would lead to an understanding of the dual of the tensor called the *par* which seems to represent a concept of "concurrent debts". We mentioned earlier the unary operator ! which represents an unlimited resource. This operator plays a subtle role in the theory we have exposited; work of Carolyn Brown [4] provides helpful insight. All of the linear logic connectives seem to have their own significance in terms of computation on nets. (We have included a list of some of the rules of linear logic in Appendix B.) Work on the exploitation of these ideas is likely to be a profitable for the study of both concurrency and proof theory.

References

[1] A. Asperti. A logic for concurrency (extended abstract). Unpublished manuscript.

[2] A. Asperti, G. L. Ferrari, and R. Gorrieri. Implicative formulae in the "proofs as computations" analogy. To appear P. Hudak, editor, *Principles of Programming Languages*, pages ??–??, ACM, 1990.

[3] A. Avron. The semantics and proof theory of linear logic. *Theoretical Computer Science*, 57:161–184, 1988.

[4] C. Brown. Relating Petri nets to formulae of linear logic. Unpublished manuscript.

[5] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing net computations and processes. In R. Parikh, editor, *Logic in Computer Science*, pages 175–185, IEEE Computer Society, 1989.

[6] H. J. Genrich and E. Stankiewicz-Wiechno. A dictionary of some basic notions of net theory. In W. Brauer, editor, *Net Theory and Applications*, pages 519–535, *Lecture Notes in Computer Science vol. 84*, Springer-Verlag, 1980.

[7] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[8] J. Y. Girard. *Proofs and Types.* Cambridge University Press, 1989.

[9] J. Hopcroft and J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8:135–159, 1979.

[10] S. R. Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proc. 14th Annual ACM Symp. on Theory of Computing*, pages 267–281, 1982.

[11] N. Martí-Oliet and J. Meseguer. *From Petri nets to linear logic.* Research Report SRI-CSL-89-4, SRI International, Menlo Park, March 1989.

[12] E. W. Mayr. An algorithm for general Petri net reachability problem. In *Proc. 13th Annual ACM Symp. on Theory of Computing*, pages 238–246, 1981.

[13] H. Muller. On Kosaraju's proof of the decidability of the reachability problem for VAS. In L. Priese, editor, *Report on the 1st GTI-workshop*, Uni-GH Paderborn, *Reihe Theoretische Informatik Nr. 13*, 1983.

[14] D. Prawitz. Ideas and results in proof theory. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–307, Volume 63 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1971.

[15] W. Reisig. *Petri Nets.* Volume 4 of *EATCS Monographs on Theoretical Computer Science*, Spriger-Verlag, 1985.

[16] G. S. Sacerdote and R. L. Tenney. The decidability problem for VAS. In *Proc. 9th Annual ACM Symp. on Theory of Computing*, 1977.

[17] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen. Studies in Logic and the Foundations of Matematics*, North-Holland, 1969.

[18] G. Takeuti. *Proof Theory.* Volume 81 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, second edition, 1987.

[19] J. van Leeuwen. A partial solution to the reachability problem for vector addition systems. In *Proc. 6th Annual ACM Symp. on Theory of Computing*, pages 303–309, 1974.

A Sample Proof Transformation

We give some examples of cut-reduction below. At each step of the reduction, the inessential cut to which a reduction rule is applied is denoted $\boxed{Cut}$. Other choices of inessential cuts, if any, at a step to which a rule could have been applied are denoted $\overline{Cut}$. Remaining inessential cuts are denoted $\underline{Cut}$.

Example 1

$$\dfrac{\dfrac{\dfrac{A\vdash B \qquad A\vdash C}{A,A\vdash B\otimes C}{\scriptstyle\otimes\mathrm{R}} \qquad \dfrac{\dfrac{B\vdash D \qquad C\vdash E}{B,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{B\otimes C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes E}\boxed{\scriptstyle Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

$$\overset{3}{\Longrightarrow}\quad \dfrac{\dfrac{A\vdash C \qquad \dfrac{A\vdash B \qquad \dfrac{B\vdash D \qquad C\vdash E}{B,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{A,C\vdash D\otimes E}\boxed{\scriptstyle Cut}}{A,A\vdash D\otimes E}{\scriptstyle\underline{Cut}} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

$$\overset{2.2.2.1}{\Longrightarrow}\quad \dfrac{\dfrac{A\vdash C \qquad \dfrac{\dfrac{A\vdash B \qquad B\vdash D}{A\vdash D}{\scriptstyle Cut} \qquad C\vdash E}{A,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{A,A\vdash D\otimes E}\boxed{\scriptstyle Cut} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

$$\overset{2.2.2.2}{\Longrightarrow}\quad \dfrac{\dfrac{\dfrac{A\vdash B \qquad B\vdash D}{A\vdash D}{\scriptstyle Cut} \qquad \dfrac{A\vdash C \qquad C\vdash E}{A\vdash E}{\scriptstyle Cut}}{A,A\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

Example 2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A\vdash A \qquad A\vdash B}{A,A\vdash A\otimes B}{\scriptstyle\otimes\mathrm{R}} \qquad \dfrac{\dfrac{A\vdash C \qquad B\vdash B}{A,B\vdash C\otimes B}{\scriptstyle\otimes\mathrm{R}}}{A\otimes B\vdash C\otimes B}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash C\otimes B}\boxed{\scriptstyle Cut} \qquad \dfrac{\dfrac{B\vdash D \qquad C\vdash C}{B,C\vdash D\otimes C}{\scriptstyle\otimes\mathrm{R}}}{C\otimes B\vdash D\otimes C}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes C}{\scriptstyle\underline{Cut}} \qquad \dfrac{\dfrac{D\vdash D \qquad C\vdash E}{D,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{D\otimes C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes E}{\scriptstyle\underline{Cut}} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

$$\overset{3}{\Longrightarrow}\quad \dfrac{\dfrac{\dfrac{\dfrac{A\vdash B \qquad \dfrac{A\vdash A \qquad \dfrac{A\vdash C \qquad B\vdash B}{A,B\vdash C\otimes B}{\scriptstyle\otimes\mathrm{R}}}{A,B\vdash C\otimes B}\boxed{\scriptstyle Cut}}{A,A\vdash C\otimes B}{\scriptstyle\underline{Cut}} \qquad \dfrac{\dfrac{B\vdash D \qquad C\vdash C}{B,C\vdash D\otimes C}{\scriptstyle\otimes\mathrm{R}}}{C\otimes B\vdash D\otimes C}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes C}{\scriptstyle\underline{Cut}} \qquad \dfrac{\dfrac{D\vdash D \qquad C\vdash E}{D,C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{R}}}{D\otimes C\vdash D\otimes E}{\scriptstyle\otimes\mathrm{L}}}{A,A\vdash D\otimes E}{\scriptstyle\underline{Cut}} \qquad D\otimes E\vdash F}{A,A\vdash F}{\scriptstyle Cut}$$

$$\xRightarrow{1.1} \dfrac{\dfrac{\dfrac{A \vdash B \qquad \dfrac{A \vdash C \qquad B \vdash B}{A, B \vdash C \otimes B}\otimes R}{A, A \vdash C \otimes B}\boxed{Cut} \qquad \dfrac{\dfrac{B \vdash D \qquad C \vdash C}{B, C \vdash D \otimes C}\otimes R}{C \otimes B \vdash D \otimes C}\otimes L}{A, A \vdash D \otimes C}Cut \qquad \dfrac{\dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\otimes R}{D \otimes C \vdash D \otimes E}\otimes L}{\dfrac{A, A \vdash D \otimes E \qquad D \otimes E \vdash F}{A, A \vdash F}Cut}Cut$$

$$\xRightarrow{2.2.2.2} \dfrac{\dfrac{\dfrac{A \vdash C \qquad \dfrac{A \vdash B \qquad B \vdash B}{A \vdash B}\boxed{Cut}}{A, A \vdash C \otimes B}\otimes R \qquad \dfrac{\dfrac{B \vdash D \qquad C \vdash C}{B, C \vdash D \otimes C}\otimes R}{C \otimes B \vdash D \otimes C}\otimes L}{A, A \vdash D \otimes C}Cut \qquad \dfrac{\dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\otimes R}{D \otimes C \vdash D \otimes E}\otimes L}{\dfrac{A, A \vdash D \otimes E \qquad D \otimes E \vdash F}{A, A \vdash F}Cut}Cut$$

$$\xRightarrow{1.2} \dfrac{\dfrac{\dfrac{A \vdash C \qquad A \vdash B}{A, A \vdash C \otimes B}\otimes R \qquad \dfrac{\dfrac{B \vdash D \qquad C \vdash C}{B, C \vdash D \otimes C}\otimes R}{C \otimes B \vdash D \otimes C}\otimes L}{A, A \vdash D \otimes C}\boxed{Cut} \qquad \dfrac{\dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\otimes R}{D \otimes C \vdash D \otimes E}\otimes L}{\dfrac{A, A \vdash D \otimes E \qquad D \otimes E \vdash F}{A, A \vdash F}Cut}Cut$$

$$\xRightarrow{3} \dfrac{\dfrac{A \vdash B \qquad \dfrac{A \vdash C \qquad \dfrac{B \vdash D \qquad C \vdash C}{B, C \vdash D \otimes C}\otimes R}{A, B \vdash D \otimes C}\boxed{Cut}}{A, A \vdash D \otimes C}Cut \qquad \dfrac{\dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\otimes R}{D \otimes C \vdash D \otimes E}\otimes L}{\dfrac{A, A \vdash D \otimes E \qquad D \otimes E \vdash F}{A, A \vdash F}Cut}Cut$$

$$\xRightarrow{2.2.2.1} \dfrac{\dfrac{A \vdash B \qquad \dfrac{\dfrac{A \vdash C \qquad C \vdash C}{A \vdash C}\boxed{Cut} \qquad B \vdash D}{A, B \vdash D \otimes C}\otimes R}{A, A \vdash D \otimes C}Cut \qquad \dfrac{\dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\otimes R}{D \otimes C \vdash D \otimes E}\otimes L}{\dfrac{A, A \vdash D \otimes E \qquad D \otimes E \vdash F}{A, A \vdash F}Cut}Cut$$

$$\xRightarrow{1.2} \dfrac{\dfrac{A \vdash B \qquad \dfrac{A \vdash C \qquad B \vdash D}{A, B \vdash D \otimes C}\otimes R}{A, A \vdash D \otimes C}\boxed{Cut} \qquad \dfrac{\dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\otimes R}{D \otimes C \vdash D \otimes E}\otimes L}{\dfrac{A, A \vdash D \otimes E \qquad D \otimes E \vdash F}{A, A \vdash F}Cut}Cut$$

$$\xRightarrow{2.2.2.2} \dfrac{\dfrac{A \vdash C \qquad \dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}Cut}{A, A \vdash D \otimes C}\otimes R \qquad \dfrac{\dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\otimes R}{D \otimes C \vdash D \otimes E}\otimes L}{\dfrac{A, A \vdash D \otimes E \qquad D \otimes E \vdash F}{A, A \vdash F}Cut}\boxed{Cut}$$

$$\xRightarrow{3} \quad \dfrac{\dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}\,\mathit{Cut} \qquad \dfrac{A \vdash C \qquad \dfrac{D \vdash D \qquad C \vdash E}{D, C \vdash D \otimes E}\,\otimes\mathrm{R}}{A, D \vdash D \otimes E}\,\boxed{\mathit{Cut}}}{\dfrac{A, A \vdash D \otimes E}{}\qquad}\,\underline{\mathit{Cut}}$$

$$\dfrac{A, A \vdash D \otimes E \qquad D \otimes E \vdash F}{A, A \vdash F}\,\mathit{Cut}$$

$$\xRightarrow{2.2.2.2} \quad \dfrac{\dfrac{\dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}\,\mathit{Cut} \qquad \dfrac{D \vdash D \qquad \dfrac{A \vdash C \qquad C \vdash E}{A \vdash E}\,\mathit{Cut}}{A, D \vdash D \otimes E}\,\otimes\mathrm{R}}{A, A \vdash D \otimes E}\,\boxed{\mathit{Cut}} \qquad D \otimes E \vdash F}{A, A \vdash F}\,\mathit{Cut}$$

$$\xRightarrow{2.2.2.2} \quad \dfrac{\dfrac{\dfrac{\dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}\,\mathit{Cut} \qquad D \vdash D}{A \vdash D}\,\boxed{\mathit{Cut}} \qquad \dfrac{A \vdash C \qquad C \vdash E}{A \vdash E}\,\mathit{Cut}}{A, A \vdash D \otimes E}\,\otimes\mathrm{R} \qquad D \otimes E \vdash F}{A, A \vdash F}\,\mathit{Cut}$$

$$\xRightarrow{1.2} \quad \dfrac{\dfrac{\dfrac{A \vdash C \qquad C \vdash E}{A \vdash E}\,\mathit{Cut} \qquad \dfrac{A \vdash B \qquad B \vdash D}{A \vdash D}\,\mathit{Cut}}{A, A \vdash D \otimes E}\,\otimes\mathrm{R} \qquad D \otimes E \vdash F}{A, A \vdash F}\,\mathit{Cut}$$

B Rules of Linear Logic

Structural Rules

$$\frac{\Gamma, A, B, \Delta \vdash \Theta}{\Gamma, B, A, \Delta \vdash \Theta}\,\text{Exchange} \qquad \frac{\Gamma \vdash A, \Theta \qquad \Delta, A \vdash \Lambda}{\Gamma, \Delta \vdash \Theta, \Lambda}\,\mathit{Cut}$$

Logical Rules

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A^{\perp}, \Delta}\,\perp\text{R} \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma, A^{\perp} \vdash \Delta}\,\perp\text{L}$$

$$\frac{\Gamma \vdash A, \Theta \qquad \Delta \vdash B, \Lambda}{\Gamma, \Delta \vdash A \otimes B, \Theta, \Lambda}\,\otimes\text{R} \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta}\,\otimes\text{L}$$

$$\frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \parr B}\,\parr\text{R} \qquad \frac{\Gamma, A \vdash \Theta \qquad \Delta, B \vdash \Lambda}{\Gamma, \Delta, A \parr B \vdash \Theta, \Lambda}\,\parr\text{L}$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \multimap B, \Delta}\,\multimap\text{R} \qquad \frac{\Gamma \vdash A, \Theta \qquad \Delta, B \vdash C, \Lambda}{\Gamma, \Delta, A \multimap B \vdash C, \Theta, \Lambda}\,\multimap\text{L}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \,\&\, B, \Delta}\,\&\text{R} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A \,\&\, B \vdash \Delta}\,\&\text{L} \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \,\&\, B \vdash \Delta}\,\&\text{L}$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta}\,\oplus\text{R} \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta}\,\oplus\text{R} \qquad \frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta}\,\oplus\text{L}$$

NORMAL PROCESS REPRESENTATIVES

Vijay Gehlot *Carl Gunter* *

University of Pennsylvania
Department of Computer and Information Sciences
Philadelphia, PA 19104 U.S.A.

October 1989

Abstract

This paper discusses the relevance of a form of cut elimination theorem for linear logic tensor theories to the concept of a process on a Petri net. We base our discussion on two definitions of processes given by Best and Devillers. Their notions of process correspond to equivalence relations on linear logic proofs. It is noted that the cut reduced proofs form a process under the finer of these definitions. Using a strongly normalizing rewrite system and a weak Church-Rosser theorem, we show that each class of the coarser process definition contains exactly one of these finer classes which can therefore be viewed as a canonical or *normal* process representative. We also discuss the relevance of our rewrite rules to the categorical approach of Degano, Meseguer, and Montanari.

1 Introduction

It has often been useful to take ideas from proof theory and look at their computational significance. One very fruitful line of investigation has been the use of the Curry-Howard correspondence—the "propositions as types" idea—as a way of seeing proofs as programs and types as specifications. This correspondence reveals an analogy between cut elimination in systems of natural deduction and the reduction of lambda-terms, thus strongly connecting the study of a central proof-theoretic idea (with a history dating back at least to the 1930's) with a central computational concept in sequential functional programmming.

Another, more recent, line of investigation with a kinship to this sequential theory concerns the relationship between certain kinds of proofs and concepts in *concurrency.* A number of authors have discussed the idea of relating concurrent computations as represented by *Petri nets* to proofs in *linear logic* [7]. One line of research seeks to use the fact that nets give rise to a monoid structure and can therefore be used to model linear logic through the use of a phase semantics [6]. In this way a net can be viewed as a model of the linear connectives in which there is a correspondence between the *truth* of a linear sequent in the model and the *reachability* relation on the net. However, most of the research [8, 9, 1, 4] has focused on the idea that a net may be viewed as a *theory* in a fragment of linear logic (the tensor theory to be precise). In particular, when things are viewed in this way, there is a precise correspondence between *concurrent computations* on a Petri net and linear logic

*Research of both authors is supported by the Office of Naval Research. Electronic mail addresses for the authors are gunter@cis.upenn.edu and gehlot@cis.upenn.edu

proofs in its associated theory. This opens a way to investigate a transfer of ideas between proof theory and the theory of concurrent computation.

The purpose of this paper is to look at the significance of the cut elimination theorem of linear logic tensor theories in the context of computations on Petri nets. The role of cut in this theory is somewhat different from cut in the context of "propositions as types". A linear proof corresponds to a net computation and the elimination of a cut corresponds to a *transformation* of that computation, rather than an *enactment* of the computation. The authors of this paper have suggested before [8] that cut elimination corresponds a form of optimization in which a computation is transformed into a "more concurrent" computation. It is our goal in this paper to show how this view of the significance of the cut fits into a theory of Petri net processes.

In particular, we show that linear logic cut elimination provides a way of understanding the relationship between two definitions of the notion of a net process studied in Best and Devillers [3]. We will define a pair of relations $\mathcal{S}$ and $\mathcal{T}$ on linear logic proofs which corresponds to two of the concepts of net process discussed in [3]. The relation $\mathcal{T}$ is coarser than $\mathcal{S}$ and relates some computations which display different levels of dependency in their descriptions (i.e. one description permits two things to be done at the same time while the other description sequentializes them). What we wish to show is that there is a *unique* $\mathcal{S}$-equivalence class of processes in each $\mathcal{T}$-equivalence class τ which can be viewed as a "maximally concurrent" *representative* of τ. Moreover, the members of this $\mathcal{S}$-equivalence class will be exactly the set of cut reduced proofs in τ. Since we can demonstrate a strongly normalizing rewrite system for linear proofs which preserves $\mathcal{T}$-equivalence, we can therefore view the distinguished $\mathcal{S}$-equivalence class in τ as a *normal representative process* for it. In general, normal representatives of classes of $\mathcal{T}$ may be viewed as the "maximally concurrent" computations on a net.

Our first primary technical result is a strong normalization theorem (Theorem 2) for a set of rewrites which preserve $\mathcal{T}$-equivalence. Termination for the rewrites can be proved using a measure on cut formulas and the number of nodes in a proof tree. Our second primary technical result is a weak Church-Rosser theorem (Theorem 3) for the action of the rewrite system on equivalence classes of $\mathcal{S}$. Confluence of our rewrite system therefore follows from Newman's Lemma. This result is then used to show that there is a unique normal process representative in each equivalence class of $\mathcal{T}$(Theorem 4).

We begin by discussing some relevant work and definitions in section 2 where we also define the two equivalence relations. In section 3 we give a set of rewrite rules on proofs which are shown to be strongly normalizing. Section 4 contains our second main result where we prove the Church-Rosser property for the induced rewrite rules on $\mathcal{S}$-equivalent classes. We then define an equivalence based on this notion of reduction and show that this equivalence coincides with $\mathcal{T}$-equivalence giving us the desired theorem about unique process representatives. Finally, in section 5 we discuss relationship between reduction on proofs and rewrite rules for arrows which arise by interpreting proofs as arrows.

2 Two theories of processes.

In this section we introduce the background on processes and linear logic needed to understand the central results of the paper. First, let's start with an example. Consider the net pictured in Figure 1. It has six *places,* drawn as circles and marked A, B, C, A', B', C' and it has five *transitions,* written as rectangles and marked r, s, r', s', t. The two closed circles on the places A and A' are tokens which indicate the availability of the "resources" A and A'. In the configuration in the picture, the transitions r and r' are *enabled* by the fulfillment of their preconditions A and A'

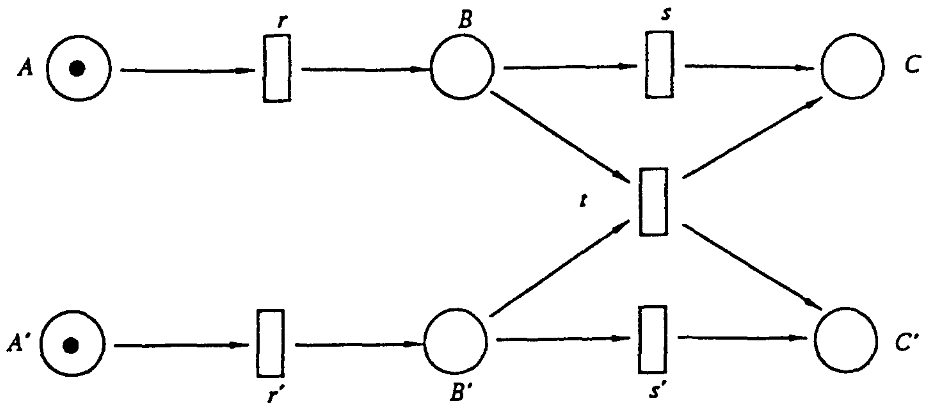


Figure 1: Net N with two processes of type $A \otimes A' \to C \otimes C'$.

respectively. Dynamically, computations proceed by the *firing* of transitions. If transition r fires, for example, then the token is removed from A and placed on its *postcondition* B; the transition r is now disabled since its precondition A is no longer filled. We may also speak of the *concurrent* firing of r and r' in the starting configuration of Figure 1 since there is no dependency between their pre-conditions.

For formal definitions we refer the reader to recent publications in LICS [10, 5]. For this paper we will take it as a working definition that a net (or, to be more precise, a place/transition net) is a set of pairs of multisets over a set S of places. This is really a special case of the definition of a net in [10, 5] where distinct transitions with the same pre and post conditions are permitted, but the restriction simplifies our notation since it avoids the need to label the linear sequents to preserve a precise correspondence between nets and linear tensor theories. For this preliminary discussion, it will be convenient to utilize their categorical treatment of nets and write transitions as arrows in a category with a binary operator $\otimes$ on its objects. In this notation, the transitions in the figure may be viewed as arrows:

$$\begin{array}{ccc} r : A \to B & & s : B \to C \\ & t : B \otimes B' \to C \otimes C' & \\ r' : A \to B & & s' : B \to C \end{array}$$

There are two operations on arrows. If $f : X \to Y$ and $g : Y \to Z$, then $f;g : X \to Z$ is the composition of f and g. If $f : X \to X'$ and $g : Y \to Y'$, then $f \otimes g : X \otimes Y \to Y \otimes Y'$ is the *tensor product* of f and g. Starting with the basic transitions, these operations generate a language of computations on the net. Intuitively we read $f;g$ as the *sequentialization* of f and g: "first do f and then do g". We read $f \otimes g$ as the *concurrent* performance of operations f and g: "do f and g at the same time".

Looking again at Figure 1, here are four sample computations of type $A \otimes A' \to C \otimes C'$ on the net N:

$$\begin{array}{rcl} f & = & (r \otimes A'); (s \otimes A'); (C \otimes r'); (C \otimes s') \\ f' & = & (r \otimes r'); (s \otimes s') \\ g & = & (r \otimes A'); (B \otimes r'); t \\ g' & = & (r \otimes r'); t \end{array}$$

where the idle transition (identity map) on a place X is written simply as X. Much of the research on nets (and, indeed, concurrency as a whole) has focused on the question of when two computations such as the ones above are "essentially the same". In the case of the computations above, one may well expect to distinguish between processes f and g, for example, since one of these computations

Structural Rules

$$\frac{}{A \vdash A}\,(\text{Identity}) \qquad \frac{\Gamma \vdash A \qquad \Delta, A \vdash B}{\Gamma, \Delta \vdash B}\,(\text{Cut})$$

Logical Rules

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,(\otimes\text{R}) \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\,(\otimes\text{L})$$

Figure 2: Structural and logical rules for the tensor fragment of linear logic.

performs transitions s and s' but not t, while the other performs t but neither s nor s'. On the other hand, it is debatable whether f and f' or g and g' should be identified. What this comes down to is a question of whether $\otimes$ is a *functor* or not, *i.e.* is it the case that the equation

$$(u \otimes v); (u' \otimes v') = (u; u') \otimes (v; v')$$

holds for arbitrary computations u, u', v, v' which make the equation type correct? Arguing one way, this is a pleasing equational property which is supported by a computational intuition that the only real difference between the left and right hand sides of this equality is the order in which things are done. Arguing against functoriality, it seems that computations f' and g' are somehow "better" than f and g, respectively, since they allow more independent computations to be performed concurrently. For example, f does not perform the transition r' until after r is complete, but this is unnecessary since r and r' can be done at the same time as in f'.

The dividing line between a theory of processes which identifies f with f' and g with g' and a theory which does not make these identifications has been carefully examined in [3] and [5]. We will not introduce the theory in the form that they have done, but instead present it as an equational theory of *proofs*. The reader can check that our presentations of the relations $\mathcal{S}$ and $\mathcal{T}$ for proofs as given below correspond to the equational theories with these names presented by Degano, Meseguer, and Montanari [5] in the last **LICS** symposium.

A linear *tensor formula* is either a propositional atom or has the form $A \otimes B$ where A and B are tensor formulas. A linear logic tensor *sequent* is a pair $\Gamma \vdash A$ where A is a tensor formula and Γ is a multiset. The rules for deriving sequents of this fragment of linear logic are given in Figure 2. A tensor theory is a set of pairs $A \vdash B$ where A and B are tensor formulas. Given a net N with places S, the *associated* tensor theory is the set of all sequents $A \vdash B$ where A and B are tensor formulas formed over atoms from S such that the pair of multisets $(M(A), M(B))$ determined by A and B is an element of the net N. There is a precise correspondence between computations on N and proofs over the associated theory in which uses of axioms in a proof correspond to firings of transitions on the net and uses of the rules for tensor on the right and cut correspond to the tensor and composition of computations respectively. We omit a further discussion of this correspondence here since it has been exposited adequately elsewhere ([9] provides a rigorous treatment for example). The remainder of this paper will be concerned with computations as represented by proofs.

We now begin a more formal discussion of proofs and the cut rule. In the presence of proper axioms it is not possible to *eliminate* cuts. Our concept of *cut reduction* will be based on a definition of a normal form for proofs and a system of rewrite rules for proofs not in normal form. We say that an instance of the cut rule in a proof is *trivial* if at least one of the premisses is an axiom of the form $A \vdash A$. A linear formula A is said to be a *netformula* (of T) if it appears in one of the formulas of the theory T. An instance of a cut rule is said to be *essential* (in a proof in the theory T) if it is non-trivial and has the form

$$\dfrac{\Gamma\vdash A \qquad A\vdash B}{\Gamma\vdash B}\,{\scriptstyle Cut}$$

where A is a netformula. A proof is said to be in *normal* form if all of its cuts are essential. We will discuss normalization of proofs in the next section.

Definition 1 Let N be a net. The equivalence relation $\mathcal{S}(N)$ on proofs is defined as the smallest equivalence relation satisfying the following equations between proof trees.

$$(1)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C\vdash D}\qquad \overset{\Pi'}{\Delta\vdash E}}{\Gamma,B,C,\Delta\vdash D\otimes E}\,{\scriptstyle\otimes\mathrm{R}}}{\Gamma,B\otimes C,\Delta\vdash D\otimes E}\,{\scriptstyle\otimes\mathrm{L}} \quad=\quad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C\vdash D}}{\Gamma,B\otimes C\vdash D}\,{\scriptstyle\otimes\mathrm{L}}\qquad \overset{\Pi'}{\Delta\vdash E}}{\Gamma,B\otimes C,\Delta\vdash D\otimes E}\,{\scriptstyle\otimes\mathrm{R}}$$

$$(2)\qquad \dfrac{\overset{\Pi}{\Gamma\vdash A}\qquad \overset{\Pi'}{\Delta\vdash B}}{\Gamma,\Delta\vdash A\otimes B}\,{\scriptstyle\otimes\mathrm{R}} \quad=\quad \dfrac{\overset{\Pi'}{\Delta\vdash B}\qquad \overset{\Pi}{\Gamma\vdash A}}{\Delta,\Gamma\vdash B\otimes A}\,{\scriptstyle\otimes\mathrm{R}}$$

$$(3)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C,\Delta,D,E\vdash F}}{\Gamma,B\otimes C,\Delta,D,E\vdash F}\,{\scriptstyle\otimes\mathrm{L}}}{\Gamma,B\otimes C,\Delta,D\otimes E\vdash F}\,{\scriptstyle\otimes\mathrm{L}} \quad=\quad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C,\Delta,D,E\vdash F}}{\Gamma,B,C,\Delta,D\otimes E\vdash F}\,{\scriptstyle\otimes\mathrm{L}}}{\Gamma,B\otimes C,\Delta,D\otimes E\vdash F}\,{\scriptstyle\otimes\mathrm{L}}$$

$$(4)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma\vdash A}\qquad \overset{\Pi'}{\Delta\vdash C}}{\Gamma,\Delta\vdash A\otimes B}\,{\scriptstyle\otimes\mathrm{R}}\qquad \overset{\Pi''}{\Lambda\vdash C}}{\Gamma,\Delta,\Lambda\vdash A\otimes B\otimes C}\,{\scriptstyle\otimes\mathrm{R}} \quad=\quad \dfrac{\overset{\Pi}{\Gamma\vdash A}\qquad \dfrac{\overset{\Pi'}{\Delta\vdash C}\qquad \overset{\Pi''}{\Lambda\vdash C}}{\Delta,\Lambda\vdash B\otimes C}\,{\scriptstyle\otimes\mathrm{R}}}{\Gamma,\Delta,\Lambda\vdash A\otimes B\otimes C}\,{\scriptstyle\otimes\mathrm{R}}$$

$$(5)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma\vdash A}\qquad \overset{\Pi'}{\Delta,A\vdash B}}{\Gamma,\Delta\vdash B}\,{\scriptstyle Cut}\qquad \overset{\Pi''}{\Lambda,B\vdash C}}{\Gamma,\Delta,\Lambda\vdash C}\,{\scriptstyle Cut} \quad=\quad \dfrac{\overset{\Pi}{\Gamma\vdash A}\qquad \dfrac{\overset{\Pi'}{\Delta,A\vdash B}\qquad \overset{\Pi''}{\Lambda,B\vdash C}}{\Delta,A,\Lambda\vdash C}\,{\scriptstyle Cut}}{\Gamma,\Delta,\Lambda\vdash C}\,{\scriptstyle Cut}$$

$$(6)\qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C\vdash D}\qquad \overset{\Pi'}{D\vdash E}}{\Gamma,B,C\vdash E}\,\text{e-cut}}{\Gamma,B\otimes C\vdash E}\,{\scriptstyle\otimes\mathrm{L}} \quad=\quad \dfrac{\dfrac{\overset{\Pi}{\Gamma,B,C\vdash D}}{\Gamma,B\otimes C\vdash D}\,{\scriptstyle\otimes\mathrm{L}}\qquad \overset{\Pi'}{D\vdash E}}{\Gamma,B\otimes C\vdash E}\,\text{e-cut}$$

In equation (6) above, 'e-cut' means that the corresponding cut is an essential one. The equations above may be read as defining associativity and commutativity of proofs in presence of proper axioms. We now define another equivalence relation $\mathcal{T}(N)$ on proofs to be the least equivalence relation generated by the equations defining $\mathcal{S}$ plus the following equation.

$$(7)\quad \dfrac{\dfrac{\begin{array}{c}\Pi\\ \Gamma\vdash A\end{array}\quad \begin{array}{c}\Pi'\\ \Delta\vdash B\end{array}}{\Gamma,\Delta\vdash A\otimes B}\otimes\mathrm{R} \quad \dfrac{\dfrac{\dfrac{\begin{array}{c}\Pi''\\ A\vdash C\end{array}\quad \begin{array}{c}\Pi'''\\ B\vdash D\end{array}}{A,B\vdash C\otimes D}\otimes\mathrm{R}}{A\otimes B\vdash C\otimes D}\otimes\mathrm{L}}{}}{\Gamma,\Delta\vdash C\otimes D}\mathit{Cut} \;=\; \dfrac{\dfrac{\begin{array}{c}\Pi\\ \Gamma\vdash A\end{array}\quad \begin{array}{c}\Pi''\\ A\vdash C\end{array}}{\Gamma\vdash C}\mathit{Cut}\quad \dfrac{\begin{array}{c}\Pi'\\ \Delta\vdash B\end{array}\quad \begin{array}{c}\Pi'''\\ B\vdash D\end{array}}{\Gamma\vdash D}\mathit{Cut}}{\Gamma,\Delta\vdash C\otimes D}\otimes\mathrm{R}$$

This last rule corresponds to the functoriality of the tensor operation. Via translation, the relations $\mathcal{S}(N)$ and $\mathcal{T}(N)$ as we have just defined them correspond to the theories $\mathcal{S}[N]$ and $\mathcal{T}[N]$ as given in [5]. Their results there demonstrate the correspondence between these equivalence classes of proofs and the processes in [3]. We will therefore refer to equivalence classes of proofs in $\mathcal{S}(N)$ and $\mathcal{T}(N)$ as $\mathcal{S}$-processes and $\mathcal{T}$-processes respectively (dropping the N when it is understood as fixed).

3 Strongly normalizing cut reduction.

In this section we give a set of reduction rules for proofs and show that they are strongly normalizing. This will provide the desired algorithm for finding the normal representative of a $\mathcal{T}$-process.

Assume that a proof $\mathcal{P}$ ends with an inessential cut and has the following form:

$$\mathcal{P}\colon\quad \dfrac{\Gamma\vdash A \qquad A,\Delta\vdash B}{\Gamma,\Delta\vdash B}\,\mathit{Cut{:}A}$$

We will refer to the left and right sub-proofs as $\mathcal{P}'$ and $\mathcal{P}''$, respectively. We will divide these reduction rules in three classes—axiom, permutation, and logical—and give an illustrative transformation in each class.

1. *Axioms.* This case is applicable when at least one of the sub-proofs is an axiom. When $\mathcal{P}'$ is an axiom, we have the following transformation:

$$\dfrac{A\vdash A \qquad A,\Delta\vdash B}{A,\Delta\vdash B}\,\mathit{Cut{:}A} \qquad\Rightarrow\qquad A,\Delta\vdash B$$

2. *Permutation.* This rule is applied when at least one of the sub-proofs $\mathcal{P}'$ and $\mathcal{P}''$ terminates with a logical rule with the main formula being different from the cut formula A or with an essential cut. For the case when the last rule of $\mathcal{P}''$ is a $\otimes$R and cut formula A is in upper left sequent of the last rule of $\mathcal{P}''$, we have the following rewrite:

$$\dfrac{\Gamma\vdash A \qquad \dfrac{\Delta',A\vdash B \qquad \Delta''\vdash C}{\Delta',\Delta'',A\vdash B\otimes C}\otimes\mathrm{R}}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\,\mathit{Cut{:}A} \quad\Rightarrow\quad \dfrac{\dfrac{\Gamma\vdash A \qquad \Delta',A\vdash B}{\Gamma,\Delta'\vdash B}\,\mathit{Cut{:}A} \qquad \Delta''\vdash C}{\Gamma,\Delta',\Delta''\vdash B\otimes C}\otimes\mathrm{R}$$

3. *Logical.* This is the case where the cut formula is the main formula of a logical rule in both $\mathcal{P}'$ and $\mathcal{P}''$ and is introduced only by this instance of the rule. The transformation in this case

depends on the outermost logical symbol of the cut formula and since we only have one logical connective, there is only one case to consider here.

$$\dfrac{\dfrac{\Gamma' \vdash A_1 \qquad \Gamma'' \vdash A_2}{\Gamma', \Gamma'' \vdash A_1 \otimes A_2}\otimes R \qquad \dfrac{A_1, A_2, \Delta \vdash B}{A_1 \otimes A_2, \Delta \vdash B}\otimes L}{\Gamma', \Gamma'', \Delta \vdash B}\,{\scriptstyle Cut: A_1 \otimes A_2}$$

$$\Rightarrow \qquad \dfrac{\Gamma'' \vdash A_2 \qquad \dfrac{\Gamma' \vdash A_1 \qquad A_1, A_2, \Delta \vdash B}{\Gamma', A_2, \Delta \vdash B}\,{\scriptstyle Cut: A_1}}{\Gamma'', \Gamma', \Delta \vdash B}\,{\scriptstyle Cut: A_2}$$

The following property of the rewrite rules is not difficult to check:

Proposition 1 (Soundness of Rewrite Rules) *The above rewrite rules preserve the $\mathcal{T}$-equivalence of proofs.* ▌

We now show that the these reduction rules are strongly normalizing. We will need the following definition in the proof of the strong normalization theorem.

Definition 2 The *grade* g of a formula A is the number of occurrences of $\otimes$ contained in A. The grade of an inessential cut is the grade of its cut formula.

Thus, by the definition above, grade of an essential cut is 0.

Theorem 2 (Strong Normalization) *There is no infinite reduction sequence beginning with any proof $\mathcal{P}$.*

Proof: Let the *complexity* of a proof be a pair (a, b), where

- $a =$ sum of the grade g of cut formulas of all inessential cuts in the proof.
- $b =$ sum of the nodes above all inessential cuts (including the premisses and conclusion of the cut).

Clearly, a cut-reduced proof has complexity (0,0). We now show that each step of reduction on a proof reduces its complexity. Consider the three classes of the transformations above. It is easy to see that application of these transformations in each case to a proof reduces its complexity.

Axiom: Both a and b are reduced.

Permutation: b is reduced keeping a the same.

Logical: a is reduced.

Thus, all reduction sequences terminate. ▌

In the following section we show that the induced reduction relation on the equivalence classes modulo the relation $\mathcal{S}(N)$ on proofs enjoys the Church-Rosser property. We will then show that every $\mathcal{T}$- equivalence class has a unique normal process representative by showing that the equivalence defined by the reduction relation on $\mathcal{S}$-equivalence classes coincides with the relation $\mathcal{T}$. The $\mathcal{S}$-equivalence class of normal forms will then be the unique process representative of a $\mathcal{T}$-equivalence class.

4 Normal process representatives.

Let $\Rightarrow$ be the reduction relation on proofs and let $\Rightarrow_{\mathcal{S}}$ be the induced reduction relation on the equivalence classes of proofs modulo the equivalence relation $\mathcal{S}$. Our aim is to show that $\Rightarrow_{\mathcal{S}}$ is weakly Church-Rosser.

Theorem 3 (Weak Church-Rosser) *The relation $\Rightarrow_{\mathcal{S}}$ satisfies the weak diamond property, that is*

$$\begin{tikzcd}
& \Pi \arrow[dl, "\Rightarrow_{\mathcal{S}}"'] \arrow[dr, "\Rightarrow_{\mathcal{S}}"] & \\
\Pi' \arrow[dr, "\Rightarrow^*_{\mathcal{S}}"'] & & \Pi'' \arrow[dl, "\Rightarrow^*_{\mathcal{S}}"] \\
& \Sigma &
\end{tikzcd}$$

Proof: (Sketch) The result is proved by induction on the structure of the proof tree by analyzing the various cases that can arise. The case where the last rule used is a $\otimes$L or a $\otimes$R follows from the induction hypothesis. The same holds when the last rule is an essential cut or when the last rule is an inessential cut and a reduction cannot be applied to this last rule. The case where latter does not hold is the only interesting case. In this case, a permutation rule may be applied in two different ways. Analyzing different possibilities, we show the existence of Σ above by applying further reduction steps to Π' and Π''. For example, let Π be of the form

$$\Pi = \dfrac{\dfrac{\overset{\Pi_1}{\Gamma, B, C \vdash A}}{\Gamma, B \otimes C \vdash A}{\scriptstyle \otimes L} \qquad \dfrac{\overset{\Pi_2}{\Delta', A \vdash D} \qquad \overset{\Pi_3}{\Delta'' \vdash E}}{\Delta', \Delta'', A \vdash D \otimes E}{\scriptstyle \otimes R}}{\Gamma, B \otimes C, \Delta', \Delta'' \vdash D \otimes E}{\scriptstyle Cut}$$

then let Π' and Π'' be obtained by application of the permutation rule for $\otimes$L and $\otimes$R, respectively. That is,

$$\Pi' = \dfrac{\dfrac{\overset{\Pi_1}{\Gamma, B, C \vdash A} \qquad \dfrac{\overset{\Pi_2}{\Delta', A \vdash D} \qquad \overset{\Pi_3}{\Delta'' \vdash E}}{\Delta', \Delta'', A \vdash D \otimes E}{\scriptstyle \otimes R}}{\Gamma, B, C, \Delta', \Delta'' \vdash D \otimes E}{\scriptstyle Cut}}{\Gamma, B \otimes C, \Delta', \Delta'' \vdash D \otimes E}{\scriptstyle \otimes L}$$

and

$$\Pi'' = \dfrac{\dfrac{\dfrac{\overset{\Pi_1}{\Gamma, B, C \vdash A}}{\Gamma, B \otimes C \vdash A}{\scriptstyle \otimes L} \qquad \overset{\Pi_2}{\Delta', A \vdash D}}{\Gamma, B \otimes C, \Delta' \vdash D}{\scriptstyle Cut} \qquad \overset{\Pi_3}{\Delta'' \vdash E}}{\Gamma, B \otimes C, \Delta', \Delta'' \vdash D \otimes E}{\scriptstyle \otimes R}$$

Now Π' can be reduced to

$$\Sigma' = \dfrac{\dfrac{\dfrac{\overset{\Pi_1}{\Gamma,B,C\vdash A} \qquad \overset{\Pi_2}{\Delta',A\vdash D}}{\Gamma,B,C,\Delta'\vdash D}\,{\scriptstyle Cut} \qquad \overset{\Pi_3}{\Delta''\vdash E}}{\Gamma,B,C,\Delta',\Delta''\vdash D\otimes E}\,{\scriptstyle \otimes\mathrm{R}}}{\Gamma,B\otimes C,\Delta',\Delta''\vdash D\otimes E}\,{\scriptstyle \otimes\mathrm{L}}$$

by another application of a permutation rule and similarly Π'' can be reduced to

$$\Sigma'' = \dfrac{\dfrac{\dfrac{\overset{\Pi_1}{\Gamma,B,C\vdash A} \qquad \overset{\Pi_2}{\Delta',A\vdash D}}{\Gamma,B,C,\Delta'\vdash D}\,{\scriptstyle Cut}}{\Gamma,B\otimes C,\Delta'\vdash D}\,{\scriptstyle \otimes\mathrm{L}} \qquad \overset{\Pi_3}{\Delta''\vdash E}}{\Gamma,B\otimes C,\Delta',\Delta''\vdash D\otimes E}\,{\scriptstyle \otimes\mathrm{R}}$$

It is easy to see that $\Sigma'\mathcal{S}\Sigma''$ and thus the required existence of Σ has been shown. ▮

Since the reduction rules are strongly normalizing by Theorem 2, we use the Newman's Lemma (see [2] on page 58) which says that WCR and SN implies CR to conclude that $\Rightarrow^*_{\mathcal{S}}$ satisfies the following diamond property which will be used in the proof of Theorem 4 below.

$$\begin{tikzcd} & \Pi \arrow[dl, "\Rightarrow^*_{\mathcal{S}}"'] \arrow[dr, "\Rightarrow^*_{\mathcal{S}}"] & \\ \Pi' \arrow[dr, "\Rightarrow^*_{\mathcal{S}}"'] & & \Pi'' \arrow[dl, "\Rightarrow^*_{\mathcal{S}}"] \\ & \Sigma & \end{tikzcd}$$

Definition 3 A *normal process representative* is an $\mathcal{S}$-equivalence class of normal forms.

Theorem 4 (Unique Process Representative) *Let N be a net. In every $\mathcal{T}$-equivalence class, there is a unique normal process representative.*

Proof: Let Π and Π' be two $\mathcal{S}$-equivalent classes. Define $\Pi \Downarrow \Pi'$ if they both reduce to same normal form modulo the equivalence $\mathcal{S}$. To prove the theorem, we only have to show that $\Pi \Downarrow \Pi'$ iff $\Pi\ \mathcal{T}\Pi'$,*i.e.* the two equivalences coincide. Since the only if part follows from the soundness of the rewrite rules, we are only left with the if part. To prove the if part, we show that if two proofs are equivalent by virtue of equation (7) in section 2, then there is a sequence of reduction $\Rightarrow^*_{\mathcal{S}}$ from one to another. We thus rewrite the left-hand side of the equation (7) to a form which is $\mathcal{S}$-equivalent to the right-hand side of the equation.

$$\dfrac{\dfrac{\overset{\Pi}{\Gamma\vdash A} \qquad \overset{\Pi'}{\Delta\vdash B}}{\Gamma,\Delta\vdash A\otimes B}\,{\scriptstyle \otimes\mathrm{R}} \qquad \dfrac{\dfrac{\overset{\Pi''}{A\vdash C} \qquad \overset{\Pi'''}{B\vdash D}}{A,B\vdash C\otimes D}\,{\scriptstyle \otimes\mathrm{R}}}{A\otimes B\vdash C\otimes D}\,{\scriptstyle \otimes\mathrm{L}}}{\Gamma,\Delta\vdash C\otimes D}\,{\scriptstyle Cut}$$

$$\Rightarrow_{\mathcal{S}} \quad \dfrac{\overset{\Pi'}{\Delta \vdash B} \qquad \dfrac{\overset{\Pi}{\Gamma \vdash A} \qquad \dfrac{\overset{\Pi''}{A \vdash C} \qquad \overset{\Pi'''}{B \vdash D}}{A, B \vdash C \otimes D}\,{\otimes}\mathrm{R}}{\Gamma, B \vdash C \otimes D}\,\mathit{Cut}}{\Gamma, \Delta \vdash C \otimes D}\,\mathit{Cut}$$

$$\Rightarrow_{\mathcal{S}} \quad \dfrac{\overset{\Pi'}{\Delta \vdash B} \qquad \dfrac{\dfrac{\overset{\Pi}{\Gamma \vdash A} \qquad \overset{\Pi''}{A \vdash C}}{\Gamma \vdash C}\,\mathit{Cut} \qquad \overset{\Pi'''}{B \vdash D}}{\Gamma, B \vdash C \otimes D}\,{\otimes}\mathrm{R}}{\Gamma, \Delta \vdash C \otimes D}\,\mathit{Cut}$$

$$\Rightarrow_{\mathcal{S}} \quad \dfrac{\dfrac{\overset{\Pi}{\Gamma \vdash A} \qquad \overset{\Pi''}{A \vdash C}}{\Gamma \vdash C}\,\mathit{Cut} \qquad \dfrac{\overset{\Pi'}{\Delta \vdash B} \qquad \overset{\Pi'''}{B \vdash D}}{\Gamma \vdash D}\,\mathit{Cut}}{\Gamma, \Delta \vdash C \otimes D}\,{\otimes}\mathrm{R}$$

Thus we have shown the existence of a unique normal $\mathcal{S}$-class for each $\mathcal{T}$-class. ▮

In the next section we briefly sketch how rewrites on proofs can be viewed as (typed) rewrite on arrows in a suitable category. A detailed analysis of this relationship will be discussed elsewhere.

5 A note on arrows vs. proofs.

As we mentioned before, some authors have found it convenient to work with arrows (in strictly symmetric strict monoidal categories to be exact) rather than proofs as we have done in this paper. To some extent this is a matter of taste, but it can be illuminating to see things in both ways. For example, the rewrite rules for proofs in section 3 are translated respectively as the following rewrite on arrows.

1.
$$i^{A \to A} \,;\, f^{A \to B} \Rightarrow f^{A \to B}$$

2.
$$(f^{G \to A} \otimes i^{D \otimes E \to D \otimes E})\,;(g^{D \otimes A \to B} \otimes h^{E \to C}) \Rightarrow ((f^{G \to A} \otimes i^{D \to D})\,;g^{D \otimes A \to B}) \otimes h^{E \to C}$$

3.
$$((f^{G \to A} \otimes g^{E \to C}) \otimes i^{D \to D})\,;h^{A \otimes C \otimes D \to B} \Rightarrow (g^{E \to C} \otimes i^{G \otimes D \to G \otimes D})\,;((f^{G \to A} \otimes i^{E \otimes D \to E \otimes D})\,;h^{A \otimes C \otimes D \to B})$$

These may at first sight seem rather unwieldy, but our proof-theoretic results show that they will work. Moreover, we found that working with proofs helped us to get the right definition of normal form. Concerning the rewrite system, the proof of Theorem 4 suggests that if one is given associativity and commutativity of arrows for free, the following rewrite will work whenever the right-hand side is defined.

$$(f_1 \otimes f_2)\,;(g_1 \otimes g_2) \Rightarrow (f_1\,; g_1) \otimes (f_2\,; g_2)$$

In other words, this rewrite rule will give a unique "normal" $\mathcal{S}[N]$ arrow for each $\mathcal{T}[N]$ equivalent class of arrows of [5]. In the rewrite above, the left-hand side is always defined whenever the right-hand side is defined but not vice-versa. In particular, subject reduction fails drastically, so the rewrite system must maintain the types of the terms.

References

[1] A. Asperti, G. L. Ferrari, and R. Gorrieri. Implicative formulae in the "proofs as computations" analogy. In P. Hudak, editor, *Principles of Programming Languages*, pages ??-??, ACM, 1990.

[2] H. Barendregt. *The Lambda Calculus: Its syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, Elsevier, revised edition, 1984.

[3] E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55(1):87–136, November 1987.

[4] C. Brown. *Relating Petri nets to formulae of linear logic.* Technical Report ECS-LFCS-89-87, University of Edinburgh, 1989.

[5] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing net computations and processes. In R. Parikh, editor, *Logic in Computer Science*, pages 175–185, IEEE, IEEE Computer Society, June 1989.

[6] U. Engberg and G. Winskel. On linear logic and Petri nets. Unpublished manuscript.

[7] J. Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[8] C. A. Gunter and V. Gehlot. Nets as tensor theories. (preliminary report). In G. De Michelis, editor, *Applications of Petri Nets*, pages 174–191, 1989. Also University of Pennsylvania, Logic and Computation Report Number 17.

[9] N. Martí-Oliet and J. Meseguer. *From Petri nets to linear logic.* Research Report SRI-CSL-89-4, SRI International, Menlo Park, March 1989.

[10] J. Meseguer and U. Montanari. *Petri Nets Are Monoids.* Research Report SRI-CSL-88-3, SRI International, Menlo Park, January 1988.

Reference Counting as a Computational Interpretation of Linear Logic

(To appear in: **Journal of Functional Programming.**)

Jawahar Chirimar
Department of CIS
University of Pennsylvania
Philadelphia, PA 19104

Carl A. Gunter
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Jon G. Riecke
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

April 1995

Abstract

We develop formal methods for reasoning about memory usage at a level of abstraction suitable for establishing or refuting claims about the potential applications of linear logic for static analysis. In particular, we demonstrate a precise relationship between type correctness for a language based on linear logic and the correctness of a reference-counting interpretation of the primitives that the language draws from the rules for the 'of course' operation. Our semantics is 'low-level' enough to express sharing and copying while still being 'high-level' enough to abstract away from details of memory layout. This enables the formulation and proof of a result describing the possible run-time reference counts of values of linear type.

Contents

1 Introduction

There have been a variety of efforts to exploit ideas from linear logic for the design and analysis of programming languages. It is our contention that a perspective on these proposals can be found in the view that linear logic is a tool for analyzing a structure we call a *memory graph* which is used to represent the run-time data of a program. A memory graph is simply a directed graph together with a finite set of functions that map finite sets called *roots* into the nodes of the graph. It is a mathematical abstraction of the run-time structure that holds such data as the activation records of procedures, heap-allocated objects, and so on. We argue a programming language based on linear logic yields fine-grained information about how the memory graph evolves at run-time, thus providing information that could be exploited in program analysis. In particular, the information provided by the linear connectives concerns the *reference counts* of nodes in the memory graph, where the reference count of a node is the sum of the in-degree of the node in the graph and the number of root elements mapped to it. Reference counting has a long [Col60, DB76], albeit controversial [WHHO92, Bak88, App92], history as a technique for avoiding garbage collection. But, aside from its direct use in managing memory, reference counting can offer a unifying view of many code optimizations, and various code generation strategies can be seen as attempts to control reference counts so that optimizations can be performed. For example, the correctness of *in-place updating* relies on ensuring that the reference count of an object is one.

Attempts to study programming languages like the one in this paper fall roughly into two groups. There are those that use some analog of the Curry-Howard correspondence [How80] as the basis for the design of a language based on linear logic [Abr, Hol88, Laf88, LM92, Mac91], and those that consider systems similar to linear logic (hereafter called 'LL') for specific applications (for instance, [GH90] and [Wad91b] consider systems to detect single-threading). The system presented in this paper falls into the former category, except to the extent that we have added some additional constructs, such as recursive functions, that bring us closer to traditional functional programming languages.

To convey some of the spirit of the ideas and constructs discussed in the literature just mentioned, let us look at a concrete example. Consider the program on the left side of Table 1. The code implements the addition function in terms of functions for incrementing and decrementing. The syntax is that of SML using a set of familiar computational primitives such as recursive definition, branching conditional, and local definition. Looking closely at the definition of addition, it is possible to note a difference between how the formal parameters of `add`, the variables `x` and `y`, are used in the body of the definition. The value of `x` is needed in the test of the conditional, which is always evaluated, and in the `else` branch of the conditional, which may not be evaluated, but not in the `then` branch of the conditional. On the other hand, the variable `y` is needed regardless of whether the `then` or `else` branch of the conditional in the body is taken. In particular, its value is needed *only* once, not twice—as may be the case with `x`. This brings out two aspects of the difference between `x` and `y`: first, that the two variables may be *used* a different number of times (`y` exactly once and `x` either once or twice) and, second, that the value of `x` must be *shared* between its two separate uses.

The program on the right is a version of the addition function written in a program with 'linear logic annotations' (using a slight simplifcation of the notation that we will define precisely later). There are four new primitives used here: `share`, `dispose`, `store`, and `fetch`. The `share` primitive indicates that `x` is needed twice: the first use is bound to the variable `w` and the second

Table 1: Translating to a Linear-Logic-Based Language.

```
let fun add x y =
      if x = 0
      then y
      else add (x-1) (y+1)
in add 2 1
end
```

```
let fun add x y =
      share w,z as x in
        if fetch w = 0
        then dispose z, add before y
        else (fetch add)
               (store ((fetch z)-1))
               (y+1)
in add (store 2) 1
end
```

to the variable z. These two variables *share* the value to which x is bound. The dispose primitive indicates that one of these sharing variables, z, is not used in the first branch of the conditional. The primitive store creates a sharable value and fetch obtains a shared value. In our interpretation, the LL-specific operations share and dispose explicitly manage reference counts of the share'able and dispose'able objects that are created and consulted by being store'd and fetch'ed. For the example in Table 1, the occurence of share indicates that two pointers are needed for the value associated with x (so the reference count of the associated value is incremented), but in the then branch of the conditional, one of the pointers is no longer needed (so the reference count of the associated value is decremented).

Analogs to the store and fetch operations are the *delay* and *force* operations that appear in many functional programming languages. In such languages, the delay primitive postpones the evaluation of a term until it is supplied to the force primitive as an argument. When this happens, the value of the delayed term is computed, returned, and memoized for any other applications of force. Abramsky [Abr] has argued that this is a natural way to view the operational semantics of the store and fetch operations of LL; we will follow this approach as well. The dispose primitive has an analog (and namesake) in several programming languages. Typically, an object is disposed by being deallocated; this operation is unsafe because it can lead to dangling pointers. In our LL language the primitive dispose will only deallocate memory if this is safe since its semantics will be to decrement a reference count; deallocation only happens when this count falls to zero. The share command is unique to LL, and its name accurately reflects the way in which it will be interpreted.

One of our primary goals in this paper is to offer an approach for rigorously expressing and proving optimizations obtained by analyzing an LL-based language. In particular, there is an adage that 'linear values have only one pointer to them' or 'linear values can be updated in place'. Wadler [Wad90] has informally observed that these claims must be stated with some care: a reference count of one can be maintained by copying, but this would negate the advantage of in-place updating. Our operational semantics allows us to check the claim rigorously: in particular, we show that linear variables may *fail* to have a count of one in our reference-counting operational semantics, which uses sharing heavily; when this is the case, a linear variable does not have a unique pointer to it and cannot safely be updated in place. The problem arises when a linear variable falls within the scope of an abstraction over a non-linear variable. We express a theorem asserting precisely when the value of a linear variable does indeed maintain a reference count of at most one.

A broader theme of our investigation is developing a level of abstraction in the semantics of programming languages that permits 'low-level' concepts to be formalized in a clear and relevant way. There has been significant progress in formulating theorems about programming languages and memory ([GG92] and [WO92] are recent examples treating garbage collection and run-time storage representation respectively). It is our hope that we can contribute to a foundation for further advances in this direction.

We will be concerned only with the question of a computational interpretation of *intuitionistic* linear logic, the fragment of the language without negation and the 'par' operation. In fact, we will restrict ourselves to the language obtained from the linear implication ($s \multimap t$) and 'of course' ($!s$) operations, although our results can be extended to all of intuitionistic LL. For the rest of the paper, read 'linear logic' to mean the implicational fragment of intuitionistic linear logic. We present our language and its properties in stages. The second section of the paper discusses the operational semantics of memoization with the aim of putting in place the basic notation and approach that will be used in subsequent sections. The third section describes the syntax, typing rules, and 'high-level' operational semantics for our LL-based language. The fourth section describes the 'low-level' operational semantics of the language. The invariants that express the basic properties of the memory graph in this semantics are precisely expressed and proved. The fifth section of the paper demonstrates further basic properties of this semantics, including its correspondence to the high-level semantics and its independence from the scheme used to allocate new memory. The sixth section uses the operational semantics to prove a static condition under which a linear value will always have a reference count of one; this shows that the LL-based language is indeed amenable to analysis about memory usage. The seventh section discusses various aspects of the technical results of the paper and attempts to provide additional perspective. Some of the most technical proofs have been deferred to an appendix.

2 Operational Semantics with Memory

Here we give a preview of the operational semantics of the LL-based language by describing the familiar operational semantics of a simple functional language with `store` (delay) and `fetch` (force) operations. We base this preliminary discussion on a language with the grammar

$$\begin{array}{rcl} M & ::= & x \mid (\lambda x.\, M) \mid (M\ M) \mid \\ & & n \mid \mathsf{true} \mid \mathsf{false} \mid (\mathsf{succ}\ M) \mid (\mathsf{pred}\ M) \mid (\mathsf{zero?}\ M) \mid \\ & & (\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M) \mid (\mathsf{fix}\ M) \mid \\ & & (\mathsf{store}\ M) \mid (\mathsf{fetch}\ M) \end{array}$$

where x and n are from primitive syntax classes of variables and numerals respectively. This is a variant of PCF [Sco, Plo77, BGS90] augmented by primitive operations for forcing and delaying evaluations. The expression $(\mathsf{fix}\ M)$ is used for recursive definitions.

The key to providing a semantics for this language is to represent the memoization used in computing the `fetch` primitive so that certain recomputation is avioded. We aim to provide a semantics at a fairly high level of abstraction using what is sometimes known as a *natural semantics* [Des86, Kah87]. Such a semantics has been described in [PS91] using explicit substitution and in [Lau93] through the use of an intermediate representation in which all function applications have variables as arguments. Both of these approaches are appealingly simple but slightly more abstract than we would like for our purposes in this paper. Our own approach, first described in [CGR92], is based on a distinction between an *environment* which is an association of variables with locations and a *store* which is an association of values with locations. Sharing of computation results is achieved through creating multiple references to a location that holds a delayed computation called a *thunk*. When the value delayed in the thunk is needed, it is calculated and memoized for future reference. To define this precisely we must begin with some notation and basic operations for environments, stores, and memory allocation.

Fix an infinite set of locations Loc, with the letter l denoting elements of this set. Let us say that a partial function is finite just in case its domain of definition is finite.

- An **environment** is a finite partial function from variables to locations; ρ denotes an environment, and Env denotes the set of all environments. The notation $\rho(x)$ returns the location associated with variable x in ρ, and to update an environment, we use the notation

$$(\rho[x \mapsto l])(y) = \begin{cases} l & \text{if } x = y \\ \rho(y) & \text{otherwise.} \end{cases}$$

  The symbol $\emptyset$ denotes the empty environment; we also use $[x \mapsto l]$ as shorthand for $\emptyset[x \mapsto l]$.

- A **value** is a
  - numeral k,
  - boolean b,
  - pointer $\mathsf{susp}(l)$ or $\mathsf{rec}(l, f)$, or
  - closure $\mathsf{closure}(\lambda x.\, M, \rho)$ or $\mathsf{recclosure}(\lambda x.\, M, \rho)$.

  The letter V denotes a value, and Value denotes the set of values.

- A **storable object** is either a value or a thunk $\mathsf{thunk}(M, \rho)$. We use Storable to denote the set of storable objects.
- A **store** is a finite partial function σ from Loc to Storable. The symbol σ denotes a store, $\emptyset$ denotes the empty store, and Store denotes the set of stores. We will use the same notation to update stores as for updating environments.

Given a store σ and a location l, we define $\sigma[l \mapsto S]$ to be the store obtained by updating σ by binding l to the storable object S. We also need a relation for allocating memory cells. A subset R of the product $(\mathsf{Storable} \times \mathsf{Store}) \times (\mathsf{Loc} \times \mathsf{Store})$ is an **allocation relation** if, for any store σ and storable object S, there is an l' and σ' where $(S, \sigma)\ R\ (l', \sigma')$ and

- $l' \notin \mathsf{dom}(\sigma)$ and $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \cup \{l'\}$;
- for all locations $l \in \mathsf{dom}(\sigma)$, $\sigma(l) = \sigma'(l)$; and
- $\sigma'(l') = S$.

This definition abstracts away from the issue of exactly how new locations are found. For specificity, we choose an allocation relation new that is a function, and write $\mathsf{new}(S, \sigma)$ for the pair (l', σ') such that $(S, \sigma)\ \mathsf{new}\ (l', \sigma')$. Of course, our operational semantics should be independent of the choice of allocation relation, a point we will formalize after describing the semantics of our LL-based language below.

The operational rules for our language could be given using a natural semantics with rules of the form $(M, \rho, \sigma) \Downarrow (l, \sigma')$ where the domain of ρ contains the set of free variables of M, and l is a location in the domain of σ' that holds the result of evaluation. Writing the semantics in the form of rules (*e.g.*, as in the appendix of [CGR92]) becomes somewhat cumbersome, so we use a kind of primitive pseudo-code that can readily be translated into a natural semantics. As a first example, consider how the store primitive is evaluated:

```
meminterp((store $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = new(thunk($M$, $\rho$), $\sigma$)
  in new(susp($l_0$), $\sigma_0$)
```

Read this as follows. To evaluate (store M) in the environment ρ and store σ, first allocate a new location holding a thunk composed of M and the environment ρ. Let σ_0 be the new store and l_0 be the location in which the thunk is held. The result of the evaluation is a store obtained from σ_0 by allocating a new location holding the storable value $\mathsf{susp}(l_0)$ paired with this new location. Note, in particular, that M is not evaluated. The structure that has been added to the memory is depicted in Figure 1.

The interesting part of the evaluator and the essence of memoization is given by the way in which the fetch primitive is handled. The argument of fetch is evaluated to return a storable value of the form $\mathsf{susp}(l_1)$. The content of location l_1 is then examined to determine whether the suspension has been evaluated to a value or whether it has not yet been evaluated, in which case it has the form $\mathsf{thunk}(N, \rho)$. If the content is a value, a pointer to the value is returned, otherwise the thunk is evaluated and l_1 duly updated with its value. A pointer to the value of the thunk is then returned as the result. Here is the pseudo-code description:

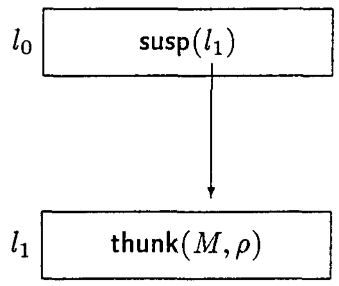


Figure 1: Structure generated by (store M)

```
meminterp((fetch $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp($M$, $\rho$, $\sigma$)
  in  case $\sigma_0(l_0)$ of susp($l_1$) =>
         case $\sigma_0(l_1)$
           of thunk($N$, $\rho'$) =>
                let ($l_2$, $\sigma_1$) = meminterp($N$, $\rho'$, $\sigma_0$)
                in  ($l_2$, $\sigma_1[l_0 \mapsto$ susp($l_2$)])
            | _ => ($l_1$, $\sigma_0$)
```

Note that there is no clause for the case when $\sigma_0(l_0)$ is *not* a suspension. In this case, we assume that the behavior of the interpreter on (fetch M) is undefined. This assumption simplifies the rules, and allows us to ignore what are, in effect, run-time type errors. Our other rules will also ignore run-time type errors.

There is another approach we might have taken to modelling memoization. The interpretation of (store M) allocates a location l_0 that holds a thunk, and returns a location l_1 that holds a pointer susp(l_0) to this location. Could we instead have returned l_0 as the value? That is, the rule could read

```
meminterp'((store $M$), $\rho$, $\sigma$) = new(thunk($M$, $\rho$), $\sigma$)
```

The answer to this question is instructive, since it relates to the way in which we will represent the distinction between copying and sharing in our model. If we choose to return the location holding the thunk as the value of the store (as opposed to returning a location holding the pointer to this thunk), then this would require a change in the fetch command. In particular, when the location l_2 is obtained there, it would be essential to put the value $\sigma(l_2)$ in the location where the value of the thunk may be sought later:

```
meminterp'((fetch $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp'($M$, $\rho_1$, $\sigma$)
  in  case $\sigma_0(l_0)$
        of thunk($N$, $\rho'$) =>
             let ($l_2$, $\sigma_1$) = meminterp'($N$, $\rho'$, $\sigma_0$)
             in  ($l_0$, $\sigma_1[l_0 \mapsto \sigma_1(l_2)]$)
         | _ => ($l_0$, $\sigma_0$)
```

Note that in the second line from the bottom of the program, the values of l_0 and l_2 in the store are the same and we will say that the value of the thunk has been *copied* from location l_2 to l_0. In the case that $\sigma_1(l_2)$ is a 'small' value, like an integer that occupies only a word of storage, there is little difference between copying the value from l_2 to l_0 versus returning a pointer to l_2 as we did in the earlier implementation. If the value $\sigma_1(l_2)$ is 'large', however, then copying may be expensive. In the language we are considering, this might involve copying a closure, which would be a modest expense, but in a fuller language it might involve copying a string or functional array, which could be very expensive. (If $\sigma_1(l_2)$ is a mutable value, then the copying is probably incorrect—but this is not a problem for the functional language at hand.) Our semantics does not directly represent the cost associated with copying because it abstracts away from a measure of the size of a value; instead, we will treat copying as if it is something to be avoided in favor of sharing (indirection) whenever this is feasible. This suggests yet a third approach to the semantics of fetch where store is implemented as with `meminterp'` but where the interpretation of fetch uses an indirection for the returned value:

```
meminterp''((fetch $M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp''($M$, $\rho_1$, $\sigma$)
  in  case $\sigma_0(l_0)$
        of thunk($N$, $\rho'$) =>
             let ($l_2$, $\sigma_1$) = meminterp''($N$, $\rho'$, $\sigma_0$)
             in  ($l_0$, $\sigma_1[l_0 \mapsto @l_2]$)
         | _ => ($l_0$, $\sigma_0$)
```

where $@l_2$ is to be viewed as a boxed value. This is possibly closer to the way memoization would be implemented in most compilers. For the semantics we use for the LL-based language in the next section this approach complicates the semantics slightly and is less efficient because of the way the reference counting is done. Otherwise, our approach could accomodate this alternative without major changes.

The implementation of memoization involves the idea of mutating a store. Even the 'functional' parts of the language must respect the potential side effects to the store that memoization may cause. Hence these operations must pass the store along in an appropriate manner. Doing this correctly may save recomputation. Here, for instance, is how the application operation is described:

```
meminterp(($M$ $N$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = meminterp($M$, $\rho$, $\sigma$)
      ($l_1$, $\sigma_1$) = meminterp($N$, $\rho$, $\sigma_0$)
  in  case $\sigma_1(l_0)$ of closure($\lambda x.N$, $\rho'$) =>
        meminterp($N$, $\rho'[x \mapsto l_1]$, $\sigma_1$)
```

The store resulting from evaluating M is used in evaluating N; similarly, the store resulting from evaluating N is used in evaluating the application.

There are a variety of ways to implement recursion. A reasonably efficient approach is to create a circular structure. This approach is simplified by restricting the interpreter to programs such that, in constructs of the form (fix N), the term N has the form $\lambda f.\lambda x.M$. The restriction is not necessary, but it is typical for call-by-value programming languages. The semantics for such recursions is given by

```
meminterp((fix $\lambda f.\lambda x.M$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = new(0, $\sigma$)
      ($l_1$, $\sigma_1$) = new(recclosure($\lambda x.M$, $\rho[f \mapsto l_0]$), $\sigma_0$)
  in  ($l_0$, $\sigma_1[l_0 \mapsto$ rec($l_1$, $f$)])
```

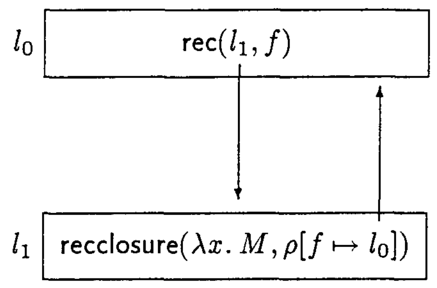


Figure 2: Structure generated by ($\mathsf{fix}\ \lambda f.\, \lambda x.\, M$)

which creates the circular structure in Figure 2. For this language we could create a single cell holding the recclosure that looped back to itself; we use two cells, though, since the additional cell holding rec will be used in the semantics of the LL-based language to facilitate connections with the type system. We also need here to change the semantics of applications so that if the operator evaluates to a rec, the pointer is traced to a recclosure; in turn, if the operator evaluates to a recclosure, the operator is used in the same way as a closure.

In the implementation of actual functional programming languages, a single recursion such as the one above would probably make its recursive calls through a jump instruction. This would be quite difficult to formalize with the source-code-based approach we are using to describe the interpreter. The important thing, for our purposes, is that recursive calls to f do not allocate further memory for the recursive closure. This means that, as far as memory is concerned, there is little difference between implementing the recursion with the jump and implementing it with a circular structure. The cycle created in this way introduces extra complexity into the structure of memory, of course, but the cycles introduced in this way must have precisely the form pictured in Figure 2.

It is easy to provide a clean type system for the language described above. One technical convenience is to tag certain bindings with types (such as the binding occurence in an abstraction $\lambda x : s.\, M$) to ensure that a given program has a unique type derivation. When it is not important for the discussion at hand, we will often drop the tags on bound variables to reduce clutter. The types for the language include ground types Nat and Bool for numbers and booleans respectively, higher types $(s \to t)$ for functions between s and t, and a unary operation $!s$ for the delayed programs of type s. The typing rules for store and fetch are introduction and elimination operations respectively:

$$\frac{M : s}{(\mathsf{store}\ M) : !s} \qquad \frac{M : !s}{(\mathsf{fetch}\ M) : s}.$$

These operations will also be found in our LL-based language with essentially the same types.

3 A Programming Language Based on Linear Logic

Term assignment for linear logic.

If a programming language L is to be based on LL, it seems reasonable to attempt the completion of an analogy based on the Curry-Howard correspondence: intuitionistic logic is to traditional functional programming languages (such as ML or Haskell) as LL is to L. Basing a language on the Curry-Howard correspondence for LL immediately becomes problematic, as LL was originally described by Girard [Gir87] using a sequent calculus. Most programming languages have a syntax and typing system like the natural deduction (hereafter called 'ND') formulation of intuitionistic logic rather than its sequent calculus formulation, since type-checking algorithms are easier to describe for ND formulations. Progress on an ND form for intuitionistic LL has been gradual, in part because **substitutivity** fails for the obvious formulations:

Definition 1 A type system satisfies the **substitutivity** property if well-typed programs are closed under substitution, *i.e.*, if $\Gamma \vdash M : t$ and $\Delta,\ x : t \vdash N : u$ and all variables in Γ and Δ are distinct, then $\Gamma, \Delta \vdash N[x := M] : u$.

Here $M[x := N]$ denotes substitution of N for x in M with the bound variables of M renamed to avoid capture of the free variables of N. SML [MTH90, MT91] is a prototypical example of a language based on an ND presentation that satisfies the substitutivity property.

Merely coming up with a ND presentation of LL that satisfies substitutivity has been an outstanding problem. In the absence of such a system, Lincoln and Mitchell [LM92], Mackie [Mac91], Wadler [Wad91a], and the authors of this paper in a preceeding work [CGR92] employed approaches that obtain some of the virtues of an ND system for LL. The system used in this paper is based on a proposal of Benton, Bierman, de Paiva, and Hyland [BBdH92] that *does* satisfy the substitutivity property, even though it lacks some of the desirable properties of the ND presentation of intuitionistic logic (such as freedom from the need to use commuting conversions [GLT89]). We refer the reader to their paper for a fuller discussion.

The propositions of the fragment of linear logic we consider are given by the grammar

$$s \ ::= \ a \mid (s \multimap s) \mid\ !s$$

where a ranges over atomic propositions. The proofs of linear propositions are encoded by terms in the grammar

$$\begin{aligned} M \ ::= \ & x \mid (\lambda x : s.\, M) \mid (M\ M) \mid \\ & (\mathsf{store}\ M\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n) \mid (\mathsf{fetch}\ M) \mid \\ & (\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ M) \mid (\mathsf{dispose}\ M\ \mathsf{before}\ M). \end{aligned}$$

Our notation here essentially corresponds to that in [CGR92, LM92] modulo incorporating adjustments from [BBdH92]. The store operation,

$$(\mathsf{store}\ M\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n),$$

binds the variables $x_1, \ldots, x_n$ in the expression M and the share operation

$$(\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ N)$$

Table 2: Natural deduction rules and term assignment for linear logic.

$$x : s \vdash x : s$$

$$\frac{\Gamma, x : s \vdash M : t}{\Gamma \vdash (\lambda x : s.\, M) : (s \multimap t)} \qquad \frac{\Gamma \vdash M : (s \multimap t) \qquad \Delta \vdash N : s}{\Gamma, \Delta \vdash (M\, N) : t}$$

$$\frac{\Gamma \vdash M : !s \qquad \Delta \vdash N : t}{\Gamma, \Delta \vdash (\mathsf{dispose}\ M\ \mathsf{before}\ N) : t} \qquad \frac{\Gamma \vdash M : !s \qquad \Delta, x : !s, y : !s \vdash N : t}{\Gamma, \Delta \vdash (\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ N) : t}$$

$$\frac{\Gamma_1 \vdash M_1 : !s_1 \quad \ldots \quad \Gamma_n \vdash M_n : !s_n \qquad x_1 : !s_1, \ldots, x_n : !s_n \vdash N : t}{\Gamma_1, \ldots, \Gamma_n \vdash (\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n) : !t}$$

$$\frac{\Gamma \vdash M : !s}{\Gamma \vdash (\mathsf{fetch}\ M) : s}$$

binds the variables x and y in N. The notation for **store** can be somewhat unwieldy when writing programs, but most programs involving **store** bind the variables in the **where** clause to other variables. Thus, if the free variables of M are $x_1, \ldots, x_n$, then (**store** M) is shorthand for the expression (**store** M **where** $x_1 = x_1, \ldots, x_n = x_n$).

The typing rules for the language appear in Table 2, where the symbols Γ and Δ denote **type assignments**, which are lists of pairs $x_1 : s_1, \ldots, x_n : s_n$, where each x_i is a distinct variable and each s_i is a type. Each of the rules is built on the assumption that all left-hand sides of the $\vdash$ symbol are legal type assignments, *e.g.*, in the rule for typing applications, the type assignments Γ and Δ, which appear concatenated together in the conclusion of the rule, must have disjoint variables. Each type-checking rule corresponds to a proof rule in the ND presentation of linear logic. For instance, the rules for **share** and **dispose** essentially correspond to the proof rules generally called *contraction* and *weakening* respectively, while those for **store** and **fetch** correspond to the LL rules called *promotion* and *dereliction.* Due to the presence of explicit rules for weakening and contraction—the rules for type-checking **dispose** and **share**—one can easily see that the free variables of a well-typed term are *exactly* those contained in the type assignment. A particular note should be taken of the form of the rule for **store**; this operation puts the value of its body with bindings for its free variables in a location that can be shared by different terms during reduction—the type changes correspondingly from t to $!t$. The construct (**fetch** M) corresponds to reading the stored value—the type changes from $!t$ to t.

There may be other ND presentations of LL on which one could base a type system. It is our belief that results in this paper are robust with respect to the exact choice of term assignment and type-checking rules. All of the results in this paper—including negative results that say that values of linear type may have more than one pointer to them—hold in the system described in [CGR92], and we expect that they are true for the languages described in [LM92] and [Mac91].

Table 3: Typing rules for non-logical constructs.

$$\vdash n : \mathsf{Nat} \qquad\qquad \vdash \mathsf{true}, \mathsf{false} : \mathsf{Bool}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash (\mathsf{succ}\ M) : \mathsf{Nat}} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash (\mathsf{pred}\ M) : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash (\mathsf{zero?}\ M) : \mathsf{Bool}} \qquad \frac{\Gamma \vdash M : !(!s \multimap s)}{\Gamma \vdash (\mathsf{fix}\ M) : s}$$

$$\frac{\Gamma \vdash L : \mathsf{Bool} \quad \Delta \vdash M : s \quad \Delta \vdash N : s}{\Gamma, \Delta \vdash (\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N) : s}$$

A programming language based on linear logic.

To fully realize the ideas of LL as the basis for a programming language, it is essential to go beyond the core language. First of all, the language could be extended to one that includes the linear logic connectives for pairing and sums, namely *tensor* $\otimes$, *plus* $\oplus$, and *with* &. Suitable ND proof rules for these connectives and term assignments for proofs using these rules are described in several places [Mac91, LM92, BBdH92]. A more challenging question is how to extend the language to include constructs for which the use of the Curry-Howard correspondence is less useful as a guide. Examples that fall in this category are arrays, general recursive datatypes involving linear implication, and recursive definitions of functions. In this paper we treat only recursive function definitions; the question of the proper treatment of recursive definitions in an LL-based language is likely to be simpler than that of general recursive datatypes, and more fundamental than that of arrays.

Our language is essentially a synthesis of PCF and the term language for encoding LL natural deduction proofs. The types are given by the following grammar:

$$s \quad ::= \quad \mathsf{Nat} \mid \mathsf{Bool} \mid (s \multimap s) \mid\ !s$$

Types without leading !'s, *e.g.*, Nat and (Nat $\multimap$ Bool), are called **linear** and those of the form $!s$ are called **non-linear**. We use the letters s, t, u, and v to denote types. The set of raw terms in the language is given by the grammar

$$\begin{aligned} M \quad ::= \quad & x \mid (\lambda x : s.\, M) \mid (M\ M) \mid \\ & n \mid \mathsf{true} \mid \mathsf{false} \mid (\mathsf{succ}\ M) \mid (\mathsf{pred}\ M) \mid (\mathsf{zero?}\ M) \mid (\mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M) \mid (\mathsf{fix}\ M) \mid \\ & (\mathsf{store}\ M\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n) \mid (\mathsf{fetch}\ M) \mid \\ & (\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ M) \mid (\mathsf{dispose}\ M\ \mathsf{before}\ M) \end{aligned}$$

where the letter x denotes any variable, and n denotes a numeral in $\{0,1,2,,\ldots\}$. The last four operations correspond to the special rules of linear logic; the other term constructors are those of PCF. The usual definitions of free and bound variables for PCF also apply here for the first three lines of the grammar.

The typing rules for our language are given by combining Tables 2 and 3. Two of these rules deserve special explanation. First, the rule for checking the expression if L then M else N checks both branches in the same type assignment, *i.e.*, the terms M and N must contain the same free variables. This is the only type-checking rule that allows variables to *appear* multiple times; it does not, however, violate the intuition that variables are *used* once, since only one branch will be taken during the execution of the program. Second, the slightly mysterious form of the typing rule for recursions is related to the idea that the formal parameter of a recursive definition must be share'd and dispose'd if there is to be anything interesting about it. Consider, for example, the rendering of the program of Table 1 into our language:

```
(fix (store λ add : !(!Nat⊸Nat⊸Nat). λx : !Nat. λy : Nat.
   share w, z as x in
      if zero? (fetch w)
      then dispose z before dispose add before y
      else (fetch add) (store (pred (fetch z))) (succ y)))
(store 2) 1
```

(where some liberties have been taken in dropping a few of the parentheses to improve readability). The recursive function *add* being defined gets used only in one of the branches; thus, the recursive call must have a non-linear type.

The definition of the addition function is a prototypical example of how one programs recursive functions in this language. In fact, both the high-level and low-level semantics will only interpret recursions (fix M) where M has the form

$$(\mathsf{store}\ (\lambda f : !s \multimap t.\ \lambda x : s.\ M)\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n).$$

This restriction is closely connected to the restriction on interpreting recursion mentioned in the previous section; the only difference here is the occurrence of the **store**. As before, this restriction is not essential, but it does simplify the semantic clause for the recursion somewhat without compromising the way programs are generally written.

Natural semantics.

Tables 4 and 5 give a high-level description of an interpreter for our language, written using natural semantics. A natural semantics describes a partial function $\Downarrow$ via proof trees. The notation $M \Downarrow c$, read 'the term M halts at the final result c', is used when there is a proof from the rules with the conclusion being $M \Downarrow c$. The terms at which the interpreter function halts are called **canonical forms**; it is easy to see from the form of the rules that the canonical forms are n, true, false, $(\lambda x.\, M)$, and (store M).

The natural semantics in Tables 4 and 5 describes a *call-by-value* evaluation strategy. That is, operands in applications are evaluated to canonical form before the substitution takes place. A basic property of the semantics is that types are preserved under evaluation:

Theorem 2 *Suppose* $\vdash M : s$ *and* $M \Downarrow c$, *then* $\vdash c : s$.

The proof can be carried out by an easy induction on the height of the proof tree of $M \Downarrow c$.

Table 4: Interpreting the Linear Core

$$\lambda x.\, M \Downarrow \lambda x.\, M \qquad \frac{M \Downarrow \lambda x.\, P \quad N \Downarrow d \quad P[x := d] \Downarrow c}{(M\ N) \Downarrow c}$$

$$\frac{M \Downarrow d \quad N \Downarrow c}{(\mathsf{dispose}\ M\ \mathsf{before}\ N) \Downarrow c} \qquad \frac{M \Downarrow d \quad P[x, y := d] \Downarrow c}{(\mathsf{share}\ x, y\ \mathsf{as}\ M\ \mathsf{in}\ P) \Downarrow c}$$

$$\frac{M_1 \Downarrow c_1 \quad \ldots \quad M_n \Downarrow c_n}{(\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n) \Downarrow (\mathsf{store}\ N[x_1 := c_1, \ldots, x_n := c_n])}$$

$$\frac{M \Downarrow (\mathsf{store}\ N) \qquad N \Downarrow c}{(\mathsf{fetch}\ M) \Downarrow c}$$

Table 5: Interpreting the PCF Extensions

$$\mathsf{true} \Downarrow \mathsf{true} \qquad \mathsf{false} \Downarrow \mathsf{false} \qquad n \Downarrow n$$

$$\frac{M \Downarrow n}{(\mathsf{succ}\ M) \Downarrow (n+1)} \quad \frac{M \Downarrow (n+1)}{(\mathsf{pred}\ M) \Downarrow n} \quad \frac{M \Downarrow 0}{(\mathsf{pred}\ M) \Downarrow 0}$$

$$\frac{M \Downarrow 0}{(\mathsf{zero?}\ M) \Downarrow \mathsf{true}} \qquad \frac{M \Downarrow (n+1)}{(\mathsf{zero?}\ M) \Downarrow \mathsf{false}}$$

$$\frac{L \Downarrow \mathsf{true} \qquad M \Downarrow c}{(\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N) \Downarrow c} \quad \frac{L \Downarrow \mathsf{false} \qquad N \Downarrow c}{(\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N) \Downarrow c}$$

$$\frac{M_1 \Downarrow c_1 \quad \ldots \quad M_n \Downarrow c_n \qquad M' \equiv M[x_1 := c_1, \ldots, x_n := c_n]}{\begin{array}{c}(\mathsf{fix}\ (\mathsf{store}\ (\lambda f.\, \lambda x.\, M)\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n)) \\ \Downarrow (\lambda x.\, M')[f := (\mathsf{store}\ (\mathsf{fix}\ (\mathsf{store}\ \lambda f.\, \lambda x.\, M')))]\end{array}}$$

4 Semantics

The high-level natural semantics is useful as a specification for an intepreter for our language, and for proving facts like Theorem 2. One would not want to implement the semantics directly, however: explicit substitution into terms can be expensive, and one would therefore use some standard representation of terms like closures or graphs in order to perform substitution more efficiently. But there is another problem with the high-level semantics: it does not go very far in providing a computational intuition for the LL primitives in the language. For example, the dispose operation is treated essentially as 'no-op'. As such, there is no apparent relationship between these connectives and memory; indeed, the semantics entirely suppresses the concept of memory.

In order to understand what the constructs of linear logic have to do with memory, we construct a semantics that relates the LL primitives to reference counting. In this semantics, the linear logic primitives dispose and share maintain reference counts. The basic structure of the reference-counting interpreter is the same as the one outlined in Section 3. Environments, values, and storable objects have the same definition as before. Because we now want to maintain reference counts, however, the definition of stores must change. A **store** is now a function

$$\sigma : \mathsf{Loc} \rightarrow (\mathbf{N} \times \mathsf{Storable}),$$

where the left part of the returned pair denotes a reference count. Abusing notation, we use $\sigma(l)$ to denote the storable object associated with location l, and $\sigma[l \mapsto S]$ to denote a new store which is the same as σ except at location l, which now holds the storable object S with the reference count of l left unaffected. The reference count of a cell is denoted by $\mathsf{refcount}(l, \sigma)$. The **domain** of a store σ is the set

$$\mathsf{dom}(\sigma) = \{l \in \mathsf{Loc} : \mathsf{refcount}(l, \sigma) \geq 1\}.$$

The change in the definition of 'store' forces an adjustment in the definition of 'allocation relation'. A subset R of the product $(\mathsf{Storable} \times \mathsf{Store}) \times (\mathsf{Loc} \times \mathsf{Store})$ is an **allocation relation** if, for any store σ and storable object S, there is an l' and σ' where $(S, \sigma)\ R\ (l', \sigma')$ and

- $l' \notin \mathsf{dom}(\sigma)$ and $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \cup \{l'\}$;
- for all locations $l \in \mathsf{dom}(\sigma)$, $\sigma(l) = \sigma'(l)$ and $\mathsf{refcount}(l, \sigma) = \mathsf{refcount}(l, \sigma')$; and
- $\sigma'(l') = S$ and $\mathsf{refcount}(l', \sigma') = 1$.

The basic structure underlying a store may be captured abstractly by a graph. Formally, a **graph** is a tuple (V, E, s, t) where V and E are sets of **vertices** and **edges** respectively and s, t are functions from E to V called the **source** and **target** functions respectively. (Note that there may be more than one edge with the same source and target; such 'multiple edge' graphs are sometimes called *multigraphs.*) Given $v \in V$, the **in-degree** of v is the number of elements $e \in E$ such that $t(e) = v$. A vertex v is **reachable** from a vertex v' if $v = v'$ or there is a path between them, that is, there is a list of edges $e_1, \ldots, e_n$ such that $v = s(e_1)$, $v' = t(e_n)$ and $t(e_i) = s(e_{i+1})$.

A **memory graph** $\mathcal{G}$ is a tuple $(V, E, s, t, [\rho_1, \ldots, \rho_n])$ where (V, E, s, t) is a graph together with a list of functions ρ_i such that each ρ_i is a function with a finite domain and with V as its codomain. The functions ρ_i are called the **root set** of the memory graph. Given $v \in V$ and ρ_i,

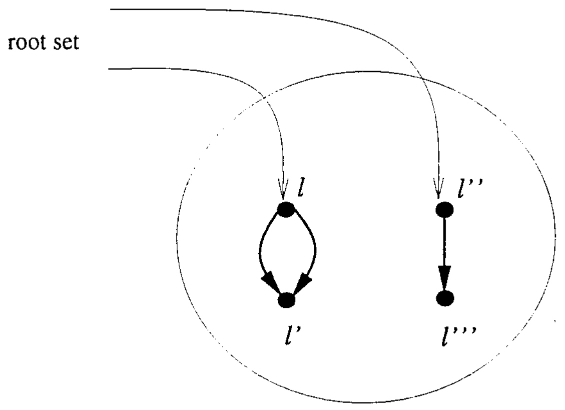


Figure 3: A memory graph.

let $|\rho_i^{-1}(v)|$ be the number of elements x in the domain of ρ_i such that $\rho_i(x) = v$. The **reference count** of a vertex $v \in V$ is the sum

$$\textbf{in-degree}(v) + \sum_{i=1}^{n} |\rho_i^{-1}(v)|.$$

A vertex in a memory graph is said to be reachable from ρ_i if it is reachable from an element in the image of ρ_i.

A **state** is a triple $(\bar{l}, \bar{\rho}, \sigma)$ where $\bar{l}$ is a list of locations, $\bar{\rho}$ is a list of environments and σ is a store. It is assumed that the set of locations in $\bar{l}$ and the image of each environment in $\bar{\rho}$ are contained in $\mathsf{dom}(\sigma)$.

Definition 3 If $S = (\bar{l}, \bar{\rho}, \sigma)$ is a state where $\bar{l} = [l_1, \ldots, l_n]$ and $\bar{\rho} = [\rho_1, \ldots, \rho_m]$, then the **memory graph** $\mathcal{G}(S)$ **induced by** S is defined as follows. The vertices of the graph are the locations in $\mathsf{dom}(\sigma)$, and the edges are determined by the following definition.

- If $l \in \mathsf{dom}(\sigma)$ is such that $\sigma(l) = \mathsf{susp}(l')$ or $\sigma(l) = \mathsf{rec}(l', f)$, there is an edge from l to l'.
- Suppose $l \in \mathsf{dom}(\sigma)$ is such that $\sigma(l) = \mathsf{closure}(N, \rho)$ or $\mathsf{thunk}(N, \rho)$. Then for every $x \in \mathsf{dom}(\rho)$, there is an edge from l to $\rho(x)$.

Let $f : \{1, \ldots, n\} \to V$ be given by $f : i \mapsto l_i$. The root set of the induced memory graph is the list $[f, \rho_1, \ldots, \rho_m]$.

For instance, the state (l, ρ, σ) where $\mathsf{dom}(\rho) = \{x\}$, $\rho(x) = l''$, $\sigma(l) = \mathsf{thunk}(M, [y, z \mapsto l'])$, $\sigma(l') = 3$, $\sigma(l'') = \mathsf{susp}(l''')$, and $\sigma(l''') = \mathsf{true}$ induces the memory graph in Figure 3. We will abuse notation and sometimes write $\mathcal{G}(\sigma)$ for the graph induced by σ alone (with no root set).

We are primarily concerned with states that satisfy a collection of basic invariants.

Definition 4 A state $S = (\bar{l}, \bar{\rho}, \sigma)$ is **count-correct** if, for each $l \in \mathsf{dom}(\sigma)$, $\mathsf{refcount}(l, \sigma)$ is equal to the reference count of l in $\mathcal{G}(S)$.

Definition 5 A state $S = (\bar{l}, \bar{\rho}, \sigma)$ is called **regular**, written $\Re(S)$, provided the following conditions hold:

$\Re1$ S is count-correct.

$\Re2$ $\mathsf{dom}(\sigma)$ is finite.

$\Re3$ For each $l \in \mathsf{dom}(\sigma)$, if $\sigma(l) = \mathsf{thunk}(M, \rho)$, then $\mathsf{refcount}(l, \sigma) = 1$.

$\Re4$ A cycle in the memory graph induced by S arises only in the form of a rec and $\mathsf{recclosure}$ as in Figure 2: that is, it has two nodes l_0 and l_1 such that $\sigma(l_0) = \mathsf{rec}(l_1, f)$ and $\sigma(l_1) = \mathsf{recclosure}(\lambda x.\, M, \rho[f \mapsto l_0])$ for some f, x, M, and ρ.

$\Re5$ For each $l \in \mathsf{dom}(\sigma)$, if $\sigma(l) = \mathsf{thunk}(M, \rho)$, then the domain of ρ is the set of free variables of M, and M is typeable. Similarly, if $\sigma(l) = \mathsf{closure}(\lambda x.\, M, \rho)$ or $\mathsf{recclosure}(\lambda x.\, M, \rho)$, then the domain of ρ is the set of free variables of $\lambda x.\, M$, and $\lambda x.\, M$ is typeable.

Here, a term M is said to be **typeable** if there is some type context Γ and type t such that $\Gamma \vdash M : t$.

It is convenient to abuse notation slightly in denoting states by writing locations, environments, and store without grouping them as in the official definition. For example, $(l_1, l_2, \rho, \sigma, \bar{l}, \bar{\rho})$ should be read as $(l_1 :: l_2 :: \bar{l}, \rho :: \bar{\rho}, \sigma)$ (where :: is the 'cons' operation that puts a datum at the head of a list). There is no chance of confusion so long as the lexical conventions distinguish the parts of the tuple, and the locations and environments are properly ordered from left to right. However, the order of these lists is irrelevant for regularity: if $\Re(\bar{l}, \bar{\rho}, \sigma)$ and $\bar{l}', \bar{\rho}'$ are permutations of $\bar{l}$ and $\bar{\rho}$ respectively, then $\Re(\bar{l}', \bar{\rho}', \sigma)$. We will use this fact without explicit mention.

Basic reference-counting operations.

Our interpreter will need four auxiliary functions to manipulate reference counts. Two of these functions, inc and dec, increment and decrement reference counts. More formally, $\mathsf{inc}(l, \sigma)$ increments the reference count of l and returns the resultant store, while $\mathsf{dec}(l, \sigma)$ decrements the reference count of l and returns the resultant store. The other two operations, $\mathsf{inc\text{-}env}(\rho, \sigma)$ and $\mathsf{dec\text{-}ptrs}(l, \sigma)$, increment or decrement the reference counts of multiple cells. The formal definition of the first of these is

$$\mathsf{inc\text{-}env}(\rho, \sigma) = \begin{cases} \sigma_n, & \text{where the domain of } \rho \text{ is } \{x_1, \ldots, x_n\}, \text{ and} \\ & \sigma_1 = \mathsf{inc}(\rho(x_1), \sigma) \\ & \vdots \\ & \sigma_n = \mathsf{inc}(\rho(x_n), \sigma_{n-1}) \end{cases}$$

In words, $\mathsf{inc\text{-}env}(\rho, \sigma)$ increments the reference counts of the locations in the range of ρ and returns the resultant store. Note that a location's reference count may be incremented more than once by this operation, since two variables x_i, x_j may map to the same location l according to ρ.

The operation $\mathsf{dec\text{-}ptrs}(l, \sigma)$, which also returns an updated store, first decrements the reference count of location l. If the reference count falls to zero, it then recursively decrements the reference counts of all cells pointed to by l. The formal definition appears in Table 6; an example appears in Figure 4 where the left side of Figure 4 (assumed to be part of the graph of the store σ) is transformed into the right side by calling $\mathsf{dec\text{-}ptrs}(l, \sigma)$. The operation $\mathsf{dec\text{-}ptrs}(l, \sigma)$ is the single

Table 6: The Definition of dec-ptrs.

$$\mathsf{dec\text{-}ptrs}(l,\sigma) = \begin{cases} \mathsf{dec}(l,\sigma) & \text{if } \sigma(l) = n, \mathsf{true}, \text{ or } \mathsf{false} \\ \mathsf{dec\text{-}ptrs}(l', \mathsf{dec}(l,\sigma)) & \text{if } \sigma(l) = \mathsf{susp}(l'), \mathsf{refcount}(l,\sigma) = 1 \\ \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \mathsf{dec}(l,\sigma)) & \text{if } \sigma(l) = \mathsf{thunk}(M,\rho), \mathsf{refcount}(l,\sigma) = 1 \\ \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \mathsf{dec}(l,\sigma)) & \text{if } \sigma(l) = \mathsf{closure}(\lambda x.\, M,\rho), \mathsf{refcount}(l,\sigma) = 1 \\ \mathsf{dec\text{-}ptrs\text{-}env}(\rho, & \text{if } \sigma(l) = \mathsf{recclosure}(\lambda x.\, N, \rho[f \mapsto l']), \\ \qquad \mathsf{dec}(l', \mathsf{dec}(l, \mathsf{dec}(l,\sigma)))) & \quad \sigma(l') = \mathsf{rec}(l,f), \\ & \quad \mathsf{refcount}(l,\sigma) = 2, \mathsf{refcount}(l',\sigma) = 1 \\ \mathsf{dec\text{-}ptrs\text{-}env}(\rho, & \text{if } \sigma(l) = \mathsf{rec}(l',f), \\ \qquad \mathsf{dec}(l', \mathsf{dec}(l, \mathsf{dec}(l,\sigma)))) & \quad \sigma(l') = \mathsf{recclosure}(\lambda x.\, N, \rho[f \mapsto l]), \\ & \quad \mathsf{refcount}(l,\sigma) = 2, \mathsf{refcount}(l',\sigma) = 1 \\ \mathsf{dec}(l,\sigma) & \text{otherwise} \end{cases}$$

$$\mathsf{dec\text{-}ptrs\text{-}env}(\rho,\sigma) = \begin{cases} \sigma_n, & \text{where the domain of } \rho \text{ is } \{x_1, \ldots, x_n\}, \text{ and} \\ & \sigma_1 = \mathsf{dec\text{-}ptrs}(\rho(x_1), \sigma) \\ & \vdots \\ & \sigma_n = \mathsf{dec\text{-}ptrs}(\rho(x_n), \sigma_{n-1}) \end{cases}$$

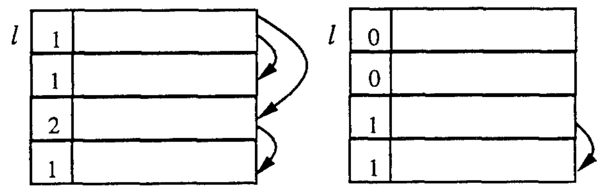


Figure 4: An Example of the dec-ptrs Operation.

most complex operation used in the interpreter. Other operations are 'local' to parts of the memory graph and do not require a recursive definition. A key characteristic of our semantics is the fact that $\mathsf{dec\text{-}ptrs}(l,\sigma)$ is only used in the rule for evaluating ($\mathsf{dispose}\ M\ \mathsf{before}\ N$).

The basic laws that capture the relationships maintained by the reference-counting, allocation, and update operations on states are given in Table 7. Most of the laws are proven in the appendix, but we give the proof for the Attenuation Law A1 here to show how the proofs go. Suppose $\Re(l,\bar{l},\bar{\rho},\sigma)$, $\mathsf{refcount}(l,\sigma) = 1$ and $\sigma(l) = \mathsf{closure}(\lambda x.\,N,\rho)$, $\mathsf{recclosure}(\lambda x.\,N,\rho)$, or $\mathsf{thunk}(N,\rho)$. Note first that the state $S' = (\bar{l},\rho,\bar{\rho},\mathsf{dec}(l,\sigma))$ is count-correct: the environment ρ has been placed in the root set, accounting for the edges coming out of the closure or thunk which has now disappeared from the memory graph. Thus, property $\Re 1$ holds of state S'. Since $\mathsf{dom}(\sigma) \supseteq \mathsf{dom}(\mathsf{dec}(l,\sigma))$, each of the properties $\Re 2$–$\Re 5$ follow directly from the hypothesis. Thus, $\Re(S')$. The property is called an "attenuation law" because pointers previously held inside the store are drawn out to the root set.

The next goal is to define an interpreter for the LL-based programming language. To understand the interpreter it is essential to appreciate how the invariants influence its design. We therefore describe the theorem that the interpreter is expected to satisfy, and mingle the proof of the theorem with the definition of the interpreter itself. The interpreter is a function **interp** which takes as its arguments a term M, an environment ρ, and a store σ. It is assumed that the domain of ρ is the set of free variables in M and that the image of ρ is contained in the domain of σ. The result of $\mathtt{interp}(M,\rho,\sigma)$ is a pair (l',σ') where σ' is a store and l' is a location in the domain of σ' such that $\sigma'(l')$ is a value, which can be viewed as the result of the computation. We use a binary infix @ for appending two lists. The theorem is stated as follows:

Theorem 6 *Let $S = (\rho,\sigma,\bar{l},\bar{\rho})$ be a state and suppose M is a typeable term. If $\Re(S)$ and* $\mathtt{interp}(M,\rho,\sigma) = (l',\sigma')$, *then $\Re(l',\sigma',\bar{l},\bar{\rho})$.*

Moreover, if $\bar{\rho} = \bar{\rho}_1 @ \bar{\rho}_2$, $\bar{l} = \bar{l}_1 @ \bar{l}_2$ and $l \in \mathsf{dom}(\sigma)$ is not reachable from $\rho :: \bar{\rho}_1$ or $\bar{l}_1$ in the memory graph induced by S, then the contents and reference count of l remain unchanged and l is not reachable from $\bar{\rho}_1$ or $l' :: \bar{l}_1$ in the memory graph induced by $(l',\sigma',\bar{l},\bar{\rho})$.

The first part of the theorem says that regularity is preserved under execution of typeable terms. The second part of the theorem expresses what we will call the **reachability property.** The special case of interest says that the evaluation of a program M in environment ρ and store σ does not affect locations in $\mathsf{dom}(\sigma)$ that are not reachable from ρ. The extra complexity of the statement is required to maintain a usable induction hypothesis in the proof of the property. A simplified version of Theorem 6 can be expressed as follows:

Corollary 7 *Suppose M is a closed, typeable term. If* $\mathtt{interp}(M,\emptyset,\emptyset) = (l',\sigma')$, *then $\Re(l',\sigma')$.*

The assumption that M is typeable is crucial in the proof of the theorem, because untypeable terms may not maintain reference counts correctly. For instance, the term

$$(\lambda x.\,(\mathsf{dispose}\ x\ \mathsf{before}\ x))\ (\mathsf{store}\ 1)$$

would cause a run-time error in the maintenance of reference counts—after the dispose, we would try to access a portion of memory with reference count zero and get a 'dangling pointer' error. This example shows that untypeable terms may cause premature deallocations. Another untypeable term

$$(\lambda x.\,(\mathsf{share}\ y,z\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{dispose}\ y\ \mathsf{before}\ 2)))\ (\mathsf{store}\ 1)$$

Table 7: Memory Graph Laws.

Attenuation Laws Suppose $\Re(l, \bar{l}, \bar{\rho}, \sigma)$ and $\mathsf{refcount}(l, \sigma) = 1$.

A1 If $\sigma(l) = \mathsf{closure}(\lambda x.\, N, \rho)$, $\mathsf{recclosure}(\lambda x.\, N, \rho)$, or $\mathsf{thunk}(N, \rho)$, then $\Re(\bar{l}, \rho, \bar{\rho}, \mathsf{dec}(l, \sigma))$.

A2 If $\sigma(l) = \mathsf{susp}(l')$, then $\Re(l', \bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$.

Laws of Decrement

D1 If $\Re(l, \bar{l}, \bar{\rho}, \sigma)$ and $\sigma(l)$ is a constant, then $\Re(\bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$.

D2 If $\Re(l, \bar{l}, \bar{\rho}, \sigma)$ and $\mathsf{refcount}(l, \sigma) \neq 1$, then $\Re(\bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$.

D3 If $\Re(l, \bar{l}, \bar{\rho}, \sigma)$, then $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.

Laws of Increment

I1 If $\Re(\bar{l}, \bar{\rho}, \sigma)$ and $l \in \mathsf{dom}(\sigma)$, then $\Re(l, \bar{l}, \bar{\rho}, \mathsf{inc}(l, \sigma))$.

I2 Suppose $\Re(\bar{l}, \bar{\rho}, \sigma)$ and $\rho(x) \in \mathsf{dom}(\sigma)$ for all $x \in \mathsf{dom}(\rho)$. Then $\Re(\bar{l}, \rho, \bar{\rho}, \mathsf{inc\text{-}env}(\rho, \sigma))$.

Environment Law

E Suppose $x \notin \mathsf{dom}(\rho)$. Then $\Re(l, \bar{l}, \rho, \bar{\rho}, \sigma)$ iff $\Re(\bar{l}, \rho[x \mapsto l], \bar{\rho}, \sigma)$.

Allocation Laws

N1 If $\Re(\bar{l}, \bar{\rho}, \sigma)$ and $(l', \sigma') = \mathsf{new}(c, \sigma)$ for some constant c, then $\Re(l', \bar{l}, \bar{\rho}, \sigma')$.

N2 Suppose $\Re(\bar{l}, \rho, \bar{\rho}, \sigma)$ and (l', σ') is equal to $\mathsf{new}(\mathsf{closure}(N, \rho), \sigma)$, $\mathsf{new}(\mathsf{thunk}(N, \rho), \sigma)$, or $\mathsf{new}(\mathsf{recclosure}(N, \rho), \sigma)$ where $FV(N) = \mathsf{dom}(\rho)$. If N is typeable, then $\Re(l', \bar{l}, \bar{\rho}, \sigma')$.

N3 If $\Re(l, \bar{l}, \bar{\rho}, \sigma)$ and $(l', \sigma') = \mathsf{new}(\mathsf{susp}(l), \sigma)$ or $\mathsf{new}(\mathsf{rec}(l, f), \sigma)$, then $\Re(l', \bar{l}, \bar{\rho}, \sigma')$.

Update Laws

U1 Suppose $S = (\bar{l}, \bar{\rho}, \sigma)$ and $\Re(S)$ and $\sigma(l)$ is a constant and $l' \in \mathsf{dom}(\sigma)$.

- If l is not reachable from l' in the memory graph induced by S, then $\Re(l', \bar{l}, \bar{\rho}, \mathsf{inc}(l', \sigma[l \mapsto \mathsf{susp}(l')]))$.
- If $\sigma(l') = \mathsf{recclosure}(\lambda x.\, N, \rho[f \mapsto l])$, then $\Re(l', \bar{l}, \bar{\rho}, \mathsf{inc}(l', \sigma[l \mapsto \mathsf{rec}(l', f)]))$.

U2 If $\Re(l, \bar{l}, \bar{\rho}, \sigma)$, $\mathsf{refcount}(l, \sigma) \neq 1$, $\sigma(l) = \mathsf{susp}(l')$, and $\sigma(l') = \mathsf{thunk}(N, \rho)$, then $\Re(\rho, \bar{l}, \bar{\rho}, \mathsf{dec}(l', \mathsf{dec}(l, \sigma[l \mapsto c])))$.

causes a 'space leak', *i.e.*, the reference count of the cell holding (store 1) is still greater than zero even though it is garbage at the end of the execution.

Interpreting the linear core.

The proof of Theorem 6 is by induction on the number of calls to the interpreter. The proof proceeds by considering each case for the program to be evaluated.

The interpretation of a variable is obtained by looking up the variable in the environment:

(1) $\texttt{interp}(x,\ \rho,\ \sigma) = (\rho(x),\ \sigma)$

That the store $(\rho(x), \sigma', \bar{l}, \bar{\rho})$ is regular is a consequence of the Environment Law E because of the assumption that the domain of ρ is $\{x\}$. The reachability condition is clearly satisfied, since the output store is the same as the input store.

To evaluate an abstraction we create a new closure, place it in a new cell, and return the location together with the updated store:

(2) $\texttt{interp}(\lambda x.\, P,\ \rho,\ \sigma) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P,\ \rho),\ \sigma)$

To prove that regularity of the state is preserved, suppose that $(l', \sigma') = \mathsf{new}(\mathsf{closure}(\lambda x.\, P, \rho), \sigma)$, then $\Re(l', \sigma', \bar{l}, \bar{\rho})$ by Allocation Law N2. The reachability condition is satisfied because the output store differs from the input store only by extending it.

Given a term P and an environment ρ whose domain includes the free variables of P, let $\rho \,|\, P$ be the restriction of the environment ρ to the free variables of P. The evaluation of an application is given as follows:

```
(3)   interp((P Q), ρ, σ) =
        let (l_0, σ_0) = interp(P, ρ|P, σ)
            (l_1, σ_1) = interp(Q, ρ|Q, σ_0)
        in  case σ_1(l_0) of closure(λx. N, ρ') or recclosure(λx. N, ρ') =>
              if refcount(l_0, σ_1) = 1
              then interp(N, ρ'[x ↦ l_1], dec(l_0, σ_1))
              else interp(N, ρ'[x ↦ l_1], inc-env(ρ', dec(l_0, σ_1)))
```

The reader may compare this rule to the rule for application given in Section 3. The key difference in the semantic clauses is the manipulation of reference counts: in the rule here, a conditional breaks the evaluation of the function body into two cases based on the reference count of the location that holds the value of the operator, and each branch of the conditional performs some reference-counting arithmetic. The resulting semantics clause looks similar to a denotational semantics such as that given in [Hud87] where information about reference counts is included in the semantics clauses. Note that the environment ρ has been split between the two subterms P and Q. The fact that $(P\ Q)$ is typeable implies that $\rho = (\rho \,|\, P) \cup (\rho \,|\, Q)$. In various forms this sort of property will be used repeatedly in the semantic clauses below.

To prove the preservation of regularity of the state for application, we start with the assumption that $\Re(\rho, \sigma, \bar{l}, \bar{\rho})$. This is equivalent to $\Re(\rho \,|\, P, \rho \,|\, Q, \sigma, \bar{l}, \bar{\rho})$. Now $\Re(l_0, \rho \,|\, Q, \sigma_0, \bar{l}, \bar{\rho})$ and $\Re(l_1, l_0, \sigma_1, \bar{l}, \bar{\rho})$ both hold by induction hypothesis (let us abbreviate 'induction hypothesis' as 'IH'). Now, there are two possibilities for the reference count of l_0 in σ_1, either it is equal to one or it is more than one. If $\mathsf{refcount}(l_0, \sigma_1) = 1$, then the first Attenuation Law, A1, says that $\Re(l_1, \rho', \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{\rho})$. By the Environment Law, E, this implies that $\Re(\rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{\rho})$

and the desired conclusion then follows from IH. If, on the other hand, $\mathsf{refcount}(l_0, \sigma_1) \neq 1$, then $\Re(l_1, \rho', \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l, \sigma)), \bar{l}, \bar{\rho})$ by I2 and D2. Hence $\Re(\rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l, \sigma)), \bar{l}, \bar{\rho})$ by E, so we are done by IH.

To see that the reachability property holds for the interpretation of application, suppose $l \in \mathsf{dom}(\sigma)$ is unreachable from $\rho :: \bar{\rho}_1$ where $\bar{\rho} = \bar{\rho}_1 @ \bar{\rho}_2$ and unreachable from $\bar{l}_1$ where $\bar{l} = \bar{l}_1 @ \bar{l}_2$. If l is unreachable from $(\bar{l}_1, \rho :: \bar{\rho}_1)$, then it is unreachable from $(\bar{l}_1, (\rho \,|\, P) :: (\rho \,|\, Q) :: \bar{\rho}_1)$, so, by IH, it is unreachable from $(l_0 :: \bar{l}_1, (\rho \,|\, Q) :: \bar{\rho}_1)$ in the memory graph induced by the state resulting from the evaluation of P. A second application of IH allows us to conclude that it is also unreachable from $(l_1 :: l_0 :: \bar{l}_1, \bar{\rho}_1)$ in the memory graph induced by $(l_1, l_0, \sigma_1, \bar{l}, \bar{\rho})$. By the definition of the memory graph, this implies that l is unreachable from ρ' as well, so it is unreachable from $(\bar{l}_1, \rho'[x \mapsto l_1], \bar{\rho}_1)$ in the memory graphs induced by the states $(\rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{\rho})$ and $(\rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l, \sigma)), \bar{l}, \bar{\rho})$. The desired conclusion therefore follows from IH. The proof of the reachability is similar for all of the remaining cases, so we will omit arguing it in the rest of the discussion.

The expression (store N where $x_1 = M_1, \ldots, x_n = M_n$) is interpreted by first evaluating the terms $M_1, \ldots, M_n$ to locations $l_1, \ldots, l_n$, building an environment that maps x_i to l_i for all i, creating a thunk out of this environment and N, and finally returning a location holding a suspension of this thunk:

```
(4)    interp((store $N$ where $x_1 = M_1, \ldots, x_n = M_n$),  $\rho$,  $\sigma$) =
         let ($l_1$,  $\sigma_1$) = interp($M_1$,  $\rho \,|\, M_1$,  $\sigma$)
             ...
             ($l_n$,  $\sigma_n$) = interp($M_n$,  $\rho \,|\, M_n$,  $\sigma_{n-1}$)
             $\rho'$  =  $[x_1, \ldots, x_n \mapsto l_1, \ldots, l_n]$
             ($l_{n+1}$,  $\sigma_{n+1}$) = new(thunk($N$,  $\rho'$),  $\sigma_n$)
         in  new(susp($l_{n+1}$),  $\sigma_{n+1}$)
```

To prove that the desired property is maintained, note that repeated application of the induction hypothesis allows us to conclude that $\Re(\rho', \sigma_n, \bar{\rho}, \bar{l})$. Therefore, $\Re(l_{n+1}, \sigma_{n+1}, \bar{\rho}, \bar{l})$ by N2. Let $(l_{n+2}, \sigma_{n+2}) = \mathsf{new}(\mathsf{susp}(l_{n+1}), \sigma_{n+1})$. Then $\Re(l_{n+2}, \sigma_{n+2}, \bar{\rho}, \bar{l})$ by N3.

The fetch of a suspended object is the most complex of all the operations. It must evaluate a thunk if the suspension holds one. The code is again similar to that for the interpreter in Section 3 we examined earlier, but, in addition to the reference-counting arithmetic, there is a clause dealing with recursion:

```
(5)    interp((fetch $P$),  $\rho$,  $\sigma$) =
         let ($l_0$,  $\sigma_0$) = interp($P$,  $\rho$,  $\sigma$)
         in  case $\sigma_0(l_0)$
               of susp($l_1$) =>
                    case $\sigma_0(l_1)$
                      of thunk($R$,  $\rho'$) =>
                           if refcount($l_0$,  $\sigma_0$) = 1
                           then interp($R$,  $\rho'$,  dec($l_1$,  dec($l_0$,  $\sigma_0$)))
                           else let ($l_2$,  $\sigma_1$) = interp($R$,  $\rho'$,  dec($l_1$,  dec($l_0$,  $\sigma_0[l_0 \mapsto 0]$)))
                                in  ($l_2$,  inc($l_2$,  $\sigma_1[l_0 \mapsto$ susp($l_2$)$]$)))
                       _ => if refcount($l_0$,  $\sigma_0$) = 1
                              then ($l_1$,  dec($l_0$,  $\sigma_0$))
                              else ($l_1$,  inc($l_1$,  dec($l_0$,  $\sigma_0$)))
                | rec($l_1$,  $f$) => ($l_1$,  dec($l_0$,  inc($l_1$,  $\sigma_0$)))
```

By IH, we have $\Re(l_0, \sigma_0, \bar{l}, \bar{\rho})$. Suppose $\sigma_0(l_0) = \mathsf{susp}(l_1)$ and $\sigma_0(l_1) = \mathsf{thunk}(R, \rho')$. If $\mathsf{refcount}(l_0, \sigma_0) = 1$, then $\Re(\rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0)), \bar{l}, \bar{\rho})$ by A1 and A2 so we are done by IH. Suppose, on the other hand, that $\mathsf{refcount}(l_0, \sigma_0) \neq 1$. By U2, $\Re(\rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])), \bar{l}, \bar{\rho})$ so $\Re(l_2, \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]), \bar{l}, \bar{\rho})$ by IH and U1; the reachability property is used to ensure the applicability of U1. More specifically, in σ_0 the location l_0 is not reachable from ρ'; thus, it is not reachable from l_2 in σ_1 either, and so $\sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]$ does not create an illegal loop in the memory graph. The cases when $\sigma_0(l_1)$ is a value or $\sigma_0(l_0) = \mathsf{rec}(l_1, f)$ are left to the reader.

The share command increments the reference count of a location:

```
(6)    interp((share $x, y$ as $P$ in $Q$),  $\rho$,  $\sigma$) =
         let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho \mid P$,  $\sigma$)
         in   interp($Q$,  $(\rho \mid Q)[x, y \mapsto l_0]$,  inc($l_0$,  $\sigma_0$))
```

$\Re(l_0, \rho \mid Q, \sigma_0, \bar{l}, \bar{\rho})$ by IH, so $\Re(l_0, l_0, \rho \mid Q, \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$ by I1. Thus it follows from the Environment Law E that $\Re((\rho \mid Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$, so the result follows from IH.

The dispose command decrements the reference count of a location. The requires calculating the consequences of possibly removing a node from the memory graph if its reference count of the disposed node falls to 0.

```
(7)    interp((dispose $P$ before $Q$),  $\rho$,  $\sigma$) =
         let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho \mid P$,  $\sigma$)
         in   interp($Q$,  $\rho \mid Q$,  dec-ptrs($l_0$,  $\sigma_0$))
```

Now, $\Re(l_0, \rho \mid Q, \sigma_0, \bar{l}, \bar{\rho})$ by IH, so $\Re(\rho \mid Q, \mathsf{dec\text{-}ptrs}(l_0, \sigma_0), \bar{l}, \bar{\rho})$ by D3. The result therefore follows from IH.

Interpreting PCF extensions.

The interpreter evaluates a constant simply by creating a cell holding the value of the constant.

```
(8)    interp($n, \rho, \sigma$) = new($n, \sigma$)

(9)    interp(true, $\rho, \sigma$) = new(true, $\sigma$)

(10)   interp(false, $\rho, \sigma$) = new(false, $\sigma$)
```

That regularity is preserved for these cases follows immediately from N1.

The rules for the arithmetic and boolean operations of PCF mimic the rules of the high-level operational semantics.

```
(11)   interp((succ $P$),  $\rho$,  $\sigma$) =
         let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho$,  $\sigma$)
         in   new($\sigma_0(l_0) + 1$,  dec($l_0$,  $\sigma_0$))

(12)   interp((pred $P$),  $\rho$,  $\sigma$) =
         let  ($l_0$,  $\sigma_0$) = interp($P$,  $\rho$,  $\sigma$)
              $n = \sigma_0(l_0)$
         in   if $n = 0$
              then new(0,  dec($l_0$,  $\sigma_0$))
              else new($n - 1$,  dec($l_0$,  $\sigma_0$))
```

```
(13)  interp((zero? $P$), $\rho$, $\sigma$) =
        let ($l_0$, $\sigma_0$) = interp($P$, $\rho$, $\sigma$)
        in if $\sigma_0(l_0) = 0$
           then new(true, dec($l_0$, $\sigma_0$))
           else new(false, dec($l_0$, $\sigma_0$))
```

To prove the desired property for the successor operation, note that $\Re(l_0, \sigma_0, \bar{l}, \bar{\rho})$ follows from IH so we are done by D1 and N1. Proofs for the other two cases are similar.

The conditional statement has the expected form, but the reference count of the condition must be decremented in each of the branches:

```
(14)  interp(if $N$ then $P$ else $Q$, $\rho$, $\sigma$) =
        let ($l_0$, $\sigma_0$) = interp($N$, $\rho\,|\,N$, $\sigma$)
        in  if $\sigma_0(l_0)$ = true
            then interp($P$, $\rho\,|\,P$, dec($l_0$, $\sigma_0$))
            else interp($Q$, $\rho\,|\,Q$, dec($l_0$, $\sigma_0$))
```

The IH implies $\Re(l_0, \sigma_0, \bar{l}, \bar{\rho})$. Whether or not $\sigma_0(l_0) =$ true, the desired conclusion follows from D1.

Finally, to interpret recursion, we will need a rule similar to the rule for interpreting store.

```
(15)  interp((fix (store ($\lambda f.\,\lambda x.\,M$) where $x_1 = M_1, \ldots, x_n = M_n$)), $\rho$, $\sigma$) =
        let ($l_1$, $\sigma_1$) = interp($M_1$, $\rho\,|\,M_1$, $\sigma$)
            ...
            ($l_n$, $\sigma_n$) = interp($M_n$, $\rho\,|\,M_n$, $\sigma_{n-1}$)
            $\rho'$ = $[x_1, \ldots, x_n \mapsto l_1, \ldots, l_n]$
            ($l_{n+1}$, $\sigma_{n+1}$) = new(0, $\sigma_n$)
            ($l_{n+2}$, $\sigma_{n+2}$) = new(recclosure($\lambda x.\,M$, $\rho'[f \mapsto l_{n+1}]$), $\sigma_{n+1}$)
        in  ($l_{n+2}$, inc($l_{n+2}$, $\sigma_{n+2}[l_{n+1} \mapsto$ rec($l_{n+2}$, $f$)]))
```

As with the interpretation of store, repeated application of the IH and E implies that $\Re(\rho', \sigma_n, \bar{l}, \bar{\rho})$. By N1, E, and N2, we therefore also have $\Re(l_{n+2}, \sigma_{n+2}, \bar{l}, \bar{\rho})$. The desired conclusion now follows from U1.

5 Properties of the Semantics

In order for the reference-counting interpreter to make sense, it must satisfy a number of invariants and correctness criteria. In this section we describe these precisely.

No space leaks.

As a short example of the kind of property one expects the semantics to satisfy, let us consider how the idea that 'there are no space leaks' can be expressed in our formalism. Given a state $S = (\bar{l}, \bar{\rho}, \sigma)$, we say that a location l is reachable from $(\bar{l}, \bar{\rho})$ if it is reachable in $\mathcal{G}(S)$ from some $l_i \in \bar{l}$ or from some $\rho_j \in \bar{\rho}$. The desired property can now be expressed as follows:

Theorem 8 *Suppose $(\rho, \sigma, \bar{l}, \bar{\rho})$ is a regular state such that each $l \in \mathsf{dom}(\sigma)$ is reachable from $(\rho, \bar{l}, \bar{\rho})$. If M is typeable and $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$, then every $l \in \mathsf{dom}(\sigma')$ is reachable from $(l', \bar{l}, \bar{\rho})$.*

The theorem is proved by induction on the number of calls to the interpreter.

Invariance under different allocation relations.

If the design of the interpreter is correct, the exact memory usage pattern should be unimportant to the final answers returned by the interpreter. Since the allocation relation new completely determines memory usage—*i.e.*, which cell (with reference count 0) will be filled next—it should not matter which allocation relation is used. We set this up formally as follows: if f is an allocation relation, let $\mathtt{interp}_f$ be the partial interpreter function defined by using f in the place of new. Recall that the environment and store with empty domains are denoted by $\emptyset$. Then we would like to prove something like the following statement by induction on the number of calls to $\mathtt{interp}_f$:

> Suppose f and g are allocation relations. If $\mathtt{interp}_f(M, \emptyset, \emptyset) = (l_f, \sigma_f)$, then $\mathtt{interp}_g(M, \emptyset, \emptyset) = (l_g, \sigma_g)$. Moreover, if $\sigma_f(l_f) = n$, true, or false, then $\sigma_f(l_f) = \sigma_g(l_g)$.

A naive induction runs afoul, though, since the interpreter can return intermediate results that are neither numbers nor booleans. We therefore need to strengthen the induction hypothesis. If $\mathtt{interp}_f$ returns a closure or suspension, the result returned by $\mathtt{interp}_g$ may not literally be the same: for instance, $\mathtt{interp}_f$ may return a location holding $\mathsf{susp}(l_0)$ and $\mathtt{interp}_g$ may return a location holding $\mathsf{susp}(l_1)$. Nevertheless, these values should be the same up to a renaming of the locations in the domain of the returned store σ'_f.

Formalizing the notion of when two stores are 'equivalent' up to renaming of their locations can be done using the underlying graphs. Two stores are 'equivalent' if their underlying graph representations are isomorphic via some function h, and the values held at the cells are 'equivalent' under h. More formally,

Definition 9 Two states $S = (\bar{l}, \bar{\rho}, \sigma)$ and $S' = (\bar{l}', \bar{\rho}', \sigma')$ are **congruent** if there is an isomorphism $h : \mathcal{G}(\sigma) \to \mathcal{G}(\sigma')$ such that for any $l \in \mathsf{dom}(\sigma)$, $\mathsf{refcount}(l, \sigma) = \mathsf{refcount}(h(l), \sigma')$ and for any $l \in \mathsf{dom}(\sigma)$,

1. For all i, $h(l_i) = l'_i$;
2. For all i, $\mathsf{dom}(\rho_i) = \mathsf{dom}(\rho'_i)$ and for all $x \in \mathsf{dom}(\rho_i)$, $h(\rho_i(x)) = \rho'_i(x)$;

3. If $\sigma(l) = n$, true, or false, then $\sigma(l) = \sigma'(h(l))$;
4. If $\sigma(l) = \mathsf{susp}(l')$, then $\sigma'(h(l)) = \mathsf{susp}(h(l'))$;
5. If $\sigma(l) = \mathsf{rec}(l', f)$, then $\sigma'(h(l)) = \mathsf{rec}(h(l'), f)$;
6. If $\sigma(l) = \mathsf{closure}(\lambda x.\, P, \rho)$, then $\sigma'(h(l)) = \mathsf{closure}(\lambda x.\, P, \rho')$, $\mathsf{dom}(\rho) = \mathsf{dom}(\rho')$, and for any $x \in \mathsf{dom}(\rho)$, $\rho'(x) = h(\rho(x))$;
7. If $\sigma(l) = \mathsf{recclosure}(\lambda x.\, P, \rho)$, then $\sigma'(h(l)) = \mathsf{recclosure}(\lambda x.\, P, \rho')$, $\mathsf{dom}(\rho) = \mathsf{dom}(\rho')$, and for any $x \in \mathsf{dom}(\rho)$, $\rho'(x) = h(\rho(x))$; and
8. If $\sigma(l) = \mathsf{thunk}(P, \rho)$, then $\sigma'(h(l)) = \mathsf{thunk}(P, \rho')$, $\mathsf{dom}(\rho) = \mathsf{dom}(\rho')$, and for any $x \in \mathsf{dom}(\rho)$, $\rho'(x) = h(\rho(x))$.

Then one may prove

Lemma 10 *Suppose* $(\bar{l}', \rho_f, \bar{\rho}', \sigma_f)$ *and* $(\bar{l}'', \rho_g, \bar{\rho}'', \sigma_g)$ *are congruent. If* $\mathtt{interp}_f(M, \rho_f, \sigma_f) = (l'_f, \sigma'_f)$, *then* $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l'_g, \sigma'_g)$ *and the resultant states* $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ *and* $(l'_g, \bar{l}'', \bar{\rho}'', \sigma'_g)$ *are congruent.*

The proof is deferred to the appendix. From this lemma, the following theorem follows directly:

Theorem 11 *Suppose* f *and* g *are allocation relations. If* $\mathtt{interp}_f(M, \emptyset, \emptyset) = (l_f, \sigma_f)$, *then* $\mathtt{interp}_g(M, \emptyset, \emptyset) = (l_g, \sigma_g)$. *Moreover, if* $\sigma_f(l_f) = n$, true, *or* false, *then* $\sigma_f(l_f) = \sigma_g(l_g)$.

Correctness of the interpreter.

Finally, we need to verify that the reference-counting semantics implements the natural semantics of Tables 4 and 5, *i.e.*, evaluating a closed term of base type yields the same result in either semantics. The proof proceeds by induction on the number of steps in the evaluation (the height of the proof tree for the $(\Rightarrow)$ direction, and the number of calls to $\mathtt{interp}$ for the $(\Leftarrow)$ direction). We again need an expanded induction hypothesis to carry out the proof, one in which we can relate the values held in memory locations to terms. To this end, we define the extraction functions $\mathsf{valof}(M, \rho, \sigma)$ and $\mathsf{valofcell}(l, \sigma)$. Intuitively, the function valofcell extracts a term from the storable value held at location l in store σ, and the function valof replaces the free variables of M with the extracted versions of the cells bound to the free variables according to ρ. The idea is easy to understand intuitively from an example. Suppose, for instance, cell l_0 holds $\mathsf{thunk}((\mathsf{dispose}\ x\ \mathsf{before}\ y), [x \mapsto l_1, y \mapsto l_2])$, l_1 holds $\mathsf{susp}(l_3)$, l_3 holds 0, and l_2 holds true in the store σ. Then $\mathsf{valofcell}(l_0, \sigma) = (\mathsf{dispose}\ ((\mathsf{store}\ 0))\ \mathsf{before}\ \mathsf{true})$. A larger example appears in Figure 5 (where reference counts have been ignored); if σ is the store depicted there, then

$$\begin{aligned}&\mathsf{valofcell}(l, \sigma) = \\ &\quad \lambda f.\, (((\lambda h.\, \lambda y.\, (\mathsf{share}\ h_1, h_2\ \mathsf{as}\ h\ \mathsf{in}\ h_1(h_2\ y)))\ f)\ (\mathsf{store}\ ((\lambda x.\, x)\ \mathsf{true})))\end{aligned}$$

Formal defintions for valof and valofcell are given by simultaneous induction in Table 8 in the appendix. A similar definition is given in [Plo75] for unwinding a closure relative to an SECD machine state.

Since we will be interpreting terms of arbitrary type, the induction hypothesis must relate values returned by the natural semantics to values returned by the reference-counting interpreter. The

l_g	$\mathsf{closure}(\lambda h.\, \lambda y.\, (\mathsf{share}\ h_1, h_2\ \mathsf{as}\ h\ \mathsf{in}\ h_1(h_2\ y)),\ \emptyset)$
l	$\mathsf{closure}(\lambda f.\, ((g\ f)\ x),\ [g \mapsto l_g,\ x \mapsto l_x])$
l_x	$\mathsf{susp}(l'_x)$
l'_x	$\mathsf{thunk}((f\ \mathsf{true}),\ [f \mapsto l'])$
l'	$\mathsf{closure}(\lambda x.\, x,\ \emptyset)$

Figure 5: Store for Example of the valofcell Operation.

key definition missing here is the definition of 'related values'. One might attempt to extend the statement of the theorem directly—that is, for closed terms, $M \Downarrow c$ iff $\texttt{interp}(M, \emptyset, \emptyset) = (l', \sigma')$ and $\mathsf{valofcell}(l', \sigma') = c$. While this statement holds for basic values, it *does not* hold for values of other types. The problem arises because the reference-counting interpreter memoizes the results of evaluating under store's whereas the natural semantics does not. For instance, evaluating the term

$$(\lambda x.\, (\mathsf{share}\ y, z\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{if}\ (\mathsf{zero?}\ (\mathsf{fetch}\ y))\ \mathsf{then}\ z\ \mathsf{else}\ z))\ (\mathsf{store}\ (\mathsf{succ}\ 5))$$

in the natural semantics returns the value (store (succ 5)), whereas evaluating the expression in the reference-counting semantics returns the value (after unwinding) (store 6). The proof thus requires relating terms that are 'less evaluated' to terms that are 'more evaluated'.

Definition 12 $M \geq N$, read 'N requires less evaluation than M', iff $M = C[M']$, $N = C[c]$, M' is closed, and $M' \Downarrow c$.

where $C[\,]$ denotes a term with a missing subterm and $C[M']$ the term resulting from using M' for that subterm. Let $\geq^*$ be the reflexive, transitive closure of $\geq$. This relation is necessary in order to express the desired property:

Theorem 13 *Suppose M is typeable, $\mathsf{dom}(\rho) = FV(M)$, M' is closed, and $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$. Suppose also that $\Re(\bar{l}', \rho, \bar{\rho}', \sigma)$.*

1. *If $M' \Downarrow c$, then $\texttt{interp}(M, \rho, \sigma) = (l', \sigma')$ and $c \geq^* \mathsf{valofcell}(l', \sigma')$.*
2. *If $\texttt{interp}(M, \rho, \sigma) = (l', \sigma')$, then $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$.*

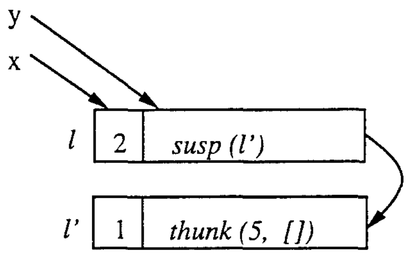
The extra assumptions about the state $(\bar{l}', \rho, \bar{\rho}', \sigma)$—namely that it satisfies the invariants above—are used in constructing an execution in the reference-counting interpreter. The proof is deferred to the appendix.

6 Linear Logic and Memory

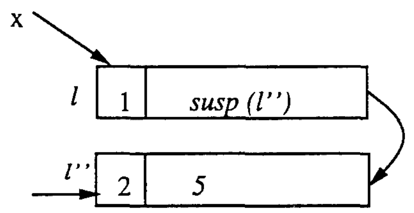
Let us now examine the question of the circumstances under which we are ensured that a location holding a value of linear type will maintain a reference count of at most one. In general, there is no guarantee that locations holding linear values will always have a reference count of one during the evaluation of a program. Consider, for example, the term

$$(\lambda w.\,(\mathsf{share}\ x, y\ \mathsf{as}\ w\ \mathsf{in}\ \mathsf{if}\,(\mathsf{zero?}\ (\mathsf{fetch}\ y))\ \mathsf{then}\ x\ \mathsf{else}\ x))\ (\mathsf{store}\ 5).$$

During evaluation, a suspension is placed in a location l, which in turn holds a pointer to a location l' holding a thunk containing the value 5. This location l is then passed to w, and two pointers called x and y are then created by the **share** which reference l. Pictorially,



When the evaluation continues to the point of (**fetch** y), the contents of the location l' are evaluated to a location l'' holding 5, the suspension in l is updated to point to l'', and a pointer to l'' is then passed to the evaluation of **zero?**. Pictorially,



Thus, the cell containing 5 now has two pointers to it, even though it has linear type, Nat.

Clearly the issue here is whether the location holding a linear value is accessible from a location holding a non-linear one, like a **susp**. We would like a static condition under which we know that this does not happen. This seems difficult because, on the face of it, there are circumstances where a computation can alter the memory graph so that a linear value is brought into a location that is referenced by a non-linear value. Consider the term:

$$M \equiv \lambda x : \mathsf{Nat}.\,\lambda f : \mathsf{Nat}\multimap !\mathsf{Nat}.\,(\mathsf{store}\ y\ \mathsf{where}\ y = (f\ x)) \qquad (1)$$

If N is a term of type $\mathsf{Nat}\multimap!\mathsf{Nat}$, then the evaluation of $((M\ 0)\ N)$ may create a memory graph in which the location holding 0 has been brought into precisely the circumstance above; so its reference count might be increased by pointers passed through a **susp**. We need to know when this can happen if we are to have any way to ensure that a linear value maintains a reference count of at most one.

There is some help on this point to be found in the proof theory of linear logic. Note, that the problem with term M in (1) relies on having a term N of type $\mathsf{Nat}\multimap !\mathsf{Nat}$. From the stand-point of linear logic and its translation under the Curry-Howard correspondence, this is a suspicious assumption, however. The proposition $A \multimap !A$ is *not* provable in LL, and the situation illustrated by M runs contrary to proof-theoretic facts about what propositions are moved through 'boxes' in a proof net during cut elimination [Gir87]. This does not directly prove that a static property exists for the LL-based programming language, but it does suggest that there is hope.

To assert the desired property precisely, we will need some more terminology. Let us say that a storable object is **linear** if it is a numeral, boolean, closure, or recclosure and say that it is **non-linear** if it has the form $\mathsf{susp}(l)$, $\mathsf{rec}(l, f)$, or $\mathsf{thunk}(M, \rho)$. We say that a location l is **non-linear in store** σ if $\sigma(l)$ is a non-linear object; similarly, a location l is **linear in store** σ if $\sigma(l)$ is a linear object. The key property concerns the nature of the path in the memory graph between a location and the root set.

Definition 14 Suppose $S = (l, \sigma, \bar{l}, \bar{\rho})$ is a regular state and $\hat{l} \in \mathsf{dom}(\sigma)$. The location $\hat{l}$ is said to be **linear from** l **in** S if there is a path p from l to $\hat{l}$ in $\mathcal{G}(S)$ such that each l' on p satisfies the following two properties:

1. $\sigma(l')$ is linear and
2. $\mathsf{refcount}(l', \sigma) = 1$.

Note that the two conditions satisfied by the path p could only be satisfied by a *unique* path from l to $\hat{l}$; if there were more than one such path, condition (2) could not be satisfied. It will be convenient to say that a path satisfying these conditions is linear. Given a regular state $S = (\rho, \sigma, \bar{l}, \bar{\rho})$, we also say that $\hat{l}$ is linear from ρ in S if there is an x in the domain of ρ such that there is a (unique) linear path from $\rho(x)$ to $\hat{l}$.

To prove the desired property we will need to know some basic facts about types and evaluation. For the high-level semantics we already expressed the Subject Reduction Theorem 2 for the LL-based programming language. In conjuction with the Correctness Theorem 13 we have a version of the result for the low-level semantics as well:

Lemma 15 *Suppose* $S = (l, \sigma, \bar{l}, \bar{\rho})$ *is a regular state,* $\mathsf{dom}(\rho) = FV(M)$, $\vdash \mathsf{valof}(M, \rho, \sigma) : t$, *and* $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$. *Then* $\vdash \mathsf{valofcell}(l', \sigma') : t$.

The theorem we wish to express says that if a program is evaluated in an environment from which a location $\hat{l}$ is linear, then the value at the location is either used and deallocated or not used and linear from the location returned as the result of the evaluation. This statement is intended to formally capture the idea that a location that is linear from an environment is used once *or* left untouched with a reference count of one. Unfortunately, the assertion contains the term 'deallocate', which needs to be made precise. If we assert instead that the reference count of the location is 0 or linear from the result at the end of the computation, then there is a problem in the case where reference count falls to 0 because the allocation relation might *reallocate* the location $\hat{l}$ to hold a value that is unrelated to the one placed there originally. This would make it impossible to assert anything interesting about the outcome of the computation. To resolve this worry, we can make a restriction on the allocation relation insisting that $\hat{l}$ is not in its range. This assumption is harmless in a sense made precise by Theorem 10. The result of interest can now be asserted precisely as follows:

Theorem 16 *Suppose $S = (\rho, \sigma, \bar{l}, \bar{p})$ is a regular state, $\mathsf{dom}(\rho) = FV(M)$, and $\mathsf{valof}(M, \rho, \sigma)$ is typeable. If $\hat{l}$ is linear from ρ in S, and $\hat{l}$ is not in the range of* new, *and $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$, then one of the following two properties holds of the regular state $S' = (l', \sigma', \bar{l}, \bar{p})$:*

1. *Either* $\mathsf{refcount}(\hat{l}, \sigma') = 0$, *or*
2. $\mathsf{refcount}(\hat{l}, \sigma') = 1$ *and $\hat{l}$ is linear from l' in S'.*

Proof: The proof is by induction on the number of calls to `interp`. We exhibit only a few of the key cases here and leave the others for the reader.

1. $M = (P\ Q)$. The evaluation of M begins as follows:

$$\begin{aligned}&\mathtt{interp}(P, \rho \,|\, P, \sigma) = (l_0, \sigma_0)\\ &\mathtt{interp}(Q, \rho \,|\, Q, \sigma_0) = (l_1, \sigma_1)\\ &\sigma_1(l_0) = \mathsf{closure}(\lambda x.\, N, \rho') \text{ or } \mathsf{recclosure}(\lambda x.\, N, \rho').\end{aligned}$$

The fact that $\hat{l}$ is linear from ρ means that it is reachable from exactly one of $\rho \,|\, P$ or $\rho \,|\, Q$. We consider the two cases separately.

(a) $\hat{l}$ is reachable from $\rho \,|\, P$. By the induction hypothesis ('IH'), one of the following two subcases applies:

i. $\mathsf{refcount}(\hat{l}, \sigma_0) = 0$. By assumption, $\hat{l}$ is never reallocated by new, and hence it follows that $\mathsf{refcount}(\hat{l}, \sigma') = 0$.

ii. $\mathsf{refcount}(\hat{l}, \sigma_0) = 1$ and $\hat{l}$ is linear from l_0. Then in the memory graph, there is a linear path

$$l_0 = l'_0, l'_1, \ldots, \hat{l}$$

(where we list only the locations associated with the path since the fact the reference counts are all equal to one means that the edges are uniquely determined). None of the locations l'_i can be reachable from $\rho \,|\, Q$ since that would imply that the reference count of at least one of them is greater than one. By Theorem 6, the contents and reference counts of the locations l'_i therefore do not change during the evaluation of Q. Now, $\hat{l}$ is linear from l_0 in $(l_1, l_0, \sigma_1, \bar{l}, \bar{p})$ and $\sigma_1(l_0)$ has the form $\mathsf{closure}(\lambda x.\, N, \rho')$ or $\mathsf{recclosure}(\lambda x.\, N, \rho')$, so $\hat{l}$ must be linear from ρ' in $(\rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1), \bar{l}, \bar{p})$ as well. Since we know that $\mathsf{refcount}(l_0, \sigma_1) = 1$, we conclude that

$$\mathtt{interp}(N, \rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1)) = (l', \sigma')$$

and the desired conclusion follows from IH.

(b) $\hat{l}$ is reachable from $\rho \,|\, Q$. By assumption, there is a linear path

$$l'_0, l'_1, \ldots, \hat{l}$$

such that l'_0 is in the image of $\rho \,|\, Q$. None of the locations on this path is reachable from $\rho \,|\, Q$ because they all have reference count equal to one. Thus, by Theorem 6, their values are unchanged by the evaluation of P, and each l'_i is still unreachable from l_0 in σ_0. By IH, there are two possibilities regarding the regular state $(l_1, l_0, \sigma_1, \bar{l}, \bar{p})$ obtained after evaluating P and Q.

i. $\mathsf{refcount}(\hat{l}, \sigma_1) = 0$. By assumption $\hat{l}$ is never reallocated by new, so $\mathsf{refcount}(\hat{l}, \sigma') = 0$ as needed.

ii. $\mathsf{refcount}(\hat{l}, \sigma_1) = 1$, In this case, the IH implies that there is a linear path from l_1 to $\hat{l}$. There are now two subcases to consider: either $\mathsf{refcount}(l_0, \sigma_0) = 1$ or $\mathsf{refcount}(l_0, \sigma_0) > 1$. We consider only the second and leave the first to the reader. By laws D2, I2, and E, we know that the state

$$S' = (\rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l_0, \sigma_1)), \bar{l}, \bar{\rho})$$

is regular and it is not hard to check that $\hat{l}$ is linear from $\rho'[x \mapsto l_1]$ in S'. Since we must have

$$\mathtt{interp}(N, \rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l_0, \sigma_1))) = (l', \sigma')$$

we are done by IH.

2. $M = (\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n)$. In this case, $\hat{l}$ is reachable from exactly one of the environments $\rho \,|\, M_i$. In the evaluation of M, we have

$$\mathtt{interp}(M_1, \rho \,|\, M_1, \sigma) = (l_1, \sigma_1)$$
$$\vdots$$
$$\mathtt{interp}(M_i, \rho \,|\, M_i, \sigma_{i-1}) = (l_i, \sigma_i)$$

By IH, there are two possibilities for the regular state

$$(l_1, \ldots, l_i, \rho \,|\, M_{i+1}, \ldots, \rho \,|\, M_n, \sigma_i, \bar{l}, \bar{\rho})$$

arising after the evaluation of M_i. Either the reference count of $\hat{l}$ is zero in σ_i or it is one and there is a linear path from l_i to $\hat{l}$. If the first case holds, then we are done, since $\hat{l}$ is not reallocated in the remainder of the computation, and therefore the conclusion of the theorem is satisfied. On the other hand, the second case is impossible: by Lemma 15, $\mathsf{valofcell}(l_i, \sigma_i)$ has type $!t$ and $\sigma_i(l_i)$ is a value, so it has the form $\mathsf{susp}(l'')$ or $\mathsf{rec}(l'', f)$. This contradicts the assumption that $\hat{l}$ is linear from l_i. Therefore reference count of $\hat{l}$ must be 0 in σ_i and hence we are done, since new never reallocates $\hat{l}$.

3. $M = (\mathsf{share}\ x, y\ \mathsf{as}\ P\ \mathsf{in}\ Q)$. In the evaluation of M we compute

$$\mathtt{interp}(P, \rho \,|\, P, \sigma) = (l_0, \sigma_0)$$
$$\mathtt{interp}(Q, (\rho \,|\, Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0)) = (l', \sigma')$$

Now $\hat{l}$ is reachable for exactly one of the environments $\rho \,|\, P$ or $\rho \,|\, Q$. We consider the two cases separately.

(a) $\hat{l}$ is reachable from $\rho \,|\, P$. For the same reasons discussed in the case for store above, IH implies that $\mathsf{refcount}(\hat{l}, \sigma_0) = 0$, and thus we are done since new never reallocates $\hat{l}$.

(b) $\hat{l}$ is reachable from $\rho \,|\, Q$. Then there is a linear path from $\rho \,|\, Q$ to $\hat{l}$ which, by Theorem 6, is unaffected by the evaluation of P. In particular, $\hat{l}$ is not reachable from l_0, so it is linear from $\rho \,|\, Q$ in the regular state $((\rho \,|\, Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0), \bar{l}, \bar{\rho})$ so we are done by IH.

The remaining cases are treated similarly. ■

To see an example of how the theorem can be applied to reasoning about properties that depend on the memory graph, suppose we want to evaluate *add* (store 2) 3 in the empty environment and empty store. The key steps are

- *add* (store 2) evaluates to (l_0, σ_0) with $\sigma_0(l_0) = \mathsf{closure}(\lambda x.\, N, \rho)$.
- 3 in σ_0 evaluates to (l_1, σ_1) such that $\sigma_1(l_1) = 3$.
- The body of *add* is evaluated with y mapped to l_1.

At this point the conditions required for the theorem above are true. Hence we know that the reference count of l_1 does not exceed one (so long as it is not deallocated and then reallocated). This implies that it is safe to update y in place during the recursive call. Similar analysis applies to definitions of multiplication and other recursive functions where we use a variable as an accumulator to store the result. This technique of proof allows us to achieve goals like those for which Hudak [Hud87] defined a collecting interpretation for reference counts.

7 Discussion

For this paper we have chosen a particular natural deduction presentation of linear logic. Others have proposed different formulations of linear logic, and it would be interesting to carry out similar investigations for those formulations. For instance, Abramsky [Abr] has used the sequent formulation of linear logic. His system satisfies substitutivity because this is essentially a rule of the sequent presentation (the *cut rule* to be precise), but there is no clear means of doing type inference for his language. Others [Mac91, LM92] have attempted to reconcile the problems of type inference and substitutivity by proposing restricted forms of these properties. Another approach has been to modify linear logic by adding new assumptions. For instance, [Wad91a] and [O'H91] propose taking $!!A$ to be isomorphic to $!A$; from the perspective of this paper, such an identification would collapse two levels of indirection and suspension into one and hence fundamentally change the character of the language. Other approaches to the presentation of LL seem to have compatible explanations within our framework, but might yield slightly different results. For example, there is a way to present LL using judgements of the form $\Gamma; \Delta \vdash s$ where Γ is a set of 'intuitionistic assumptions' (types of non-linear variables) and Δ is a multi-set of 'linear assumptions' (types of linear variables). This approach might suit the results of Section 6 better than the presentation we used in this paper because it singles out the linear variables more clearly and provides what might be a simpler term language. On the other hand, the connection with reference counts is less clear for that formulation.

It is also possible to fold reference-counting operations into the interpretation of a garden variety functional programming language (that is, one based on intuitionistic logic). The ways in which the result differs from the semantics we have given for an LL-based language are illuminating. First of all, there are several choices about how to do this. One approach is to maintain the invariant that interp is evaluated on triples (M, ρ, σ) where the domain of ρ is exactly the set of free variables of M. When evaluating an application $M \equiv (P\ Q)$, for example, it is essential to account for the possibility that some of the free variables of M are shared between P and Q. This means that when P is interpreted, the reference counts of variables they have in common must be incremented (otherwise they may be deallocated before the evaluation of Q begins):

```
interp(($P\ \ Q$), $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = interp($P$, $\rho\,|\,P$, inc-env($\rho\,|\,P \cap \rho\,|\,Q$, $\sigma$))
      ($l_1$, $\sigma_1$) = interp($Q$, $\rho\,|\,Q$, $\sigma_0$)
  in  case $\sigma_1(l_0)$ of closure($\lambda x.\,N$, $\rho'$) or recclosure($\lambda x.\,N$, $\rho'$) =>
        if refcount($l_0$, $\sigma_1$) = 1
        then interp($N$, $\rho'[x \mapsto l_1]$, dec($l_0$, $\sigma_1$))
        else interp($N$, $\rho'[x \mapsto l_1]$, inc-env($\rho'$, dec($l_0$, $\sigma_1$)))
```

The deallocation of variables is driven by the requirement that only the free variables of M can lie in the domain of ρ; this arises particularly in the semantics for the conditional:

```
interp(if $N$ then $P$ else $Q$, $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = interp($N$, $\rho\,|\,N$, inc-env($\rho|N \cap (\rho|P \cup \rho|Q)$, $\sigma$))
  in  if $\sigma_0(l_0)$ = true
      then interp($P$, $\rho\,|\,P$, dec($l_0$, dec-ptrs-env(($\rho|P$) − ($\rho|Q$), $\sigma_0$)))
      else interp($Q$, $\rho\,|\,Q$, dec($l_0$, dec-ptrs-env(($\rho|Q$) − ($\rho|P$), $\sigma_0$)))
```

An alternative approach to providing a reference-counting semantics for an intuitionistic language would be to delay the deallocation of variables until 'the last minute' and permit the application of interp to triples (M, ρ, σ) where the domain of ρ includes the free variables of M but may also include other variables. This makes it possible to simplify the interpretation of the conditional:

```
interp(if $N$ then $P$ else $Q$, $\rho$, $\sigma$) =
  let ($l_0$, $\sigma_0$) = interp($N$, $\rho$, $\sigma$)
  in  if $\sigma_0(l_0)$ = true
      then interp($P$, $\rho$, dec($l_0$, $\sigma_0$))
      else interp($Q$, $\rho$, dec($l_0$, $\sigma_0$))
```

but the burden of disposal then shifts to the evaluation of constants:

```
interp($n$, $\rho$, $\sigma$) = new($n$, dec-ptrs-env($\rho$, $\sigma$))
```

The basic difference between a 'reference-counting interpretation of intuitionistic logic' following one of the approaches just described versus reference counting and linear logic is the way in which the LL primitives make many distinctions *explicit* in the code. The LL primitives make it possible to describe certain kinds of 'code motion' that concern when memory is deallocated. For example, the program

$$\lambda x : s.\, \mathsf{if}\ B\ \mathsf{then}\ (\mathsf{dispose}\ x\ \mathsf{before}\ P)\ \mathsf{else}\ (\mathsf{dispose}\ x\ \mathsf{before}\ Q)$$

can be shown to be equivalent *in the higher-level semantics* to

$$\lambda x : s.\, (\mathsf{dispose}\ x\ \mathsf{before}\ \mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q)$$

but the latter program can be viewed as preferable in the reference-counting semantics, because it may deallocate the locations referenced by x sooner. As another example, the program

$$\lambda x : s.\, (\mathsf{dispose}\ y\ \mathsf{before}\ M)$$

is equivalent to

$$(\mathsf{dispose}\ y\ \mathsf{before}\ \lambda x : s.\, M)$$

if x and y are different variables. The transformation may be significant if the value of y would be deallocated rather than needlessly held in a closure.

Proving that programs like the ones above are equivalent as far as the high-level semantics is concerned can be facilitated by a fixed-point (denotational) semantics for the LL-based language. A reasonable semantics of this kind can be given as an extension of the semantics of call-by-value with the operation $!s$ being interpreted as the *lifting* operation on domains. For such a semantics it is possible to extend the adequacy result using the standard techniques (as in section 6.2 of [Gun92] or 11.4 of [Win93]).

The question of whether an LL-based language could be useful as an intermediate language for compiler analysis for intuitionistic programming languages is certainly related to the techniques for translating between them. By analogy, there have been various studies of the subtleties of transformation to CPS ([LD93] is a one recent example). A closer analogy is the translation of a language meant to be executed in call-by-name into a call-by-value language with primitives for delaying (store'ing) and forcing (fetch'ing). There is a standard translation for this purpose and many of the issues that arise for that translation also arise in the translation from intuitionistic to linear logic. For instance, a pair of programs that are strongly reminiscent of those in Table 1 appears in the discussion of the ALFL compiler in [BHY88] based on a sample from the test suite in [Gab85]. This problem is addressed by the technique of *strictness analysis* [AH87]: with strictness analysis the translation can be made more efficient or the translated program can be optimized. There are several techniques known for translating intuitionistic logic into linear logic. To illustrate, consider the combinator S (here written in ML syntax):

```
fn x => fn y => fn z => (x z)(y z)
```

When we apply Girard's translation, the result (using a syntax similar to the one in Table 1) is the following program:

```
fn x => fn y => fn z =>
  share z1,z2 as z in
    ((fetch x) (store (fetch z1)))
    (store ((fetch y) (store (fetch z2))))
```

However, another program having S as its 'erasure' is

```
fn x => fn y => fn z =>
  share z1,z2 as z in (x z1)(y z2)
```

which is evidently a much simpler and more efficient program. An analog of strictness analysis that applies to the LL translation is clearly needed if an LL intermediate language is to be of practical significance in analyzing 'intuitionistic' programs.

Our reference-counting interpreter and the associated invariance properties can easily be extended to the linear connectives &, $\otimes$, and $\oplus$ (although it is unclear how to handle the 'classical' connectives). Extending the results to dynamic allocation of references and arrays is not difficult if such structures do not create cycles. For instance, it can be assumed that only integers and booleans are assignable to mutable reference cells. To see this in a little more detail, if we assume that o is Nat or Bool, then typing rules can be given as follows:

$$\frac{\Gamma \vdash M : o}{\Gamma \vdash \mathsf{ref}(M) : \mathsf{ref}(o)} \qquad \frac{\Gamma \vdash M : \mathsf{ref}(o) \qquad N : o}{\Gamma \vdash M := N : \mathsf{ref}(o)} \qquad \frac{\Gamma \vdash M : \mathsf{ref}(o)}{\Gamma \vdash\, !M : o}$$

To create a reference cell initialized with the value of a term M, the term M is evaluated and its value is *copied* into a new cell:

```
(16)  interp(ref(M), ρ, σ) =
        let (l₀, σ₀) = interp(M, ρ|M, σ)
        in  new(σ₀(l₀)), dec(l₀, σ₀))
```

The location l_0 holds the *immutable* value of M; a new *mutable* cell must be created with the value of M as its initial value. Assignment mutates the value associated with such a cell:

```
(17)  interp(M := N, ρ, σ) =
        let (l₀, σ₀) = interp(M, ρ|M, σ)
            (l₁, σ₁) = interp(N, ρ|N, σ₁)
        in  (l₀, dec(l₁, σ₁[l₀ ↦ σ₁(l₁)]))
```

To obtain the value held in a mutable cell denoted by M, the contents of the cell must be copied to a new immutable cell:

```
(18)  interp(!$M$, $\rho$, $\sigma$) =
        let ($l_0$, $\sigma_0$) = interp($M$, $\rho \mid M$, $\sigma$)
        in  new($\sigma_0(l_0)$), dec($l_0$, $\sigma_0$))
```

Although the code for creating a reference cell and the code for dereferencing look the same, they are dual to one another in the sense that cell creation, ref(M), copies the contents of an immutable cell to a mutable one while dereferencing, $!M$ copies the contents of a mutable cell to an immutable one. The language designed in this way is similar to Scheme with force and delay primitives, but with restrictions like those of ML on which values are mutable. The restriction on the types of elements held in reference cells is similar to those made for block-structured languages, which do not permit higher-order procedures to be assigned to variables (reference cells).

In conclusion, we have demonstrated that a language whose design is guided by an analog of the Curry-Howard correspondence applied to linear logic can be interpreted as providing fine-grained information about reference counts in the memory graphs produced by the program during run-time. As such, the LL-based language may be useful for detecting or proving the correctness of forms of program analysis that rely on reference counts of nodes of memory graphs. As a secondary theme we have illustrated an approach to expressing and proving properties of programs at a level of abstraction in which properties of memory graphs are significant but some lower-level properties, such as memory layout, are abstracted away. Isolating this level of abstraction could be useful for correctness proofs of lower levels, such as the correctness of a memory allocation scheme.

Acknowledgements

This research was partially supported by an ONR Young Investigator Award number N00014-88-K-0557, NSF grant number CCR-8912778, NRL grant number N00014-91-J-2022, and NOSC grant number 19-920123-31. A preliminary version of the paper appeared in the Proceedings of the 1992 ACM Conference on LISP and Functional Programming. For discussions that contributed to the investigation and exposition in this paper we thank Samson Abramsky, Andrew Appel, Val Breazu-Tannen, Amy Felty, Elsa Gunter, James Hicks, Dave MacQueen, Andre Scedrov, Phil Wadler, and David Wise.

A Proofs of the Main Theorems

Verification of the Basic Laws in Table 7

Proposition 17 *Each of the laws A1, A2, D1, D2 given in Section 4 hold.*

Proof: The proof of A1 may be found in Section 4, and the proof of A2 is similar. We thus need only to verify D1 and D2.

D1 Suppose $S = (l, \bar{l}, \bar{\rho}, \sigma)$, $\Re(S)$ holds, $\sigma(l)$ is a numeral or boolean, and $S' = (\bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$. Note that there are no outgoing edges from l in the memory graph induced by S; thus, even if $l \notin \mathsf{dom}(\mathsf{dec}(l, \sigma))$, the state S' is count-correct. Since $\mathsf{dom}(\sigma) \supseteq \mathsf{dom}(\mathsf{dec}(l, \sigma))$, each of the properties $\Re2$–$\Re5$ follow directly from the hypothesis. Thus, $\Re(S')$.

D2 Suppose $\Re(l, \bar{l}, \bar{\rho}, \sigma)$ and $\mathsf{refcount}(l, \sigma) \neq 1$, and let $S' = (\bar{l}, \rho, \bar{\rho}, \mathsf{dec}(l, \sigma))$. By hypothesis, it follows that $\mathsf{refcount}(l, \sigma) > 1$ since l is in the root set. Thus, $\mathsf{refcount}(l, \mathsf{dec}(l, \sigma)) \geq 1$ and hence S' is count-correct, satisfying $\Re1$. Since $\mathsf{dom}(\sigma) = \mathsf{dom}(\mathsf{dec}(l, \sigma))$, each of the properties $\Re2$–$\Re5$ follow directly from the hypothesis. Thus, $\Re(S')$.

This completes the verification of each part. ■

Proposition 18 *Law D3 holds; more generally,*

1. *If* $\Re(l, \bar{l}, \bar{\rho}, \sigma)$, *then* $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.
2. *If* $\Re(\bar{l}, \rho, \bar{\rho}, \sigma)$, *then* $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \sigma))$.

Proof: By induction on the total number of calls to dec-ptrs and dec-ptrs-env. In the basis, suppose the number of calls is one; there are two cases:

1. dec-ptrs is called. Then there are three subcases:
   (a) $\sigma(l) = n$, true, or false. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec}(l, \sigma)$. By D1, $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.
   (b) $\sigma(l) = \mathsf{susp}(l')$, $\mathsf{thunk}(M, \rho)$, or $\mathsf{closure}(\lambda x. M, \rho)$, and $\mathsf{refcount}(l, \sigma) > 1$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec}(l, \sigma)$, and hence by D2, $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.
   (c) $\sigma(l) = \mathsf{rec}(l', f)$ or $\mathsf{recclosure}(\lambda x. N, \rho)$, and $\mathsf{refcount}(l, \sigma) > 2$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec}(l, \sigma)$, and hence by D2, $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.
2. dec-ptrs-env is called. Then since dec-ptrs is not called, $\mathsf{dom}(\rho)$ must be the empty set. Thus, $\mathsf{dec\text{-}ptrs\text{-}env}(\rho, \sigma) = \sigma$ and hence $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \sigma))$.

For the induction hypothesis, suppose the total number of calls to dec-ptrs and dec-ptrs-env is greater than one. There are again two main cases:

1. dec-ptrs is called. There are five subcases depending on the reference count and the value stored at l.
   (a) $\sigma(l) = \mathsf{susp}(l')$ and $\mathsf{refcount}(l, \sigma) = 1$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec\text{-}ptrs}(l', \mathsf{dec}(l, \sigma))$. By A2, $\Re(l', \bar{l}, \bar{\rho}, \mathsf{dec}(l, \sigma))$ and so by induction, $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l', \mathsf{dec}(l, \sigma)))$. Thus, $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.

(b) $\sigma(l) = \mathsf{thunk}(M, \rho)$, $\mathsf{refcount}(l, \sigma) = 1$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \mathsf{dec}(l, \sigma))$. By A1, $\Re(\bar{l}, \rho, \bar{\rho}, \mathsf{dec}(l, \sigma))$ and so by induction, $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \mathsf{dec}(l, \sigma)))$. Thus, $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.

(c) $\sigma(l) = \mathsf{closure}(\lambda x.\, M, \rho)$ and $\mathsf{refcount}(l, \sigma) = 1$. Similar to the previous case.

(d) $\sigma(l) = \mathsf{recclosure}(\lambda x.\, N, \rho[f \mapsto l'])$, $\sigma(l') = \mathsf{rec}(l, f)$, $\mathsf{refcount}(l, \sigma) = 2$, and $\mathsf{refcount}(l', \sigma) = 1$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \mathsf{dec}(l', \mathsf{dec}(l, \mathsf{dec}(l, \sigma))))$. Let $\sigma_0 = \mathsf{dec}(l', \mathsf{dec}(l, \mathsf{dec}(l, \sigma)))$; then the state $S = (\bar{l}, \rho, \bar{\rho}, \sigma_0)$ is count-correct, since both l and l' have disappeared from the memory graph. Also, S satisfies properties $\Re$2-$\Re$5, since $\mathsf{dom}(\sigma_0) \subseteq \mathsf{dom}(\sigma)$. Thus, $\Re(S)$, and hence by induction $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \sigma_0))$. Thus, $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(l, \sigma))$.

(e) $\sigma(l) = \mathsf{rec}(l', f)$, $\sigma(l') = \mathsf{recclosure}(\lambda x.\, N, \rho[f \mapsto l])$, $\mathsf{refcount}(l, \sigma) = 2$, and $\mathsf{refcount}(l', \sigma) = 1$. Then $\mathsf{dec\text{-}ptrs}(l, \sigma) = \mathsf{dec\text{-}ptrs\text{-}env}(\rho, \mathsf{dec}(l', \mathsf{dec}(l, \mathsf{dec}(l, \sigma))))$. Similar to the previous case.

2. dec-ptrs-env is called. Since the number of calls is greater than one, $\mathsf{dom}(\rho) = \{x_1, \ldots, x_n\}$ for $n \geq 0$. Since $\Re(\rho(x_1), \ldots, \rho(x_n), \bar{l}, \bar{\rho}, \sigma)$ and

$$\mathsf{dec\text{-}ptrs\text{-}env}(\rho, \sigma) = \mathsf{dec\text{-}ptrs}(\rho(x_n), \mathsf{dec\text{-}ptrs}(\ldots \mathsf{dec\text{-}ptrs}(\rho(x_1), \sigma)\ldots))$$

it follows from repeated applications of the induction hypothesis that $\Re(\bar{l}, \bar{\rho}, \mathsf{dec\text{-}ptrs}(\rho, \sigma))$.

This completes the induction hypothesis and hence the proof. ■

Proposition 19 *Each of the laws I1, I2, E, N1, N2, N3, U1, and U2 in Section 4 hold.*

Proof: We verify each law individually.

I1 Suppose $\Re(\bar{l}, \bar{\rho}, \sigma)$ and $l \in \mathsf{dom}(\sigma)$. Let $S' = (l, \bar{l}, \bar{\rho}, \mathsf{inc}(l, \sigma))$. Since there is one more pointer to l in the root set of S' and the reference count has been incremented, S' is count-correct. Since $\mathsf{dom}(\sigma) = \mathsf{dom}(\mathsf{inc}(l, \sigma))$, each of the properties $\Re$2–$\Re$5 follow directly from the hypothesis. Thus, $\Re(S')$.

I2 Suppose $\Re(\bar{l}, \bar{\rho}, \sigma)$ and $\rho(x) \in \mathsf{dom}(\sigma)$ for all $x \in \mathsf{dom}(\rho)$. Then $\Re(\bar{l}, \rho, \bar{\rho}, \mathsf{inc\text{-}env}(\rho, \sigma))$ follows by an easy induction on the size of $\mathsf{dom}(\rho)$ using an arguments similar to the last case.

E Suppose $\Re(l, \bar{l}, \rho, \bar{\rho}, \sigma)$ and $x \notin \mathsf{dom}(\rho)$, and let $S' = (\bar{l}, \rho[x \mapsto l], \bar{\rho}, \sigma)$. Then the root set points of S and S' are identical, and the memory graph induced by S and S' are hence identical. Thus, $\Re(S')$. The converse is similar and omitted.

N1 Suppose $\Re(\bar{l}, \bar{\rho}, \sigma)$ and $(l', \sigma') = \mathsf{new}(c, \sigma)$ for some constant c, and let $S' = (l', \bar{l}, \bar{\rho}, \sigma')$. Since new is an allocation relation, $\mathsf{refcount}(l', \sigma) = 0$, $\mathsf{refcount}(l', \sigma') = 1$, and for any location $l \neq l'$, $\sigma(l) = \sigma(l')$ and $\mathsf{refcount}(l, \sigma) = \mathsf{refcount}(l, \sigma')$. First, note that S' is count-correct, since the only location in σ' that is different from σ is l', and that location has a pointer in the root set. This verifies property $\Re$1. Since $\mathsf{dom}(\sigma)$ is finite, $\mathsf{dom}(\sigma')$ is also finite and so property $\Re$2 holds of S. Finally, since new does not create any additional cycles in the memory graph or thunks or closures, properties $\Re$3–$\Re$5 hold in S'. Thus, $\Re(S')$.

N2 Suppose $\Re(\bar{l},\rho,\bar{\rho},\sigma)$, $(l',\sigma') = \mathsf{new}(\mathsf{closure}(N,\rho),\sigma)$ or $\mathsf{new}(\mathsf{thunk}(N,\rho),\sigma)$, $FV(N) = \mathsf{dom}(\rho)$, and N is typeable, and let $S' = (l',\bar{l},\bar{\rho},\sigma')$. Since new is an allocation relation, $\mathsf{refcount}(l',\sigma) = 0$, $\mathsf{refcount}(l',\sigma') = 1$, and for any location $l \neq l'$, $\sigma(l) = \sigma(l')$ and $\mathsf{refcount}(l,\sigma) = \mathsf{refcount}(l,\sigma')$. To see that property $\Re1$—namely count-correctness—holds of S', note that all of the pointers from ρ are accounted for in the closure or thunk stored in l', and that l' only has reference count 1. To see $\Re2$, $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma) \cup \{l'\}$ is finite because $\mathsf{dom}(\sigma)$ is. If l' is a thunk, then $\mathsf{refcount}(l',\sigma') = 1$, which together with the hypothesis guarantees property $\Re3$. No cycles are created in the induced memory graph by new, so $\Re4$ holds. Finally, $\Re5$ holds by hypothesis. Thus, $\Re(S')$.

N3 Suppose $\Re(l,\bar{l},\bar{\rho},\sigma)$ and $(l',\sigma') = \mathsf{new}(\mathsf{susp}(l),\sigma)$ or $\mathsf{new}(\mathsf{rec}(l,f),\sigma)$. Then $\Re(l',\bar{l},\bar{\rho},\sigma')$ follows in a manner similar to the previous case.

U1 Suppose $S = (\bar{l},\bar{\rho},\sigma)$ and $\Re(S)$, $\sigma(l)$ is a constant. We prove the first statement of U1 only; the first follows similarly. So suppose $l' \in \mathsf{dom}(\sigma)$, and l is not reachable from l' in the memory graph induced by S, and let $S' = (\bar{l},\bar{\rho},\mathsf{inc}(l',\sigma[l \mapsto \mathsf{susp}(l')]))$. In S' the in-degree of l' is now one greater than in S; the in-degree of all other nodes remains the same. Thus, S' satisfies property $\Re1$. Since $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')$, the domain of σ' is finite, satisfying property $\Re2$. No new thunks are created, so property $\Re3$ holds of S'. Since l is not reachable from l' in S, there is no cycle through l in S'. Thus, S' satisfies property $\Re4$. Finally, property $\Re5$ holds since no thunks or closures are added to σ. Thus, $\Re(S')$.

U2 Suppose $S = (l,\bar{l},\bar{\rho},\sigma)$ and $\Re(S)$, $\mathsf{refcount}(l,\sigma) \neq 1$, $\sigma(l) = \mathsf{susp}(l')$, and $\sigma(l') = \mathsf{thunk}(N,\rho)$, and let $S' = (\rho,\bar{l},\bar{\rho},\mathsf{dec}(l',\mathsf{dec}(l,\sigma[l \mapsto c])))$. To verify property $\Re1$, note first that $\mathsf{refcount}(l',\sigma) = 1$ by hypothesis. Thus, since the pointers from $\sigma l'$ are mentioned in the root set of S', it follows that S' is count-correct. It is also clear that each of the properties $\Re2$–$\Re5$ hold of S'. Thus, $\Re(S')$.

This completes the verification of each part. ■

Proof of Lemma 10

Lemma 10 *Suppose* $(\bar{l}',\rho_f,\bar{\rho}',\sigma_f)$ *and* $(\bar{l}'',\rho_g,\bar{\rho}'',\sigma_g)$ *are congruent. If* $\mathtt{interp}_f(M,\rho_f,\sigma_f) = (l'_f,\sigma'_f)$, *then* $\mathtt{interp}_g(M,\rho_g,\sigma_g) = (l'_g,\sigma'_g)$ *and the resultant states* $(l'_f,\bar{l}',\bar{\rho}',\sigma'_f)$ *and* $(l'_g,\bar{l}'',\bar{\rho}'',\sigma'_g)$ *are congruent.*

Proof: By induction on the number of calls to **interp**. We cover the four cases in the core language and leave the other cases to the reader. To make the cases easier to read, let h be the isomorphism from $\mathcal{G}(\sigma_f)$ to $\mathcal{G}(\sigma_g)$ that makes the above states congruent.

1. $M = x$. Then $\mathtt{interp}_f(M,\rho_f,\sigma_f) = (\rho_f(x),\sigma_f)$. Then also $\mathtt{interp}_g(M,\rho_g,\sigma_g) = (\rho_g(x),\sigma_g)$, and the resultant states $(\rho_f(x),\bar{l}',\bar{\rho}',\sigma'_f)$ and $(\rho_g(x),\bar{l}'',\bar{\rho}'',\sigma'_g)$ are congruent via h.

2. $M = (\lambda x.\, P)$. Then $\mathtt{interp}_f(M,\rho_f,\sigma_f) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P,\rho_f),\sigma_f) = (l'_f,\sigma'_f)$. Since f is an allocation relation,

   - $l'_f \notin \mathsf{dom}(\sigma_f)$ and $\mathsf{dom}(\sigma'_f) = \mathsf{dom}(\sigma_f) \cup \{l'_f\}$;
   - for all locations $l \in \mathsf{dom}(\sigma_f)$, $\sigma_f(l) = \sigma'_f(l)$ and $\mathsf{refcount}(l,\sigma_f) = \mathsf{refcount}(l',\sigma'_f)$; and

- $\sigma'_f(l'_f) = \mathsf{closure}(\lambda x.\, P, \rho_f)$ and $\mathsf{refcount}(l'_f, \sigma'_f) = 1$.

Note that $\mathtt{interp}_g(M, \rho_g, \sigma_g) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P, \rho_g), \sigma_g) = (l'_g, \sigma'_g)$. Again, since g is an allocation relation,

- $l'_g \notin \mathsf{dom}(\sigma_g)$ and $\mathsf{dom}(\sigma'_g) = \mathsf{dom}(\sigma_g) \cup \{l'_g\}$;
- for all locations $l \in \mathsf{dom}(\sigma_g)$, $\sigma_g(l) = \sigma'_g(l)$ and $\mathsf{refcount}(l, \sigma_g) = \mathsf{refcount}(l', \sigma'_g)$; and
- $\sigma'_g(l'_g) = \mathsf{closure}(\lambda x.\, P, \rho_g)$ and $\mathsf{refcount}(l'_g, \sigma'_g) = 1$.

Let $h' = h[l'_f \mapsto l'_g]$. It is clear that h' is an isomorphism from $\mathcal{G}(\sigma'_f)$ to $\mathcal{G}(\sigma'_g)$. Now consider the resultant states $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ and $(l'_g, \bar{l}'', \bar{\rho}'', \sigma'_g)$. Using the isomorphism h, the first two conditions for congruence of states are satisfied, and so we just need to show that the last six properties, stating the relationship between the values stored at locations, is satisfied. But the contents of the cells in σ_f and σ_g do not change, and for the new locations, $\sigma'_f(l'_f) = \mathsf{closure}(\lambda x.\, N, \rho_f)$, $\sigma'_g(h'(l'_f)) = \sigma'_g(l'_g) = \mathsf{closure}(\lambda x.\, N, \rho_g)$, $\mathsf{dom}(\rho_f) = \mathsf{dom}(\rho_g)$, and for all $x \in \mathsf{dom}(\rho_f)$, $\rho_g(x) = h'(\rho_f(x))$; the last two facts follow from the hypothesis. Thus, the resultant states are congruent.

3. $M = (P\ Q)$. Since $\mathtt{interp}_f(M, \rho_f, \sigma_f) = (l'_f, \sigma'_f)$,

- $\mathtt{interp}_f(P, \rho_f \,|\, P, \sigma_f) = (l_{f,0}, \sigma_{f,0})$;
- $\mathtt{interp}_f(Q, \rho_f \,|\, Q, \sigma_{f,0}) = (l_{f,1}, \sigma_{f,1})$;
- $\sigma_{f,1}(l_{f,0}) = \mathsf{closure}(\lambda x.\, N, \rho'_f)$ or $\mathsf{recclosure}(\lambda x.\, N, \rho'_f)$.

By hypothesis the environments ρ_f and ρ_g have the same domain, can also be divided into $\rho_g \,|\, P$ and $\rho_g \,|\, Q$. By two applications of the induction hypothesis,

- $\mathtt{interp}_g(P, \rho_g \,|\, P, \sigma_g) = (l_{g,0}, \sigma_{g,0})$ and
- $\mathtt{interp}_g(Q, \rho_g \,|\, Q, \sigma_{g,0}) = (l_{g,1}, \sigma_{g,1})$,

and the states $(l_{f,0}, l_{f,1}, \bar{l}', \bar{\rho}', \sigma_{f,1})$ and $(l_{g,0}, l_{g,1}, \bar{l}'', \bar{\rho}'', \sigma_{g,1})$ are congruent. In particular, note that $\sigma_{g,0}(l_{g,0}) = \mathsf{closure}(\lambda x.\, N, \rho'_g)$ or $\mathsf{recclosure}(\lambda x.\, N, \rho'_g)$. There are now two cases:

(a) $\mathsf{refcount}(l_{f,0}, \sigma_{f,1}) = 1$. Then $\mathsf{refcount}(l_{g,0}, \sigma_{g,1})$ is also 1, since the two reference counts must be the same. Because the states $(\bar{l}', \rho'_f[x \mapsto l_{f,1}], \bar{\rho}', \sigma_{f,1})$ and $(\bar{l}'', \rho'_g[x \mapsto l_{g,1}], \bar{\rho}'', \sigma_{g,1})$ are congruent, it follows from the induction hypothesis that $\mathtt{interp}_g(N, \rho'_g[x \mapsto l_{g,1}], \mathsf{dec}(l_0, \sigma_{g,1})) = (l'_g, \sigma'_g)$ and $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ and $(l'_g, \bar{l}', \bar{\rho}', \sigma'_g)$ are congruent. Putting the pieces together, we also see that $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l'_g, \sigma'_g)$ as desired.

(b) $\mathsf{refcount}(l_{f,0}, \sigma_{f,2}) \neq 1$. Then $\mathsf{refcount}(l_{g,0}, \sigma_{g,2}) \neq 1$ also, since the two reference counts must be the same. Since $(\bar{l}', \rho'_f[x \mapsto l_{f,1}], \bar{\rho}', \sigma_{f,1})$ and $(\bar{l}'' \rho'_g[x \mapsto l_{g,1}], \bar{\rho}'', \sigma_{g,1})$ are congruent, by induction $\mathtt{interp}_g(N, \rho'_g[x \mapsto l_{g,1}], \mathsf{inc\text{-}env}(\rho'_g, \mathsf{dec}(l_{g,0}, \sigma_{g,1}))) = (l'_g, \sigma'_g)$ and $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ and $(l'_g, \bar{l}'', \bar{\rho}'', \sigma'_g)$ are congruent. Putting the pieces together, we also see that $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l'_g, \sigma'_g)$ as desired.

4. $M = (\mathsf{fetch}\ P)$. Then $\mathtt{interp}_f(P, \rho_f, \sigma_f) = (l_{f,0}, \sigma_{f,0})$. By induction, $\mathtt{interp}_g(P, \rho_g, \sigma_g) = (l_{g,0}, \sigma_{g,0})$ and the states $(l_{f,0}, \bar{l}', \bar{\rho}', \sigma_{f,0})$ and $(l_{g,0}, \bar{l}'', \bar{\rho}'', \sigma_{g,0})$ are congruent. Now there are two main cases: either $\sigma_{f,0}(l_{f,0}) = \mathsf{susp}(l_{f,1})$ or $\sigma_{f,0}(l_{f,0}) = \mathsf{rec}(l_{f,1}, x)$. We leave the second case to the reader since it is relatively straightforward and consider only the first case.

   Suppose $\sigma_{f,0}(l_{f,0}) = \mathsf{susp}(l_{f,1})$. By the definition of congruence, $\sigma_{g,0}(l_{g,0}) = \mathsf{susp}(l_{g,1})$. Now there are two subcases depending on the object held at $l_{f,1}$:

   (a) $\sigma_{f,0}(l_{f,1}) = \mathsf{thunk}(R, \rho'_f)$. Then by congruence, $\sigma_{g,0}(l_{g,1}) = \mathsf{thunk}(R, \rho'_f)$. There are two subcases depending on the reference count of $l_{f,0}$:

      i. $\mathsf{refcount}(l_{f,0}, \sigma_{f,0}) = 1$. Since the above tuples are congruent, $\mathsf{refcount}(l_{g,0}, \sigma_{g,0}) = 1$. Note that the states

      $$(\bar{l}', \rho'_f, \bar{\rho}', \mathsf{dec}(l_{f,1}, \mathsf{dec}(l_{f,0}\sigma_{f,0},)))$$
      $$(\bar{l}'', \rho'_g, \bar{\rho}'', \mathsf{dec}(l_{g,1}, \mathsf{dec}(l_{g,0}\sigma_{g,0},)))$$

      are congruent since ρ'_f and ρ'_g must have the same domain and must match via the multigraph isomorphism h on their domains. Thus, by induction, $\mathtt{interp}_g(R, \rho'_g, \mathsf{dec}(l_{g,1}, \mathsf{dec}(l_{g,0}\sigma_{g,0},))) = (l'_g, \sigma'_g)$ and the states $(l'_f, \bar{l}', \bar{\rho}', \sigma'_f)$ and $(l'_g, \bar{l}'', \bar{\rho}'', \sigma'_g)$ are congruent. Putting all the steps together, we also see that $\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l'_g, \sigma'_g)$

      ii. $\mathsf{refcount}(l_{f,1}, \sigma_{f,1}) \neq 0$. Similar to the previous case.

   (b) $\sigma_{f,0}(l_{f,1}) \neq \mathsf{thunk}(R, \rho'_f)$. Then again there are two cases depending on the reference count of $l_{f,0}$:

      i. $\mathsf{refcount}(l_{f,0}, \sigma_{f,0}) = 1$; then $\mathsf{refcount}(l_{g,0}, \sigma_{g,0}) = 1$. Thus,

      $$\mathtt{interp}_g(M, \rho_g, \sigma_g) = (l_{g,1}, \mathsf{dec}(l_{g,0}, \sigma_{g,0}))$$

      and the states $(l_{f,1}, \bar{l}', \bar{\rho}', \mathsf{dec}(l_{f,0}, \sigma_{f,0}))$ and $(l_{g,1}, \bar{l}'', \bar{\rho}'', \mathsf{dec}(l_{g,0}, \sigma_{g,0}))$ are congruent.

      ii. $\mathsf{refcount}(l_{f,0}, \sigma_{f,0}) \neq 1$. Similar to the previous case.

This completes the induction and hence the proof. ■

Proof of Theorem 13

Recall from Section 5 that, in order to prove a correctness theorem, we needed a definition of how to unwind a term from a store. The definition of two mutually-recursive functions for performing this task, valof and valofcell, appears in Table 8. It is obvious from the definitions that only the reachable cells affect the value returned by valof and valofcell. For instance, if l' is not reachable from l in store σ and $\sigma' = \mathsf{dec}(l', \sigma)$, then $\mathsf{valofcell}(l, \sigma) = \mathsf{valofcell}(l, \sigma')$. We will use this fact throughout the arguments that follow.

Also essential to the proof of Theorem 13 is a notion of when one term is 'more evaluated' than another. Section 5 defines a relation $\geq^*$ between terms which expresses this relationship. We can prove three lemmas about the relationship of $\geq$ and canonical forms.

Lemma 20 *If $c \geq P$ and c is a canonical form, then P is a canonical form. Moreover, c and P have the same shape, i.e., if c is a numeral or boolean, then $c = P$; if $c = \lambda x.\, Q$, then $P = \lambda x.\, Q'$; and if $c = (\mathsf{store}\ Q)$, then $P = (\mathsf{store}\ Q')$.*

Table 8: Definitions of valof and valofcell.

$$
\begin{array}{rcl}
\mathsf{valof}(x,\rho,\sigma) & = & \mathsf{valofcell}(\rho(x),\sigma) \\
\mathsf{valof}(\lambda x.\,P,\rho,\sigma) & = & \lambda x.\,\mathsf{valof}(M,\rho,\sigma), \quad x \notin \mathsf{dom}(\rho) \\
\mathsf{valof}((P\ Q),\rho,\sigma) & = & (\mathsf{valof}(P,\rho,\sigma)\ \mathsf{valof}(Q,\rho,\sigma)) \\
\mathsf{valof}((\mathsf{fetch}\ P),\rho,\sigma) & = & (\mathsf{fetch}\ \mathsf{valof}(P,\rho,\sigma)) \\
\mathsf{valof}((\mathsf{share}\ x,y\ \mathsf{as}\ P\ \mathsf{in}\ Q),\rho,\sigma) & = & (\mathsf{share}\ x,y\ \mathsf{as}\ \mathsf{valof}(P,\rho,\sigma)\ \mathsf{in}\ \mathsf{valof}(Q,\rho,\sigma)), \\
 & & \text{where } x,y \notin \mathsf{dom}(\rho) \\
\mathsf{valof}((\mathsf{dispose}\ P\ \mathsf{before}\ Q),\rho,\sigma) & = & (\mathsf{dispose}\ \mathsf{valof}(P,\rho,\sigma)\ \mathsf{before}\ \mathsf{valof}(Q,\rho,\sigma)) \\
\mathsf{valof}(n,\rho,\sigma) & = & n \\
\mathsf{valof}(\mathsf{true},\rho,\sigma) & = & \mathsf{true} \\
\mathsf{valof}(\mathsf{false},\rho,\sigma) & = & \mathsf{false} \\
\mathsf{valof}((\mathsf{succ}\ P),\rho,\sigma) & = & (\mathsf{succ}\ \mathsf{valof}(P,\rho,\sigma)) \\
\mathsf{valof}((\mathsf{pred}\ P),\rho,\sigma) & = & (\mathsf{pred}\ \mathsf{valof}(P,\rho,\sigma)) \\
\mathsf{valof}((\mathsf{zero?}\ P),\rho,\sigma) & = & (\mathsf{zero?}\ \mathsf{valof}(P,\rho,\sigma)) \\
\mathsf{valof}((\mathsf{fix}\ P),\rho,\sigma) & = & (\mathsf{fix}\ \mathsf{valof}(P,\rho,\sigma)) \\
\mathsf{valof}((\mathsf{if}\ N\ \mathsf{then}\ P\ \mathsf{else}\ Q),\rho,\sigma) & = & \mathsf{if}\ \mathsf{valof}(N,\rho,\sigma)\ \mathsf{then}\ \mathsf{valof}(P,\rho,\sigma)\ \mathsf{else}\ \mathsf{valof}(Q,\rho,\sigma)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{valof}((\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1,\ldots,x_n = M_n),\rho,\sigma) \\
\quad = (\mathsf{store}\ \mathsf{valof}(N,\rho,\sigma)\ \mathsf{where}\ x_1 = \mathsf{valof}(M_1,\rho,\sigma),\ldots,x_n = \mathsf{valof}(M_n,\rho,\sigma)),\ x_i \notin \mathsf{dom}(\rho)
\end{array}
$$

$$
\mathsf{valofcell}(l,\sigma) = \begin{cases}
n & \text{if } \sigma(l) = n \\
\mathsf{true} & \text{if } \sigma(l) = \mathsf{true} \\
\mathsf{false} & \text{if } \sigma(l) = \mathsf{false} \\
\lambda x.\,\mathsf{valof}(M,\rho,\sigma) & \text{if } \sigma(l) = \mathsf{closure}(\lambda x.\,M,\rho) \text{ or} \\
 & \quad \sigma(l) = \mathsf{recclosure}(\lambda x.\,M,\rho) \text{ and} \\
 & \quad x \notin \mathsf{dom}(\rho) \\
(\mathsf{store}\ \mathsf{valofcell}(l',\sigma)) & \text{if } \sigma(l) = \mathsf{susp}(l') \\
\mathsf{valof}(M,\rho,\sigma) & \text{if } \sigma(l) = \mathsf{thunk}(M,\rho) \\
\mathsf{valof}((\mathsf{fix}\ (\mathsf{store}\ (\lambda f.\,\lambda x.\,M))),\rho,\sigma) & \text{if } \sigma(l) = \mathsf{rec}(l',f), \\
 & \quad \sigma(l') = \mathsf{recclosure}(\lambda x.\,M,\rho[f \mapsto l]), \\
 & \quad x,f \notin \mathsf{dom}(\rho)
\end{cases}
$$

Proof: There are two cases to consider: either $c \Downarrow P$, or $c = C[M]$, $P = C[N]$, $C[\cdot]$ is nontrivial, and $M \Downarrow N$. In the first case, since c is canonical, $c = P$, and hence P is canonical. In the second case, for c to be canonical it must be the case that $C[\cdot] = n$, true, false, $\lambda x.\, D[\cdot]$, or $(\mathsf{store}\ C[\cdot])$. Thus, P must be canonical as well, and must have the same shape as c. ■

Lemma 21 *If c is a canonical form and $M \geq c$, then $M \Downarrow d \geq c$.*

Proof: By the definition of $M \geq c$, we know that $M = C[M']$, $c = C[d]$, and $M' \Downarrow d$. In order for c to be canonical, it must be the case that either $C[\cdot] = [\cdot]$, n, true, false, $\lambda x.\, D[\cdot]$, or $(\mathsf{store}\ D[\cdot])$. In the first case, $M' = M$ and $d = c$, so $M \Downarrow c \geq c$. For the other cases, $M \Downarrow M \geq c$. ■

Lemma 22 *If c is a canonical form and $M \geq^* c$, then $M \Downarrow d \geq^* c$.*

Proof: An easy induction on the length of $M = M_1 \geq \ldots \geq M_k \geq c$ using Lemma 20. ■

We need a similar definition of one state in the reference-counting interpreter being 'more evaluated' than another. Basically, one state is more evaluated than another if, tracing from the root set, the storable objects held at nodes are identical or thunks have been replaced by more evaluated forms. Formally,

Definition 23 We say $(\bar{l}, \bar{\rho}, \sigma) \geq^* (\bar{l}, \bar{\rho}, \sigma')$ if for all l reachable from the root set, $l \in \mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma')$ and

1. $\sigma(l) = n$, true, false, $\mathsf{closure}(\lambda x.\, N, \rho)$, or $\mathsf{recclosure}(\lambda x.\, N, \rho)$, and $\sigma(l) = \sigma'(l)$ and $(\rho, \sigma) \geq^* (\rho, \sigma')$;
2. $\sigma(l) = \mathsf{susp}(l_0)$ or $\mathsf{rec}(l_0, f)$, $\sigma(l_0)$ is not a thunk, $\sigma'(l) = \mathsf{susp}(l_0)$ and $(l_0, \sigma) \geq^* (l_0, \sigma')$; or
3. $\sigma(l) = \mathsf{susp}(l_0)$, $\sigma(l_0) = \mathsf{thunk}(R, \rho)$ and either
   (a) $\sigma'(l) = \mathsf{susp}(l_0)$, $\sigma'(l_0) = \mathsf{thunk}(R, \rho)$, and $(\rho, \sigma) \geq^* (\rho, \sigma')$; or
   (b) $\sigma'(l) = \mathsf{susp}(l')$, $\sigma'(l')$ is not a thunk, $\mathtt{interp}(R, \rho, \sigma) = (l', \sigma'')$, and $(l', \sigma'') \geq^* (l', \sigma')$

where $(\rho, \sigma) \geq^* (\rho, \sigma')$ if for every $x \in \mathsf{dom}(\rho)$, $(\rho(x), \sigma) \geq^* (\rho(x), \sigma')$.

It is not difficult to prove that $\geq^*$ is reflexive and transitive on states. It is also not difficult to prove the following two lemmas:

Lemma 24 *Suppose $\Re(\bar{l}, \rho, \bar{\rho}, \sigma)$ and $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$. Then $(\bar{l}, \bar{\rho}, \sigma) \geq^* (\bar{l}, \bar{\rho}, \sigma')$.*

Lemma 25 *If $Q' \geq^* \mathsf{valof}(Q, \rho, \sigma)$ and $(\bar{l}', \rho, \bar{\rho}', \sigma) \geq^* (\bar{l}', \rho, \bar{\rho}', \sigma')$, then $Q' \geq^* \mathsf{valof}(Q, \rho, \sigma')$.*

The proof of the first is an easy induction on the number of calls to $\mathtt{interp}$; the proof of the second is an easy induction on the definition of valof.

We now have enough machinery to prove the main correctness theorem.

Theorem 13 *Suppose M is typeable, $\mathsf{dom}(\rho) = FV(M)$, M' is closed, and $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$. Suppose also that $\Re(\bar{l}', \rho, \bar{\rho}', \sigma)$.*

1. *If $M' \Downarrow c$, then $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$ and $c \geq^* \mathsf{valofcell}(l', \sigma')$.*

2. *If* $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$, *then* $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$.

Proof: The first part is proven by induction on the height of the proof of $M' \Downarrow c$. We consider the cases for the core language and leave the cases for the PCF extensions to the reader. To ease the readability of the various cases, we can separate each induction case into two cases based on whether or not M is a variable or a canonical form. The first of these cases can be seen immediately. If M is a canonical form or variable, then the form of the rules guarantees that $\mathtt{interp}(M, \rho, \sigma)$ returns a result (l', σ') and $\mathsf{valofcell}(l', \sigma') = \mathsf{valof}(M, \rho, \sigma)$. Thus, by Lemma 22, it follows that $c \geq^* \mathsf{valofcell}(l', \sigma')$. For instance, if $M = (\lambda x.\, P)$, then $\mathtt{interp}(M, \rho, \sigma) = \mathsf{new}(\mathsf{closure}(\lambda x.\, P, \rho), \sigma) = (l', \sigma')$. Since $\Re(\bar{l'}, \rho, \bar{\rho'}, \sigma)$ and $l' \notin \mathsf{dom}(\sigma)$, the new cell l' in σ' cannot be reached from σ. Thus,

$$\mathsf{valofcell}(l', \sigma') = \mathsf{valof}(\lambda x.\, P, \rho, \sigma') = \mathsf{valof}(\lambda x.\, P, \rho, \sigma)$$

as desired.

If, on the other hand, M is not a variable or canonical form, then there is some interpretation required in the reference-counting interpreter. Now we divide into cases depending on the last rule used in the proof of $M' \Downarrow c$.

1. $M' = (P'\ Q')$, where $P' \Downarrow (\lambda x.\, N')$, $Q' \Downarrow d$, and $N'[x := d] \Downarrow c$. The only case to consider is $M = (P\ Q)$, where $P' \geq^* \mathsf{valof}(P, \rho, \sigma)$ and $Q' \geq^* \mathsf{valof}(Q, \rho, \sigma)$. Since M is typeable, the free variables of P and Q are disjoint. The first step is to evaluate the operator and operand. By induction,

   $$\mathtt{interp}(P, \rho \,|\, P, \sigma) = (l_0, \sigma_0)$$

   and $(\lambda x.\, N') \geq^* \mathsf{valofcell}(l_0, \sigma_0)$. We need to show that l_0 really holds a closure. By Lemma 20, $(\lambda x.\, N')$ and $\mathsf{valofcell}(l_0, \sigma_0)$ must have the same shape. Since $\Re(\bar{l'}, \rho \,|\, P, \rho \,|\, Q, \bar{\rho'}, \sigma)$, by Theorem 6 $\Re(l_0, \bar{l'}, \rho \,|\, Q, \bar{\rho'}, \sigma_0)$ and so $\sigma_0(l_0)$ cannot be a thunk. Thus, the only possibility left is that

   $$\sigma_0(l_0) = \mathsf{closure}(\lambda x.\, N, \rho') \text{ or } \mathsf{recclosure}(\lambda x.\, N, \rho').$$

   Next we need to evaluate the operand. By Lemma 24 we know $(\bar{l'}, \rho \,|\, Q, \bar{\rho'}, \sigma) \geq^* (\bar{l'}, \rho \,|\, Q, \bar{\rho'}, \sigma_0)$. Since $Q' \geq^* \mathsf{valof}(Q, \rho \,|\, Q, \sigma)$, by Lemma 25 we have $Q' \geq^* \mathsf{valof}(Q, \rho \,|\, Q, \sigma_0)$. By the induction hypothesis,

   $$\mathtt{interp}(Q, \rho \,|\, Q, \sigma_0) = (l_1, \sigma_1)$$

   where $d \geq^* \mathsf{valofcell}(l_1, \sigma_1)$. Since $\mathtt{interp}(Q, \rho \,|\, Q, \sigma_0) = (l_1, \sigma_1)$, it follows from Lemma 24 that $(l_0, \bar{l'}, \bar{\rho'}, \sigma_0) \geq^* (l_0, \bar{l'}, \bar{\rho'}, \sigma_1)$; thus,

   $$\sigma_1(l_0) = \mathsf{closure}(\lambda x.\, N, \rho') \text{ or } \mathsf{recclosure}(\lambda x.\, N, \rho').$$

   To evaluate the application, there are two subcases: either $\mathsf{refcount}(l_0, \sigma_1) = 1$ or $\mathsf{refcount}(l_0, \sigma_1) > 1$. We do the first case and leave the other case to the reader. If $\mathsf{refcount}(l_0, \sigma_1) = 1$, then $\Re(\bar{l'}, \rho'[x \mapsto l_1], \bar{\rho'}, \mathsf{dec}(l_0, \sigma_1))$ by laws A2 and E. It follows from

   $$(\bar{l'}, \bar{\rho'}, \sigma_0) \geq^* (\bar{l'}, \bar{\rho'}, \sigma_1) \geq^* (\bar{l'}, \bar{\rho'}, \mathsf{dec}(l_0, \sigma_1))$$

   and Lemma 25 that $N'[x := d] \geq^* \mathsf{valof}(N, \rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1))$. Thus, by induction,

   $$\mathtt{interp}(N, \rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1)) = (l', \sigma')$$

   where $c \geq^* \mathsf{valofcell}(l', \sigma')$. This shows that $\mathtt{interp}(M, \rho, \sigma) = (l', \sigma')$ and $c \geq^* \mathsf{valofcell}(l', \sigma')$.

2. $M' = (\mathsf{store}\ N'\ \mathsf{where}\ x_1 = M'_1, \ldots, x_n = M'_n)$, $c = (\mathsf{store}\ N'[x_1, \ldots, x_n := d_1, \ldots, d_n])$, and $M'_i \Downarrow d_i$. We only need to consider the case when $M = (\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n)$, where $N' \geq^* \mathsf{valof}(N, \emptyset, \sigma)$ and $M'_i \geq^* \mathsf{valof}(M_i, \rho \,|\, M_i, \sigma)$. Since M is typeable, the free variables of each M_i are disjoint. Since $M'_1 \geq^* \mathsf{valof}(M_1, \rho \,|\, M_1, \sigma)$, by induction

   $$\mathtt{interp}(M_1, \rho_1, \sigma) = (l_1, \sigma_1),$$

   where $d_1 \geq^* \mathsf{valofcell}(l_1, \sigma_1)$. By Lemma 24,

   $$(\bar{l}', \rho \,|\, M_2, \ldots, \rho \,|\, M_n, \sigma) \geq^* (\bar{l}', \rho \,|\, M_2, \ldots, \rho \,|\, M_n, \sigma_1)$$

   and by Theorem 6, $\Re(l_1, \bar{l}', \rho \,|\, M_2, \ldots, \rho \,|\, M_n, \sigma_1)$. Since $M'_2 \geq^* \mathsf{valof}(M_2, \rho \,|\, M_2, \sigma)$, by Lemma 25, $M'_2 \geq^* \mathsf{valof}(M_2, \rho \,|\, M_2, \sigma_1)$. Using similar repeated applications of the induction hypothesis,

   $$\mathtt{interp}(M_i, \rho_i, \sigma_{i-1}) = (l_i, \sigma_i)$$

   where $d_i \geq^* \mathsf{valofcell}(l_i, \sigma_i)$, and by Lemma 24,

   $$(l_1, \ldots, l_{i-1}, \bar{l}', \bar{\rho}', \sigma_{i-1}) \geq^* (l_1, \ldots, l_{i-1}, \bar{l}', \bar{\rho}', \sigma_i)$$

   Finally, let

   $$\begin{aligned} &\rho' = \emptyset[x_1, \ldots, x_n \mapsto l_1, \ldots, l_n] \\ &\mathsf{new}(\mathsf{thunk}(N, \rho'), \sigma_n) = (l_{n+1}, \sigma_{n+1}) \\ &\mathsf{new}(\mathsf{susp}(l_{n+1}), \sigma_{n+1}) = (l', \sigma') \end{aligned}$$

   Then using Lemma 25, we find that $(\mathsf{store}\ N'[x_1, \ldots, x_n := d_1, \ldots, d_n]) \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

3. $M' = (\mathsf{fetch}\ N')$, where $N' \Downarrow (\mathsf{store}\ Q')$ and $Q' \Downarrow c$. Then the only case to consider is $M = (\mathsf{fetch}\ N)$ where $N' \geq^* \mathsf{valof}(N, \rho, \sigma)$. By induction,

   $$\mathtt{interp}(N, \rho, \sigma) = (l_0, \sigma_0)$$

   where $(\mathsf{store}\ Q') \geq^* \mathsf{valofcell}(l_0, \sigma_0)$. By Theorem 6, $\Re(l_0, \bar{l}', \bar{\rho}', \sigma_0)$. Since $(\mathsf{store}\ Q') \geq^* \mathsf{valofcell}(l_0, \sigma_0)$ and $\sigma_0(l_0)$ must be a value, it follows from Lemma 20 that $\sigma_0(l_0) = \mathsf{susp}(l_1)$ or $\mathsf{rec}(l_1, f)$ and $Q' \geq^* \mathsf{valofcell}(l_1, \sigma_0)$. We consider only the case when $\sigma_0(l_0)$ is $\mathsf{susp}(l_1)$ and leave the other case to the reader. There are two subcases:

   (a) $\sigma_0(l_1) = \mathsf{thunk}(R, \rho')$. There are two subcases:

      i. $\mathsf{refcount}(l_0, \sigma_0) = 1$. First, note that neither l_0 nor l_1 is reachable from ρ'—if either were, the state $S = (l_0, \bar{l}', \bar{\rho}', \sigma_0)$ would have a cycle that was not composed solely of a rec and a recclosure—and this contradicts the regularity of the state S. Thus,

      $$Q' \geq^* \mathsf{valof}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))).$$

      By laws A1 and A2, $\Re(\bar{l}', \rho', \bar{\rho}', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0)))$. Thus, it follows by induction that $\mathtt{interp}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))) = (l', \sigma')$ and $c \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

ii. $\mathsf{refcount}(l_0, \sigma_0) > 1$. First, note that neither l_0 nor l_1 is not reachable from ρ'—if either were, there would be an illegal cycle in the memory graph induced by $S = (l_0, \bar{l'}, \bar{\rho}', \sigma_0)$. Thus, $Q' \geq^* \mathsf{valof}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])))$ still holds. By law U2, $\Re(\bar{l'}, \rho', \bar{\rho}', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])))$. Thus, it follows by induction that

$$\mathtt{interp}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0]))) = (l_2, \sigma_1)$$

and $c \geq^* \mathsf{valofcell}(l_2, \sigma_1)$. Since l_0 is not reachable from ρ' in $\mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])$, by Theorem 6 it follows that l_0 is not reachable from l_2 in σ_1. Thus, $c \geq^* \mathsf{valofcell}(l_2, \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]))$ as desired.

(b) $\sigma_0(l_1) \neq \mathsf{thunk}(R, \rho')$. Then $\mathsf{valofcell}(l_1, \sigma_0)$ is a value. There are two subcases:

i. $\mathsf{refcount}(l_0, \sigma_0) = 1$. Since $Q' \geq^* \mathsf{valofcell}(l_1, \mathsf{dec}(l_0, \sigma_0))$, it follows by Lemma 22 that $c \geq^* \mathsf{valofcell}(l_1, \mathsf{dec}(l_0, \sigma_0))$.

ii. $\mathsf{refcount}(l_0, \sigma_0) > 1$. Similar to the previous subcase and hence omitted.

4. $M' = (\mathsf{share}\ x, y\ \mathsf{as}\ P'\ \mathsf{in}\ Q')$, where $P' \Downarrow d$ and $Q'[x, y := d] \Downarrow c$. Then the only case to consider is $M = (\mathsf{share}\ x, y\ \mathsf{as}\ P\ \mathsf{in}\ Q)$, where $P' \geq^* \mathsf{valof}(P, \rho \,|\, P, \sigma)$ and $Q' \geq^* \mathsf{valof}(Q, \rho \,|\, Q, \sigma)$. Since M is typeable, the free variables of P and Q are disjoint. Since $P' \geq^* \mathsf{valof}(P, \rho \,|\, P, \sigma)$, it follows by induction that
$$\mathtt{interp}(P, \rho \,|\, P, \sigma) = (l_0, \sigma_0),$$
where $d \geq^* \mathsf{valofcell}(l_0, \sigma_0)$. By Lemmas 24 and 25, $Q' \geq^* \mathsf{valof}(Q, \rho \,|\, Q, \sigma_0)$, and by Theorem 6, $(\bar{l'}, \rho \,|\, Q, \bar{\rho}', \sigma_0)$. By laws I1 and E,
$$\Re(\bar{l'}, (\rho \,|\, Q)[x, y \mapsto l_0], \bar{\rho}', \mathsf{inc}(l_0, \sigma_0)).$$
Since $Q'[x, y := d] \geq^* \mathsf{valof}(Q, (\rho \,|\, Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0))$, it follows by induction that
$$\mathtt{interp}(Q, \rho \,|\, Q[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0)) = (l', \sigma')$$
and $c \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

5. $M' = (\mathsf{dispose}\ P'\ \mathsf{before}\ Q')$, where $Q' \Downarrow c$. This case is similar to the previous case and hence omitted.

This completes the proof of the first part. The second part is proven by induction on the number of calls to `interp`. We consider the cases for the core of the language and leave the cases for the PCF extensions to the reader.

1. $M = x$. Then $\mathtt{interp}(M, \rho, \sigma) = (\rho(x), \sigma) = (l', \sigma')$. Note that $\mathsf{valofcell}(l', \sigma') = \mathsf{valof}(M, \rho, \sigma)$, and hence $M' \geq^* \mathsf{valofcell}(l', \sigma')$. Since $\Re(\bar{l'}, \rho, \bar{\rho}', \sigma)$, $\sigma'(l') = \sigma(\rho(x))$ must be a value, and hence $\mathsf{valofcell}(l', \sigma') = d$ where d is a canonical form. Thus, by Lemma 22, $M' \Downarrow c \geq^* d = \mathsf{valofcell}(l', \sigma')$ as desired.

2. $M = (\lambda x.\, N)$. Similar to the previous case.

3. $M = (P\ Q)$. Since $\texttt{interp}(M, \rho, \sigma) = (l', \sigma')$, it follows that

$$\begin{aligned}&\texttt{interp}(P, \rho \,|\, P, \sigma) = (l_0, \sigma_0)\\ &\texttt{interp}(Q, \rho \,|\, Q, \sigma_0) = (l_1, \sigma_1)\\ &\sigma_1(l_0) = \mathsf{closure}(\lambda x.\, N, \rho') \text{ or } \mathsf{recclosure}(\lambda x.\, N, \rho')\end{aligned}$$

Since $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$, it must be that $M' = (P'\ Q')$ for some closed P' and Q', where $P' \geq^* \mathsf{valof}(P, \rho, \sigma)$ and $Q' \geq^* \mathsf{valof}(Q, \rho, \sigma)$. By induction,

$$\begin{aligned}P' \Downarrow d' &\geq^* \mathsf{valofcell}(l_0, \sigma_0)\\ Q' \Downarrow d &\geq^* \mathsf{valofcell}(l_1, \sigma_1)\end{aligned}$$

By Lemmas 24 and 25, $d' \geq^* \mathsf{valofcell}(l_0, \sigma_1)$. Since $\sigma_1(l_0)$ is a closure, $\mathsf{valofcell}(l_0, \sigma_1)$ must be a λ-abstraction, and so by Lemma 20 it follows that $d' = (\lambda x.\, N')$ for some N'. If $\mathsf{refcount}(l_0, \sigma_1) = 1$, then $N'[x := d] \geq^* \mathsf{valof}(N', \rho'[x \mapsto l_1], \mathsf{dec}(l_0, \sigma_1))$. If, on the other hand, $\mathsf{refcount}(l_0, \sigma_1) > 1$, then $N'[x := d] \geq^* \mathsf{valof}(N', \rho'[x \mapsto l_1], \mathsf{inc\text{-}env}(\rho', \mathsf{dec}(l_0, \sigma_1)))$. In either case, it follows by the induction hypothesis that

$$N'[x := d] \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma').$$

Thus, we conclude $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$.

4. $M = (\mathsf{store}\ N\ \mathsf{where}\ x_1 = M_1, \ldots, x_n = M_n)$. Since M evaluates,

$$\begin{aligned}&\texttt{interp}(M_1, \rho \,|\, M_1, \sigma) = (l_1, \sigma_1)\\ &\texttt{interp}(M_2, \rho \,|\, M_2, \sigma_1) = (l_2, \sigma_2)\\ &\qquad\vdots\\ &\texttt{interp}(M_n, \rho \,|\, M_n, \sigma_{n-1}) = (l_n, \sigma_n)\\ &\rho' = \emptyset[x_1, \ldots, x_n \mapsto l_1, \ldots, l_n]\\ &(l_{n+1}, \sigma_{n+1}) = \mathsf{new}(\mathsf{thunk}(N, \rho'), \sigma_n)\\ &(l', \sigma') = \mathsf{new}(\mathsf{susp}(l_{n+1}), \sigma_{n+1})\end{aligned}$$

Since $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$, it follows that $M' = (\mathsf{store}\ N'\ \mathsf{where}\ x_1 = M_1', \ldots, x_n = M_n')$ and $N' \geq^* \mathsf{valof}(N, \emptyset, \sigma)$ and $M_i' \geq^* \mathsf{valof}(M_i, \rho \,|\, M_i, \sigma)$. By induction, $M_1 \Downarrow c_1 \geq^* \mathsf{valofcell}(l_1, \sigma_1)$. To evaluate the next term in the sequence, note that

$$M_2' \geq^* \mathsf{valof}(M_2, \rho_2, \sigma) \geq^* \mathsf{valof}(M_2, \rho_2, \sigma_1)$$

so by induction $M_2' \Downarrow c_2 \geq^* \mathsf{valofcell}(l_2, \sigma_2)$. Extending the induction hypothesis further yields that $M_i \Downarrow c_i \geq^* \mathsf{valofcell}(l_i, \sigma_i)$. Note also that by Lemmas 24 and 25, $N' \geq^* \mathsf{valof}(N, \emptyset, \sigma_n)$; it follows that

$$(\mathsf{store}\ N'[x_1, \ldots, x_n := c_1, \ldots, c_n]) = c \geq^* \mathsf{valof}(N, \rho', \sigma_n) = \mathsf{valofcell}(l', \sigma').$$

Thus $M_i \Downarrow c_i$ and so $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

5. $M = (\mathsf{fetch}\ P)$. Since M evaluates, $\texttt{interp}(P, \rho, \sigma) = (l_0, \sigma_0)$ and by Theorem 6, $\Re(l_0, \bar{l}', \bar{\rho}', \sigma_0)$. Since $M' \geq^* \mathsf{valof}(M, \rho, \sigma)$, it follows that $M' = (\mathsf{fetch}\ P')$ for some P' and $P' \geq^* \mathsf{valof}(P, \rho, \sigma)$. By induction,

$$P' \Downarrow d' \geq^* \mathsf{valofcell}(l_0, \sigma_0).$$

Note that $\sigma_0(l_0) = \mathsf{susp}(l_1)$ or $\mathsf{rec}(l_1, f)$; we consider the first case here and leave the other to the reader. Since $\sigma_0(l_0)$ is a suspension, it follows from Lemma 20 that $\mathsf{valofcell}(l_0, \sigma_0) = (\mathsf{store}\ Q)$ for some Q. Thus, $d' = (\mathsf{store}\ Q')$ for some Q'. There are now two subcases:

(a) $\sigma_0(l_1) = \mathsf{thunk}(R, \rho')$. There are two subcases depending on the reference count of l_0.

i. $\mathsf{refcount}(l_0, \sigma_0) = 1$. Then $\mathtt{interp}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0))) = (l', \sigma')$. Since the state $S = (l_0, \bar{l}', \bar{\rho}', \sigma_0)$ is regular, it follows that l_0 is not reachable from ρ'—otherwise, there would be an illegal cycle in the memory graph induced by S. Thus,

$$Q = \mathsf{valof}(R, \rho', \sigma_0) = \mathsf{valof}(R, \rho', \mathsf{dec}(l_0, \sigma_0)),$$

and hence by induction

$$Q' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma').$$

Thus, $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$ as desired

ii. $\mathsf{refcount}(l_0, \sigma_0) > 1$. Then $\mathtt{interp}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0]))) = (l_2, \sigma_1)$ and $(l', \sigma') = (l_2, \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]))$. Note that because the state $S = (l_0, \bar{l}', \bar{\rho}', \sigma_0)$ is regular, neither l_0 nor l_1 is reachable from ρ'—otherwise, there would be an illegal cycle in the memory graph induced by S. Thus,

$$Q = \mathsf{valof}(R, \rho', \sigma_0) = \mathsf{valof}(R, \rho', \mathsf{dec}(l_1, \mathsf{dec}(l_0, \sigma_0[l_0 \mapsto 0])))$$

and hence by induction

$$Q' \Downarrow c \geq^* \mathsf{valofcell}(l_2, \sigma_1).$$

Since $\Re(l', \bar{l}', \bar{\rho}', \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)]))$ holds by Theorem 6, l_0 is not accessible from l_2. Thus,

$$\mathsf{valofcell}(l_2, \sigma_1) = \mathsf{valofcell}(l_2, \mathsf{inc}(l_2, \sigma_1[l_0 \mapsto \mathsf{susp}(l_2)])) = \mathsf{valofcell}(l', \sigma')$$

and so $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

(b) $\sigma_0(l_1) \neq \mathsf{thunk}(R, \rho')$. This case is straightforward and left to the reader.

6. $M = (\mathsf{share}\ x, y\ \mathsf{as}\ P\ \mathsf{in}\ Q)$. Since M evaluates,

$$\mathtt{interp}(P, \rho \,|\, P, \sigma) = (l_0, \sigma_0)$$
$$\mathtt{interp}(Q, (\rho \,|\, Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0)) = (l', \sigma')$$

Since $M' \geq \mathsf{valof}(M, \rho, \sigma)$, it must be the case that $M' = (\mathsf{share}\ x, y\ \mathsf{as}\ P'\ \mathsf{in}\ Q')$ for some terms $P' \geq^* \mathsf{valof}(P, \rho \,|\, P, \sigma)$ and $Q' \geq^* \mathsf{valof}(Q, \rho \,|\, Q, \sigma)$. By induction,

$$P' \Downarrow d \geq^* \mathsf{valofcell}(l_0, \sigma_0).$$

Let $\sigma_1 = \mathsf{inc}(l_0, \sigma_0)$. By Lemmas 24 and 25,

$$Q' \geq^* \mathsf{valof}(Q, \rho_2, \sigma) \geq^* \mathsf{valof}(Q, \rho_2, \mathsf{inc}(l_0, \sigma_0)).$$

Hence, $Q'[x, y := d] \geq^* \mathsf{valof}(Q, (\rho \,|\, Q)[x, y \mapsto l_0], \mathsf{inc}(l_0, \sigma_0))$. By induction,

$$Q'[x, y := d] \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma').$$

Thus, $M' \Downarrow c \geq^* \mathsf{valofcell}(l', \sigma')$ as desired.

7. $M' = (\mathsf{dispose}\ P\ \mathsf{before}\ Q)$. Similar to the previous case and hence omitted.

This completes the proof of the second claim and hence the proof of the theorem. ■

References

[Abr] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science.* To appear.

[AH87] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[App92] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Bak88] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(7):11–20, 1988.

[BBdH92] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. Term assignment for intuitionistic linear logic. Announced on the **Types** electronic mailing list, 1992.

[BGS90] V. Breazu-Tannen, C. Gunter, and A. Scedrov. Computing with coercions. In M. Wand, editor, *Lisp and Functional Programming*, pages 44–60. ACM, 1990.

[BHY88] A. Bloss, P. Hudak, and J. Young. An optimizing compiler for a modern functional programming language. *Computer Journal*, 31(6), 1988.

[CGR92] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 139–150. ACM, 1992.

[Col60] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[DB76] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[Des86] Joëlle Despeyroux. Proof of translation in natural semantics. In *Proceedings, Symposium on Logic in Computer Science.* IEEE, 1986.

[Fel91] A. Felty. A logic program for transforming sequent proofs to natural deduction proofs. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, Lecture Notes in Artificial Intelligence, pages 157–178, Berlin, 1991. Springer-Verlag.

[Gab85] R. P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[GG92] B. Goldberg and M. Gloger. Polymorphic type reconstruction for garbage collection without tags. In W. Clinger, editor, *Lisp and Functional Programming*, pages 53–65. ACM, 1992.

[GH90] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343, 1990.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Sci.*, 50:1–102, 1987.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. The MIT Press, 1992.

[Hol88] S. Holmstrom. Linear functional programming. In T. Johnsson, S. Peyton-Jones, and K. Karlsson, editors, *Implementation of Lazy Functional Languages*, pages 13–32, 1988.

[How80] William A. Howard. The formulae-as-types notion of construction. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987. (Preliminary version appeared in Proceedings 1986 ACM Conference on LISP and Functional Programming, August 1986, pp. 351-363).

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lect. Notes in Computer Sci.*, New York, 1987. Springer-Verlag.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Sci.*, 59:157–180, 1988.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In S. L. Graham, editor, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154. ACM, 1993.

[LD93] J. L Lawall and O. Danvy. Separating stages in the continuation-passing style transformation. In S. L. Graham, editor, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 124–136. ACM, 1993.

[LM92] Patrick Lincoln and John C. Mitchell. Operational aspects of linear lambda calculus. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 235–247, 1992.

[Mac91] I. Mackie. Lilac: A functional programming language based on linear logic. Master's thesis, University of London, 1991.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* The MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[O'H91] P. W. O'Hearn. Linear logic and interference control (preliminary report). In D. H. Pitt, editor, *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 74–93, Berlin, 1991. Springer-Verlag.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Sci.*, 1:125–159, 1975.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[Pot77] Garrel Pottinger. Normalization as a homomorphic image of cut-elimination. *Annals of Mathematical Logic*, 12(3):223–357, 1977.

[PS91] S. Purushothaman and J. Seaman. An adequate operational semantics of sharing in lazy evaluation. Technical Report PSU-CS-91-18, Pennsylvania State University, 1991.

[Sco] D. S. Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. Unpublished manuscript, 1969.

[Wad90] P. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods.* North Holland, 1990.

[Wad91a] P. Wadler. There is no substitute for linear logic. Manuscript, 1991.

[Wad91b] Philip Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 255–273. ACM, 1991.

[WHHO92] D. S. Wise, C. Hess, W. Hunt, and E. Ost. Uniprocessor performance of reference-counting hardware heap. Unpublished manuscript, 1992.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing. The MIT Press, 1993.

[WO92] M. Wand and D. P. Oliva. Proving the correctness of storage representations. In W. Clinger, editor, *Lisp and Functional Programming*, pages 151–160. ACM, 1992.

[Zuc74] J. I. Zucker. Cut-elimination and normalization. *Annals of Mathematical Logic*, 1(1):1–112, 1974.