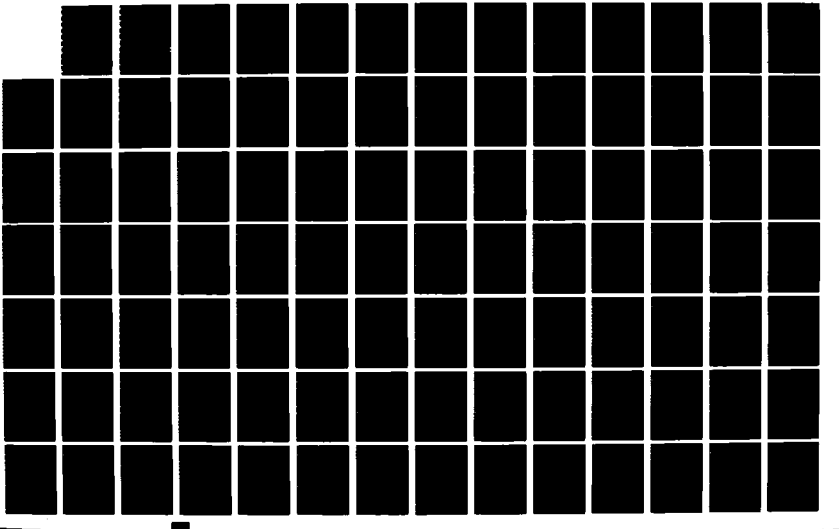
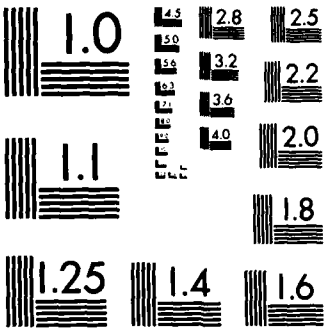
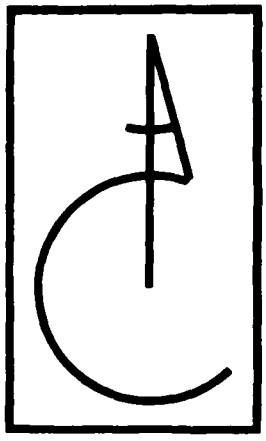
AD-A171 671 FORMAL TECHNIQUES FOR SPECIFICATION AND VALIDATION OF TACTICAL SYSTEMS(U) MASSACHUSETTS COMPUTER ASSOCIATES INC WAKEFIELD 02 JUN 86 CADD-8606-0203 1/3

UNCLASSIFIED DAAK80-81-C-0072 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



Final Technical Report
Contract No. DAAK80-81-C-0072
(Formal Techniques for Specification and Validation of Tactical Systems)

Massachusetts
Computer Associates, Inc.

MASSACHUSETTS
COMPUTER ASSOCIATES, INC.
26 PRINCESS STREET, WAKEFIELD, MASS. 01880 • 617/245-9540

Final Technical Report
Contract No. DAAK80-81-C-0072
(Formal Techniques for Specification and Validation of Tactical Systems)

June 2, 1986

CADD-8606-0203

Submitted to:

U. S. Army Communications & Electronics Command
Fort Monmouth, N. J., 07703-5008

Attn: E. Kivior
AMSEL-PC-S-A-CO(KIV)

Massachusetts Computer Associates, Inc. (COMPASS) is pleased to submit this Final Technical Report for Contract DAAK80-81-C-0072, on Formal Techniques for Specification and Validation of Tactical Systems.

The primary results of this basic research program are embodied in a published paper, "Object-Oriented Subsystem Specification", which is attached as Appendix I.

A major objective of this research was to initiate a transfer of formal specification technology into the design and development of tactical systems. To this end, the research was carried out in collaboration with engineers from CENTACS, who participated in a number of experimental case studies over the duration of the project. One of these was documented in a separate paper, attached as Appendix II; this describes our collective first attempts to formalize the design for a (hypothetical) "tactical situation - reporting system", before either the formal specification techniques or our informal conception of that system had fully matured. Essentially the same example is considered in the final paper.

The starting point for this reasearch was the mathematical approach to system specification originated by J. R. Abrial at Oxford University, which has come to be known as "Z". Much of the work on this contract was carried out in direct collabortion with the Programming Research Group at Oxford. A collection of papers on Z from this group, reflecting the state of their work at that stage, is attached as Appendix III.

Appendix I

OBJECT-ORIENTED SUBSYSTEM SPECIFICATION

S. A. Schuman & D. H. Pitt

CA - 8606 - 0202

June 2, 1986

To appear in: Meertens (ed.),
Proceedings of the IFIP Working Conference on Program Specification and Transformation
(North-Holland, 1986).

This research was supported in part by the U.S. Army Communications and Electronics Command, Fort Monmouth, N.J., under Contract Number *DAAK80-81-C-0072*.

OBJECT-ORIENTED SUBSYSTEM SPECIFICATION

S.A. Schuman *
University of East Anglia

D.H. Pitt †
University of Surrey

1. Introduction

The aim of this paper is to introduce a rigorous, mathematically based notation for supporting the earliest phases of the software design process, sometimes referred to as "systems architecture". Our objective is the development of a formal framework that could be applied successfully in practice. As such, we are consciously following in the footsteps of VDM [Bjørner & Jones, 1978] and [Jones, 1980]. The particular technique presented here is a variant on the approach originated at Oxford by J-R. Abrial, which has come to be known as "Z"; this is now documented in [Suffrin *et al.*, 1985] and [Hayes, 1986].

Underlying much of the current work in this area is the well established structuring principle of *data abstraction*, which implies some basis for modular decomposition into separately specifiable sub-units. Over the years, two somewhat distinct "schools of thought" have emerged as to the proper basis for applying such principles "in the large". We argue that this apparent divergence reflects a fundamental (technical) distinction, between *values* as opposed to *objects*. Quite obviously, both concepts have an important rôle to play.

In the case of values this decomposition is based upon "abstract data types", which serve to characterise some domain of interest in terms of certain constructor and selector functions. The abstraction then involves specifying an *equality relation* over such (immutable) values. The so-called "algebraic" approach, initiated by [Guttag & Horning, 1978] and [Goguen *et al.*, 1978] is therefore especially appropriate in this context; it is also the focus of most recent research on formal specification methods.

A "class" of abstract objects, in contrast to a "type" of values, serves to encapsulate the definition of some internal *state* in conjunction with an associated set of access operations for querying and/or updating any individual instance of that class. Thus "axiomatic" methods involving pre- and postconditions, expressed in terms of a suitable state model, would appear to be the most natural approach for specifying such abstractions. As foreshadowed by [Dahl, 1972] and [Hoare, 1972], the "object-oriented" paradigm originally embodied in SIMULA 67 (and subsequently incorporated into a number of more recent programming languages) has proved to be an extremely effective technique for decomposing and reasoning about complex systems. Our goal here is to provide a useful counterpart to those facilities at the level of formal specifications.

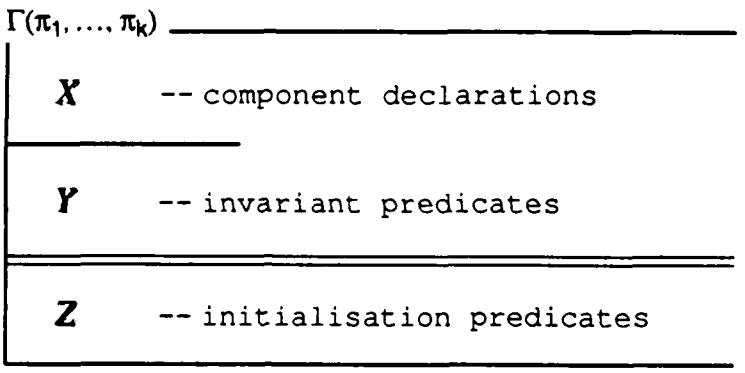
Section 2 of this paper contains an informal overview of our notations and conventions. Section 3 gives a more formal treatment, including rules of inference for reasoning about the *behaviour* implied by such specifications. Section 4 concludes with the development of an extended example, specifying the architecture of a distributed information system.

Authors' addresses:
* S.A.Schuman, School of Information Systems, University of East Anglia, Norwich NR4 7TJ, England;
† D.H. Pitt, Department of Mathematics, University of Surrey, Guildford, Surrey, GU2 5XH, England.

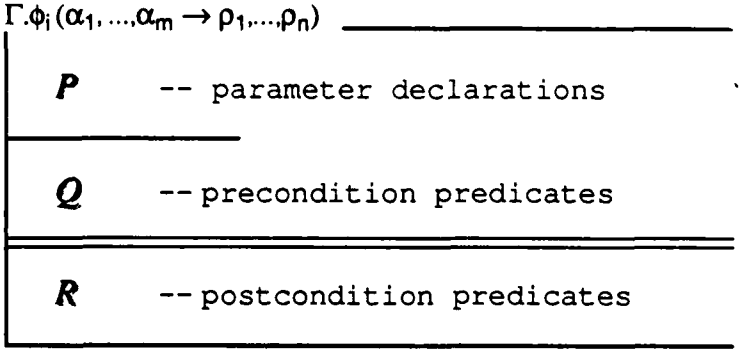
2. Notations and Conventions

Structurally, the specification of an object class comprises two distinct kinds of definition, *viz.*

(i) A single *state schema*, which has the following general form:



(ii) An associated set of *event schema*, each of which is formed as follows:



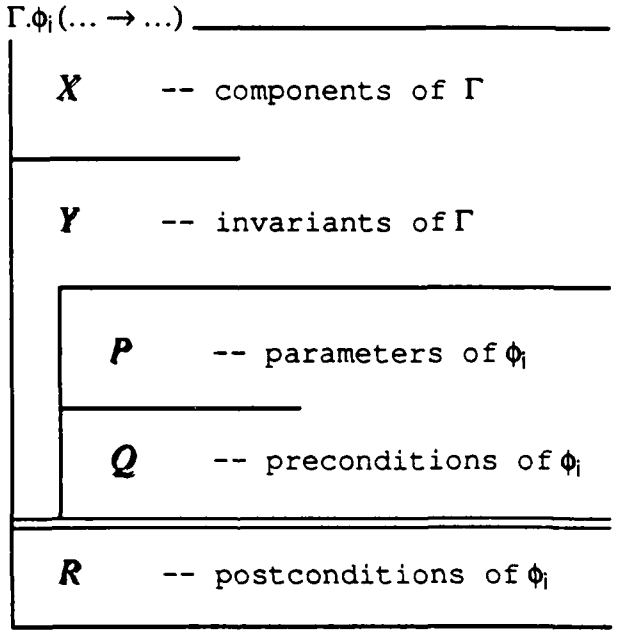
The header of a state schema identifies the class in question (i.e. it gives it a name), as well as naming any formal parameters for that class. The event schema headers introduce the name of each separate operation, in conjunction with names for any input and/or output parameters figuring in its signature. The association alluded to above is indicated by simply prefixing the class name Γ to the individual operation names $\phi_1 \ \ldots \ \phi_p$, which are for the moment assumed to be unique in the context of a given specification. No provisions for explicitly grouping such schema texts into larger "modules" are considered here.

Inside a schema definition the different lists of declarations or predicates are normally set out vertically, as independent items appearing one under another, wherein the order is not significant; alternatively, several such items may be written on the same line, separated by semicolons. It often arises that a particular list is empty, whereupon the corresponding part of the schema may be omitted entirely. These latter conventions will be illustrated by a succession of examples, to follow.

This "box-like" presentation of schema definitions was adopted mainly as a means for setting off the enclosed formal text from its surrounding (natural language) explanations – without which a specification is neither complete nor comprehensible. Within this minimal and strictly syntactic framework, the actual content of every such specification is expressed using essentially the traditional notations of predicate calculus and classical (ZF) set theory, extended as and when necessary by additional constructs defined in terms of those notations. Although we are assuming one specific development of that underlying formalism [Abrial 1982], any other axiomatic formulation of this well understood mathematical basis would in principle be equally appropriate.

Whilst we intend that these specifications should have a fully rigorous (and therefore formal) interpretation, it is perfectly acceptable – and usually quite helpful – to start out with a much more "operational" perspective. The state schema, which is where one begins in order to specify some class of interest, might well be viewed as defining an abstract "data model" for all objects belonging to that class. Support for abstraction comes firstly from the fact that this model is expressed in terms of a "very high-level" and purely "declarative" language, in which the "data types" correspond to (presumably familiar) mathematical constructions as opposed to the more concrete structures normally provided by a programming language. Any formal parameters of the state schema stand for constant components, the values of which are independently fixed for each separate object instance. The remaining components declared within that schema may then be construed as "state variables" of an individual object, in that their values can be selectively modified as a consequence of applying particular operations. The invariant predicates assert that certain relationships amongst the component values must always hold; as such, they serve to impose constraints upon how the state of an object may be changed by any associated operation. Finally, the initialisation predicates establish whatever conditions are to be assumed at the outset, when an object of the class in question is first instantiated.

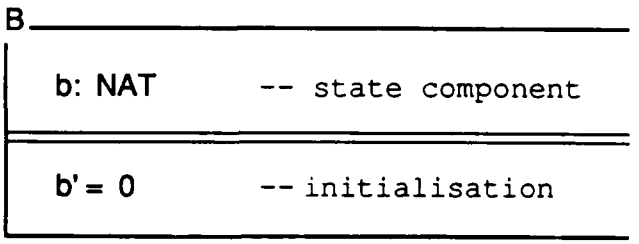
The requisite operations are then specified with respect to this abstract model. Each such definition is presented as a free-standing event schema (analogous to an independent function or procedure), but expressed as if it were textually "embedded" into the scope of the corresponding state schema:



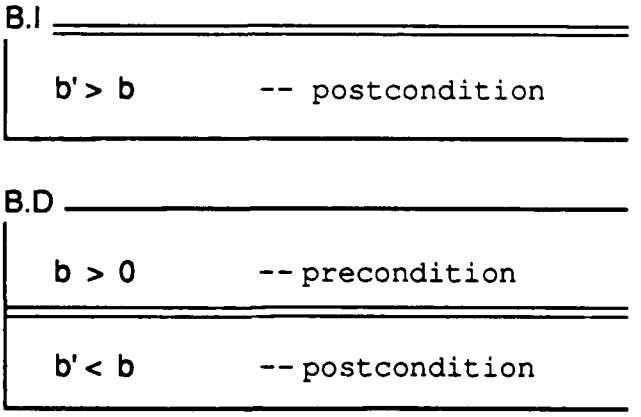
Thus all component names appearing therein refer to the current values of some representative instance; the actual object will be designated whenever this operation is subsequently applied. Names figuring in its signature denote additional constants, standing for the values which are either input (as arguments) or output (as results) in the context of any given application. In general, the precondition predicates will range over both component and parameter names. They thereby serve not only to restrict applicability of the operation to certain object states but also to constrain the possible argument and result values relative to those states: such an event may occur if and only if all of these conditions are satisfied simultaneously.

Explicit effects of this event (upon individual state variables) are established by the postcondition predicates, wherein we adopt the common convention of "dashing" (or "priming") a component name to denote its value *after* the event; for symmetry, the same convention is used in the initialisation predicates of the state schema, where a dashed name stands for the initial value of that component (after instantiation).

About the simplest possible example is that of a running *balance*, which may increase or decrease by some arbitrary but observable amount. Such balances can be modelled by a single state component, corresponding to their current value:

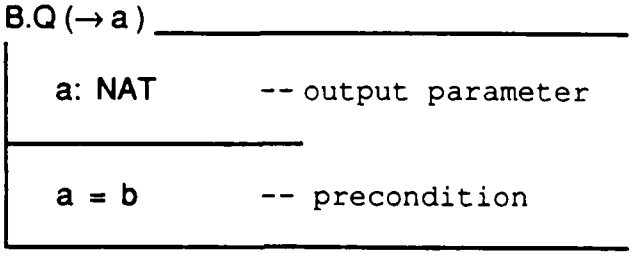


For purposes of illustration this value is specified in terms of natural numbers rather than, say, integers (or some more specialised unit of account). Thus there is no *a priori* upper bound on any balance, but "overdrafts" are precluded. As the invariant is empty, no further constraints are imposed. Within this domain, the initial value of every balance is zero. The operations of interest, namely increasing or decreasing a balance, may then be defined quite succinctly:



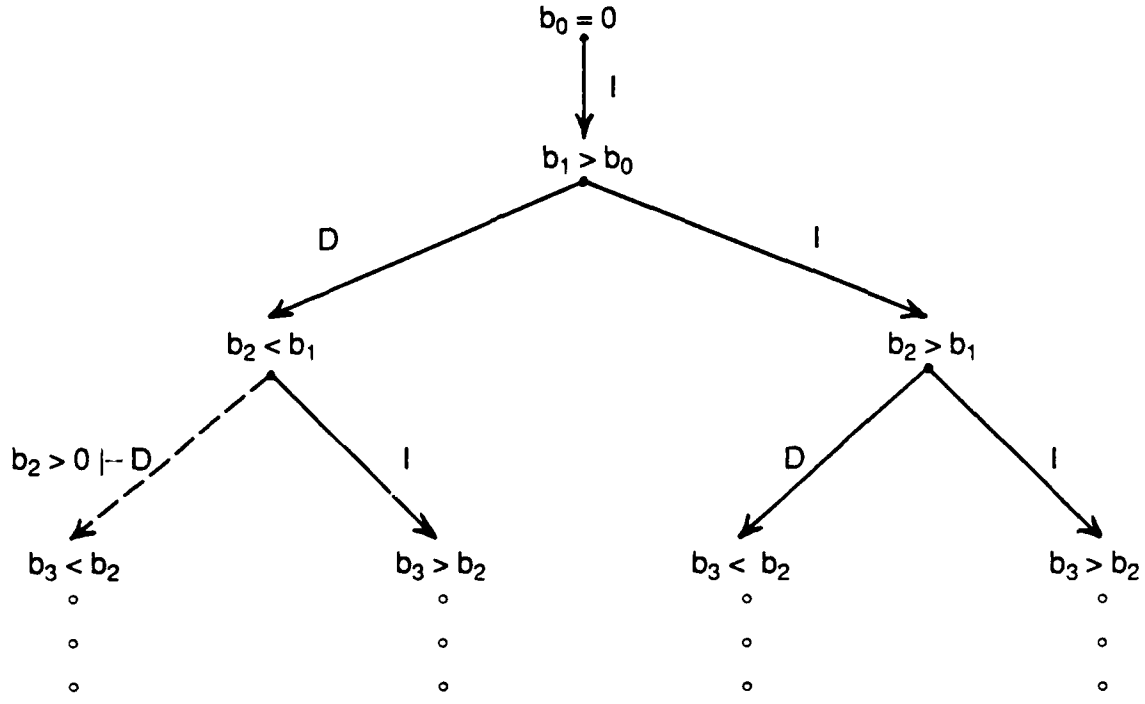
No precondition is needed for the increase operation. In the case of a decrease operation, however, the desired postcondition cannot be established consistently without requiring a non-zero balance before the event (since there is no natural number $b' < 0$ unless $b > 0$).

Occurrences of these events are only observable in the sense that an increase always results in a new balance which is strictly greater than it was beforehand, whereas a decrease has the opposite effect. But the amount by which the balance changes in each instance is (deliberately) left indeterminate, so there is no means of influencing what choice is made on any given occurrence. An operation to query the current balance (without changing its value) might also be provided, if only to avoid the pitfalls of inadvertently attempting to decrease a balance that has gone to zero:



This operation has no effect (as indicated by the absence of any postconditions), and is always applicable. But the only reason for defining such events would be to *encapsulate* completely the abstract model in question. It is often more productive to defer these considerations until later stages in the specification process, when the contextual requirements have been firmly established.

The dynamic behaviour characterised by such specifications is most easily visualised in the form of a "decision tree". Accordingly, the initial pattern for a balance belonging to the class specified above (ignoring the superfluous query function) might be depicted as follows:



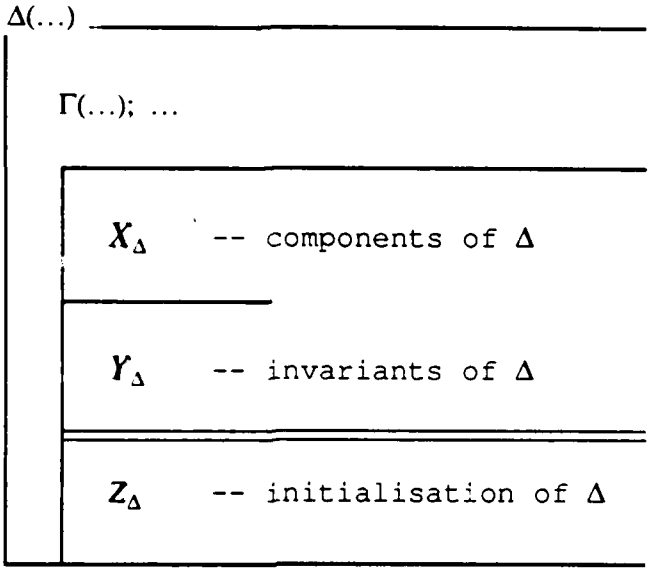
The branches of this tree are labelled by (denotations of) alternative events which may occur at distinct points in any possible *history* and those points, the nodes, are annotated with predicates reflecting the postcondition established by the immediately preceding operation; the root annotation corresponds to the initialisation predicate in the state schema. Within these annotations, component names are subscripted so as to differentiate successive points in "time" (indicated as depth in the tree). A particular history is then given by the sequence of events, or *trace*, along some path from the root of such a tree to any reachable node. The predicates associated with each successive node in that history are to be interpreted as incremental (or cumulative) assertions about the object state up to that point.

At the first step in the behaviour for this class, the only possible event is the increase operation I, which follows directly from the initial condition for the class B and the precondition of the decrease operation D. Thereafter either I or D events may in general occur, and indeed an increase is always possible. If, however, the most recent event in any trace was a decrease operation, then the admissibility of another D event at that point becomes indeterminate because its precondition may or may not be satisfied, depending on how this specification is subsequently refined (or implemented). Indeterminacy of this sort is shown above as a dotted branch, to suggest that the ensuing subtree is conditionally present; the labels appearing on such branches include an antecedent as part of the event denotation, which predicates are obtained directly from the corresponding precondition.

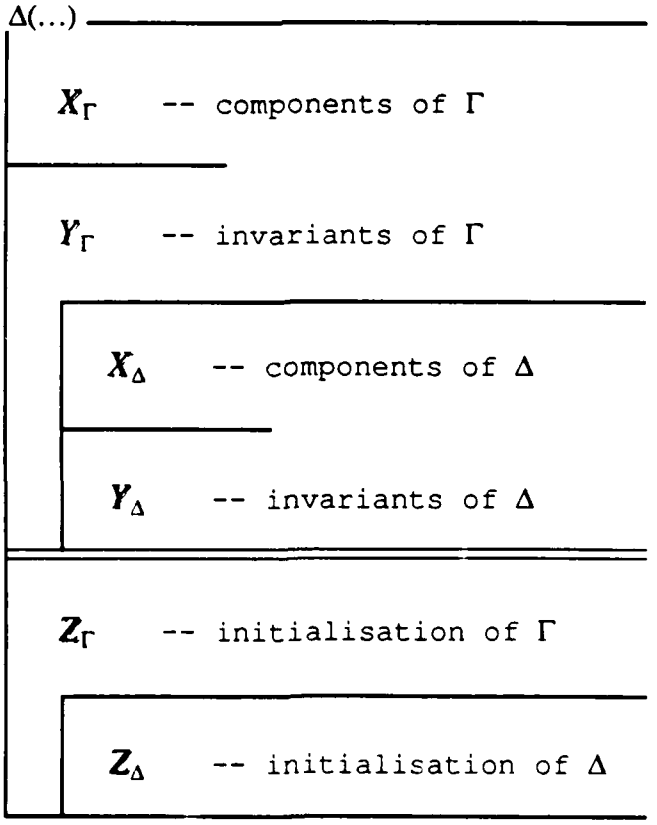
The sole purpose of this (admittedly artificial) exercise in underspecification was to defer all commitments that would in any way *quantify* the actual increase or decrease to a running balance – so as to encompass the broadest possible class of potential refinements. These include subclasses for which this indeterminate behaviour is fully resolved (that is, where every "conditional branch" in the original history is either provably admissible or, alternatively, ruled out altogether), as well as ones wherein the indeterminacy is still present (but dependent upon more specific conditions, which are only meaningful in the context of that particular subclass).

As a general rule, refining the specification of some class Γ involves introducing a new subclass Δ, for which further properties are specified *relative* to that given base. The definition of such a subclass takes the form of a state schema which simply identifies the relevant base definition(s).

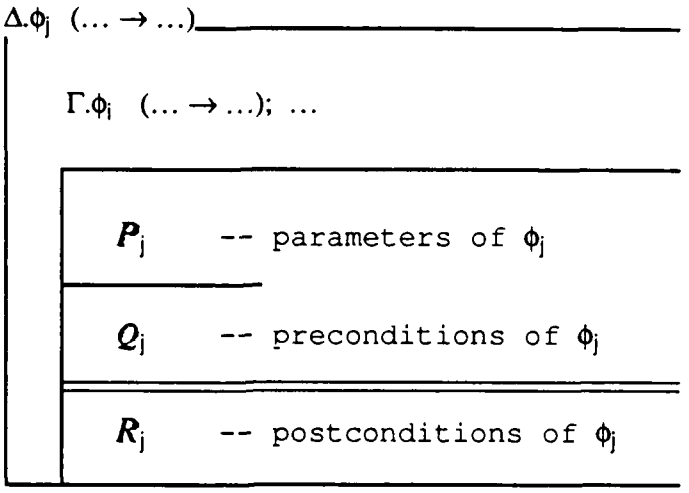
The actual *extensions* are then presented within an embedded subschema:



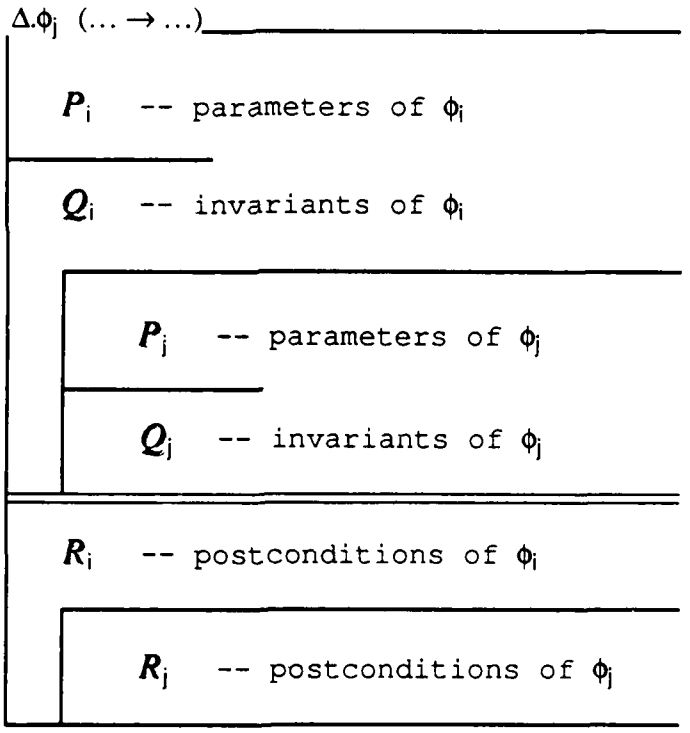
This refinement may be understood to stand for a textual expansion of the definition as written (wherein the nesting structure is preserved):



Operations associated with the new subclass would normally be defined in terms of existing operations on the base class(es). Such *promotions* can be specified in an analogous fashion, as refinements to the corresponding event schema:



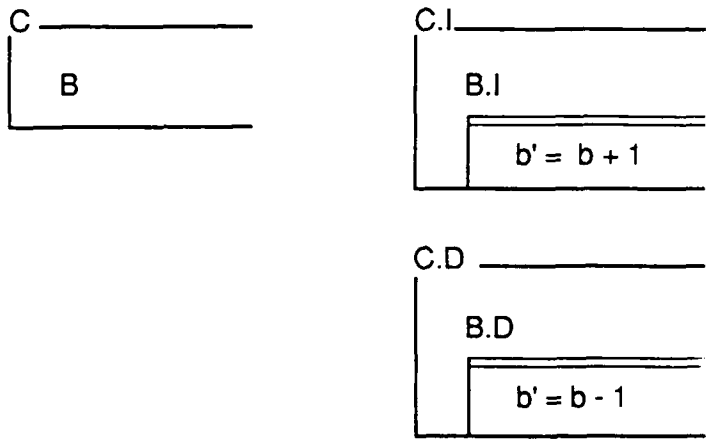
Again, the intended interpretation is most easily visualised in terms of its textual expansion:



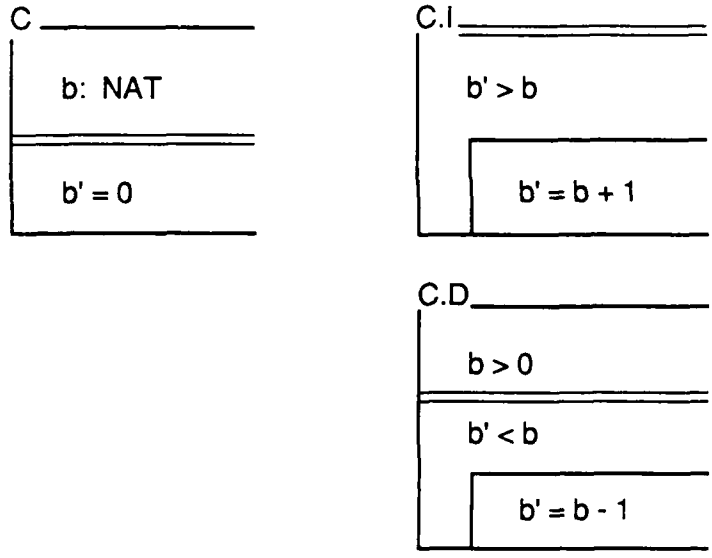
Observe that all parameter declarations and invariants for a designated base operation ϕ_i are effectively "inherited" within this definition; thus, only additional parameters or constraints associated with the new operation need to be specified explicitly. This applies equally to the context of postconditions.

As suggested by the ellipses above, our conventions allow for so-called "multiple inheritance", where more than one base may be identified in conjunction with such refinements. This provides a means for combining several (compatible) definitions at the same level, and thereby specifies a subclass which belongs to many different classes all at once.

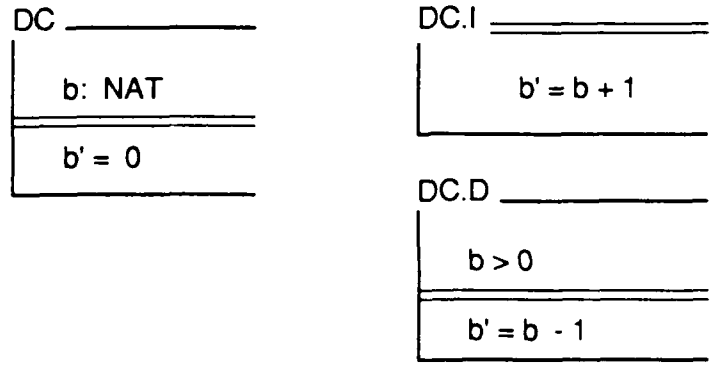
An obvious refinement of the previously specified balances is the subclass of *counters*, for which the abstract model is identical (whence the state schema is in effect just a renaming of that class). The operations for incrementing or decrementing such a counter could then be expressed as promotions of the corresponding increase and decrease events, but with stronger postconditions:



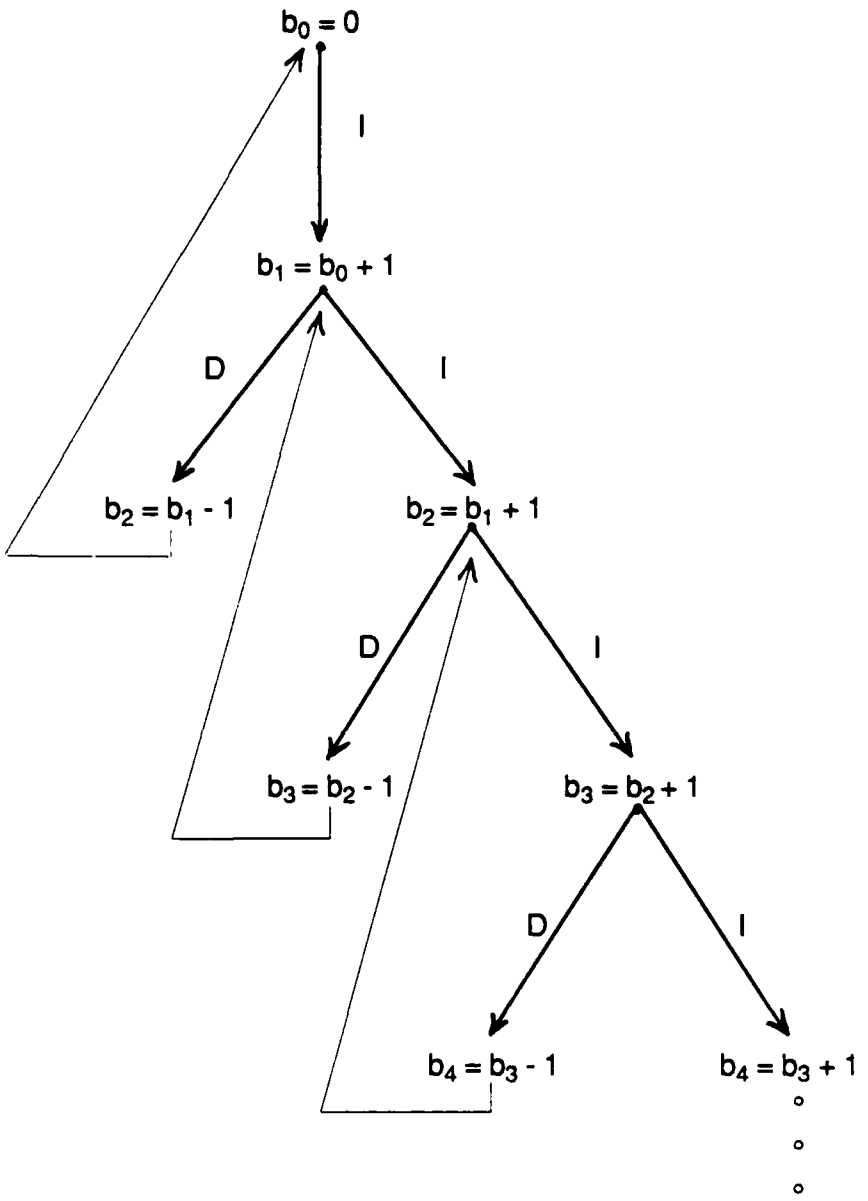
Both the initial conditions for this new subclass as well as the necessary precondition for the decrement operation are thereby inherited, as can be seen by expanding its specification in full:



It emerges that the weaker "observability" postconditions of the balance operations are also inherited. These properties would be (trivial) theorems of the event definitions if such counters had instead been specified directly, without reference to the class of balances:



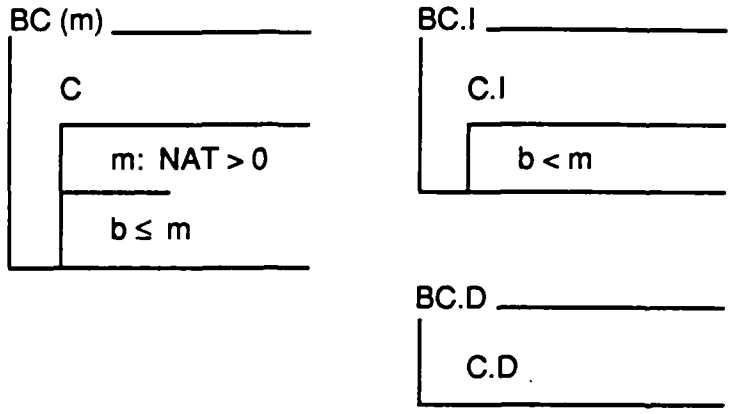
This example is of course so simple that it hardly matters which way one expresses the specification. The behaviour is the same in both cases:



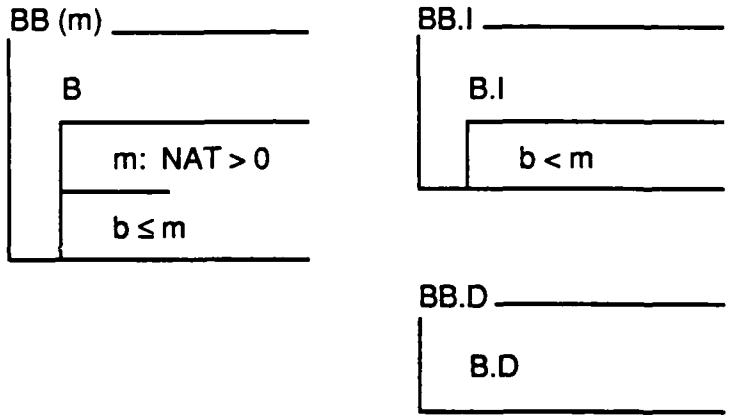
Not surprisingly, the increment and decrement operations on counters correspond respectively to the successor and predecessor functions on natural numbers – which is precisely what was specified! (The backward-pointing edges above, indicating that the nodes in question may be equated on the basis of their associated predicates, suggest various recursive formulations for these functions.)

It should be observed that this counter behaviour is in fact a fully resolved (albeit still unbounded) refinement of the balance behaviour specified at the outset, since the more restrictive postconditions now determine whether a decrement event is or is not admissible at every point.

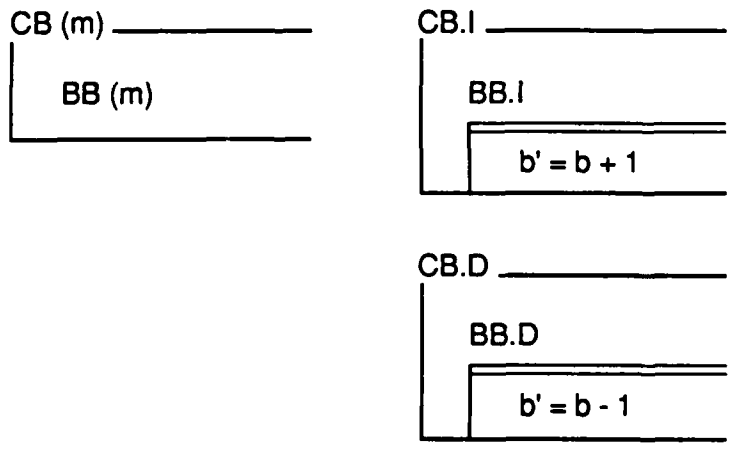
Should one subsequently wish to impose an upper bound upon such counters, this could be specified as a further refinement of the unbounded class, corresponding to a restriction on the original (balance) model. The increment operation then requires a precondition, so as to preserve this new invariant, whereas the decrement operation is just a direct promotion:



Alternatively, this might have been approached by first specifying a subclass of *bounded balances*:

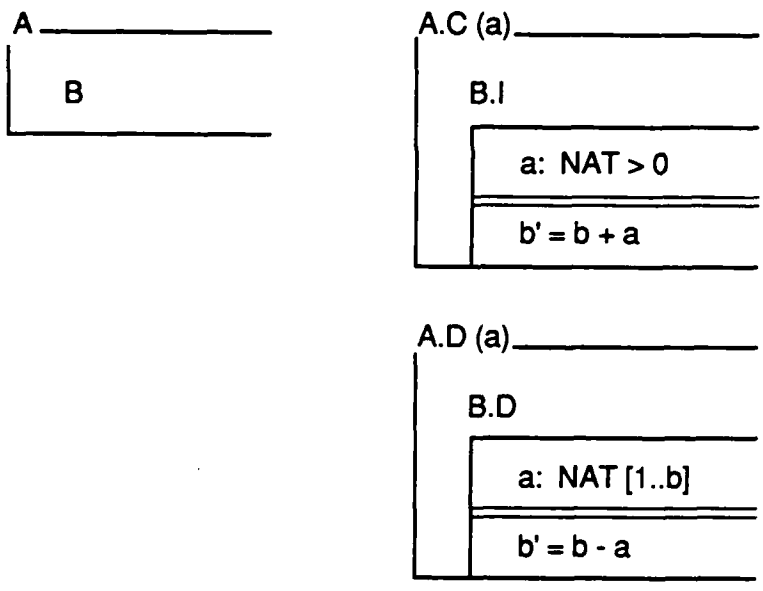


This new basis can then be refined, as before, by adding the postconditions specific to counters:



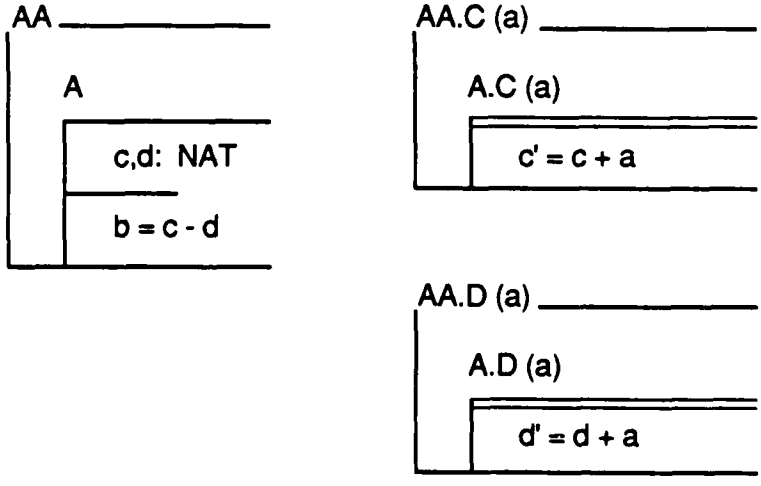
With either approach the resultant behaviour is indeed bounded, but not fully resolved – in that the admissibility of an increment operation now depends upon the value of the formal parameter m, which may be different for each separate instantiation of both BC and CB.

Another interesting subclass of running balances is the more traditional notion of an *account* that may be credited or debited by some specific amount, which is supplied as an argument each time one of these operations is invoked. Such accounts could also be specified by refinement:



The amount to be credited or debited must be non-zero in every case, in order to satisfy the observability properties imposed in the context of balances; moreover, the amount of a debit must not exceed the current balance since overdrafts were ruled out from the beginning. Thus the admissibility of a credit or debit operation is always indeterminate, in that it depends upon the actual value of an input parameter.

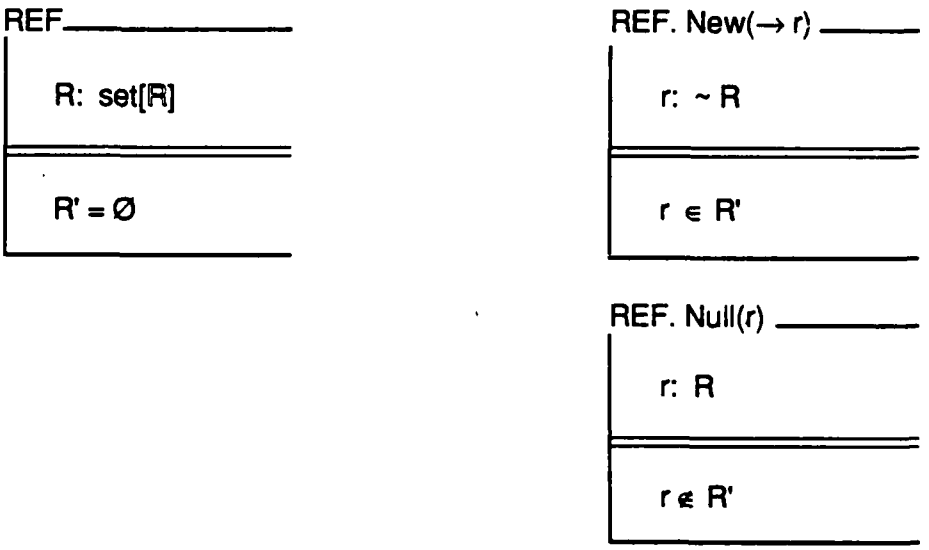
Refinement of a class specification more typically involves proper extension, in the sense of adding new components to the underlying model (some of which may be wholly or partially redundant). This corresponds to specifying additional "attributes" for a given subsystem, possibly (but not necessarily) associated with its implementation. But such extensions might also be introduced solely to reason about the implied behaviour. Suppose, for example, that one wished to establish that the sum of all debits to an account never exceeds the total of actual credits. A subclass of *audited accounts* is probably more appropriate for these purposes:



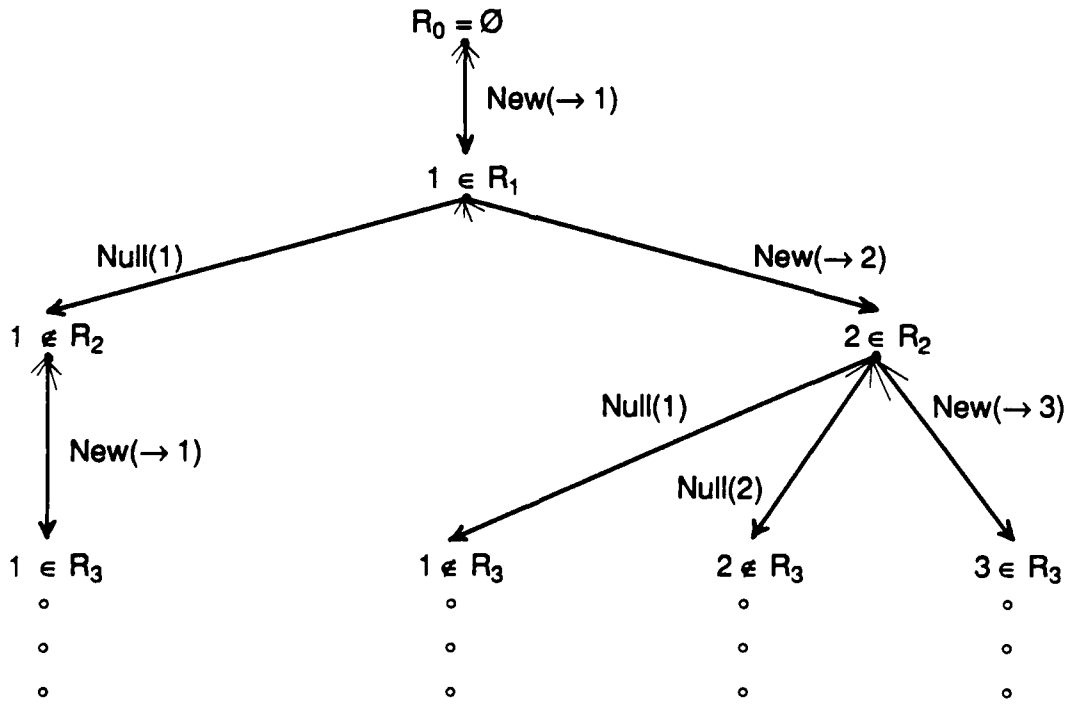
3. Reasoning About Specifications

In this section, we outline the formal basis for reasoning about subsystem specifications, both *statically* (considering only the individual schema comprising a class definition) and *dynamically* (over their implied behaviour); in particular, we develop the special rules of "historical" inference which underlie our conventions.

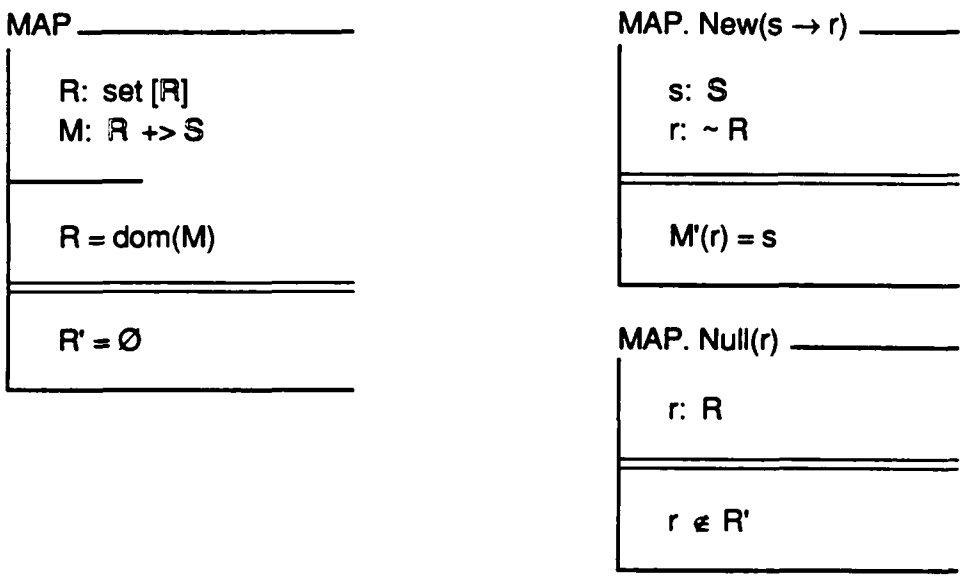
For these purposes we shall also introduce another simple but much more typical example, dealing with an issue that arises in the design of a great many systems, viz. *unique identification*. The class of interest therefore corresponds to a "generator" for the requisite "reference" values, which may well be abstracted merely as elements drawn from some entirely arbitrary "carrier set", say $\mathbb{R}$. The state of such a generator would then be modelled as a (finite) subset R of this given carrier set, comprising the references outstanding at any point, where R is initially empty. Generation of a new reference is then some (indeterminate) selection from the complement of R (i.e. $\mathbb{R} \setminus R$, where $\setminus$ denotes set difference), whilst nullifying (or "freeing") a reference is just its deletion from R:



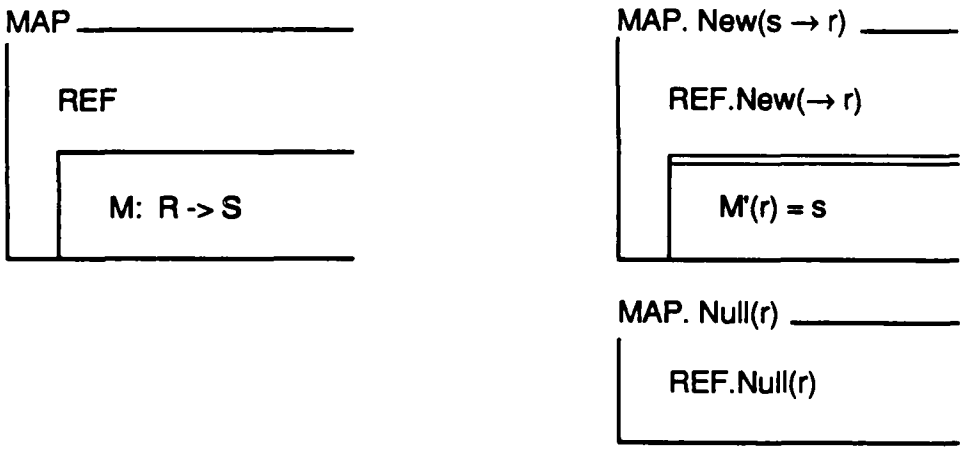
Taking $\mathbb{R}$ as NAT>0 (and showing other choices as "fan-outs"), the initial behaviour includes:



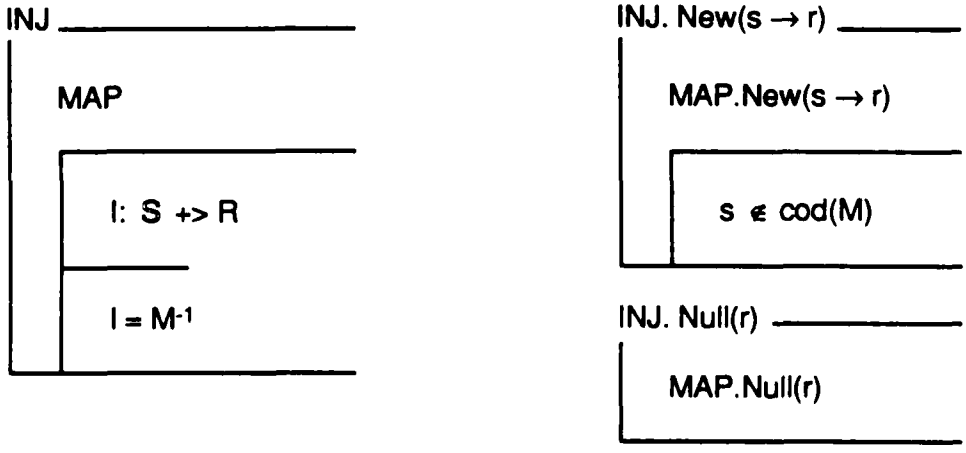
The most common application of such references is to identify (or "index") occurrences of values belonging to some other set, say S. This sort of association is normally modelled as a finite map (i.e. a partial function):



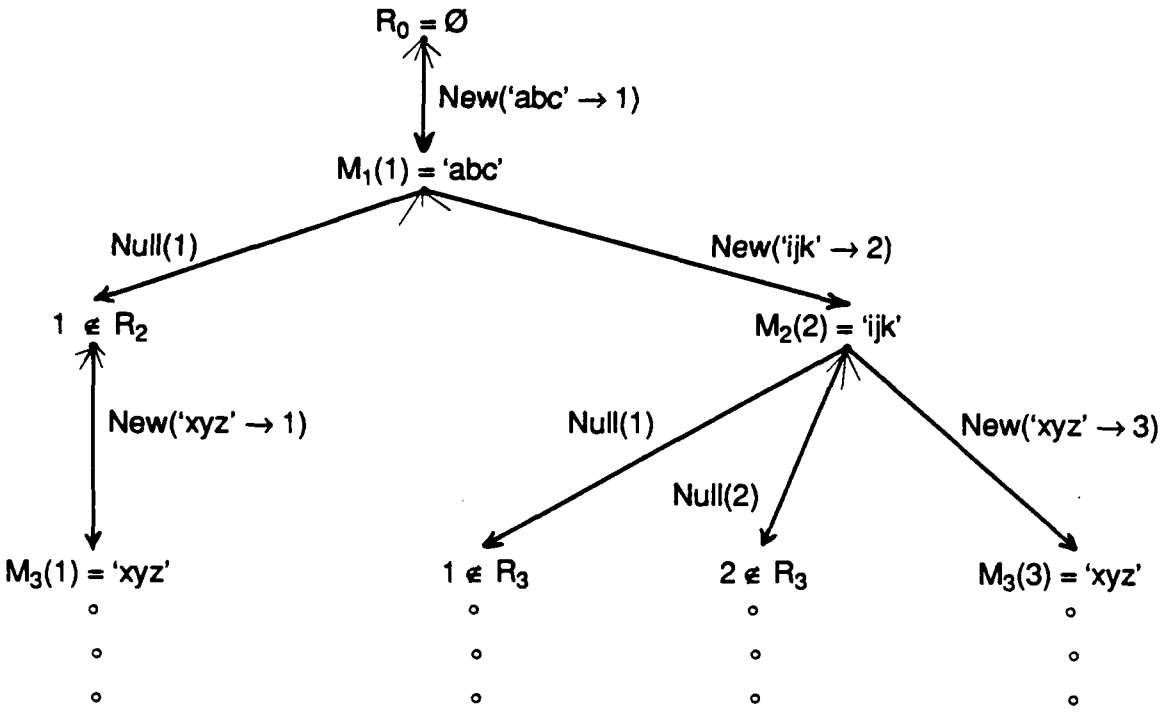
It would perhaps be more natural to specify this association directly as a new subclass of REF:



(Note that the map is now specified as a total function). This may be further refined by requiring that the association be *injective* (one-to-one), which is expressed by asserting that its converse is also functional:



Taking $\mathbb{S}$ to be *strings* (as in a "symbol table"), the initial behaviour of this subclass now includes:



Such behaviours are to be modelled in terms of traces of event denotations (the sequence of labels along any possible path in the decision tree depicted above), where every trace is associated to a corresponding history (predicates on the state, as suggested by the node annotations) from which properties holding "at that point" may then be inferred.

The different forms of schema definition making up a class specification are nothing more than a highly stylised framework for setting out the various predicates entering into any possible history. The predicates in question range over (mathematical) variables which are introduced by a list of declarations, as follows:

$$v_1 : T_1;\ \ldots;\ v_q : T_q$$

wherein $v_1 \ldots v_q$ are just the names of these variables and $T_1 \ \ldots \ T_q$ stand for their corresponding set-theoretic "types". Such types are expressed in terms of certain given or "generic" set names (herein denoted by letters from a distinguished alphabet $\mathbb{A} \ldots \mathbb{Z}$), the names of externally defined types (e.g. NAT) and the usual set constructions over such types – including products, partial or total functions and relations, finite subsets and sequences etc. Inside a schema definition, each variable so declared is thereby asserted to belong to some set having the particular structural properties associated with its specified type. Thus these assertions can be embodied into a set of predicates having the following form:

$$\{v_1 \in T_1, \ldots, v_q \in T_q\}$$

where $v_1 \ldots v_q$ are just free variables within that definition. As such, they may be implicitly quantified at the level of the schema as a whole; embedded declarations could then be thought of as corresponding to nested quantification.

The characteristic predicates for a given schema are derived by combining these conditions with additional predicates (over the same free variables) which are obtained from other parts of that schema definition. All such derived sets stand for the *conjunction* of their constituent predicates (whence the empty set is logically equivalent to *true*).

Two separate sets of predicates are derived from the state schema for a particular class, as follows:

- the *state predicates* $S = X \cup Y$
  (where X is obtained from the component declarations and Y from the state invariants);
- the *initial predicates* $I = S \cup Z$
  (where S is the state, given above, and Z is obtained from the initialisation predicates).

The state predicates are *undashed*, in that none of the free variables occurring therein are dashed. But the initial predicates are *mixed*, which is to say that they will in general contain both dashed and undashed variables. This is because initialisation is specified as a "pseudo-event" (and will therefore be discussed in that context, below).

Definition. A *model* for the state as specified in the state schema for a given class is some association of actual sets to the generic names appearing (directly or indirectly) within those declarations, together with any substitution of values from those sets (or constructed sets) for all free variables within that model.

The state so specified must be *consistent* in the usual sense: there is such a model, and values for the state variables within that model, satisfying all of the predicates in S (i.e. both types and invariants); otherwise, that specification is contradictory and no such class exists.

The operations ϕ_i specifically associated with a given class are expressed as relations between undashed and dashed names, standing respectively for the value of a state variable before and after some occurrence of the event in question. This convention implies a wholly independent set of free variables $v_1', \ldots, v_q'$ which have exactly the same declarations as their undashed counterparts. The corresponding types and invariants, denoted S', are therefore obtained by simply dashing each such undashed name occurring free within S. The characteristic predicates for each event ϕ_i are then derived from its schema definition as follows:

- the *guard predicates* $G = S \cup (P \cup Q)$
  (which incorporates the parameter declarations P and the preconditions Q);
- the *final predicates* $F = S' \cup R$
  (which introduces the dashed names in conjunction with the postconditions R);
- the *event predicates* $E = G \cup F$
  (which thereby relates the state values *before* and *after* any occurrence of that event).

Each associated event must also be consistent in its own right, meaning that there is a model (obtained in the same way as for a state) satisfying all of the predicates in E. The same applies to any specified initialisation, characterised by the set I as defined above. We shall speak of these characteristic sets as standing for what they serve to describe: "a state S", "a guarded state G", "a final state F", "an event E" etc. The consistency requirement on all such predicates is essentially static, in that it can be established for each separate schema in isolation using only axioms of the underlying set theory and classical rules of inference.

Implicit in every event is some "admissibility" condition, under which it is allowed to occur at all; this is given by simply positing the existence of a corresponding final state.

Definition. Let E be an event; then its *implied precondition*, denoted *ipc(E)*, is obtained by existentially quantifying over all dashed variable names occurring free within E. The event E is *admissible* (and may therefore *occur*) in any current state C which is consistent with *ipc(E)*.

It follows by construction that $ipc(E) \Rightarrow G \Rightarrow (P \wedge Q \wedge S)$, where G is the guarded state embodied in E; hence that event may only occur if its parameter declarations P and explicit preconditions Q (as well as the underlying state invariants S) are satisfied.

It is desirable, however, that a given event can indeed occur (i.e."complete" successfully) *whenever* its preconditions are satisfied, meaning that there must be a consistent final state for all admissible occurrences. This leads us to impose a much stronger consistency requirement.

Definition. Let W be a set of predicates, and V be the subset of undashed predicates within W; then W is *end-consistent* iff V is consistent and, for any current state in any model that satisfies all the predicates in V, there is a corresponding final state which satisfies all predicates in $W \setminus V$.

Consider, for example, some predicates over a state variable N, assumed to be of type set [NAT]:

(i) $\{ N \neq \emptyset,\ 1 \notin N' \}$

(ii) $\{ N \neq \emptyset,\ 1 \notin N',\ N' = N \}$

(iii) $\{ N \neq \emptyset,\ 1 \notin N',\ N' \setminus \{1\} = N \setminus \{1\} \}$

Both (i) and (iii) are end-consistent, whereas (ii) is not – because it fails to cover the case $N = \{1\}$.

We are especially interested in specifications for which end-consistency can also be established statically. A sufficient condition is that the event in question is what we shall call "well-formed".

Proposition. Let E be an event, and G be the guard for that event; then E is end-consistent if E is consistent and $G \Rightarrow ipc(E)$; in this case it follows that $ipc(E) \equiv G$.

The relation characterised by a mixed predicate W may be conceptually "inverted" through another renaming over its free variables, denoted as W°, which is obtained by specially superscripting all undashed variables (so that v° now stands for the "previous" value of v) and undashing all dashed names (so that v now stands for its new "current" value v'). Thus E° expresses, as an undashed predicate, the net "effect" once some event E has occurred, conditioned on the existence of a previous state in which that event was admissible.

Definition. Let E be an event; then its *weakest postcondition*, denoted $wpc(E)$, is obtained by existentially quantifying over all previous variable names (of the form v°) occurring free within E°.

The term "weakest" is used here to emphasise that this condition does not depend upon any more specific properties of that previous state. However, by construction we have that $wpc(E) \Rightarrow wpc(F) \Rightarrow (wpc(R) \wedge S)$, which is strong enough not only to establish the (potentially weaker) postconditions R , as explicitly specified within the corresponding event schema, but also to preserve the invariants associated with its underlying state$S$, since these are both included into the corresponding final state F for that event.

Part of the conciseness of our conventions comes from the fact that postconditions serving solely to "restore" the state invariants may be omitted, as they are present implicitly. More important is the omission of conditions which state only that some aspect of the state remains "unchanged", since these latter properties may be inferred (and are therefore inherited) from previous "history".

Definition. A *history* is a finite sequence of event predicates, $h = \langle E_1,\ldots,E_k \rangle$, where each E_i corresponds to an individual occurrence of some event belonging to the class in question.

We wish to consider situations in which $h = h^\circ \wedge \langle E \rangle$, for a given initial history h° and event E, such that a set B, comprising all predicates which hold after h°, characterises the previous state *before* that occurrence of E. For h to exist, E must be admissible at that point, implying that B is consistent with $ipc(E)$. The properties holding *after* this event are given by $wpc(E)$.

The weakest postcondition for any event depends upon how its explicit postconditions $r \in R$ are actually written. Compare the effect of different expressions over a state variable N of type set[NAT], according as some predicate b (say, $0 \in N$) is or is not known to hold in B:

r	$wpc(r)$	$wpc(b \wedge r)$
$1 \in N'$	$1 \in N$	$1 \in N$
$N' = N \cup \{1\}$	$1 \in N$	$0 \in N \wedge 1 \in N$
$1 \notin N'$	$1 \notin N$	$1 \notin N$
$N' = N \setminus \{1\}$	$1 \notin N$	$0 \in N \wedge 1 \notin N$

Such examples show that $wpc(p1 \wedge p2)$ is not in general equivalent to $wpc(p1) \wedge wpc(p2)$, even where the conjuncts of the latter are equivalent, although we do the following:

Proposition: If $p1 \Rightarrow p2$ then $ipc(p1) \Rightarrow ipc(p2)$ and also $wpc(p1) \Rightarrow wpc(p2)$, whence equivalent predicates have equivalent implied preconditions and weakest preconditions.

We would argue, however, that the main use of such forms as $N' = N \cup \{1\}$ and $N' = N \setminus \{1\}$ above is simply to "carry forward" history, which is tantamount to overspecification if the only intended effect of the operations in question is to insert or delete the value 1. It is for this reason that we have introduced inheritence of properties which can be inferred from history. A consequence is that the weakest postconditions for $(b \wedge r)$ above then become the same for each operation, i.e. $(0 \in N \wedge 1 \in N)$ on insertion and $(0 \in N \wedge 1 \notin N)$ on deletion, irrespective of how r is expressed.

In order to inherit any (undashed) predicates holding in B, the event E as obtained from its schema must be augmented by some set A of (mixed) predicates serving to re-establish those properties for the state variables after that event, as reflected in $wpc(E \cup A)$; but A must be such that this conjunction remains end-consistent. The strongest possible augments assert that nothing changes.

Definition. Let B be some previous state; then any predicate of the form $(v' = v)$, where v is an undashed variable name occurring free in B, is termed an *identity* for B. The set of all such predicates is denoted *identity(B)*.

This is the default, when R is empty. But if any "updates" at all are specified, $E \cup identity(B)$ will not be end-consistent. Some such identities may be appropriate (and indeed this is how the parameters of a schema are held constant, since their names cannot be written in dashed form); normally, however, a weaker set of "neutral" augments must be chosen.

Definition. Let B be some previous state; then a mixed predicate p over the state variables of B is termed *neutral* with respect to B iff $identity(B) \Rightarrow p$. The set of all such p is denoted *neutral(B)*.

For the example above, neutral predicates over N will have the general form $(N' \cap M = N \cap M)$, for some specific $M \subseteq NAT$, or equivalently $(N' \setminus \sim M = N \setminus \sim M)$; thus they include $(N' \cap \{0\} = N \cap \{0\})$.

Definition. Let B be some previous state and E be an event; then a predicate $p \in$ *neutral(B)* is termed *neutral* with respect to B and E iff the set $B \cup E \cup \{ p \}$ is end-consistent. The set of all such p is denoted *neutral(B,E)*.

Proposition. The set *neutral(B,E)* is closed under disjunction.

However, *neutral(B,E)* is not in general closed under conjunction. Consider, for example, the case where $B = \{1 \notin N , 2 \notin N \}$ and $E = \{ p1 \vee p2 \}$, with $p1 \equiv (N'=N\backslash\{1\})$ and $p2 \equiv (N'=N\backslash\{2\})$; then $p1$ and $p2$ are both contained in *neutral(B,E)*, but $(p1 \wedge p2)$ is not.

This means that not all neutral predicates are mutually compatible. To ensure that only such forms are chosen, there is a need to "filter" these choices, as follows.

Definition. Let B be some previous state and E be an event; then a predicate $q \in$ *neutral(B)* is termed *central* with respect to B and E iff, for all predicates $p \in$ *neutral(B)*, the set $B \cup E \cup \{p,q\}$ is end-consistent. The set of all such q is denoted *central(B,E)*.

Proposition. The set *central(B,E)* is closed under disjunction and conjunction.

With regard to the previous example, the form $(N'\backslash\{1,2\}=N\backslash\{1,2\}) \equiv (p1 \wedge p2)$ is included in *central(B,E)*. The required inference rule may now be defined in terms of this set.

Definition. Let h be a history and r be an undashed predicate; then $h \models\!\!\equiv r$ (r is inferred from h) is defined inductively (over initial histories h° and events E) by the following rules:

(i) $\langle\rangle \quad \models\!\!\equiv true$

(ii) $h^\circ \,\hat{}\, \langle E \rangle \quad \models\!\!\equiv r$ iff $\exists\, p \in H(h^\circ),\ q \in central(\, H(h^\circ), E\,) \bullet wpc(E \cup \{p,q\}\,) \Rightarrow r$

wherein *H(h)* denotes the set of all predicates inferred by these rules.

Proposition. The set *H(h)* is closed under conjunction. (This follows from the closure of *central(B,E)*, by induction on the length of h.)

A valid objection to the inference rule $\models\!\!\equiv$ given above is that the choice of neutral augments depends upon the particular history. An apparently weaker, but *static* (and thus practically applicable) inference rule is developed below, using only neutral predicates which are central to the event E itself (where we assume that *identity (B)* and *neutral(B)* are extended to mixed predicates in the obvious way).

Definition. Let E be an event; then a predicate $q \in$ *neutral(E)* is *central* to E iff $q \in$ *central(B,E)* for all states B such that $B \cup E$ is end-consistent.The set of all such q is denoted *central(E)* .

Definition. Let h be a history and r be an undashed predicate; then $h \models r$ is defined inductively by the following rules:

(i) $\langle\rangle \quad \models true$

(ii) $h^\circ \,\hat{}\, \langle E \rangle \quad \models r$ iff $\exists\, p \in H(h^\circ),\ q \in central(E) \bullet wpc(\, E \cup \{p,q\}\,) \Rightarrow r$

wherein $H(h)$ now denotes the set of all predicates inferred by these static rules.

Proposition. If $h \models r$ then $h \models\!\!\equiv r$ (since *central(B,E)* $\subseteq$ *central(E)* for all states B).

We are now in a position to define consistency over histories, using whichever set of historical inferences proves most convenient.

Definition. A history $h^\circ \,\hat{}\,\langle E\rangle$ is *end-consistent* iff $H(h^\circ) \cup E$ is end-consistent; $\langle\rangle$ is vacuously end-consistent. A history h is *consistent* (and therefore plausible) iff all of its prefixes are end-consistent.

Proposition. If E is end-consistent and $ipc(E) \in H(h^\circ)$ then $h^\circ \,\hat{}\,\langle E\rangle$ is end-consistent.

If E is statically "well-formed", $h^\circ \,\hat{}\,\langle E\rangle$ will be end-consistent whenever the guard G for that event can be inferred from h°.

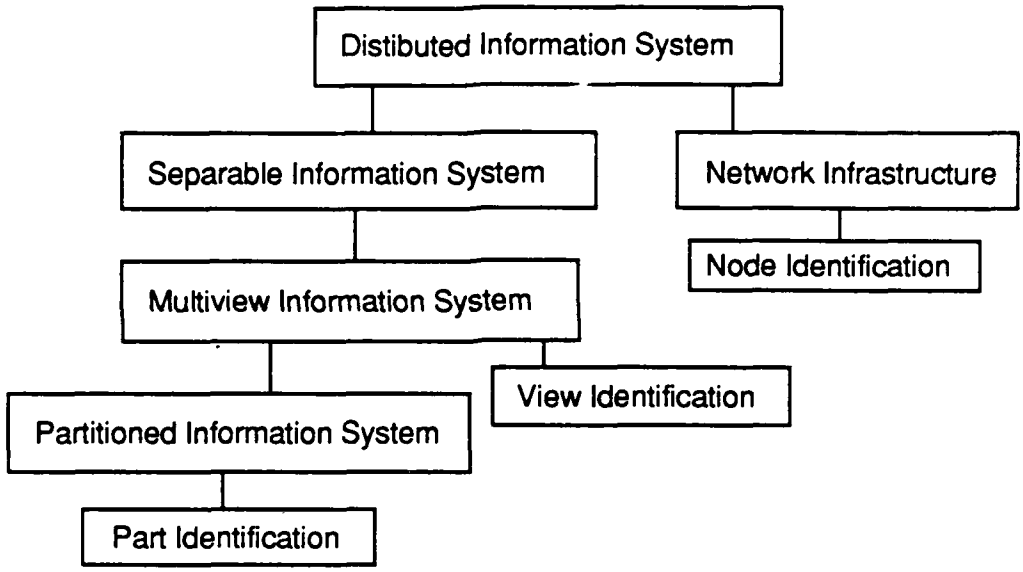
The behaviour characterised by a class specification can now be modelled in terms of traces.

Definition. Assuming some association of sets to the generic names appearing within a given class specification, a *denotation* for an event belonging to that class is just its signature with any systematic substitution of values from these sets for the parameter names; an *initial denotation* is just the signature of the class after such substitution. The *traces* for that class are the prefix-closed set of finite sequences of event denotations, each of which begins with the same initial denotation .

The *projection* of a denotation is a model for (an occurrence of) the denoted event, wherein the free variables corresponding to its parameters have the values which were substituted for their names within that signature. A trace $\varepsilon_0\hat{}\langle\varepsilon_1,\ldots,\varepsilon_k\rangle$ *projects* onto a history $E_0\hat{}\langle E_1,\ldots,E_k\rangle$ iff E_i is a projection of ε_i. A *model* for the behaviour of a class is then a set of traces, each of which projects onto a consistent history.

4. A More Realistic Example

This section is entirely devoted to the development of a somewhat more realistic example. The following "roadmap" gives an overview of both the example to be considered and the resultant system architecture.



The specification is developed exclusively by composition and refinement, which imposes a "bottom-up" method of proceeding; otherwise, the order of presentation is arbitrary. As usual, the first stage in this process is concerned with unique identification.

Partitions are uniquely identified as elements drawn from some carrier set $\mathbb{P}$; initially, there are no such *parts* in existence.

PID

Part: set [$\mathbb{P}$]

Part' = $\emptyset$

Whenever a new partition is first defined, its identification is distinct from that of any other part currently in-use.

PID. New Part ($\rightarrow$ p)

p: ~ Part

---

p $\in$ Part'

An existing part identifier may be nullified, in which case it is deleted from the set of those currently in-use.

PID. Null Part(p)

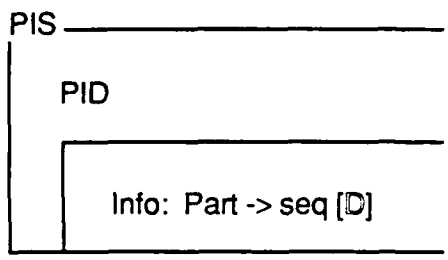
p: Part

---

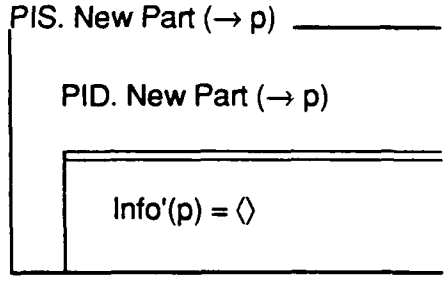
p $\notin$ Part'

Based on this part identification, the structure of a *partitioned information system* is now specified.

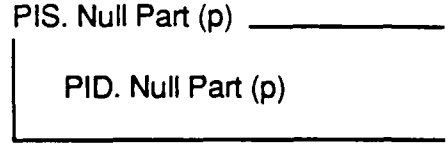
The information in the system consists of a separate sequence of data for each defined part, corresponding to the (historical) order in which successive data were added to that particular partition. The actual data values belong to some underlying set $\mathbb{D}$, which is not further specified at this level.



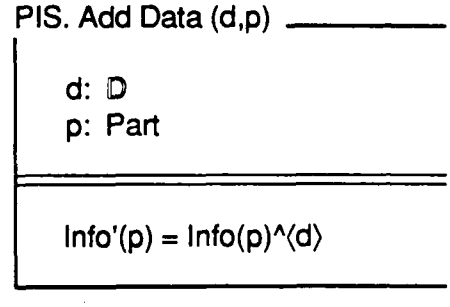
When a new part is first created, the associated sequence of information is empty.



When an existing part is nullified, information previously associated with that partition is no longer known within the system.



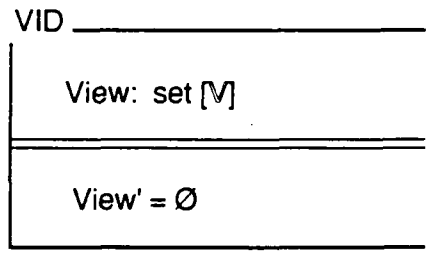
When a new datum is added to an existing part, it becomes the latest in the sequence associated with that partition.



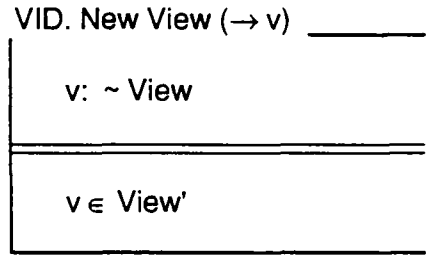
Specifying the structure of the information itself is an overriding concern in the design process for many such systems. Here, however, these issues have been abstracted almost completely (so that they may be addressed as an orthogonal aspect at some later stage). The only commitment is to partitioning, where the information associated with each part is truly independent; thus this abstraction would not be adequate for relational structures. We have also postulated an historical interpretation, as opposed to "overwriting semantics", for reasons which will become apparent. This decision could be reversed later, by encapsulating the model so as to allow access to only the most recent data for each partition.

A characteristic of most database-like systems is the ability to support multiple *views* of the same information. Such views must first be identified.

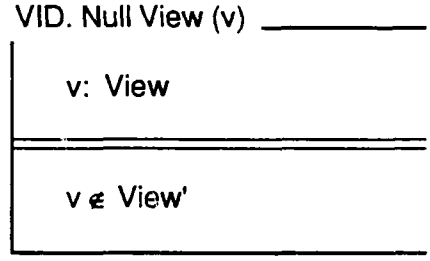
Views are uniquely identified as elements drawn from some carrier set $\mathbb{V}$; initially, no views are defined.



When a new view is defined, its identification is distinct from any other view in existence.

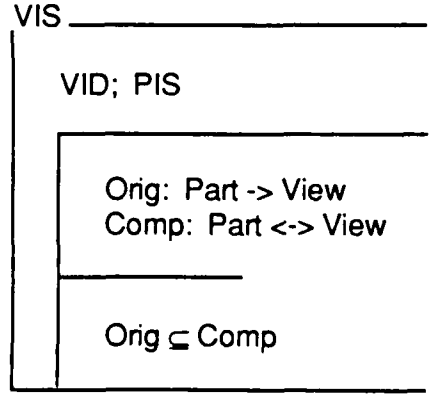


When a view is nullified, its identifier is no longer known.



This view identification is then combined with the partitioned structures specified previously to define a *multiview information system*, followed by promotion of the operations on VID and PIS.

In an information system with multiple views, each partition *originates* in one particular view, but may be visible (as a *component*) in many views; every part is a component of the view in which it originates.



New views may still be created at any time (without further restriction).

VIS. New View ($\rightarrow$ v)

VID.New View ($\rightarrow$ v)

Only views which are not currently in the rôle of the originator for any existing partition may be nullified.

VIS. Null View (v)

VID.Null View (v)

$v \notin cod(Orig)$

Each new part is created from some specified view, which thereby becomes the originator for that partition.

VIS. New Part (v $\rightarrow$ p)

PIS.New Part ($\rightarrow$ p)

v: View

Orig'(p) = v

A partition may only be nullified from the view in which that part originates.

VIS. Null Part (p,v)

PIS.Null Part (p)

v: View

p Orig v

Only the originating view for a given part may add data to that particular partition.

VIS. Add Data (d,p,v)

PIS.Add Data (d,p)

v: View

p Orig v

Additional operations, specific to a multiview system, are also introduced at this level.

The originating view for any partition may be changed by its current originator, provided that the part in question is already a component of the view designated as the new originator.

VIS.Chg Orig (p,v,u)

p: Part
v,u: View

$u \neq v$
p Orig v
p Comp u

Orig'(p) = u

A part may be made newly visible in some view *v* from another view *u* wherein it is already a component.

VIS. In View (p,v,u)

p: Part
v,u: View

p Comp u
$(p,v) \notin Comp$

$(p,v) \in Comp'$

A component part may be excluded from any view which is not (currently) the originator for that partition.

VIS. Ex View (p,v)

p: Part
v: View

$(p,v) \in Comp \backslash Orig$

$(p,v) \notin Comp'$

All of the foregoing operation definitions embody design decisions which are, in some sense, arbitrary; they could just as easily have been made (and specified) differently.

At this point we turn to specifying the underlying network. Again the first step is concerned with uniquely identifying the *nodes* in such a configuration.

Nodes are uniquely identified as elements of some set $\mathbb{N}$; a subset of the nodes so identified are said to be "up" at any given time.

NID

Node, Up: set [$\mathbb{N}$]

$Up \subseteq Node$

$Node' = \emptyset$

When a new node is introduced it is given a distinct identifier; such newly defined nodes are initially "down" (i.e. not up).

NID. New Node ($\rightarrow$ n)

n: ~ Node

$n \in Node'$

An identified node may be (definitively) deleted from the set of known nodes within the system.

NID. Null Node (n)

n: Node

$n \notin Node'$

Nodes which are down may be brought up.

NID. Node Up (n)

n: Node\Up

$n \in Up'$

Nodes which are up may also go down.

NID. Node Down (n)

n: Up

$n \notin Up'$

The *network infrastructure* is then characterised as a further refinement, which introduces the interconnection of individual nodes.

Every defined node is *connected* to itself, and may also be connected to other such nodes; these connections are considered to be bi-directional. A node is said to be *accessible* from another node if there exists some sequence of interconnected nodes leading from one to the other, where all of the nodes in question are up.

NIS

NID

Conn: Node <-> Node
Acc: Node <+> Node

$Conn = Conn^{-1}$
$id[Node] \subseteq Conn$
$Acc = (Conn \cap Up \times Up)^{+}$

Additional operations at this level might include events such as the following:

It is possible to establish a new connection between previously unconnected nodes.

NIS.MakeConn (n,m)

n,m: Node

$(n,m) \notin Conn$

==========

$(n,m) \in Conn'$

Existing connections may be broken at any time (but a node cannot be disconnected from itself).

NIS.BreakConn (n,m)

n,m: Node

$n \neq m$
n Conn m

==========

$(n,m) \notin Conn'$

Quite obviously, this network definition is merely a "placeholder", which is introduced more for purposes of illustration than for its actual substance. In practice, it should be replaced by a (sufficiently abstract) specification for the real network infrastructure upon which the system of interest is meant to be constructed.

Before proceeding to our objective of a physically distributed system, it is helpful to introduce an intermediate abstraction corresponding to a *separable information system.* This is formulated as a further refinement to the multiview systems already specified.

There is an independent *projection* for each separate component (part, view) so that the sequences of data associated with the same partition in different views need not be identical; the length of a given projection is the *span* of that component. The set of such projections for the views in which each part originates is exactly equal to the information within the system as a whole. The projections for other components may *lag behind.*

SIS

VIS

Proj: Comp -> seq[ID]
Span: Comp -> NAT

Span = (#)∘Proj
Proj∘(id[Part] & Orig) = Info
Proj∘(id[Part] & Comp) ⊆ $pref^{-1}$∘Info

These latter invariants are expressed in terms of the operator to *join* two relations with a common domain:

_&_: (𝔸 <+> 𝔹) × (𝔸 <+> ℂ) -> (𝔸 <+> (𝔹 × ℂ))

$a(R_1 \& R_2)(b,c) \Leftrightarrow aR_1b \wedge aR_2c$

The "lag" is specified as some *prefix* of the original information, where this relation has its usual definition:

pref: seq[𝔻] <-> seq[𝔻]

---

p pref x ⇔ ∃s: seq[𝔻]•(p^s=x)

It follows from the above specification that component projections for the same partition differ only in their respective spans; they must agree on prefixes of equal length, as it is not possible to "rewrite history". All operations on VIS may be promoted directly (since each part can only be extended from the view in which it originates). In addition, some means must be provided for dealing with out-of-date projections:

A particular component part in view *v* may be brought up-to-date with respect to that same part in another view *u*, provided that no information would be lost as a result.

SIS.Update (p,v,u)

p: Part
v,u: View

---

p Comp v
p Comp u
Span (p,v) ≤ Span (p,u)

Proj' (p,v) = Proj (p,u)

Finally, a *distributed information system* is obtained by mapping the separable views (established at the previous level) onto some given network infrastructure.

A particular node is designated as the *host* for each view, which thereby determines a corresponding *base* for every component. However, there is at most one *representative* sequence of data for each partition at any node, whence the projections of that component part are identical in all views sharing the same host.

DIS

SIS; NIS

Host: View -> Node
Base: Comp -> (Part × Node)
Repr: (Part × Node) +> seq[𝔻]

Base ⊆ (id[Part] ⊗ Host)
Proj = Repr ∘ Base

The required correspondence, (Part × View) -> (Part × Node), is specified above in terms of a *relational product*:

$$\_\otimes\_ : (\mathbb{A} <+> \mathbb{C}) \times (\mathbb{B} <+> \mathbb{D}) \to ((\mathbb{A} \times \mathbb{B}) <+> (\mathbb{C} \times \mathbb{D}))$$

$$(a,b)(R_1 \otimes R_2)(c,d) \Leftrightarrow aR_1c \wedge bR_2d$$

The operations appropriate to such a distributed information system would mainly be introduced as promotions from the previous level, whilst taking into account any additional constraints imposed by the physical distribution. For instance:

A component may be brought up-to-date with respect to another view provided that the hosts for both views are mutually accessible. (NB: this will update that component part for all views on the same host.)

DIS.Update (p,v,u)

SIS.Update (p,v,u)

Host(u) Acc Host(v)

Certain technically more complex issues (e.g. relating to restart and recovery) might also begin to be tackled at this level, in conjunction with promoting specific events inherited from the underlying network definition. At this point, however, the overall system structure announced at the outset is essentially complete.

It should be observed that further development of the foregoing specification is limited by its somewhat "Olympian" perspective, wherein only rather global properties of the system have so far been characterised. Despite this objection, formalisation of such a top-level design is a useful first step. For the particular example considered here, subsequent refinements (leading towards an actual implementation) ought properly to be formulated as a decomposition into independent *processes*, based on the concepts of communication and synchronisation developed by Milner[1980] and Hoare[1985]. These questions will be addressed in a sequel to this paper.

Acknowledgements. This work was supported in part by the U.S. Army Communications and Electronics Command under a research contract with Massachusetts Computer Associates, Inc., and by the U.K. Science and Engineering Research Council through a part-time visiting fellowship with the Programming Research Group at Oxford University. Significant contributions were made (from both sides of the Atlantic) by S. Amoroso, B. Cohen and T. Wheeler, who participated in experimental case studies over a long period of time. We are especially grateful to Gillian Hall and Jane Vergette for the most tangible contribution of all: taming the untried technology used to produce the typescript of this paper.

REFERENCES

1. Abrial, J-R., A Theoretical Foundation to Formal Programming, Programming Research Group (Oxford University, 1982).

2. Bjørner, D. and Jones, C.B., *The Vienna Development Method*, LNCS No. 61 (Springer-Verlag, 1978).

3. Dahl, O-J., Hierarchical Program Structures, in *Structured Programming* (Academic Press, 1972) pp. 175-220.

4. Goguen, J.A., Thatcher, J.W. and Wagner, E.G., An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, in Yeh (ed.), *Current Trends in Programming Methodology*, Vol. IV: Data Structuring, (Prentice-Hall, 1978) pp. 80-149.

5. Guttag, J.V. and Horning, J.J., The Algebraic Specification of Abstract Data Types, *Acta Informatica*, Vol. 10 (Springer-Verlag, 1978) pp. 27-52.

6. Hayes, I. (ed.), *Specification Case Studies* (Prentice-Hall Int'l., 1986).

7. Hoare, C.A.R., *Communicating Sequential Processes* (Prentice-Hall Int'l., 1985).

8. Hoare, C.A.R., Proof of Correctness of Data Representations, *Acta Informatica*, Vol. 1 (Springer-Verlag, 1972) pp. 271-281.

9. Jones, C.B., *Software Development: A Rigorous Approach* (Prentice-Hall Int'l., 1980).

10. Milner, R., *A Calculus of Communicating Systems*, LNCS No. 92 (Springer-Verlag, 1980).

11. Suffrin, B., Morgan, C., Sørensen, I.H. and Hayes, I., Notes for the Z Handbook, Programming Research Group (Oxford University, 1985).

APPENDIX II

An Experiment with an Approach to Formal Specifications

Serafino Amoroso
Thomas Wheeler
Center for Tactical Computer Systems
U.S.A. CECOM
Ft. Monmouth N.J.
&
Stephen Schuman
Massachusetts Computer Assoc.
Wakefield Mass.

Abstract

An experiment is described involving a new approach to system specifications. The process of developing a formal specification from an informally specified distributed information system concept forms the basis of the experiment. The new approach, which is based on formal mathematical techniques, is introduced gradually as the experiment is described. An attempt is made to document the experience and lessons learned as the experiment progressed.

Introduction

An investigation was begun a few years ago into potential applications for the newly emerging distributed systems technology to the Army's Air/Land Battle Forces. This investigation's purpose was to help motivate and give direction to a research effort on distributed systems that was beginning at the time. The first application considered was a distributed communication/database system for battalion situation-status reporting in a battlefield environment.

An informal specification of a somewhat abstract version of this situation-status reporting system formed the starting point for the experiment that is reported here on a new approach to system specification using formal mathematical techniques[1]. Although the approach[2, 3] has been under investigation for several years (mainly at the Programming Research Group at Oxford University in England), it is still evolving and is not yet in its final state.

Although there were no distinct roles given to each of the authors, the intent was to simulate a small team containing representation from three disciplines. The first are the "user representatives", those who understand the need for the system being designed and the intended application, the second were the designers, those who are responsible for the intended implementation. Both of these are assumed to have an intuitive understanding of the formal techniques that will be used to reason about the design that is sought. The third are the "analysts" who are assumed to be fluent in the formal techniques described here.

The formal techniques used here are still under development and the intent primarily was to experiment with the approach evaluating its current utility. This possibly might clarify some aspects of the approach that could be improved. The hope of gaining a deeper understanding of the distributed system itself was not an insignificant part of our motivation.

The preceding paragraph was the primary motivation for the experiment, however the process described in this paper can be viewed by the reader as a design effort for a certain aspect of the system, namely the distributed nature of an information management system. Viewed in this way, it is hoped that the reader will gain some of the appreciation that was gained by the authors, of the insite into solving complex problems of system design provided by the precision and clarity of mathematical reasoning.

The formal notation used in the approach will be introduced gradually as the paper progresses.

An Informal Specification of a Distributed Database System

The informal description of a distributed communication/database system concept given here, was written before the formal specification was attempted. Since one of the expected benefits of doing a formal

specification is additional insights into the concepts being modeled, it should be of some interest to revisit this starting point after the formal specification is developed.

Consider a collection of computer work stations physically separated over some distance (e.g., a few miles) interconnected by some electronic communications medium (e.g., packet radio). The interconnection pattern need not be total but it should be nonetheless possible for any work station to communicate with any other work station. The only physical devices (hardware) used to implement the system are those physically located with each work station. This would include at least a computer, a keyboard and screen, some secondary storage, and the communication equipment. The hardware at each station is identical. The system must be mobile in the sense that the stations will be moving from time to time to different physical locations. Moving will probably involve a logging out process followed later by a logging in process.

Each station maintains a local file system. The basic unit managed by these file systems is the report, which is similar to the programming language concept of a record, as found for example in Pascal. A report consist of a fixed number of fields, each being an information unit of some type. The number of the fields and the type assigned to any field can be changed only at what we will call "system initialization time". Such alterations to the structure of a report will not be a frequent occurrence. Examples of field values are: a numeric value, say INTEGER or REAL, or a tuple of numeric values, or a string of characters.

The local file system will manage a collection of such reports, one of which will be designated as the current report. The file system will be able to modify the values of any of the reports in the collection, create new reports, delete reports, etc. An editing capability will be available at each station.

The basis for intercommunication among the stations is the notion of "global data areas". The collection of stations making up a particular system can involve any number of global data areas. Any global data area, say G, has a number of stations inputing information to it and a number of stations outputing information from it. An inputing station for G is not necessarily also an outputing station and vice versa. To be an inputing station to G means that W can send a copy of a report to G. The totality of all information held by G is the collection of all reports sent by the inputing stations for G. An inputing station can have at most one report in G at any time. It can have no reports in G, it can remove a report and replace it by a new report or not replace it at all. Outputing stations for G can query the information in G. Precisely what the query capabilities are is open at this time. We may want to restrict the query capabilities of particular outputing stations. Since the only hardware for these systems is at the physical location of the stations, processing support for the global data areas must be at the stations.

The most important attribute for the intended application of these systems is "robustness", that is, the ability of a system to maintain a continuity of service even under severe operating conditions and as individual stations go down. A station can go down either by an orderly log-out, or abruptly (by malfunctioning or by being destroyed e.g. by hostile action). If G has input stations W1,...,Wm and output stations W1',...,Wm', then if Wi goes down,

its most current report in G (if any) continues to be accessible to the output stations which can continue to function to the maximum extent possible. Likewise, if some Wj' goes down, the rest of the system must function with no change other than the fact that Wj' will not be performing queries. In other words, if any station goes down, whether or not it is an inputing or an outputing station for G, the rest of the system must be able to function normally. All this implies that a global data area cannot be implemented at any one station. The situation is similar for multiple stations going down at once. Other problems are aggrevated when multiple stations go down (e.g., problems of communication connectivity). We leave these issues open for now.

The question of the robustness of a system also includes the question of data consistency and currency. If W updates a report in G, then after "some reasonable period of time", all outputing stations for G should get this new report instead of the older version. At any moment of time, two outputing stations for G should always get information extracted from the same versions of a given report.

Beginning the Experiment

We decided to begin by trying to formulate the essential concepts underlying this technology, as opposed to beginning the specification of a particular system. We thus stepped back from the particular system with the expectation that the essential concepts would form a framework for the resulting system specification. Essentially what we planned to do first was to look at the class of such systems, and thus avoid imposing bounds too soon.

(This use of generalization, which follows naturally from the mathematical nature of the approach to specification, leads to better long term maintainability properties in the resulting system, as requirements changes are easily accommodated. The maintainability of a system is a direct consequence of its ability to accomodate changes in the requirements. The existence of an explicit framework for a system's design based on the class of systems it belongs to, leads to a high probability that a change fits within that framework, unless, of course, the change places the system outside that class.)

After some discussion, the following three concepts seemed central.

(1) different "views" inside a distributed system,

(2) "virtual nodes" that need not physically exist, and

(3) "replication" or redundancy.

The first goal was to give some clarity and precision to these notions. Hopefully we could then study their implications to the design.

Basic Concepts.

The users of the system are visualized as the "originators" of information

in the system. Each information unit so generated is a "report" and for now there is no need to consider the structure of individual reports.

Each user will define a sequence of such reports, namely the history of all the reports this user has generated. Each user is capable of appending another report to this sequence at any time (i.e., generating another report).

INF =def seq RPT

(Certain mathematical concepts are considered standard and will be used freely, among these is the set of sequences of elements from some other set, for example, the above expression states that INF is the name of a set defined as the set of all finite sequences over a set RPT. Looking back, we should have used the identifier INFO instead of INF, and we should have kept things more general by using the identifier DATA rather than RPT. The choice of identifiers for the various concepts introduced is not a trivial matter. It is an important factor in the readability and clarity of the notation. We in fact use "info" for a related concept later, and we do eventually drop the name INF and change RPT to DATA. The the notational system being described has the important attribute of encouraging experimentation by making the process of changing and improving the concepts being defined rather easy. As we will see, these changes and improvements can be more substantial than just changing identifiers.)

The Initial Formal Specification.

At this early stage we chose to model the system users or "originators" by a partial mapping "orig" from a set NID (node identifiers) to INF. Note that at this stage we were conceptually identifying the concept of "node" and "user"

Orig : NID -|-> INF

The set NID is intended as the collection of all possible names to be used for originators, this includes originators that may enter the system in the future. Any element of NID in the domain of Orig is being used to name an originator currently in the system, and its value under the mapping is intended to model the sequence of reports issued by the originator from the time he entered the system. Note this sequence of reports can be empty.

Continuing we defined

Curr : NID -|-> RPT

The mapping Curr (current report) has this form (its "signature"). Inuitively, Curr assigns to an element of NID in its domain of definition the most current report issued by the "node". Curr is defined by:

Curr = Last(Orig)

which gives the last element in the sequence (Orig).

Conceptual Modules and Semantic Operators

An important aspect of the notation used in this approach to formal specification is that of a "conceptual module". The following was the first conceptual module attempted for this application. It was an initial attempt at a definition of the concept of a "node".

```
Node(n)
| n   : NID
| Own : INF
| Oth : VIEW
```

Each node n is considered to consist of its name (an element from the set NID of node identifiers), a sequence in INF (the node's "own" information history), and a "view" of the totality of all information in the system. The concept of VIEW was one of the notions considered central and at this point was yet to be made precise.

Another conceptual module considered at this early stage was:

```
INFO
| Orig : NID -> seq RPT
|
| range(Orig) = {< >}
```

This module was an attempt to characterize the totality of all information in the system. The mapping "Orig" (which here is total) associates with each element of NID a sequence of reports. Here all of the potential node identifiers are already associated with RPT sequences, the unused names are be mapped to empty sequences. The expression below the double line in the module specifies "initial conditions". Here the report sequences are all initially empty.

INFO contained a "semantic operator" which makes possible the generation of a report by a particular user.

```
INFO.Issue(n, r)
|   n : NID
|   r : RPT
|
|   Orig'(n) = Orig(n) * <r>
```

The meaning of this "semantic operator" (for the conceptual module INFO) is that given a node identifier n and a report r, r is placed as the latest entry in the node's history of reports. Hence, a new report has been generated. In the notation the prime (') symbol is used to indicate the condition of a mathematical entity after an operation has been applied.

The concept of a "view" mentioned above is intuitively the idea of a node "seeing" some part of the total information available in the system. Our first attempt at a formal definition took the following form.

```
VIEW(n)_____________
|
| n : NID
|
| Own : seq RPT
|
| Inf : NID -|-> seq RPT
|_____________________
|
| Inf(n) = own
|
| Inf contained in Orig
|_________________________
```

The identifier "inf" (information) which was earlier used to name the set of all finite sequences from RPT, is now being used for a different role. The constraint "Inf contained in Orig" states that Inf is some part of the total system information. This attempt to capture the concept of a view was soon improved to a rewrite of the INFO module:

```
INFO----------------------------------------
|
| Orig : NID -> seq RPT
|
| View : NID -> ( NID -|-> seq RPT ),
|_____________________________
|
| View(n)(m) contained in  pre( Orig(m)),     where   n /= m,
|
| View(n)(n) = Orig(n),    and
|
| n in dom( View(n)).
|_____________________________________________
|_____________________________________________
|
| range( Orig) = {<>}
|_____________________________________________
```

The operator "pre" on sequences is the set of all initial parts (prefixes) of its sequence argument. The first constraint above states that "the view that n has of m is always some initial part of the sequence of reports issued by m". The "view" that a node has, "view(n)", is specified to be a set of some of initial prefixes of the histories of some the nodes of the system. A node can always view all of its own information. A node can always view its own latest information, but the latest information generated by some of the nodes may not be locally available yet.

Refining The System.

A period of consolidation and redefinition resulted in the specification which follows. Such consolidation and redefinition seems to be a vital part of the formal specification process. It is here that most of "progress" seems to be made.

The specification, whose exposition makes up the main body of the paper, is composed of four main parts. The first is the specification of the functional aspects of the system, the second describes the distributional aspects of systems, the third combines these to form a distributed information system and the fourth extends this to address the issue of robustness.

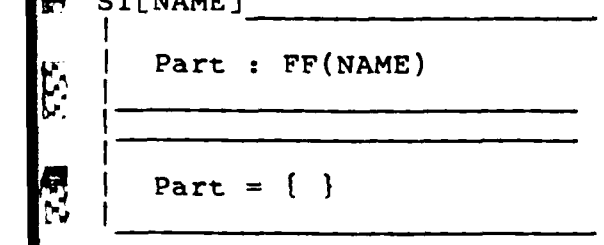
The Formal Specification.

Both the formal text (conceptual modules and semantic operators) and the prose which follows each module and operator are considered to be parts of the specification, with the prose being commentary and interpretation of the formal text.

System Functions.

The first facet of the system to be specified was the functionality, which is specified independent of the idea of distribution which is added later.

Concept S1: System made up of Parts.



A conceptual module S1 defines the Parts of the system in terms of a "generic" set NAME. (By a "generic set" we mean that no properties of the set are specified other that the fact that the set contains a supply of elements, and that it can be determined whether or not two arbitrary elements of such a set are equal.) A set "Part" (the collection of parts of the system) is meant to be an abstraction of the users or uning programs of the system and is specified with "signature" FF(NAME) which means that Part will always be a finite subset of the set NAME. (The use of the term "nodes" for "the system users" was not a good choice, and was changed to separate the two concepts it embodied, the logical parts, users or user programs and data, and the physical parts, or nodes, as will be seen below.) The set Part

is initially empty. (The empty set = { }).

```
S1.NewPart( -> p )_____________________________
|
|  p : NAME
|
|  p not in Part
|_____________________________________________
|_____________________________________________
|
|  p in Part'
|_____________________________________________
```

New parts can be added to the system. Parts are always to be named by an element from NAME. (The symbol "->" indicates that "p" is an explicit result of this operation, ie. an out parameter.)

```
S1.DelPart( p )_____________________________
|
|  p : NAME
|_____________________________________________
|_____________________________________________
|
|  p not in Part'
|_____________________________________________
```

Parts can be deleted from the system.

Concept S2: Describing Information in the System.

```
S2[DATA]____________________________________
|
|  S1
|_____________________________________________
|
|  Info : Part -|-> seq DATA
|
|  Comp : FF(Part)
|_____________________________________________
|
|  Comp = dom(Info)
|_____________________________________________
|_____________________________________________
```

This module extends the first module S1. In this way new modules can build directly on previously defined modules. Info and Comp are implicitly empty initially since Part is initially empty.

The vague terms "Part" and "Comp" (the components which have information) were used in an attempt to keep the development as general as possible. The plan was that we would give the identifiers finally adopted a great deal of thought once the system concepts were clearly understood.

S2.MakeComp(p)

p : Part
p not in Comp
p in Comp'
Info'(p) = { }

System "parts" can be made into system "components" which can then issue information.

S2.Issue(c, d)

c : Comp
d : DATA
Info'(c) = Info(c) * <d>

Only system "components" can issue information. This source of information is the only way information will come into the system. (Notice that the formal statements are not difficult to read, "Conceptual Modules" containing definitions, constraints on values(invariants) and (after the double line) initial conditions, while "semantic operators" contain definitions, conditions of operation(preconditions) and (after the double line) results(postconditions).

Concept S3: Entry of Information.

S3

S2
Orig : FF(Comp)

The group of current originators is a subset of "Comp" and is initially empty (since Comp is).

S3.AuthOrig(c)

c : Comp
═══
c in Orig'

S3.NonOrig(c)

c : Comp
═══
c not in Orig'

Ability to origionate information can be given and withdrawn.

S3.Issue(o, d)

o : Orig
d : DATA
═══
S2.Issue(o, d)

Only authorized Originators can issue information. (Note the replacement of a "semantic operation" by a restricted version.)

Concept S4: Viewing Information.

S4

S3
Vis : Part <-|-> Comp
View : Part -> (Comp -|-> seq DATA)
Id(Comp) contained in Vis
All p, All c : (p, c) in Vis <=> View(p)(c) contained in Pre(Info(c))

Two concepts are introduced at this level. Visibility describes which parts

can see which components' information. View describes what information a part sees for each component. The invariants state that components can see their own information and a part sees a prefix of the information for those components in its view.

S4.Open(pp, cc)

pp : FF(Part)

cc : FF(Comp)

---

cc contained in vis'(pp)

All p, All c : (p, c) not in Vis => View'(p)(c) = < >

Visibility can be extended. Added views are initially empty.

S4.Close(pp, cc)

pp : FF(Part)

cc : FF(Comp)

---

cc not contained in Vis'(pp)

Visibility can be removed.

S4.MakeComp(p)

S2.MakeComp(p)

(p, p) in Vis'

When system parts are made components, they are able to see the information which they issue (replacement of operation by extended version, so as to maintain the invariant).

S4.Sync(c, s, d)

```
c : Comp
s, d : Part
-----------------------------------------
c in im(Vis){s}
c in im(Vis){d}
#(View(d)(c)) <= #(View(s)(c))
-----------------------------------------
-----------------------------------------
View'(d)(c) = View(s)(c)
-----------------------------------------
```

It is possible to make the view that a part(destination) has of a component's(c) information the same as the view another part(source) has of that component's information (ie. to synchronize their views), as long as both have views of that component's information and the source's view was later (ie. had more recent versions).

This concludes the specification of the functionality of the system. As was stated earlier this is not meant to be a specification for an operational system but rather a class of systems. As such it includes only those functional properties that were deemed to be essential to this class of systems, ie. distributed information systems.

Distribution of a System.

The next set of "Conceptual Modules" addresses a different aspect of the distributed information system, those concepts essential to the distributed nature of the system. As this aspect looks at the system from a different perspective, ie. is orthogonal to the functional view, this specification is independent of the functional specification. These two specifications will be combined later to specify both aspects of the system.

Concept P1: Physical Distribution of Nodes.

P1[NN]

Node : FF(NN)

Conn : NN <-|-> NN

Path : NN <-|-> NN

Conn contained in Node X Node

Conn in symm(NN)

Path = Conn*

Node = { }

This conceptual module defines the basis of a distributed system as a collection of Nodes which are Connected together allowing those Nodes with a Path between them to interact. (Note that the term Node is used here in a much more restricted sense than its use in the first attempt at a specification, here it is an abstraction of a single computer in a network of computers.) Node will always be a finite subset of names from another generic set of names (NN). Only existing Nodes can be Connected, with the existence of a path depending on the closure of the individual Connections. Conn is a symmetric relation. Node, and therefore Conn and Path, is empty.

P1.NewNode(-> n)

n : NN

---

n not in Node

---

n in Node'

P1.DelNode(n)

n in Node

---

n not in Node'

Nodes can be added and deleted.

P1.Connect(n, m)

n, m : Node
n, m not in Conn
(n, m) in Conn'

P1.DisConn(n, m)

n, m : Node
(n, m) in Conn
(n, m) not in Conn'

Individual Connections, and therefore Paths, can be added to and removed from the network of Nodes.

Concept P2: Activation/Deactivation of Nodes.

Conceptual module P2 builds on the network of Connected Nodes provided by P1 to add the concept of parts of this system, either nodes or Connections, not being available for use at times during the life of the system.

P2

P1
Up : FF(Node)
Avl : FF(Conn)
Acc : FF(Path)
Avl contained in Up X Up
Acc = Avl*

The Nodes that are working are a finite subset of the existing Nodes. The Available Connections are that finite subset of the Connections whose Nodes are Up. The Accessible Paths are, likewise, the closure of the available Connections.

P2.NodeUp(n)

n : Node
n not in Up
----
n in Up'

P2.NodeDn(n)

n : Up
----
n not in Up'

Nodes can come Up and go down, implicitly causing Connections to become Available or not resulting in making accessible/inaccessible other Nodes.

P1 and P2 contain the concepts we have chosen as essential in the modeling of the distributional aspects of systems. Note that we have, for instance, chosen not to explicitly consider that connections could go down independent of Nodes, on the basis that to a user of a connection, either the connection or the node going down appears the same.

Distributed Information System: Combining Concepts.

The functional conceptual modules which provided for the Issuing and Viewing of information are combined with the distributional conceptual modules to form the concept of a distributed information system. This allows us to consider those aspects of an information system which either follow from its distributed nature (modules D1 through D3), or can use this distributed nature to advantage (module R1).

Concept D1: Distributing Parts.

Now that the system is distributed, Parts (users or programs) are considered to exist at a Node. Note that this choice precludes explicit duplication of Parts, as will be seen later (conceptual module R1) the normal operation of the system provides enough redundancy so that explicit duplication is not needed.

```
DSys1-----------------------------------------
|
|  S4
|
|  P2
|-----------------------
|  Home : Part -> Node
|_____________________________________________
|_____________________________________________
```

The first Distributed System conceptual module combines and directly extends both the final functional and distributed configuration modules. Every Part has a Home Node. (Again since Part and Node are both initially empty, Home is also.)

```
DSys1.NewPart(n ->p)_____________________________
|
|  S1.NewPart( ->p )
|
|  n : Up
|_____________________________________________
|_____________________________________________
|
|  Home'(p) = n
|_____________________________________________
```

When Parts come into existence, they do so at a Home Node. (Promotion of a semantic operation to include the effects of a new concept.)

```
DSys1.MoveHome(p, n)_____________________________
|
|  p : Part ;  n : Up
|_____________________________________________
|
|  Home(p) /= n
|
|  (Home(p), n) in Acc
|_____________________________________________
|_____________________________________________
|
|  Home'(p) = n
|_____________________________________________
```

The Home of a Part can move.

Concept D2: Activation/Deactivation

In addition to the long term existence of Parts on Nodes, there is a shorter term Activation/Deactivation of these Parts based both on the desires of the Part and the status of the Node where it is located.

DSys2

DSys1
Site : Part -> Up
Sess : FF(Part)
Sess = dom(Site)

Parts can only be active (in Session) when they are at a Site which is Up. A Part's Site can be different from its Home.

DSys2.LogIn(P, n)

p : Part ; n : Up
p not in Sess
Site'(p) = n

DSys2.LogOut(p)

p : Sess
p not in Sess'

Parts can Log in or out at a Node which is Up.

DSys2.NodeDown(n)

P2.NodeDn(n)
n not in range(Site')

When a Node goes down, there are no Parts at that node in Session, ie. activity at that Node ceases.

Concept D3: Distributed Entry and Retrieval.

The activity of the Parts and Components in the distributed system must take into account the effects of the distributed nature of the system on the

operations. Both the effects on existing operations and the addition of new operations must be considered.

DSys3

DSys2

No concepts are added here, only operations.

DSys3.Issue(s, d)

S3.Issue(s, d)

s in Sess

(Site(s), Home(s)) in Acc

In order to Issue information, a source must be in session and have access to its Home.

DSys3.Sync(c, s, d)

S4.Sync(c, s, d)

d in Sess

(Home(s), Site(d)) in Acc

Synchronization requires that the Home of the source be accessible to the destination, but not that the Component that they are synchronizing on be accessible to either.

DSys3.Query(s, c, ->d)

s :Sess
c :Comp
d : DATA
(s, c) in Vis
#(View(s)(c)) > ∅
(Site(s), Home(s)) in Acc
d = last (View(s)(c))

Query is not an essentian operation since all of the information is available and therefore all queries are possible, but it is included here to indicate and examine restrictions which should be included when queries are specified. (Note that if this were the only Query, only the last Data item would have to be kept in any implementation of this system.)

Robust Distributed System.

A robust distributed system extends the distributes information system to allow for the possibility of reconstructing a Part which happened to be (at Home) at a Node which went Down..

Concept R1: Reconstruction of Information.

RbDSys1

DSys3
Best : Node -> (Comp -|-> seq DATA)
Temp : FF(Part)
Best contained in View o inv (home)
All n, All p(at n), All c : #Best(n)(c)>= #View(p)(c)

Best (at each node) contains the latest view of each component that any Part at that Node has. (The two constraints are a precise but inelegant way of saying this, elegance requires reformulation in terms of more sophisticated

relational operators.)

RbDSysl.TempPart(n, p, ->q)

n : Up; p : Sess; q : Name
q not in Part
q in Part' and q in Temp'
View'(q) = Best(n) restricted to im(Vis){p}
p not in Sess'

TempPart creates a temporary Part whose View is as good (up to date) as any that exists on Node n and removes P from being in session.

RbDSysl.ReplPart(q, p)

q, p : Part
q in Temp
Info'(p) = Info(q)
View'(p) = View(q)
Home'(p) = Home(q)
Site'(p) = Site(q)
q not in Part'

ReplPart replaces the "value" of p with that of q and deletes q from the legal Parts.

These two operations can be used together to reconstruct a Part from operationally available implicit replication thus obviating the need for explicit replication whose sole purpose is for the reconstruction of Parts.

Conclusions:

In our description of this experiment, we have avoided speaking of the approach used as being a "methodology". The entire intent of the approach is to develop a useful notational framework in which a wide variety of system concepts can be expressed and in which one can reason about these concepts.

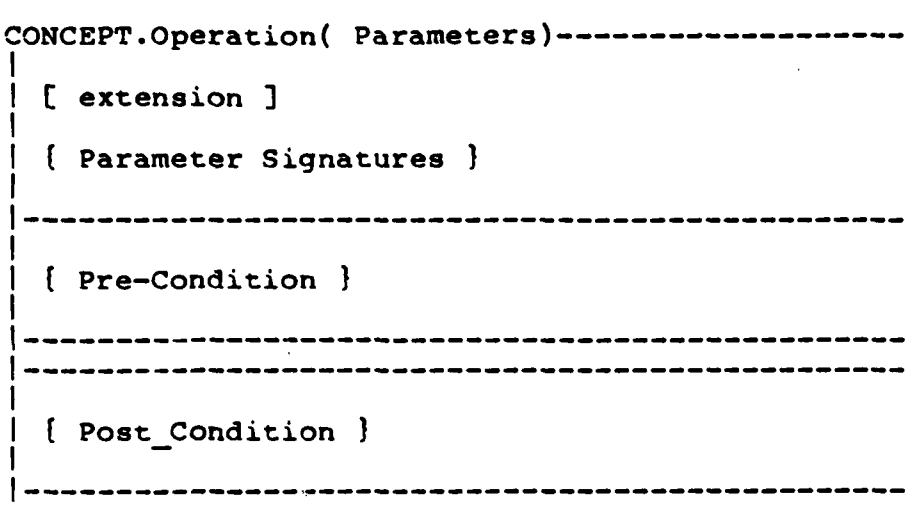
We envision a small team of highly trained designers, with the close cooperation of the potential users, reasoning out the system design. A formal system of notation would be used to express with mathematical precision the concepts agreed on so far, up for discussion, etc. Having such precise documentation would focus the design, and its dynamic nature would make it serve as a growing baseline for the design. The fact that the notation enables implications of the design to be formulated and proven, allows the team to experiment with issues that are usually not possible to consider until the design has been implemented.

Another way of describing "specification" is as a process that a design team goes through in developing a system specification, not only for the document that will result. The process itself is of more value than the resulting specification in the sense that it requires a treatment of the concepts that must help to ensure their consistency and correctness. More specifically, as each conceptual module is formulated and developed, the thought process is anchored and the discussion focuses on the appropriate issues. The resulting document should not be dismissed. A specification of the desired system will exist with a degree of precision seldom approached.

Appendix

Syntax of "Conceptual Module" and "Semantic Operation".

```
CONCEPT[GENERIC_SET]-----------------------------
|
|  { included_concept }
|
|-------------------------------------------------
|
|  { id: SIGNATURE }
|
|-------------------------------------------------
|
|  { invariant descriptions }
|
|-------------------------------------------------
|-------------------------------------------------
|
|  { initial condition }
|
|-------------------------------------------------
```



References.

1. The Vienna Development Method, D. Bjorner and C.B. Jones, Springer-Verlag lecture notes in computer science, No. 61, 1978.

2. Towards a Mathematical Semantics of Computer Languages. D. Scott and C. Strachey, Oxford University T.M. PRG-6.

3. Formal Specification of a Editor B. Sufrin, Oxford University T.M. PRG-21.

Appendix 3: Papers on Z, Programming Research Group, Oxford University

1. Mathematics for System Specification
   by Bernard Sufrin

2. Notes for a Z Handbook
   Part 1: The Mathematical Language
   by Bernard Sufrin, Carroll Morgan, Ib Sorensen, Ian Hayes

3. The Schema Language
   (same authors)

4. An Example of Data Refinement:
   Implementing a Two-dimensional Array as
   a One-dimensional Vector
   by Carroll Morgan

5. Examples of Specification Using Mathematics
   by Ian Hayes

6. A Message System
   by Ian Hayes

7. CICS Temporary Storage
   by Ian Hayes

8. Formal Specification of a Simple Assembler
   by Bernard Sufrin

9. Case Studies in Formal System Specification
   by Bernard Sufrin

10. Z Reference Card

Mathematics for System Specification

Bernard Sufrin

Oxford, 1983/84

Preface

This course and its companions "System Specification and Development", and "Program Correctness" are intended to show you how to use mathematics (and in this course, more specifically *set theory*) in the design and development of computer-based systems. The principal message of this first course is that by using the notation and reasoning methods of mathematics it is possible both to present understandable and coherent system specifications and to discover design flaws long before a system goes into production. Later we will demonstrate that programs themselves can be viewed as mathematical objects, and show how it is possible to *prove* whether or not programs meet their specifications.

The emphasis in the first course is on learning to use mathematics *as a tool*, so our initial presentation of the basic notation and reasoning methods will be intuitive and informal. Whilst this may be unfashionable in some quarters, it is an approach we share with the teachers of many forms of applied mathematics. We introduce the basic notation by an informal characterisation of the meaning of its sentences. We extend the basic notation by using its definitional power to construct a toolkit which is powerful enough to let us begin to describe and reason about some simple, practical systems. During this part of the course we are careful to present convincing *informal* arguments that what we claim are theorems are theorems in fact, but we do not introduce the idea of a *formal* proof until much later. By this time the need for rigour will have become clear, and we will spend a short time in outlining the formal basis for the notation and reasoning methods presented earlier.

1. Introduction

The use of natural language as a vehicle for the specification (or description) of computer-based systems has serious limitations. Anybody who has ever been the victim of bad or inadequately documented software will confirm that the manuals which purport to describe the behaviour of a system never tell the whole story. Almost every programmer who starts to use a new machine, programming language, or operating system sets up a number of experiments, in which they attempt to discover how it "really" behaves. It is a commonplace observation that computer systems (be they large or small) accumulate around themselves a body of folklore — necessary knowledge for anybody who wishes to use them effectively — and a number of "experts" — people who understand (or claim to) the hidden secrets of the system because they have read _ the source code!

But even knowledge gained this way is transient because systems never remain remain stable. Consider, for example, an applications program built using a database package which itself relies on an operating system. Because there is no definitive and unambiguous record of the precise nature of the facilities which the operating system must provide, the manufacturer's system programmers may decide arbitrarily that a certain behaviour is "accidental", and may remove it during a rewrite — perhaps thereby triggering a rewrite of parts of the database package, and thence (by the same unhappy process) a rewrite of the application program. So an enormous amount of time, energy and talent is wasted in simply running to stand still; an activity which is given the name "maintenance" -- as if programs were subject to the action of the weather!

The fact that natural language permits a variety of interpretations to be made of a given specification is partly responsible for this confusion. Agreements made in good faith between system designers and their clients, turn out to be based on mutual mystification (or exhaustion) rather than a common understanding of the nature of a requirement; the same situation obtains for agreements between designers and implementers, between groups of implementers working on different layers of a system, and between systems- and applications-programmers.

As well as wasting hours in perplexing discussions caused by differing interpretations of terminology, a design team which records its decisions in natural language is often unable to foresee serious negative consequences of bad decisions taken early in the life of a system. These often emerge only after a great deal of work has been done on system implementation — by which time it has become too costly to remedy them fully.

The end result is what it has become fashionable to call the Software crisis: the paying customers rarely get what they thought they were going to get, the price is usually higher than they thought they were going to pay, and the end users suffer more misery than they had dreamed was possible!

It has long been conjectured that formalisation can and should play a role in the process of system design and construction, the expectation being that its employment would mitigate at least some of the problems outlined above. But except in certain rather specialised areas (for example compiler construction, numerical algorithms) the problems of putting this precept into practise have come to be regarded as almost insurmountable by the majority of practising programmers and designers.

In this course we present a *formal language* — the language of set theory — and show how to use its notations to record decisions about the intended behaviour of computer-based systems, and its reasoning methods to elucidate the consequences of such decisions. In order to illustrate this, we will apply the notation to the description of a number of systems, some of which are more than just academic examples. Finally we will consider formal criteria by which the correctness of a program relative to its specification may be judged.

2. The Language of Set Theory

"Every mathematician agrees that every mathematician must know some set theory; the disagreement begins in trying to decide how much is some. ... The student's task in learning set theory is to steep himself in unfamiliar but essentially shallow generalities until they become so familiar that they can be used with almost no conscious effort. In other words, general set theory is pretty trivial stuff really, but if you want to be a mathematician you need some, and here it is; read it, absorb it, and forget it."

From the introduction to "Naive Set Theory" by Paul R. Halmos

"For the logician, a main virtue of a theory is that it be concise, so as to be easier to study and characterise; notation is typically devoid of all intuitive content, so that a sentence will not be confused with its meaning. It is common that intuitively evident sentences are quite difficult to prove in such theories.

For the computer scientist, it is more important that a theory be easy to use. Proofs of evident sentences within the theory should be easy to discover, and possibly to automate, and should reflect the intuition behind them"

From "The Logical Basis for Computer Programming" by Zohar Manna and Richard Waldinger.

2.1 Introduction

The study of logic (and later of set theory) arose out of the desire of mathematicians to produce rules which enabled them to say which arguments were valid and which were not. In view of its importance to mathematicians, it may come as rather a shock to a Computer Scientist to discover that despite the fact that it has been studied as a topic in its own right for at least a century, mathematicians have not yet agreed upon a concrete syntax for the language of set theory! If you understand that concrete syntax isn't really very important you may be more surprised to learn that they haven't agreed on an abstract syntax or a semantics (more precisely an axiomatisation) either.

If we want to use the language of set theory as a means of communicating ideas, then we're obliged to choose an existing variant or to invent one of our own. It turns out -- though it might not have -- that it doesn't really matter which variant we choose; the differences between the rival axiomatisations will not drastically affect the way we work or the style of reasoning we are able to use. This is because the differences between axiomatisations only become apparent in the curious realms of the transfinite, and computation is done in the realm of the finite (or at worst the countably infinite).

Apart from a little syntactic sugaring, the language presented in the first part of the course follows that presented in the first part of *[Abrial]* -- to whom those concerned with a more formal approach may turn. It will later become apparent to cognoscenti that there are differences with that language which are not merely cosmetic. For the moment, though, those differences may not be perceived, and in any case can safely be ignored.

Other informal presentations will be found in *[Halmos]*, and *[Gries]*.

If we were programmers studying a programming language with the intention of programming in it we might examine it in terms of expressions, declarations and commands and their respective meanings, and try to discover what it had in common with the languages we already knew. We'd also try to discover the purpose of any glaring unorthodoxies. As programmers and mathematicians studying one dialect of the language of set theory with the intention of writing specifications in it, we will find it useful to begin by examining its four interrelated sublanguages, namely the language of *terms*, the language of *predicates*, the language of *definitions*, and the language of *theorems*. In subsequent sections we will show how the languages are *extended*.

Assuming some familiarity with the ideas, vocabulary and symbolism of logic and set theory, we will try to give a general idea of the flavour of these component languages by means of a few simple examples. After reading these you should be able to point out the main differences between our notation and the "standard" notations. Understanding our reasons for introducing the differences will certainly take longer.

2.2 Term language

A term is a phrase of the language which corresponds intuitively to a *set* or to an *element* of a set. Examples of sets are a pack of wolves, a bunch of grapes, a flock of pigeons or a collection of books. An element of a set may be a wolf, a grape or a pigeon, a book, a number, or a function. A set may have elements which are other sets -- for example in classical geometry a line is a set of points so the set of lines in a plane is an example of a set of sets.

The simplest kind of term is a name, for example:

```
N
Bernard
```

The language of definitions (which we discuss later) is used for introducing new terms into a document written in the language of set theory, and giving them meaning. In order to simplify what follows, we will assume that N has already been defined in such a way that it denotes the set of Natural Numbers (nonnegative integers).

One way to specify a (finite) set of Numbers is to write down its elements one by one -- this is often called an *extensional* definition or specification. For example:

```
{1, 2, 3, 5, 7}
```

It is obviously out of the question to give extensional specifications for sets (even finite sets) above a certain size. We cannot, for example, write down all the elements of the set of prime numbers. Another kind of term is used to denote such large sets, and an example of this kind of term is:

```
{ n:N | (divisors n) = {1, n} }
```

which reads "the set of natural numbers n whose divisors are 1 and n". This style of set description is called *comprehensive*, it is used to specify *subsets* of a certain set by giving the characteristic properties of their elements.

The first part of such a term is a *signature*, in the case of our example:

```
n:N
```

which *introduces a new variable* n which may take values from the set of natural numbers (N) and whose scope is the second part of the term, namely the predicate

```
(divisors n) = {1, n}
```

In fact it doesn't matter what the name of the variable is: a number is in the set denoted by this term exactly when the divisors of that number are 1 and the number itself; indeed the same set can be denoted by:

```
{ m:N | (divisors m) = {1, m} }
```

or by:

```
{ cabbage:N | (divisors cabbage) = {1, cabbage} }
```

Yet another way to read this term is: "the set, each of whose elements is a Number whose divisors are 1 and itself".

2.3 Predicate language

A predicate is a phrase of the language which corresponds to a statement (about sets and/or elements) which may be true or false. For example: "x is smaller than 4", "There is no number which is larger than all the prime numbers", "All prime numbers are of the form n^2+m^2+1 for some numbers n and m", "The inverse of the successor function is a function".

A predicate is either a *primitive predicate* or is constructed from simpler predicates by means of propositional connectives which are denoted by the signs:

... ∧ ...	... *and* ...
... ∨ ...	... *or* ...
¬ ...	*not* ...
... ⇒ ...	*if* ... *then* ...
... ⇔ ...	... *exactly when* ...

Supposing that P stands for a predicate which corresponds to "There is no number which is larger than all the prime numbers", and that Q stands for a predicate which corresponds to "All prime numbers are of the form _", then the predicate:

```
P ∧ Q     .....    (P1)
```

corresponds to the statement "There is no number which is larger than all the prime numbers *and* all prime numbers are of the form _". The predicate

```
P ∨ Q     .....    (P2)
```

corresponds to the statement "There is no number which is larger than all the prime numbers, *or* all prime numbers are of the form _".

By definition in order to *prove* the truth of the first of these predicates (P1), of course, we would have to prove the truth of both its constituents, that is P and Q. (If you think you *can* prove Q then come and see me). By definition, in order to prove the truth of the second predicate (P2), all that is needed is to prove *one* of its constituents; in other words the symbol which we pronounce "or" corresponds to the idea "either _ or _ or both".

It turns out that the propositional connectives can all be defined in terms of *not* and *and* by syntactic equivalence, thus if P and Q are predicates we have:

$$P \vee Q \quad \triangleq \quad \sim(\sim P \wedge \sim Q)$$
$$P \Longrightarrow Q \quad \triangleq \quad \sim P \vee Q$$

The "exactly when" relationship between two predicates, P and Q (sometimes read "P if and only if Q") is written

$$P \Longleftrightarrow Q$$

it is the conjunction (and) of the predicates which correspond to "if P then Q" and "if Q then P", that is

$$P \Longleftrightarrow Q \quad \triangleq \quad P \Longrightarrow Q \wedge Q \Longrightarrow P$$

Evidently one strategy for proving an "exactly when" predicate would be to prove both its "if _ then _" components.

It is very important to understand that the truth of the statement "if P then Q" does not guarantee the truth of the statement "if not P then not Q". You might be able to convince yourself of this by considering, for example, the "real life" fact "*if it is raining then the roof is wet*": if "*it is not raining*" is the only fact we know, then we *still* can't conclude anything about the wetness of the roof (it may just have stopped raining, or a flock of herons with bladder trouble may have just flown over). Perhaps it would be easier to convince oneself of this if instead of saying "if P then Q" we said "Q must hold whenever P holds".

The first *primitive predicate* we will introduce is the one which corresponds to the concept of of *membership* or *belonging*. If the element x belongs to the set S then the predicate

$$x \in S$$

is true; otherwise the predicate is false. For example,

$$3 \in \text{primes}$$

is true, but

$$2345678 \in \text{primes}$$

is false.

Next we introduce the predicate which corresponds to the *equality* relationship between sets: two sets are said to be equal if they have exactly the same elements. If S and T denote subsets of a set which have exactly the same elements then we write S=T.

The third predicate to be introduced is the "is a subset of" predicate: if every element of a set S is also an element of the set T then we say that S is a subset of T and write:

$$S \subseteq T$$

If S is a subset of T which does not have exactly the same elements as T then we say it is a *proper subset* of T and write:

$$S \subset T$$

Later in this section we will formalise the connection between these three kinds of predicate.

The predicate which corresponds to the notion "all _ have the property _" is written with a symbol which frightens some people. For example, the statement "all numbers when added to themselves produce a prime number" is expressed as the predicate:

$$\forall\ n{:}\mathbb{N}\ .\ (n+n) \in primes$$

Of course this predicate — and the statement to which it corresponds — is false, but that doesn't mean we can't write it down.

Here are some more examples:

$$(a+1 = b+1)$$
$$divisors\ n = \{1, 5, 7\}$$
$$n \in primes$$
$$\{\ n{:}\mathbb{N} \mid (divisors\ n) = \{1, n\}\ \} \neq \{\}$$
$$\exists\ n{:}\mathbb{N}\ .\ (divisors\ n) = \{1, n\}$$
$$\forall\ a{:}\mathbb{N};\ b{:}\mathbb{N}\ .\ (a+1 = b-1)$$

The last predicate — which formalises the statement that for all natural numbers a and b, a+1 is equal to b-1 — is *false*, whereas the first three may be true or false depending on the values of n, a, and b — we need to know more about these values in order to discover whether or not the predicates are true. There are many situations in which it may not be possible to demonstrate the truth or falsity of a predicate.

The alert reader will have noticed that the predicates above aren't independent of each other. The penultimate predicate — which is also written with a symbol which some people claim to be frightened of — may be read "there is a natural number, n, which has divisors 1 and n". It is true exactly when the fourth predicate

$$\{\ n{:}\mathbb{N} \mid (divisors\ n) = \{1, n\}\ \} \neq \{\}$$

which may be read "the set, each of whose elements is a Number whose divisors are _ is not the empty set" is true (and false exactly when the fourth predicate is false).

We can use "exactly when" to encapsulate the connection between the subset relationship and a "for all _" predicate more formally. The following predicate is always true for any sets S and T:

$$(S \subseteq T) \quad \leftrightarrow \quad (\forall\ x{:}S\ .\ x \in T)$$

We can do the same for equality of sets; the following predicate is always true for any sets S and T:

$$((\forall\ x{:}S\ .\ x \in T) \wedge (\forall\ x{:}T\ .\ x \in S)) \Leftrightarrow (S = T)$$

Since "exactly when" is a transitive relationship (that is for any predicates P, Q and R, if P holds "exactly when" Q holds, and if Q holds "exactly when" R holds, then P holds "exactly when" R holds) the following connection between the subset relation and the equality relation follows from the two connections just outlined:

$$((S \subseteq T) \wedge (T \subseteq S)) \Leftrightarrow (S = T)$$

Finally, we can express a general relationship between statements of the form "there is a _ for which the property _ holds" and "all _ have the property _", namely that for all sets S and predicates, P

$$(\forall\ x{:}S\ .\ P) \Leftrightarrow \neg(\ \exists\ x{:}S\ .\ \neg P\ )$$

In other words P holds for all elements of S exactly when there is no element of S for which the negation of P holds.

2.4 Definition language -- Syntactic Equivalences

A *simple syntactic equivalence* is a phrase of the language which associates a name with a term. For example, the following definition associates the name primes with a term denoting the set of all prime numbers.

$$\texttt{primes} \triangleq \{\ n{:}\mathbb{N} \mid \texttt{divisors}\ n = \{1, n\}\ \}$$

It signifies that primes is a shorthand for the term on the right hand side of the $\triangleq$ sign. For example, if we are asked to prove that

$$3 \in \texttt{primes}$$

then the first thing we do is to substitute the definition of primes for primes itself, and try to prove that:

$$3 \in \{\ n{:}\mathbb{N} \mid \ldots\ \}$$

which we will probably do by trying to prove that the predicate _ holds when 3 is substituted systematically for n in it.

Another form of the same definition is:

primes
$$\{\ n{:}\mathbb{N} \mid \texttt{divisors}\ n = \{1, n\}\ \}$$

It means exactly the same as the "$\triangleq$" form.

More complicated forms of syntactic equivalence will be introduced as the need arises.

2.5 Theorem language

Stripped of its prose any text in the language of set theory consists at the "top level" of several definitions, followed by several theorems. A *theorem* is a statement *about* the definitions; it asserts that a certain predicate can be (has been) proved to be true from the definitions themselves and the *rules of reasoning* of the language of set theory.

The simplest form of theorem is written as a turnstile sign followed by a predicate:

```
⊢ Predicate
```

The following theorem, for example, is tantamount to an assertion that we have *proved* that 3 is an element of the set denoted by the name primes.

```
⊢ 3 ∈ primes
```

Of course have not yet introduced the *rules of reasoning* so it is not strictly possible for our readers to prove this theorem. What is possible is to argue *informally* from an intuitive knowledge of the properties of sets and of the divisors function but this *informal argument* should not be mistaken for *formal proof*. Indeed there is another problem: we have yet to write down a definition for "divisors", any informal reasoning must at present be based on the fact that the name "divisors" suggests that the function in question maps a number into the complete set of its divisors -- which may not be so!

We will say more about the rules of reasoning later. For the moment it is sufficient to understand that *a theorem is not a predicate*, nor may a theorem be written within a predicate; for example the following pseudo-set-theoretical text is *not part of the language of set theory*.

WRONG `4 ∈ (divisors n) ⇒ ( ⊢ n ∉ primes )` **WRONG**

2.6 Definition language -- Signatures

2.6.1 -- Variables

In order to introduce a new *variable* we must first indicate the set of values over which the variable is permitted to range; we do so by means of a *signature*. For example, the following signatures associate respectively n with the set N (*natural number*) and a and b with the set *sequences of* N.

```
n: N
a, b: seq N;
```

The phrases within which signatures introduce variables include the predicates

```
∀ Signature . Predicate

∃ Signature . Predicate
```

and the terms

```
( Signature | Predicate )

λ Signature | Predicate . Term
```

The signs

$$\exists, \quad \forall$$

are called "quantifiers" in classical logic. When we use this word we also include the signs:

$$\lambda, \ (\ldots \mid \ldots)$$

We will say more about λ later.

The *scope* of the association between name and type (that is those parts of the text from which the association is visible) for variables introduced this way is the text of the Predicate, (or in the case of the λ-term the text of the Predicate and the Term) except where the association is temporarily made invisible by an intervening signature for the same name.

For example in the following predicate

$$\forall\ m{:}\mathrm{LECT}\ .\ \exists\ n{:}\mathrm{CAR}\ .\ pred_1 \wedge (m{:}\mathbf{N} \mid pred_2) = ()$$

the association m:LECT is "visible" in pred1, the association n:CAR is visible in pred1 and pred2, and the association m:N is visible in pred2.

2.6.2 -- Constants

In order to introduce a new *constant* into a piece of mathematical discourse, we also give a signature for it; this time at the "top level". The signature may be associated a predicate which constrains its value in some way. Predicates which constrain constants introduced in this way are sometimes called *axioms*.

For example this is how to introduce a constant (of type) Number, whose value we require to be between seven and nineteen, but about which we wish to be no more specific:

$$\begin{array}{|l|}\hline processorspernode: \mathbf{N} \\ \hline 7 < processorspernode < 19 \\ \hline \end{array}$$

The double horizontal line has no significance, except that it serves as typographical emphasis that the signature appears at the top level. Likewise the short horizontal line simply separates the signature from the predicates with which it is associated, and the long single horizontal line emphasises the end of the predicates.

The following signature and its associated predicate introduce a constant function from numbers to numbers, named foo, which maps every number into its square:

$$\begin{array}{|l|}\hline foo: \mathbf{N} \rightarrow \mathbf{N} \\ \hline \forall\ n{:}\mathbf{N}\ .\ foo\ n = n \times n \\ \hline \end{array}$$

Of course we haven't explained properly what we mean by the sign $\rightarrow$ or by the word *function*! We will do so later.

A constant defined in this way is visible throughout the whole of the remaining "mathematical discourse", except for those places where variables with the same name are visible.

2.7 Set Comprehension Revisited

Mathematicians sometimes use another form of comprehensive specification to denote the set of all things "of a certain kind", for example:

$$\{ n:\mathbb{N} \mid n^2 \}$$

denotes the set of all things of the form

$$n^2$$

where n ranges over the natural numbers. Another example is

$$\{ m, n:\mathbb{N} \mid m^2+n^2<44 \mid n^3-m^3 \}$$

which denotes the set of all things of the form

$$n^3-m^3$$

where m and n range over the numbers in such a way that the sum of their squares is less than 44.

In fact the general form of a comprehensive set specification is:

$$\{ \text{Signature} \mid \text{Predicate} \mid \text{Term} \}$$

There are a number of special cases of this form which can be abbreviated. If the Predicate doesn't constrain the variables of the signature (in other words if it is identically true) then we leave it out and write the comprehension as:

$$\{ \text{Signature} \mid \text{Term} \}$$

If the Signature introduces a single variable and the Term *is* that variable then we leave out the Term and and write the comprehension as:

$$\{ \text{Signature} \mid \text{Predicate} \}$$

This form of abbreviation will later be generalised.

If we leave out both Term and Predicate we just get the degenerate case:

$$\{ \text{Signature} \}$$

An example of this is

$$\{ x:X \}$$

which is an abbreviation for

```
{ x:X | true . x }
```

which is the set of elements x of X for which the predicate true holds. Of course this is just the set

```
X
```

itself.

Some mathematicians use one or both of the syntactic forms:

```
{ Term | Signature | Predicate }
{ Term | Signature }
```

to denote set comprehension.

2.8 Structured Types: Part 1

We have shown how to specify subsets of a given set using *extensional* and *comprehensive* specifications, but these (sub)-sets can never have more in them than the original set. In this section we introduce two means of constructing "bigger" sets from given sets; the remaining method (tree-constructions) will be introduced later.

2.8.1 Cross-Products -- Sets of Tuples

If T1 and T2 are both sets, then the term

$$T_1 \times T_2$$

denotes another set, namely the set of two-tuples (ordered pairs) whose first elements are drawn from T1 and whose second elements are drawn from T2. This set is sometimes called the *product* or *cross-product* of T1 and T2.

The ordered pair whose first element is a and whose second element is b is written

```
(a, b)
```

If we have somewhere defined

```
LECT ≙ { BS, TH }
CAR  ≙ { RWR360W, PVM495W, A420GBH }
```

then the term

```
LECT×CAR
```

denotes the set of ordered pairs

```
{ (BS, RWR360W), (BS, PVM495W), (BS, A420GBH),
  (TH, RWR360W), (TH, PVM495W), (TH, A420GBH) }
```

This set has six elements -- the arithmetical *product* of the number of elements in LECT and the number of elements in CAR, hence the name *cross-product*.

Another term denotes the same set, namely

```
( l:LECT; m:CAR . (l, m) )
```

It can be read as "the set of all ordered pairs (l,m) where l is an element of LECT and m is an element of CAR." By generalising the abbreviation introduced in section 2.7 we can rewrite this term as:

```
( l:LECT; m:CAR )
```

Subsets of the cross product are denoted by terms of the form:

```
( l:LECT; m:CAR | Predicate . (l, m) )
```

which (again generalising the abbreviation of section 2.7) can be shortened to

```
( l:LECT; m:CAR | Predicate )
```

We can generalise the notation for two-tuples to that for n-tuples by using the following syntactic equivalences:

$$T_1 \times T_2 \times T_3 \quad \triangleq \quad T_1 \times (T_2 \times T_3)$$

and

$$(x, y, \ldots z) \quad \triangleq \quad (x, (y, \ldots z))$$

For example:

```
LECT×CAR×{1983, 1982, 1981}
```

denotes the set of triples

```
( l:LECT; m:CAR; y:N | y∈{1983, 1982, 1981} . (l, m, y) )
```

2.8.2 Power Sets -- Sets of Subsets

The *powerset* of a given set, S say, denoted by the term

```
P S
```

is the set whose elements are the subsets of S. For example the powerset of LECT is

```
( {}, {BS}, {TH} {BS, TH} )
```

Notice that the number of elements of P S is 2 to the *power* of the number of elements of S (hence the name).

The term {} denotes the empty set, whose properties with respect to *any* set X can be summed up by the theorem:

```
⊢  ∀ x:X . x∉{}
```

In other words -- *no member of any set is a member of the empty set.*

What is more, the properties of an empty subset SS of a set X with respect to *any* predicate P can be summed up by:

$$\vdash \quad \forall\, SS{:}\mathbb{P}\, X \;.\; SS{=}\{\} \Rightarrow (\forall\, x{:}SS \;.\; P)$$

In other words *any* predicate about *all the elements* of an empty set is *true*; of course since there are no such elements this fact is not too useful!

Notice that the powerset of the empty set has exactly one element, namely the empty set itself; that is:

$$\vdash \quad \mathbb{P}\,\{\} = \{\,\{\}\,\}$$

This should demonstrate that it is important to be very clear about the distinction between a set with no elements and a set whose single element is a set with no elements (if you don't like this, then reflect on the difference between an empty tea packet, and a cupboard with only an empty tea packet inside).

The powerset of $\mathbb{N}$ is very large (so big that its size isn't expressible as a Natural number). Here is one way of specifying a constant whose value is a single element (*ie* a set of numbers) of that huge set...

$$\begin{array}{|l}\hline \quad Even: \mathbb{P}\,\mathbb{N} \\ \hline \quad \forall\, n{:}\mathbb{N} \;.\; n \in Even \Leftrightarrow (\exists\, m{:}\mathbb{N} \;.\; n = m+m) \\ \hline \end{array}$$

We read this "Even is a set of numbers, and every number n is in even exactly when it is twice some number m" Here's another example...

$$\begin{array}{|l}\hline \quad Odd: \mathbb{P}\,\mathbb{N} \\ \hline \quad Odd = \{\, n{:}Even \;.\; n+1 \,\} \\ \hline \end{array}$$

"Odd is the set of numbers of the form n+1 for even numbers n."

As an exercise try specifying Even in the style we used for specifying Odd and specifying Odd in the style we used for specifying Even.

2.8.3 Finite Subsets

The idea of a finite set is quite familiar to us, but it's surprisingly hard to find a *simple* formal definition. Informally, the finite sets are those whose elements we can "count".

Notation: If X is a set, then we write

$$\mathbb{F}\, X$$

to denote the set of finite subsets of X. Evidently if X is itself finite we have:

```
F X = P X
```

It can also be proved that

```
⊢  F X = {} ∪ { x:X; ss:F X . {x}∪ss }
```

in other words, the set of finite subsets of X contains the empty set and those made by "adding a single element of X to a finite subset of X".

The number of elements in a finite set, S, sometimes called its *cardinality*, or *size*, is written #S . The important properties of the # operator, which reflect the fact that finite sets can be built up by adding elements "one by one" to the empty set, are:

```
# {} = 0
∀ S:F X . ∀ x:X . x∉S ⟹ #({x}∪S)=1+#S
```

If T does *not* denote a set, then the term #T has no meaning.

This concludes our introduction to the flavour of the sublanguages of the language of set theory. By now you should understand the ideas to which terms and predicates correspond, what a signature is, how to specify a subset of a given set, and how to specify sets of sets and sets of tuples.

3: The Language of Relations and Functions

In this section of the course, we show how the basic language of set theory is extended to include notations which allow us to describe relations and functions.

3.1 Binary Relations

You are already familiar with the concept of a *binary relation* — for example in mathematics the relations "is less than", "is a subset of" or in law "owns a car whose registration number is". In the language of set theory, binary relations are considered to be sets of pairs.

For example, the "less than" relation on the natural numbers is the set of pairs

```
( i:N; j:N  | (∃ k:N . i+k=j ∧ k≠0) )
```

(though we wouldn't *define* it in this way).

Notation: If R is a binary relation between elements of A and of B, and if a:A and b:B then the predicate

```
a R b
```

is syntactically equivalent to the predicate

```
(a,b) ∈ R
```

Sometimes we write a maps to b under R or R maps a to b; using this terminology, of course, we see that "<" maps (for example) 3 to every number bigger than three.

Relations can be *finite*, for example suppose that OWNER is the set of all potential car owners, that REG is the set of car registrations, and that MAKER is the set of car manufacturers. Without concerning ourselves with the *internal* structure of these sets let's also suppose that we have certain distinguished constants, namely

```
TH, JS, BS, RB, IS:                        OWNER
Ford, Bentley, Renault, Morris, Datsun: MAKER
a420gbh, rwr360w, pvm495, is400p, a190: REG
```

Then we could specify relations owns and made respectively:

```
( (TH, a420gbh), (BS, rwr360w), (BS, pvm495), (IS, is400p) )

(  (Renault, a420gbh), (Morris, rwr360w)
   (Morris,  pvm495),  (Datsun, is400p), (Bentley, a190)   )
```

In which case the following predicates would be true

```
TH      owns a420gbh
BS      owns rwr360w
Morris made rwr360w
```

whereas these are false

```
BS       owns  is400p
TH       owns  pvm495
Bentley  made  pvm495
Ford     made  a420gbh
```

It is customary to use the following syntactic sugar for the the pairs which make up a relation:

```
a↦b  ≙ (a,b)
```

so owns would be written

```
( TH↦a420gbh, BS↦rwr360w, BS↦pvm495, IS↦is400p )
```

and "<" could be written

```
( i:N; j:N | (∃ k:N ... ) . i↦j )
```

Notation: Given two sets X and Y the set of relations between X and Y — which is denoted X↔Y — is the powerset of their cross product. More formally:

```
[X,Y]      X↔Y   ≙   P(X×Y)
```

This is a more complicated form of syntactic equivalence which comes in two parts; the first part reads "given two sets X and Y" and the second defines the left hand *term* to be syntactically equivalent to the right hand term. We could also have written it as

```
┌─ X↔Y [X,Y] ──────────────────────┐
│   P(X×Y)                         │
└──────────────────────────────────┘
```

3.2 Domain and Range of a Relation

Given a relation

```
R: X↔Y
```

the *domain* of R — written dom R — is the set of all elements of X which R relates to at least one element of Y. In other words:

```
dom R  = ( x:X | (∃ y:Y . x R y) )
```

The *range* of R — written ran R — is the set of all elements of Y which are related by R to at least one element of X, that is:

```
ran R  = ( y:Y | (∃ x:X . x R y) )
```

A relation whose domain and range are subsets of the *same* set is called a *homogeneous* relation. The *empty* relation ∅ is just the empty set of pairs. Its domain and range are empty; it isn't too interesting (to be more precise it's about as interesting as the number 0 or the empty set).

3.3 Partial Functions

Functions are amongst the most basic tools of mathematicians and computer programmers. As programmers you may be used to thinking of functions as "recipes for computing" certain quantities; in the language of set theory (indeed in the whole of mathematics apart from computation) it is convenient to treat functions more abstractly than this, namely as special kinds of relation.

A relation R is called a *partial function* if it maps each element of its domain to *exactly one* element of its range. More formally:

$$X \pfun Y \ [X, Y] \quad \{ R : X \rel Y \mid (\forall x : X; y_1, y_2 : Y \bullet (x R y_1 \land x R y_2) \implies y_1 = y_2) \}$$

The so-called *total functions* from a set X are defined by

$$X \fun Y \ [X, Y] \quad \{ f : X \pfun Y \mid \mathrm{dom}\ f = X \}$$

In referring to a function as total it is important to say *from what set* it is total. This is because in general we can derive many "total" functions from a partial function — one for each subset of its domain. More formally:

$$\forall f : X \pfun Y \bullet \forall S : \mathbb{P} X \bullet S \subseteq (\mathrm{dom}\ f) \implies \{ x : X; y : Y \mid x \in S \land x\ f\ y \} \in (S \fun Y)$$

We use the phrase *proper relation* to describe a *relation* which is not functional. (For example "<").

Notation: if $F : X \rel Y$ is a function, and if x is an element of the domain of F, then the term

$$F\ x$$

means the unique y in Y such that xFy. If, on the other hand, x is *not* an element of the domain of F then we cannot conclude anything about that term.

For example, consider the "unsquare" function

$$unsq = \{ x : \mathbb{N} \bullet x^2 \mapsto x \}$$

The term

$$unsq\ 4$$

means 2, because 2 is the unique number satisfying 4 unsq x. On the other hand, the term unsq 6 has no meaning (or to be more precise, *cannot be reasoned about any further*).

Infix Notation: if $F : (X \times Y) \rel Z$ is a function, and if $x : X$ and $y : Y$ then the following syntactic equivalence holds

$$x\ F\ y \quad \widehat{=} \quad F(x, y)$$

At first sight this appears to be syntactically ambiguous, since if F is a function it is also a relation, and we already defined the *predicate*

$$a\ R\ b \quad \Leftrightarrow \quad (a,b) \in R$$

The apparent ambiguity may be resolved by inspecting the "shape" of the function and of its operands. To be more precise, if x, y, z, and F are introduced by

$$x : X$$
$$y : Y$$
$$z : Z$$
$$F : X \times Y \twoheadrightarrow Z$$

then the phrase

$$(x\ F\ y)$$

is a *term* of type Z. On the other hand the phrase

$$(x,y)\ F\ z$$

is a *predicate*, as is the phrase:

$$(x,y)\ F\ (x\ F\ y)$$

For the moment our language doesn't have to be understood by computers, so there's no need when defining a function F to say whether xFy or $F(x,y)$ is the form we'll use.

3.4 Specification by Predicate

We have hitherto defined relations and functions by giving extensional or comprehensive specifications of sets of pairs. We now introduce a more convenient style of specification, namely "by predicate". For example, here is a specification of the relation "is the square of"

```
squares: N ↔ INT
─────────────────────────────────
∀ n:N; i:INT .
    (n squares i) ⇔ (i² = n)
```

and here is a specification of a fairly uninteresting total function

```
boring: N → N
─────────────────────────────────
∀ n:N . boring n = n² + 31n + 2
```

The defining predicate is equivalent to the predicate

$$boring = (n:N \;.\; n \mapsto n^2+31n+2)$$

If we were being careful we would check that the polynomial term denotes a natural number for *all* n:N for if it did not, then the defining predicate would not be consistent with the signature; this is because the signature requires that the constant boring take its value from the set of *total* functions from N.

Finally here is a specification of the function which maps a nonempty set of numbers to its minimum element (P_1 means the set of nonempty subsets)

```
min: (P₁ N) → N
─────────────────────────────────
∀ S:P₁ N .
   min S ∈ S ∧
   ∀ x:S . min S ≤ x
```

Notice that in this latter case we give no hint about how to discover the number in question, we just give its properties. Such property-oriented specifications, particularly of functions, may seem a little strange to programmers who are used to giving "computational recipes"; notwithstanding this they are widely used in mathematics.

One slight *abuse of language* which we permit is to omit the topmost universal quantification when to do so would cause no confusion. This is usually the case when specifying relations or (total) functions, for example

```
squares: N ↔ INT
boring:  N → N
─────────────────────────────────
n squares i ⇔ i²=n
boring n = n² + 31n + 2
```

3.5 λ-expressions

In some branches of computer science it is customary to denote functions by (so-called) lambda-expressions. The following syntactic equivalence defines these terms:

$$\lambda \text{ Signature} \mid \text{Predicate} \cdot \text{Term} \triangleq (\text{Signature} \mid \text{Predicate} \cdot (\text{variables of Signature}) \mapsto \text{Term})$$

For example:

$$\lambda\ m, n{:}\mathbb{N} \mid m \geq n \cdot (m+n, m-n) \triangleq (m, n{:}\mathbb{N} \mid m \geq n \cdot (m,n) \mapsto (m+n, m-n))$$

which denotes a function which maps pairs of numbers into pairs of numbers.

Exercise:

Write down a term which denotes the *range* of the function denoted by

$$\lambda \text{ Signature} \mid \text{Predicate} \cdot \text{Term}$$

Notation:

If $\square$ is a binary infix operator with signature $\_ \square \_ : X \times Y \to Z$, then

$$(\_ \square y) \triangleq \lambda\ x{:}X \cdot x \square y \quad \text{and} \quad (x \square \_) \triangleq \lambda\ y{:}Y \cdot x \square y$$

For example consider $\_ + \_ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $\_ - \_ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

$(3 + \_)$ is a function which adds 3 to its argument.
$(55 - \_)$ is a function which subtracts its argument from 55.

3.6 On Unsatisfiable Specifications

The pattern of much of the software architect's work is to specify what a problem is, using the mathematical notation, then go on to discover whether or not the specification is satisfiable. It is important to realise that the language of mathematics is sufficiently powerful to allow us to specify things which may not exist. For example, consider

$$\text{boring?} : \mathbb{N} \to \mathbb{N}$$
$$\forall\ n{:}\mathbb{N} \cdot \text{boring?}\ n = n^2 - 31n + 2$$

In this case the polynomial term doesn't denote a natural number for all values of n, so the specification is unsatisfiable. In other words, whilst there is a *partial* function on the natural numbers with the indicated property, there is *no such total function*. Now consider:

$$\text{prime?} : \mathbb{N}$$
$$\forall\ n{:}\text{primes} \cdot \text{prime?} > n$$

Unsatisfiable specifications aren't always so immediately and demonstrably unsatisfiable!

4: The Mathematical Toolkit

We're now in a position to use the definitional notation, introduced earlier, to introduce the "kit" which we'll need for subsequent specifications.

4.1 Operations on Sets

If X is a set, then we can specify the difference, union, and intersection operations, and the inclusion relation(s) on subsets of X as follows:

```
[X]
```

```
_ - _,
_ ∪ _,
_ ∩ _: (P X) × (P X)  → (P X)
```

```
S₁ - S₂ = ( x:X | x∈S₁ ∧ x∉S₂ )
S₁ ∪ S₂ = ( x:X | x∈S₁ ∨ x∈S₂ )
S₁ ∩ S₂ = ( x:X | x∈S₁ ∧ x∈S₂ )
```

```
_ ⊆ _,
_ ⊂ _: (P X) ↔ (P X)
```

```
S₁ ⊆ S₂ ⇔ ∀ x:S₁ . x∈S₂
S₁ ⊂ S₂ ⇔ (S₁≠S₂ ∧ S₁⊆S₂)
```

These operators have a large number of properties, with which every aspiring mathematician and computer scientist must become familiar. If you have any doubts about your understanding of them then do the exercises in *[Lipschutz]*.

4.2 Functions and Relations as "Data"

Functions and relations whose domains or ranges are sets of functions or relations are called "higher order". It is important to understand that higher order functions and relations are as easy to define as the "simple" functions and relations we have met up to now.

First we take a practical example: let us suppose that we wish to model a database which records the owners of cars. If we make the simplifying assumption that every registration number has an owner (perhaps the "ministry of transport") then the state of this database at any stage in its evolution can be modelled by a total function of type DB defined by

```
DB ≙ REG → OWNER
```

A family of transactions, each of which records the fact that a person a has bought a car whose registration is r can be modelled by the function...

```
buys: (OWNER×REG) → (DB→DB)
```

```
(a buys r) db = db - ( r↦(db r) ) ∪ ( r↦a )
```

This is a function of two arguments, whose result is itself a function. For each owner a and registration r there is a function

a buys r

which maps our model of the state of the database before the transaction into our model of its state after the transaction.

Since the model of the database is itself a function, we evidently have defined a function whose result is a function from functions to functions. This may be a bit mind-blowing for people who are used to programming in languages where functions aren't "first-class" objects: remembering that we are writing in a *descriptive* language rather than a programming language may help to calm you down.

Notice that we used union and set difference to describe the relationships between the functions modelling the database before and after the transaction. This is perfectly legitimate: the sets are (in this case) sets of pairs. This is where the mathematical idea of a function (relation) as a set of pairs begins to pay off; we can operate on functions and relations using the same operators as we use to operate on sets. In the next section we use this freedom in order to specify some very powerful operators indeed.

4.3 Operations on Relations and Functions

An operator which may already be familiar is (forward) composition of relations, defined by

$$\_ \fatsemi \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \rightarrow (X \leftrightarrow Z)$$

$$(x,z) \in (R_1 \fatsemi R_2) \Leftrightarrow \exists\, y:Y \;.\; (x,y) \in R_1 \wedge (y,z) \in R_2$$

For example, suppose the relation

madeby =
(a420gbh↦Renault, rwr360w↦Morris,
  pvm495↦Morris, is400p↦Datsun, a190↦Bentley)

Then the relation ownsacarmadeby, defined by

ownsacarmadeby = owns ⨾ madeby

is (in extenso)

(BS↦Morris, TH↦Renault, IS↦Datsun)

Notice that it also *happens* to be a function.

Some authors also use the sign • defined by:

$$\_ \bullet \_ : (Y \leftrightarrow Z) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Z)$$

$$R_2 \bullet R_1 = R_1 \fatsemi R_2$$

4.4 A Simple Example: Family Life

Suppose we are designing a database in which to keep information about families. Let P denote the set of all persons; each person has but a single biological mother and a single biological father and we capture this fact by introducing two partial functions

$$ma, pa: P \nrightarrow P$$

The functions are partial because we cannot hope to record this information about everybody who has ever lived. We do insist, though that everbody who has a mother has a father and vice versa, (thereby ruling out immaculate conception). We also know that nobody can be both a father and a mother, and record these real-world constraints by adding the predicates

$$\begin{aligned} &dom\ ma = dom\ pa \\ &ran\ ma \cap ran\ pa = \{\} \end{aligned}$$

to our specification. For the moment we shall ignore several other real-world constraints. We can now define several other family relationships using composition and union. For example:

$$\begin{aligned} &parent, \\ &grandma, \\ &grandpa: \quad P \leftrightarrow P \\ \hline &parent = ma \cup pa \\ &grandma = parent \fatsemi ma \\ &grandpa = parent \fatsemi pa \end{aligned}$$

In order to define brother and sister we need a few more tools.

The inverse function

$$\begin{aligned} &inv: (X \leftrightarrow Y) \rightarrow (Y \leftrightarrow X) \\ \hline &(y,x) \in (inv\ R) \Leftrightarrow (x,y) \in R \end{aligned}$$

maps a binary relation into its inverse.

Notation: if R is a relation then

$$R^{-1} \triangleq (inv\ R)$$

For example:

$$madeby = made^{-1}$$

Note that the inverse of a function is not necessarily a function. For example, consider the function:

$$\_ + \_ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

Its inverse is a relation which holds between every number n and any pair of numbers whose sum is n.

Exercise:

Write a *comprehensive* specification of the inverse of +.

The identity relation on subsets of a set is defined by

```
Id: (P X) → (X↔X)
---------------------
Id S = ( x:S . x↦x )
```

Exercise:

Summarise in prose the "effect" on a relation of composing it on the left/right with an identity on a *proper* subset of its domain/range? Suppose that R is a relation, S is a subset of its domain and T is a subset of its range. Write down terms which denote the set of pairs which comprise the relations.

```
R;(Id T)
(Id S);R
```

Resuming, for the moment, our "family life" example, let's now suppose that we keep a record of who is male and who is female; nobody is both:

```
m, f: P P
---------
m ∩ f = {}
m ∪ f = P
```

The real-world constraint which requires mothers to be female and fathers to be male is recorded by—

```
ran pa ⊆ m
ran ma ⊆ f
```

The relations sister and brother are now (almost) respectively definable by

```
parent ; parent⁻¹ ; (Id f)
parent ; parent⁻¹ ; (Id m)
```

All that is left is to prevent males from being their own brothers and females from being their own sisters:

```
sister  = (parent ; parent⁻¹ ; (Id f)) - (Id f)
brother = (parent ; parent⁻¹ ; (Id m)) - (Id m)
```

Notice that we used right-composition with an identity relation to *restrict* the size of a relation. The following operators restrict relations by specifying a restriction on their domain and range respectively:

$$\_ \upharpoonright \_ : (X \leftrightarrow Y) \times (\mathbb{P}\ X) \rightarrow (X \leftrightarrow Y)$$
$$\_ \setminus \_ : (X \leftrightarrow Y) \times (\mathbb{P}\ X) \rightarrow (X \leftrightarrow Y)$$

$$R \upharpoonright S = (\mathrm{id}\ S) \fatsemi R$$
$$R \setminus S = R \upharpoonright (X - S)$$

$$\_ \downarrow \_ : (X \leftrightarrow Y) \times (\mathbb{P}\ Y) \rightarrow (X \leftrightarrow Y)$$
$$\_ / \_ : (X \leftrightarrow Y) \times (\mathbb{P}\ Y) \rightarrow (X \leftrightarrow Y)$$

$$R \downarrow T = R \fatsemi (\mathrm{id}\ T)$$
$$R / T = R \downarrow (Y - T)$$

The domain restriction operator may be used with its operands reversed, as if defined by:

$$\_ \upharpoonright \_ : (\mathbb{P}\ X) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$$

$$S \upharpoonright R = (\mathrm{id}\ S) \fatsemi R$$

For example,

$$(\lambda\ x{:}\mathbb{N} \bullet x^2) \upharpoonright \{ 2,3 \} = \{ 2 \mapsto 4,\ 3 \mapsto 9 \}$$

$$(\lambda\ x{:}\mathbb{N} \bullet x^2) \downarrow \{ 2,3 \} = \{ \}$$

$$(\lambda\ x{:}\mathbb{N} \bullet x^2) \downarrow 1..8 = \{ 1 \mapsto 1,\ 2 \mapsto 4 \}$$

$$\{ 2,\ 3 \} \upharpoonright (\lambda\ x{:}\mathbb{N} \bullet x^2) = \{ 2 \mapsto 4,\ 3 \mapsto 9 \}$$

4.5 The Registration Database Revisited

Consider, for a moment, the database example in our original definition of buys we had to write a rather unweildy term to denote the database after the transaction. The operators we have just introduced allow us to make this a bit less cumbersome; it is easy to show that

$$db \setminus (r) \cup \{ r \mapsto a \} = (a\ \underline{\mathrm{buys}}\ r)\ db$$

In fact the idea of a relation being like another "except _" occurs so frequently in specification that we introduce another operator -- the *relational override* operator:

$$\_ \oplus \_ : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$$

$$R_1 \oplus R_2 = (R_1 \setminus (\mathrm{dom}\ R_2)) \cup R_2$$

For example, the function:

db ● (r↦a)

"behaves" like db except that it maps r to a. A slightly more interesting example is the following characterisation of a database transaction which allows "simultaneous" registration of a number of vehicles to a single owner (perhaps a car wholesaler):

```
bulkbuys: OWNER×(F REG) → (DB→DB)
(a bulkbuys S) db = db ● ( r:S . r↦a )
```

4.6 Generalised Application: Image

The *image* of a set S through a relation R (sometimes called the R-image of S) is the set of elements of R's "destination" to which R maps elements of S. The function

```
Im: (X ↔ Y) → ((P X) → (P Y))
Im R S = ( y:Y | (∃ x:S . xRy) )
```

maps a relation R into a function which maps a set S of elements into its image through R.

For example, consider the relation owns of section 3: here are some examples of *images* through it

```
Im owns (BS) =      (rwr360w, pvm495)
Im owns (BS, RB) = (rwr360w, pvm495)
Im owns (TH, BS) = (rwr360w, a420gbh, pvm495)
```

Notation: if R:X↔Y is a relation and S a subset of Y, and y an element of Y then

```
R [ S ]  ≙ Im R S
R [ y ]  ≙ Im R (y)
```

For example

```
owns [ (IS) ]          = (is400p)
owns [ IS ]            = (is400p)
owns [ (RB, IS, TH) ] = (is400p, a420gbh)
owns [ () ]            = ()
```

4.7 Properties:

If R, R1, R_2:X↔Y are relations, and if S, S_1 and S_2 are subsets of X, and if T is a subset of Y, then the following predicates (amongst others) always hold:

$$(R^{-1})^{-1} = R$$

$$\mathrm{dom}\ R^{-1} = \mathrm{ran}\ R$$

$$\mathrm{ran}\ R^{-1} = \mathrm{dom}\ R$$

$$(R_1 \cup R_2)^{-1} = R_1^{-1} \cup R_2^{-1}$$

$$(R_1 \cup R_2)[\ S\ ] = R_1[\ S\ ] \cup R_2[\ S\ ]$$
$$R[\ S_1 \cup S_2\ ] = R[\ S_1\ ] \cup R[\ S_2\ ]$$

$$\mathrm{dom}(\ R_1 \ ;\ R_2\ ) = R_1^{-1}[\ \mathrm{dom}\ R_2\ ]$$
$$\mathrm{ran}(\ R_1 \ ;\ R_2\ ) = R_2[\ \mathrm{ran}\ R_1\ ]$$

$$\mathrm{dom}(\ R \upharpoonright S\ ) = S \cap (\mathrm{dom}\ R)$$
$$\mathrm{ran}(\ R \downarrow T\ ) = T \cap (\mathrm{ran}\ R)$$

Exercises:

(1) Prove the properties outlined above.

(2) We have outlined some properties of union with respect to the image and inverse operators; what are the properties of *intersection* with respect to these operators?

(3) Siblings are people who share the same parents. Half-siblings share the same father or the same mother but not both. Specify the relations sibling, half-sibling, half-sister, cousin, great-aunt. What is interesting about the relations cousin and sibling? Can you specify the relation "childless aunt"?

4.8 Finite Sequences

Finite sequences are important because they allow us to capture the essence of entities as diverse as lists, files, arrays, memories, and histories. Although it is possible to give an abstract axiomatic characterisation of them, we have chosen to formalise them as partial functions from the natural numbers. As we shall see below, this augments the applicability of the existing toolkit.

We first define the *finite partial functions* — they are the partial functions whose domains are finite.

```
[X,Y]

    X ⇸ Y  ≙  ( f: X↔Y | dom f ∈ F X )
```

Given a set X, the *finite sequences of* X are the finite partial functions from N to X whose domains are of the form 1..n (for some n:N). More formally

```
seq[X]
  ( f: N ⇸ X | dom f = 1..#f )
```

For example:

```
( 1↦5, 2↦6, 3↦77 )    ∈ seq[N]
( 1↦Ford )             ∈ seq[MAKER]
( 1↦(), 2↦primes )     ∈ seq[P N]
```

In general the following syntactic sugar is used for extensional specifications of sequences:

```
<>              ≙  ()
<a₁>            ≙  ( 1↦a₁ )
<a₁ ... aₙ>     ≙  ( 1↦a₁, ... n↦aₙ )
```

4.8.1 Basic Sequence-Building Operators

One way of building a new sequence is to "push a new element onto the front of" an existing sequence, thus

```
cons: (X×seq[X]) → seq[X]
-----------------------------------------
∀ x:X; s:seq[X] .  x cons s = ( 1↦x ) ∪ pred⨾s
```

defines an operator which "pushes" an element x onto the front of a sequence s. It is called cons because it constructs a new sequence (it also captures at least some of the properties of Lisp's cons). We usually abbreviate this operator to the infix sign ^ — thus if x is an element of X and s is a sequence of Xs

```
x ^ s ≙ x cons s
```

We are obliged to show that our definition of *cons* is consistent with its signature, *ie* that for a sequence s and an element x, the term which defines x *cons* s really denotes a function with all the characteristic properties of a sequence.

If we recall that the function pred is the inverse of the function suc on the natural

numbers, *i.e.* that:

$$\begin{aligned} pred &= \{\ 0\mapsto 1,\ 1\mapsto 2,\ \ldots\ \}^{-1} \\ &= \{\ \ldots,\ 2\mapsto 1,\ 1\mapsto 0\ \} \end{aligned}$$

Then, by virtue of the definition of composition (see earlier),

$$\begin{aligned} dom(\ pred;s\ ) &= pred^{-1}[dom\ s] \\ &= suc[1..\#s] \\ &= 2..\#s+1 \end{aligned}$$

and so

$$\begin{aligned} dom(\ \{\ 1\mapsto x\ \} \cup (pred;s)\ ) &= 1..\#s+1 \\ \#(\ \{\ 1\mapsto x\ \} \cup (pred;s)\ ) &= 1+\#s \end{aligned}$$

which is exactly what the sequence axioms require.

A more concrete example is

$$\begin{aligned} 44^{\frown}\langle 33\ 22\rangle &= \{\ 1\mapsto 44\ \} \cup \{\ 2\mapsto 33,\ 3\mapsto 22\ \} \\ &= \langle 44\ 33\ 22\rangle \end{aligned}$$

In fact the sign $^{\frown}$ is *overloaded* (just as the sign - means both arithmetic subtraction and set difference). It also denotes an operator (pronounced "snoc" if you like) which pushes an element onto the end of a sequence, namely:

$$snoc: (seq[X]\times X) \rightarrow X$$

$$\forall\ x:X;\ s:seq[X]\ .\ s\ \underline{snoc}\ x = s \cup \{\ suc\ \#s \mapsto x\ \}$$

If x is an element of X and s is a sequence of Xs then

$$s\ ^{\frown}\ \langle x\rangle = s\ \underline{snoc}\ x$$

When the sign $^{\frown}$ is used, it should always be clear from context (or to be more precise, from the types of its operands) whether the "cons" or the "snoc" operator is meant. In cases where this is not so, we will use snoc and cons themselves; sometimes in proofs we add a little arrowhead to the $^{\frown}$ for readability -- thus

$$\begin{aligned} \overset{\rightarrow}{\frown} &\quad = \quad cons \\ \overset{\leftarrow}{\frown} &\quad = \quad snoc \end{aligned}$$

4.8.2 Operators on Sequences

Many sequence operators have domains which are the nonempty sequences, defined by:

$$[X]$$

$$seq1 = seq[X] - \{\langle\rangle\}$$

For example hd and tl which behave rather like the Lisp operators car and cdr.

```
hd, last: seq1[X] → X
tl, front: seq1[X] → seq[X]
```

```
∀ x:X; s:seq[X] .
    hd( x ^ s ) = x
    last( s ^ x ) = x
    tl( x ^ s ) = s
    front( s ^ x ) = s
```

Exercise:

We have specified these operators *nonconstructively* — that is by giving predicates which relate them to cons and snoc rather than by giving terms which denote the set of pairs to which they correspond.

Prove that the following *constructive* definitions satisfy the specifications for hd, front, tl, and last.

```
hd = λ s:seq[X] | s≠<> . (s 1)

front = λ s:seq[X] | s≠<>  . s↾(1..(pred #s))

tl = λ s:seq[X] | s≠<> . succ↑s  \  {0}

last = λ s:seq[X] | s≠<>  . s(#s)
```

We will use the same style of specification for the operator which *appends* two sequences, namely

```
_ * _ : seq[X]×seq[X] → seq[X]
```

```
∀ s:seq[X] .
      <> * s = s                                  .... app.1

∀ s₁, s₂:seq[X]; x:X .
      (x^s₁)*s₂ = x^(s₁*s₂)                       .... app.2
```

What we have done is to *specify* the result of appending any sequence to the empty sequence; and then specify the result of appending a sequence s_2 to a nonempty sequence in terms of the result of appending s_2 to the tail of the nonempty sequence; since all sequences are either empty or nonempty we have covered all possible cases, and you might think that for this very reason that a function which satisfies the above specification *must* therefore exist. This is indeed the case, but it neeed not be so in general. Later in the course we shall see that a certain class of "recursive" specification is always satisfiable, and that this specification falls into that class. For the moment, though, we will demonstrate a constructive solution to the equations above, namely:

```
_ * _ = λ s₁,s₂:seq[X] .
          s₁ ∪ (shift #s₁)↑s₂
```

```
where
    shift: N → N → N
    shift m n = n-m
```

Exercise:

Think about the strategy you would use to *prove* that this definition satisfies the specification.

The last *standard* operator on sequences is the one which *reverses* them, specified by:

```
rev: seq[X] → seq[X]
-------------------------------------------
rev <> = <>                          .... rev.1
rev (x cons s) = (rev s) snoc x      .... rev.2
```

Exercises:

(1) give a constructive definition which satisfies this specification.

(2) try to prove that the following theorems hold given that x is an element of X and s, s_1 – s_3 are all sequences of Xs –

$s_1 * (s_2 * s_3) = (s_1 * s_2) * s_3$ .... *(T1)*

$s * \langle x \rangle = s \hat{} x$ .... *(T2)*

$rev(\ s_1 * s_2\ ) = (rev\ s_2) * (rev\ s_1)$ .... *(T3)*

$(rev;rev)\ s = s$ .... *(T4)*

$\langle x \rangle = x \hat{} \langle\rangle$ .... *(T5)*

4.8.3 Reasoning about Sequences

The well-known *principle of mathematical induction* allows us to begin to reason about sequences. Summarised in set-theoretical form it states that if a certain set of natural numbers is known to contain n+1 whenever it contains n, and if it is also known to contain zero, then in fact it contains all the natural numbers. Formally, we have

```
⊢ ∀ S:P N .
  (
    0∈S ∧
    ∀ n:S . n+1∈S
  ) ⟹ S=N
```

It is clear that this principle could be extended to the sequences – for we could prove things by induction over their *lengths*. The following theorem is easily proven from the principle of mathematical induction:

```
⊢ ∀ SS:P (seq[X]) .
```

```
(
  <> ∈ SS ∧
  ∀ s:SS; x:X . x^s ∈ SS
) ⟹ SS=seq[X]
```

This result is called the *principle of finite sequence induction*. It is a special case of a much more general result which we shall examine later in the course, namely the *principle of structural induction*.

In order to show how to *use* this principle, we shall prove that the set SS of sequences s which satisfy

```
(rev;rev) s = s
```

contains all the sequences. More formally, if

```
SS ≙ { s:seq[X] | (rev;rev) s = s }
```

then

```
⊢ SS = seq[X]
```

Our proof is structured along the lines of the induction principle -- we will first prove what is called the Base Case:

```
(1)  (rev;rev) <> = rev(rev <>)      .....  composition definition
(2)  (rev;rev) <> = (rev <>)         .....  1, rev.1
(3)  (rev;rev) <> = <>               .....  2, rev.1
(4)  <> ∈ SS                         .....  3, SS definition
```

Next we perform what is known as the Induction Step. To do so we must show, given a sequence s in SS and an element x in X, that x ^ s is in SS. Suppose, then that s is a sequence in SS; this supposition is called the *induction hypothesis* and from it we proceed formally as follows

```
(5)  s∈SS                            .....  Induction Hypothesis
(6)  (rev;rev) s = s                 .....  5, SS definition
(7)  rev <x> = <x>                   .....  Lemma (see later)

(8)  rev( rev( x^ s ) ) = rev( (rev s) ^ x )     ...    rev.2
(9)    !! = rev( (rev s) ^ <x> )                 .....  8, T2
(10)   !! = (rev <x>) ^ (rev (rev s))            .....  9, T3
(11)   !! = <x> ^ s                              .....  10, ?, 5, composition def
(12)   !! = (x ^ <>) ^ s                         .....  11, T5
(13)   !! = x ^ (<> ^ s)                         .....  12, app.2
(14)   !! = x ^ s                                .....  13, app.1
(15) ∀ x:X; s:SS . s∈SS ⟹ x^s ∈ SS               .....  Generalisation
```

This completes the second part of the proof; we can now apply the sequence induction principle and conclude that SS contains all the sequences.

The ‼ on the left hand side of the equalities on lines 9-14 stands in each case for the term on the right hand side of the previous line; it just saves typing.

Each line in our proof consists of a numbered fact or hypothesis, supported by some *evidence*. In each case the evidence consists of a reference to a previously-numbered fact, and/or a predicate from a definition, and/or a theorem (or Lemma -- which is just a "local" theorem) which is proven elsewhere.

The local "lemma" was

```
⊢ ∀ x:X . rev<x> = <x>
```

Proof (left as an exercise).

4.8.4 Useful Properties

If s is a sequence of elements of X and if f is a function whose domain includes the range of s, then the composition of s and f is also a sequence. More formally:

```
⊢ ∀ s:seq[X]; f:X↔Y .
          (ran s)⊆(dom f) ⇒ (s⨾f)∈seq[Y]
```

If a sequence s_1 is a prefix of a sequence s_2, then the set of pairs which constitute s_1 is a *subset* of the set of pairs which constitute s_2. More formally:

```
⊢ ∀ s1,s2:seq[X] .
          ∃s:seq[X] .
               s1⌢s=s2  ⇔  s1⊆s2
```

For this reason we usually use the "subset" relation between sequences to mean "is a prefix of". This is an example of an *idiom*.

If s is a sequence, then the relation "are adjacent elements in s" is captured by a relatively simple formula, namely:

```
s⁻¹⨾succ⨾s
```

Exercises:

(1) prove the following theorem

```
|- ∀ s:seq[X]; x1,x2:X .
    (x1, x2) ∈ s⁻¹⨾succ⨾s
          ⇔
    ∃i:(1..#s-1) . s i = x1  ∧  s(i+1) = x2
```

(2) use the above fact to give a simple specification of the set of sequences of X which are "ordered with respect to" a homogeneous relation "<" on X.

5: Disjoint Unions, Recursive Types, Enumerated Types

5.1: Introduction

In previous sections we introduced ways of making *new* types of set from existing sets, but did not say anything about the "primitive" sets of the language. Indeed in order to avoid discourse about primitive sets we "parachuted" the natural numbers (N) into our discussion without really *defining* them. In this section we introduce the *only* notation by which new *primitive* types of set are defined. We also show how the notation may be used to define abstract forms of the tree-like structures which are common objects of discourse in Computing Science. This notation is currently used by practitioners of the "Z" style of specification; there is no corresponding "standard" notation.

5.2: Strong Typing

Although we have not yet introduced any of the consequences which flow from it, the language we have introduced so far is *strongly typed*. What this means is that certain terms and predicates are defined by us to be well-typed, whereas others are ill-typed; in general we only provide means of reasoning about well-typed terms and predicates.

The role played by type-analysis in mathematics is analogous to that played by dimensional analysis in physics and mechanics; it is a safeguard against writing utterly nonsensical mathematical terms and predicates, but is no guarantee that the mathematical model is true to reality or even consistent.

A complete explanation of the type rules is beyond the scope of these notes, but we can illustrate their spirit by briefly considering the *union* operator, which has *generic type* specified by the signature:

$$[X]$$

$$\cup : (\mathbb{P}\, X) \times (\mathbb{P}\, X) \rightarrow (\mathbb{P}\, X)$$

This means that every instance of the union operator must have operands which both have type $\mathbb{P}\, X$ — *for some primitive type of set* X — and that its result is also of type $\mathbb{P}\, X$.

Now suppose that Y and Z are *distinct* primitive types and that y is of type Y and s and t are of type $\mathbb{P}\, Z$; in other words:

$$y : Y$$
$$s : \mathbb{P}\, Z$$
$$t : \mathbb{P}\, Z$$

Then the term

$$s \cup t$$

is well-typed because the generic parameter X in the type-specification of the union operator can be instantiated with the primitive type Z. On the other hand, none of the terms

$$y \cup y$$
$$y \cup s$$
$$s \cup Z$$
$$s \cup Y$$

$Y \cup Z$

are well-typed, because in none of these cases can we find a suitable instantiation of the type variable X.

In the next section we introduce a new category of type-constructor which allows us both to overcome certain problems introduced by strong-typing, and to describe certain kinds of "recursively-specified" sets (sometimes called trees).

5.3: Disjoint Union

Suppose for a moment that we want to specify a function, find, which searches a list of words for a particular word, say foo, and either returns its index (if it appears once) or an error code indicating whether it appears more than or less than once. Let us suppose that the set of error codes, call it E, is primitive (generic), and that the set of words, call it W, is also primitive.

How do we write a signature for the function we have in mind? Clearly

$$find: seq[W] \rightarrow (E \cup N)$$

will not do, for there is no primitive type of set of which both N and E are subsets and so the $\cup$ term on the right of the function arrow is *ill-typed*. What we need on the right of the arrow is a type of set which contains "as many" elements as there in N and E combined, but in which there is no confusion between numberish and errorish things.

One such set is

$$N \times E \times (0,1)$$

where the third component of each

```
(n, e, which)
```

triple indicates whether the n or the e part is the *real* information contained in the triple. This is unsatisfactory, firstly because it is only one of a number of possible "codings", and secondly, because it is "overkill" in the sense that each number has several possible representations in the coding, namely

```
(n, errnone, 0) (n, errmany, 0) (n, erroverflow, 0) ...
```

and each error has an infinity of representations in the coding, namely:

```
(0, errnone, 1)       (1, errnone, 1)  ...  (567890, errnone, 1)
(0, errmany, 1)       (1, errmany, 1)  ...            ...
(0, erroverflow, 1)         ...        ...            ...
```

Mathematicians abhor clutter, so it is perhaps not surprising that our notation allows us to describe precisely the set which is needed, namely one which has exactly one element for each element of E and exactly one element for each element of N. The term

```
error<<E>>  |  result<<N>>
```

denotes such a set. The identifiers result and error are chosen for their mnemonic value, not for any technical reason.

If we call this set S (which we can do in the usual way by putting the name S on a box around it)_

```
┌─ S ──────────────────────────────────┐
│  error<<E>>  |  result<<N>>          │
└──────────────────────────────────────┘
```

then the following *injective* functions are defined "automatically"_

```
┌──────────────────────────────────────┐
│ error:   E  S                        │
│ result:  N ↣ S                       │
├──────────────────────────────────────┤
│ ran error ∪ ran result = S           │
│ ran error ∩ ran result = {}          │
└──────────────────────────────────────┘
```

The axioms for these functions indicate precisely that there's neither junk (the union of the ranges of the functions is exactly S) nor confusion (the ranges are disjoint).

Moreover, because the functions are injective[*See Appendix II*], their inverses are also functions, so that given an element s:S — then if it "stands for" an error code, *ie* if

$$s \in (\text{ran error})$$

then the one in question is

$$\text{error}^{-1}\ s$$

Likewise if it stands for a number,

$$s \in (\text{ran result})$$

then the one in question is

$$\text{result}^{-1}\ s$$

We are now in a position (for better or for worse) to specify the function we first thought of. First we introduce some constants to denote the word being sought and the error codes:

```
┌──────────────────────────────────────┐
│ foo:  W                              │
│ none: E                              │
│ many: E                              │
└──────────────────────────────────────┘
```

Having done so we can specify:

```
find: seq[W] → S
```

```
∀s:seq[W]
 #index=0 ⇒ find s = error none
 #index>1 ⇒ find s = error many
 #index=1 ⇒ find s ∈ result[ index ]
where
  index = s⁻¹[ ⟨ foo⟩ ]
```

5.4: General Characterisation of Disjoint Union

In general if S1 .. Sn are terms which denote sets, and if id1 .. idn are identifiers, then the term:

$$id_1\langle\langle S_1\rangle\rangle \mid \ldots \mid id_n\langle\langle S_n\rangle\rangle$$

denotes a new set. Such set-denoting terms hardly ever appear without being named, which is done in the usual manner, for example:

```
D
  id1<<S1>> | ... | idn<<Sn>>
```

In this case the following constants are defined and remain in scope for as long as the name D.

```
id1 : S1 ↣ D
id2 : S2 ↣ D
.
.
.
idn : Sn ↣ D
```

```
< ran id1  ...  ran idn > partitions D
```

and — as you might have expected — the ranges of these injective functions partition *[see Appendix I]* the new set.

5.5: Reasoning Methods

The principal method of reasoning about disjoint unions is based on the following (meta)-theorem.

$$
\begin{array}{l}
\quad P: \mathcal{P}\, D \\
\vdash \\
\quad (\\
\qquad \forall s{:}S_1 \,.\, (id_1\ s) \in P \wedge \\
\qquad . \\
\qquad . \\
\qquad . \\
\qquad \forall s{:}S_n \,.\, (id_n\ s) \in P \\
\quad) \Rightarrow P = D
\end{array}
$$

Which expressed informally reads: if a certain property (the characteristic predicate of P) holds for all ways of constructing an element of D then it holds for all elements of D.

Special Case: Enumerated Types

The term

$$id_1 \mid \ldots \mid id_n$$

denotes a "new" set. If this set is named, (E say) then the following constants are defined and remain in scope for as long as the name E.

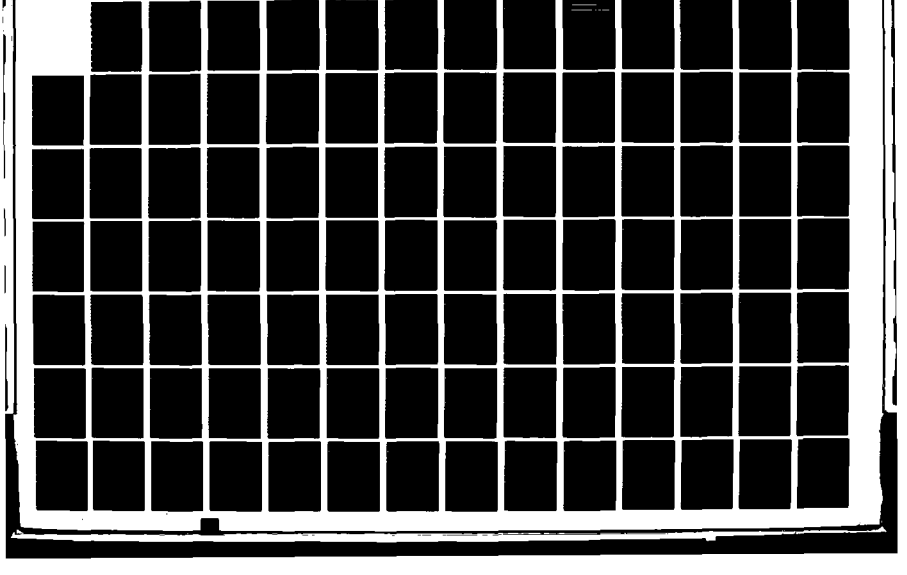
$$
\begin{array}{|l}
\hline
\quad id_1 : E \\
\quad . \\
\quad . \\
\quad . \\
\quad id_n : E \\
\hline
\quad \langle (id_1) \ldots (id_n) \rangle \ \underline{\text{partitions}}\ E \\
\hline
\end{array}
$$

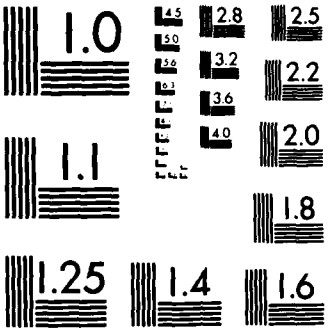
Moreover they all denote different elements of E.

> **Technical Note:** *In fact this is the only correct way to introduce a "new" type consisting of an enumeration of elements. The definition*
>
> $$E \triangleq (foo, baz, blah)$$
>
> *(which is occasionally used with the intention of defining an enumerated type) means nothing at all in our notation except in the scope of* foo baz *and* blah, *and even then it may not be well-typed!*

AD-A171 671 FORMAL TECHNIQUES FOR SPECIFICATION AND VALIDATION OF TACTICAL SYSTEMS(U) MASSACHUSETTS COMPUTER ASSOCIATES INC WAKEFIELD 02 JUN 86 CADD-8606-0203 2/3

UNCLASSIFIED DAAK80-81-C-0072 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

5.6: Hybrid Case

Terms of the form

$$id_1 \mid id_2\langle\langle S_2\rangle\rangle \mid \ldots id_j\langle\langle S_j\rangle\rangle \mid id_n$$

(in which some of the identifiers stand alone) are perfectly valid. When such a term is named (H say) the following constants are defined:

$id_1 : H$
$id_2 : S_2 \rightarrowtail H$
.
.
$id_j : S_j \rightarrowtail H$
$id_n : H$

$\langle \{ id_1 \}, \text{ran } id_2, \ldots, \text{ran } id_j, \{ id_n \} \rangle$ partitions H

The pattern should be obvious. The stand-alone identifiers correspond to constant elements of the new type, the remaining elements correspond to injective functions into the new type. The elements of the type are partitioned appropriately.

5.7: Recursive Types

Trees occur sufficiently frequently in the world of specification that it is useful to have a small library of conceptual tools with which to manipulate and reason about them. It is rather convenient to be able to specify trees "recursively", but we have to take some care when doing so.

Let us consider, for the moment, the problem of specifying what is a Lisp(kit)-list. Informally,

A list is *either* an atom *or* a pair of lists

Clearly it doesn't matter what the internal structure of the set of atoms is, we will denote this set by A and try to construct a (strictly) set-theoretic specification of the lists:

$$L = A \cup (L \times L)$$

What we are trying to convey here is that the set of lists contains all the atoms and is closed under the *pairing* operation. Unfortunately this "specification" of the set of lists has no solution because the strong typing prevents it. *(Scottophiles please note: we mean "no solution in our set theory"; the sign = means something different in Scottery.)*

We can be satisfied with a set which is rich enough to contain one representative for every A and one for every pair of L. It turns out there is a set which has just the properties we want; it is defined in our notation by:

```
L
  atom<<A>> | cons<<L×L>>
```

The theory (due to Tarski) behind such recursive specifications is simple and elegant: its details need not concern us here, but as might be expected we have recourse to the techniques we used in defining disjoint unions. In essence the equation implicit in the above definition has a solution because given a set, C say, which is big enough to "carry" the lists, and given injections:

```
atom:  A ↣ C
cons:  C×C ↣ C
(ran cons) ∩ (ran atom) = {}
```

whose ranges are disjoint, it is possible to discover a subset of C

```
L: ℙ C
```

which has the properties

```
atom[ A ]  ⊆ L
cons[ L × L ] ⊆ L
```

In other words it contains an element (atom a) for each a∈A and an element cons(x,y) for each pair of lists (x,y)∈(L × L). This set is clearly good enough.

It turns out that a lot of interesting new types can be defined in this way, for example:

```
NUM
  zero | suc<< NUM >>
```

```
BIN
  nil | node<< NUM×BIN×BIN >>
```

These will easily be recognisable as (a set isomorphic to) the natural numbers and the set of binary trees which have numbers at the nodes.

Not all such specifications denote "reasonable" sets, though. For example:

```
SILLY
  foo<< SILLY >> | baz<< SILLY >>
```

might have nothing at all in it, but might not.

In general, the set of trees specified by

```
T
  id₁<<S₁>>  |  ...  | idₙ<<Sₙ>>
```

is nonempty if there is at least one term S1 ... Sn in which T does not appear free (or if at least one of the ids stands alone).

The functional ids ... are sometimes called the *constructors* of T; those ids which stand alone are called *ground elements* of the type. The *ground constructors* of T are those whose Si terms do not contain T free; elements in their ranges are called the *ground trees* (or *structures*) of the type. The remaining constructors are called *proper constructors*; elements in their ranges are called the *proper trees* (or *structures*) of T.

> Technical Note: *It (almost) turns out that this kind of specification works if each term* Si *in which* T *appears free, has a certain property, namely that the function*
>
> $$f \triangleq \lambda T \,.\, id_i[\, S_i \,]$$
>
> *is monotonic under inclusion. That is, given two subsets* X *and* Y *of some hypothetical set, we can prove that:*
>
> $$X \subseteq Y \Rightarrow f[\, X \,] \subseteq f[\, Y \,]$$
>
> *Unfortunately there is a snag; sometimes it is just not possible to find a large enough "carrier". A sufficient condition for the existence of a large enough carrier is to restrict the* S_i *terms which contain* T *to one of the following finite forms:*
>
> | | |
> |---|---|
> | T × ... | *(... may contain T)* |
> | F T | *(finite subsets of T)* |
> | T ⇸ ... | *(finite mappings from T)* |
> | ... ⇸ T | *(finite mappings to T)* |

5.8: Reasoning Methods -- Structural Induction

The most important reasoning method applicable to recursively specified sets is the *principle of structural induction*. We will illustrate the general principle of this method by presenting instances of it which are appropriate for reasoning about the sets L, NUM, and BIN.

The list induction principle is:

```
   S: P L
⊢
   (
     ∀a:A . (atom a) ∈ S  ∧
     ∀l1, l2: S . cons(l1,l2) ∈ S
   ) ⟶ S=L
```

If a property holds for the image of every atom, and if it holds for lists consed from lists for which it holds, then it holds for all lists.

The NUM induction principle is:

```
   S: P NUM
⊢
   (
     zero ∈ S   ∧
     ∀n: S . (suc n)∈S
   ) ⟶ S=NUM
```

If a property holds for zero, and for the successor of every NUM for which it holds, then it holds for all numbers.

The BIN induction principle is:

```
   S: P BIN
⊢
   (
      nil ∈ S   ∧
      ∀ b1, b2: S; n: NUM . node(n, b1, b2) ∈ S
   ) ⟶ S=BIN
```

The general principle is a little tedious to state formally, but you can see the idea, which can be summarised as follows: To prove that a property holds for all elements of a recursive type first prove that all the *ground elements* of a type have the property; then prove that all *ground trees* have that property; finally, under the assumption that all its subtrees have the property, prove that each *proper tree* has the property.

5.9: Reasoning Methods -- Recursion Principle

Hitherto we have specified certain functions "recursively" without having any assurance that the specifications were satisfiable. For example, the following is a specification of a function which is intended to list the nodes of a binary tree in *pre-order*

```
flat: BIN → seq[NUM]
-----------------------------------------------------
flat nil = <>
∀ n: NUM; b1, b2: BIN .
 flat( node(n, b1, b2) ) = (flat b1) * <n> * (flat b2)
```

It turns out that a large class of recursive specifications of total functions on trees (including the one above) are indeed satisfiable.

We will summarise the recursion principle by giving a typical instance of its application, without going into the theory behind its validity. Suppose T is defined by

```
T
cons0 | cons1<<X>> | cons2<<Y×T>> | cons3<<Y×T×T>>
```

where X and Y and Z are type expressions which don't contain free occurences of the identifier T. Suppose we are also given:

```
g0: D
G1: X → D
H2: Y×D → D
H3: Z×D×D → D
```

The following specification of f is satisfiable:

```
f: T → D
-----------------------------------------------------
f cons0 = g0
∀x:X . f(cons1 x) = G1 x
∀y:Y; t:T . f(cons2(y,t)) = H2(y, f t)
∀z:Z; t1,t2: T . f(cons3(z,t1,t2)) = H3(y, f t1, f t2)
```

That is to say, there really is a total function which behaves as specified.

In general the principle states that in order to specify a total function over a recursively-defined type it is necessary to specify its value at all ground elements of the type, to specify its value at all ground trees of the type, and to specify its value at all proper trees of the type. In this latter case it is permissible to mention the value of the function at subtrees.

In our definition of BINary trees, the single ground element is `nil`, and the single (proper) constructor is `node`. These correspond to `cons0` and `cons3` of our "typical instance". The specification of `flat` is valid because <> corresponds to `g0`, and the function

```
λ s, n, t . s * <n> * t
```

which is of type

$$seq[NUM] \times NUM \times seq[NUM] \rightarrow seq[NUM]$$

corresponds to H3.

In our notes on adequate representations for finite partial functions we shall apply both the structural induction principle and the recursion principle extensively.

Appendix I: Partitions

In this appendix we give formal definitions of disjointness and partitions.

$$\begin{array}{l}
\textbf{disjoint:} \quad \mathbb{P}(\textbf{seq}\ (\mathbb{P}\ X)) \\
\textbf{partitions:}\ \textbf{seq}\ (\mathbb{P}\ X) \leftrightarrow (\mathbb{P}\ X) \\
\hline
\forall\ SS: \textbf{seq}\ (\mathbb{P}\ X)\ . \\
\quad SS \in \textbf{disjoint} \Rightarrow \\
\quad\quad \forall\ i_1, i_2 : \textbf{dom}\ SS\ . \\
\quad\quad\quad i_1 \neq i_2 \Rightarrow (SS\ i_1) \cap (SS\ i_2) = \{\} \\
\\
\forall\ SS: \textbf{seq}\ (\mathbb{P}\ X)\ ;\ S: \mathbb{P}\ X\ . \\
\quad SS\ \underline{\textbf{partitions}}\ S \Leftrightarrow (SS \in \textbf{disjoint}) \wedge \cup(\textbf{ran}\ SS) = S
\end{array}$$

A set of (sub)sets of X is disjoint if it is pairwise disjoint; that is if no two distinct sets have a nonempty intersection.

A disjoint set SS of (sub)sets of X partitions a (sub)set S of X if its (generalised) union is S.

Appendix II: Injections

Suppose we picture each relation by drawing an arrow from each domain element to the range elements to which it corresponds. Then we would see that the *functional* relations are those in which there are no *diverging* arrows — each domain element maps to exactly *one* range element. The *injective functions* are the functions in which there are also no *converging* arrows: every element of the range arises from just *one* element in the domain. If we reverse the arrows of such a function (take its inverse) we will immediately notice that the resulting relation is itself a function (no diverging arrows). Thus each injective function is itself a function. More formally we define:

$$[X, Y]$$

$$\begin{array}{lll}
X \rightarrowtail\!\!\!\!\!\!\rightarrow Y & \triangleq & (\ f : X \leftrightarrow Y \mid f^{-1} \in Y \rightarrow X\ ) \\
X \rightarrowtail Y & \triangleq & (\ X \leftrightarrow Y\ ) \cap (\ X \rightarrow Y\ )
\end{array}$$

The two signs we introduce denote respectively the *partial* injections and the *total* injections.

Appendix III: Syntax

The following is a summary of the syntax of those parts of the language we have so far explained.

Term	::=	Id		
		{ Signature | Predicate . Term }	.....	*set comprehension*
		{}	.....	*empty set*
		{ Termlist }	.....	*set extension*
		(Termlist)	.....	*tuple*
		Term × Term	.....	*product constructor*
		ℙ Term	.....	*subset constructor*
		𝔽 Term	.....	*finite subset constructor*
		Tree		
		# Term	.....	*size*
		FnTerm Term	.....	*function application*
		Term FnTerm Term	.....	*infix function application*
		λ Signature | Predicate . Term	.....	*lambda abstraction*
Tree	::=	Branch		
		Branch | Tree	.....	*disjoint union*
Branch	::=	Id	.....	*ground element*
		Id<<Term>>	.....	*tree constructor*
Predicate	::=	∃ Signature . Predicate	.....	*existential quantification*
		∀ Signature . Predicate	.....	*universal quantification*
		Term = Term	.....	*equality*
		Term ∈ Term	.....	*membership*
		Term ≠ Term	.....	*negation of equality*
		Term ⊂ Term	.....	*proper subset*
		Term ⊆ Term	.....	*subset*
		Predicate ∧ Predicate	.....	*conjunction*
		Predicate ∨ Predicate	.....	*disjunction*
		Predicate ⇒ Predicate	.....	*implication*
		Predicate ⇔ Predicate	.....	*logical equivalence*
		¬ Predicate	.....	*negation*
		Term RelTerm Term	.....	*infix relation membership*
RelTerm	::=	Term		
FnTerm	::=	Term		
Signature	::=	TypeAttr		
		TypeAttr; Signature		
TypeAttr	::=	IdList: Term		
TermList	::=	Term		
		Term, Termlist		
IdList	::=	Id		
		Id, Idlist		
Defn	::=	[IdList] Term≙Term	.....	*Syntactic Equivalence*
		[IdList] Signature | Predicate	.....	*Constant Specification*
		[IdList] Id ∈ Tree		

Address

Bernard Sufrin. Ext 281

LService

```
12:15:22  Connected to:          Martin R. Raskovsky (x294)
12:15:22  In service of:         3F00_0000_0200_4165
12:15:25  Receiving:             Sys:Service>3AAB_41F1_AD99_F5E7
12:15:28  Receiving:             Sys:Service>38AB_00DF_AD99_EAD3%
12:15:30  Receiving:             Sys:Service>38AB_00DF_AD99_EAD3
12:15:33  Receiving:             Sys:Service>39AB_6074_AD99_2C67%
12:15:54  Receiving:             Sys:Service>39AB_6074_AD99_2C67
12:16:24  Running:               qPrint @Sys:Service>Print.Me
12:16:24
12:16:33  Print indirect from:   Sys:Service>Print.Me
12:16:39  Context from:          Sys:Service>38AB_00DF_AD99_EAD3
12:16:43  qpDoc:                 Entry
12:16:54  Printing:              Sys:Service>39AB_6074_AD99_2C67
12:18:19                         Printed 10 pages
12:18:20  qpDoc:                 Exit
12:18:29  qPrint:                Exit
12:18:29
12:18:54  Connecting:            To Network via Data Switch
12:19:01  Connecting:            To Time Service
12:19:01  Time Service:          12:19:01 07-Sep-84
12:19:01  Reserved until:        12:49:01 07-Sep-84
12:19:02  LService:             Entry
12:19:57  Reserved until:        12:49:58 07-Sep-84


12:19:58  Connected to:          Bernard Sufrin. Ext 281
12:19:58  In service of:         4000_0000_0200_5766
12:20:01  Receiving:             Sys:Service>3AAB_DFCF_AD99_8888
12:20:08  Receiving:             Sys:Service>F7AA_AA9A_AD99_6281%
12:20:15  Receiving:             Sys:Service>F7AA_AA9A_AD99_6281
12:20:20  Receiving:             Sys:Service>F9AA_0070_AD99_885E%
12:21:03  Receiving:             Sys:Service>F9AA_0070_AD99_885E
12:21:42  Receiving:             Sys:Service>FCAA_BFB0_AD99_FEA9%
12:22:05  Receiving:             Sys:Service>FCAA_BFB0_AD99_FEA9
12:22:29  Receiving:             Sys:Service>2DAB_81C6_AD99_90AB%
12:22:51  Receiving:             Sys:Service>2DAB_81C6_AD99_90AB
12:23:13  Receiving:             Sys:Service>2FAB_0F2F_AD99_EC1A%
12:23:38  Receiving:             Sys:Service>2FAB_0F2F_AD99_EC1A
12:24:03  Receiving:             Sys:Service>31AB_84D1_AD99_2DC1%
12:24:27  Receiving:             Sys:Service>31AB_84D1_AD99_2DC1
12:24:50  Receiving:             Sys:Service>33AB_9052_AD99_0243%
12:25:19  Receiving:             Sys:Service>33AB_9052_AD99_0243
12:25:47  Receiving:             Sys:Service>35AB_03F3_AD99_43E9%
12:26:06  Receiving:             Sys:Service>35AB_03F3_AD99_43E9
12:26:27  Running:               qPrint @Sys:Service>Print.Me
12:26:27
12:26:37  Print indirect from:   Sys:Service>Print.Me
12:26:42  Context from:          Sys:Service>F7AA_AA9A_AD99_6281
12:26:46  qpDoc:                 Entry
12:26:52  Printing:              Sys:Service>F9AA_0070_AD99_885E
12:28:26                         Printed 8 pages
12:28:31  Printing:              Sys:Service>FCAA_BFB0_AD99_FEA9
12:29:31                         Printed 7 pages
12:29:36  Printing:              Sys:Service>2DAB_81C6_AD99_90AB
12:30:30                         Printed 6 pages
12:30:34  Printing:              Sys:Service>2FAB_0F2F_AD99_EC1A
12:31:35                         Printed 7 pages
12:31:40  Printing:              Sys:Service>31AB_84D1_AD99_2DC1
12:32:34                         Printed 6 pages
12:32:38  Printing:              Sys:Service>33AB_9052_AD99_0243
12:33:47                         Printed 8 pages
12:33:51  Printing:              Sys:Service>35AB_03F3_AD99_43E9
12:34:30                         Printed 4 pages
12:34:31  qpDoc:                 Exit
12:34:40  qPrint:                Exit
12:34:40
```

Notes for a Z Handbook
Part 1: the Mathematical Language

Draft 18 August 1984

Abstract

In these notes we present a concise summary of the mathematical sublanguage of the specification notation Z.

Bernard Sufrin
Carroll Morgan
Ib Sorensen
Ian Hayes

Oxford University Computing Laboratory
Programming Research Group
8 Keble Road
Oxford OX1 3QD
England

Introduction

When specifications are written in the Z notation, two complementary formal languages are employed: the mathematical language and the schema language. The mathematical language is based on standard set theory, but is "strongly typed" in a way which most Computer Scientists will find familiar. The schema language supports the systematic presentation of large-scale system specifications, or families of specifications, which embody material defined in the mathematical language.

Z has evolved a good deal since its introduction in 1979 [Abrial], and this handbook is an attempt to capture the state of the mathematical language as it stands in mid-1984. For an introduction to the mathematics which underlies Z the reader may consult any of the material in Section 1 of the bibliography. For an introduction to the style and practice of formal system specification which Z supports see section 2 of the bibliography, which is a list of case studies which for the most part use the present dialect.

1
Basic Notation

§1.0 Preliminaries

In what follows we use

```
class ::=
        def1
        def2
        ...
        defn
```

to define a new syntactic class `class`. A definition in this form should be taken to mean that wherever a phrase of class `class` is required, it can be supplied in one of the forms def_1 ... def_n.

The principal syntactic classes we shall define are `signature`, `term`, `predicate`. In specifying syntactic equivalences we shall assume that the syntactic variables

$t, t_1, t_2, \ldots t_n$	range over terms
$id, id_1, id_2, \ldots id_n$ $v, v_1, v_2 \ldots v_n$	range over symbols, as do
$sig, sig_1, sig_2, \ldots sig_n$	range over signatures
$p, p_1, p_2 \ldots p_n$	range over predicates
F, F_1, F_2	range over terms which denote functions
R, R_1, R_2	range over terms which denote relations
$T, T_1, T_2, \ldots T_n$	range over terms which denote sets

The lexical structure of the class of `symbols` is left undefined, but includes all sequences of alphanumeric characters, and all peculiar shapes which the artistically inclined may care to think up to represent mathematical objects mnemonically.

When specifying syntactic equivalences, the symbol $\widehat{=}$ between two patterns a and b means "a and b are defined to be syntactically equivalent". To be more precise, if by substituting appropriate phrases for the syntactic variables in the pattern a we match a phrase of the language, then this phrase may be replaced by the phrase formed from the pattern b by making the same substitutions for syntactic variables. Likewise if b matches a phrase of the language, then the phrase may be replaced by a suitably modified a. (See note 3 of §1.1 for a simple worked example).

> Technical Note: As presented here the syntax appears to be ambiguous, for except in two cases we have not mentioned the binding power (precedence) of the symbols of the notation. Nevertheless, since the symbols used as operators are always introduced by signatures which indicate their set-theoretic type, the structure of compound phrases can usually be inferred from context, and in awkward cases parentheses may be employed to make the structure explicit.

§1.1 Signatures

A type declaration introduces a new variable or a new constant, associates it with a symbolic identifier, and ascribes to it a type, from which may be deduced the values which it is permitted to take. A signature is a sequence of type declarations.

Syntax:

```
signature ::=
              decl
              decl ; signature

decl      ::=
              symbol : term
```

Syntactic Equivalence:

$id_1, id_2, \ldots id_n : T \;\widehat{=}\; id_1 : T; id_2 : T; \ldots id_n : T$

Examples:

```
carriagereturn: CHAR

size: Number; weight: Number; contents: P THING

size:     Number
weight:   Number
contents: P THING                                    (1)

factor: primes ↔ Number                              (2)

carriagereturn, linefeed: CHAR                       (3)

_ + _: N × N → N                                     (4)

adder: N → N → N                                     (5)
```

Notes:

(1) A declaration may be presented in vertical form, in which case the semicolons between type attributions may be omitted, and each type attribution appears on a new line.

(2) A term which appears to the right of the colon in a type attribution must denote either a type, or a set whose carrier type can be determined. (See Appendix I)

(3) This signature is syntactically equivalent to `carriagereturn: CHAR; linefeed:CHAR` the substitutions being: `CHAR` for `T`, `carriagereturn` for id_1, `linefeed` for id_2 (and 2 for `n`).

(4) When introducing a function symbol whose name is not composed of alphanumerics, it is customary to indicate its fixity by putting underline symbols in the places where its argument or arguments will appear. This signature introduces the infix symbol `_+_` which maps pairs of numbers to numbers.

(5) This signature introduces a function symbol, adder, which maps a number into a function from numbers to numbers. The function-arrow signs are right associative so the term `A→B→C` should be parsed `A → (B → C)`. Function application, denoted by juxtaposition, is left associative (see §1.2).

§1.2 Terms

A term is a phrase of the mathematical language which corresponds intuitively to a set or to an element of a set.

Syntax:

```
term   ::=
        symbol
        ( signature | predicate . term )        set comprehension
        ( term, term, ... term )                finite set extension
        ( term, term, ... term )                n-tuple
        term × term × ... × term                product set (n-tuples)
        P term                                  power set (subsets)
        term term                               function application   (6)
        μ term                                  arbitrary choice from a set
        branch | branch ... | branch            disjoint union

branch ::=
        symbol
        symbol<<term>>
```

Note:

(6) Function application is denoted by juxtaposition, which associates to the left. The term `add n n` therefore denotes the application `((add n) n)`. Of course nothing forbids the use of parenthesis around arguments, as for example in `delete(foo)`.

Syntactic Equivalences:

$v_1:T_1;\ v_2:T_2;\ \ldots\ v_n:T_n \mid p$	$\cong$	$v_1:T_1;\ v_2:T_2;\ \ldots\ v_n:T_n \mid p\ .\ (v_1 \ldots v_n)$	(1)
`sig . t`	$\cong$	`sig | true . t`	
$(\ v_1:T_1;\ v_2:T_2;\ \ldots\ v_n:T_n\ )$	$\cong$	$T_1 \times T_2 \times \ldots T_n$	
`( t . sig | p )`	$\cong$	`( sig | p . t )`	
`( t . sig )`	$\cong$	`( sig . t )`	
$t_1 \mapsto t_2$	$\cong$	(t_1, t_2)	(2)
$t_1\ F\ t_2$	$\cong$	$F(t_1, t_2)$	(3)
$\lambda\ v_1:T_1;\ v_2:T_2;\ \ldots\ v_n:T_n \mid p\ .\ t$	$\cong$	$(v_1:T_1;\ v_2:T_2;\ \ldots\ v_n:T_n \mid p\ .\ (v_1 \ldots v_n) \mapsto t)$	(4)
`t` *where* `sig | p`	$\cong$	`μ ( sig | p . t )`	(5)

Notes:

(1) In situations where the form `signature | predicate . term` appears, the term may be omitted when it is a tuple consisting of the variables of the signature, as may the predicate if it is identically true.

(2) The sign $\mapsto$ is pronounced "to", or "maps to"; terms of this form appear in the construction of relations. (see Relations, §2.2)

(3) Infix notation for function application. (see Functions, §2.3)

(4) A function is just identified with the set of domain-range pairs which it maps between. (see Functions, §2.3)

(5) This permits local definitions to be made.

Examples:

```
{ n:N | divisors n = {53, 73} . n² }                    (1)
{ n:N | divisors n = {1, n} }                           (2)
{ i,j: N | i+j ∈ primes . (i,j) }                       (3)
{ i,j: N | i+j ∈ primes }                               (4)
{53, 73}                                                (5)
(i, j, k)                                               (6)
N × N × N                                               (7)
{ i,j,k: N }                                            (8)
P {53, 73}                                              (9)
{}                                                      (10)
P {}                                                    (11)
μ { x:N | x³ = 64 . x }                                 (12)
μ { x,y: N | x=42 ^ y=45 . x³-y² }                      (13)
pair<<N × N>> | triple<<N × N × N>>                     (14)
blue | white | green                                    (15)
```

Interpretation:

(1) The squares of all numbers divisible only by 53 and 73.

(2) The numbers which are only divisible by themselves and 1 (the primes).

(3) The set of all pairs of numbers whose sum is a prime.

(4) Same as (3).

(5) The set whose two elements are 53 and 73

(6) The triplet (3-tuple) i j k.

(7) The set of ALL triplets of numbers.

(8) Same as (7).

(9) The set of all subsets of {53, 73}; that is { {}, {53}, {73}, {53,73} }.

(10) The empty set.

(11) The set of all subsets of the empty set; that is { {} }.

(12) A number whose cube is 64; either 4 or -4.

(13) The difference between 42 cubed and 45 squared.

(14) The disjoint union of the pairs of numbers and the triples of numbers. (See Appendix 1).

(15) The enumerated type whose elements are blue, white, and green. (See Appendix 1).

§1.3 Predicates

A predicate is a phrase of the mathematical language which corresponds intuitively to a statement (about sets or elements of sets), which may or may not be provable.

Syntax:

predicate ::=		
	¬ predicate	negation
	predicate ∧ predicate	conjunction (and)
	term = term	equality
	term ∈ term	set membership
	term ⊆ term	subset
	term symbol term	relation holds between terms (e)
	∃ signature . predicate	existential quantification (for some)
	predicate *where* definition	local definition

Note:
(e) It is customary to underline a symbol composed only of alphanumeric characters when it is used as an infix operator.

Syntactic Equivalences:

$p_1 \vee p_2$	$\Leftrightarrow$	$\neg(\neg\ p_1 \wedge \neg p_2)$	disjunction (or)
$p_1 \Rightarrow p_2$	$\Leftrightarrow$	$(\neg\ p_1) \vee p_2$	implication (if-then)
$p_1 \Leftrightarrow p_2$	$\Leftrightarrow$	$(p_1 \Rightarrow p_2) \wedge (p_2 \Rightarrow p_1)$	logical equivalence
$t_1 \neq t_2$	$\Leftrightarrow$	$\neg(t_1 = t_2)$	inequality
$t_1 \notin t_2$	$\Leftrightarrow$	$\neg(t_1 \in t_2)$	nonmembership
$t_1 \subset t_2$	$\Leftrightarrow$	$(t_1 \subseteq t_2) \wedge (t_1 \neq t_2)$	proper subset
(∀ sig . p)	$\Leftrightarrow$	¬ (∃ sig . (¬ p))	universal quantification (for all)
$\exists_1$ sig . p	$\Leftrightarrow$	(∃ sig . p) ∧ (∀ v_1, v_2: (sig | p) . $v_1 = v_2$)	existence of a unique element
∀ sig | p_1 . p_2	$\Leftrightarrow$	∀ sig . $p_1 \Rightarrow p_2$	(1)
∃ sig | p_1 . p_2	$\Leftrightarrow$	∃ sig . $p_1 \wedge p_2$	(2)
$p_1 \wedge p_2 \wedge \ldots \wedge p_n$	$\Leftrightarrow$	p_1 p_2 ... p_n	(3)
t_1 R t_2	$\Leftrightarrow$	$(t_1, t_2) \in R$	(4)
t_1 R_1 t_2 R_2 t_3	$\Leftrightarrow$	$(t_1\ R_1\ t_2) \wedge (t_2\ R_2\ t3)$	

Notes:
(1) The left hand side of this equivalence is read: "p_2 holds for all _ such that p_1".

(2) The left hand side of this equivalence is read: "p_2 holds for some _ such that p_1".

(3) A long sequence of conjunctions may be written vertically provided that the indentation established for the first predicate is inherited by the following predicates.

(4) The left hand side is sometimes read as "R maps t_1 to t_2". In Z a relation is identified with the pairs of elements between which it holds. (See §2.2)

Examples:

```
¬(369 ∈ primes)                                                          (1)
(mydog ∈ hasfleas) ∧ (age mydog > 14)                                    (2)
((weather here) = raining) ∨ (here = Paris)                              (3)
((weather here) ∈ awful) ⟹ (here = Oxford)                               (4)
(x divides y) ⟺ (∃ z:ℕ . x × z = y)                                      (5)
EUROPE ⊂ STATE                                                           (6)
(369, 7) ⊂ primes                                                        (7)
∃ state:EUROPE . state≠UK ∧ (head state) descendant Victoria             (8)
∀ state:EUROPE . state ∈ EEC                                             (9)
∃ state:EUROPE | state≠UK . (head state) descendant Victoria             (10)
∀ state:STATE | state ∈ EUROPE . state ∈ EEC                             (11)
∀ p: PERSON .                                                            (12)
        father p ∈ PERSON
        mother p ∈ PERSON
        children p ⊂ PERSON

Bernard descendant Samuel where descendant ≙ child*                      (13)
```

Interpretation:

(1) 369 is not a member of the set of primes.

(2) Mydog is a member of hasfleas and its age is greater than 14.

(3) The weather here is raining or here is Paris.

(4) If the weather here is a member of awful then here is Oxford.

(5) x divides y if and only if there is a number z whose product with x is y.

(6) EUROPE is a subset of the set STATE.

(7) The set whose two elements are 369 and 7 is a subset of the set of primes.

(8) There is a state in EUROPE which is not the UK and whose head is a descendant of Victoria.

(9) All states in EUROPE are members of EEC.

(10) (Same as 8)

(11) (Same as 9)

(12) The father, mother, and children of all persons are persons.

(13) Bernard is in the relation descendant with Samuel, where descendant is defined to be the transitive closure (see §2.7) of the relation child.

§1.4 Definitions

Constants are defined either by syntactic equivalence, or by axiomatic specification. Generic (ie families of type-parameterised) definitions are preceeded by a sequence of type identifiers, which indicate the formal type parameters of the definition.

Syntax:

```
definition ::=
        axiomatic
        syntactic
        generic

axiomatic ::=

        ┌──────────┐
        │signature │
        ├──────────┤
        │predicate │
        └──────────┘

syntactic ::=
        term ≙ term

        ┌─ symbol ──┐
        │  term     │
        └───────────┘

generic  ::=  [symbol, ...] definition                    (0)
```

Note:

(0) Generic definitions specify families of constants; see examples (0) and (0) below.

Examples:

(1)

```
┌──────────────────────────┐
│poly: ℕ → ℕ               │
├──────────────────────────┤
│∀ x:ℕ . poly x = x³ + 2x³ │
└──────────────────────────┘
```

(2)

```
┌──────────────────────────────┐
│processorspernode: ℕ          │
├──────────────────────────────┤
│7 < processorspernode < 19    │
└──────────────────────────────┘
```

(3)

```
┌──────────────────────────────────┐
│primes, squareprimes: ℙ ℕ         │
├──────────────────────────────────┤
│primes = { n:ℕ | divisors n = {1, n} }
│                                  │
│squareprimes = { n:primes . n² }  │
└──────────────────────────────────┘
```

(4)

```
impossible: ℕ
─────────────────────────────
∀ n:primes . impossible > n
```

(5)

```
NumberPair ≙ ℕ × ℕ
```

(6)

```
NumberPair
ℕ × ℕ
```

Interpretation

(1) The symbol poly denotes a function which maps numbers to numbers. Every number maps under poly to the sum of its square and twice its cube.

(2) The symbol processorspernode denotes a (constant) number between seven and nineteen.

(3) The symbols primes and squareprimes denote constants: primes is the set of all numbers divisible only by 1 and themselves; squareprimes is the set of numbers generated by squaring each prime.

(4) The symbol impossible denotes a number which is larger than or equal to every prime. This is an unsatisfiable specification, of course, since there is no such number.

(5) The symbol NumberPair is syntactically equivalent to the term $\mathbb{N} \times \mathbb{N}$.

(6) Same as (5).

Further Examples:

(7)

```
PrimaryColour
    Red   |   Blue   |   Yellow
```

[X]

(8)

```
_ ∪ _ : ℙ X × ℙ X → ℙ X
──────────────────────────────────────────
∀ s₁, s₂: ℙ X . s₁ ∪ s₂ = { x:X | x ∈ s₁  ∨ x ∈ s₂ }
```

[X]

(9)

```
flatten: seq (seq X)  → seq X                    flat.0
──────────────────────────────────────────────────────
flatten <> = <>                                  flat.1
∀ x:seq X; s:seq (seq X) .
   flatten(x ^ s) = x ⁀ (flatten s)              flat.2
```

Interpretation

(7) PrimaryColour is a type which contains exactly three elements, named Red, Blue, and Yellow. (See Appendix I).

(8) This is a <u>family</u> of definitions. For every type X, the infix symbol $\cup_{[X]}$ denotes union for sets of elements from X. This is a function which maps pairs of sets to sets. The result set and the argument sets are all subsets of X. It is customary to omit the subscript $_{[X]}$ when using the operator in a term since it can be inferred from the types of the operands. For example, if we have

gold: P FISH
angel: P FISH

then the ∪ in the term gold ∪ angel denotes the union function for sets of elements of type FISH, ie: $\cup_{(FISH)}$ For further details see Appendix 1.

(8) This specifies a family of functions on sequences of sequences. In order to be able to refer (for example within proofs) to the different parts of the specification we have labelled them. This is an informal practice which can improve the readability of documents when used judiciously.

§1.5 Chapters

As yet there is no formal way of modularising a Z document. Present practise is to divide the material presented in a specification into named chapters. Each chapter consists of some symbol definitions explained by prose, together with some theorems (see §1.6). Although it is rare for constants defined in different chapters to be given identical names, any confusion between such constants is resolved by qualifying their names with the name of the chapter in which they were defined. For example: if in a single Z text we have chapters named SIMPLE STORAGE and SAFE STORAGE, each of which defines something named STORE, and if we wish in a subsequent chapter to refer to both constants, then we use the symbol: SIMPLESTORAGE§STORE to denote the former and the symbol SAFESTORAGE§STORE to denote the latter.

It is also customary for Z documents to be self-contained. Any symbols which are referred to in a text are to be found explained in a chapter (perhaps an appendix) of the document. In order to simplify matters it is assumed, unless otherwise indicated, that the standard material of section 2 of this document (on sets, relations, functions, numbers, sequences iteration and transitive closure) is present. Constants defined in these standard chapters may freely be used anywhere in a document.

§1.6 Theorems

A theorem is a statement <u>about</u> the definitions which appear in its chapter and the chapters to which it refers. The statement is an assertion that a certain predicate has been proved from the definitions themselves together with the rules of reasoning of Z [1]. In order to be able to refer to theorems within prose or from proofs which use them, we sometimes label them with identifiers or numbers.

Syntax:

```
theorem ::=
            ⊢ predicate
            hypothesis ⊢ predicate

hypothesis ::=
            signature
            signature | predicate
            hypothesis; hypothesis
```

Examples:

```
⊢ 3 ∈ primes

x:ℕ  ⊢ x+1 ∈ ℕ                                                    (2)

x,y:ℕ | x≠0≠y ⊢ (pred x)=(pred y) ⇒ (x=y)                          (3)
```

Notes:

[1] A theorem <u>doesn't</u> amount to an assertion that we <u>wish</u> we could prove a certain predicate, or that we think that we might be able to but haven't quite enough to time to go through the motions. Such a statement is usually called a conjecture, and may be written with a ?⊢ sign in place of the ⊢ sign. For details of the way in which proofs should be presented, see Appendix 2.

[2] This theorem reads "given a natural number x, we can prove that x+1 ∈ ℕ".

[3] This theorem reads "given natural numbers x and y, which both differ from zero, we can prove that (pred x)=(pred y) ⇒ (x=y)".

2
Standard Chapters

§2.1 Sets

2.1.1 There are no "built-in" sets in Z. Despite this, it is customary to assume that the symbol $\mathbb{N}$ denotes the set of natural numbers (nonnegative integers). In fact $\mathbb{N}$, together with the usual operations on it, can be defined within Z (see Appendix 1 for details).

It is also customary to preface a Z document with a form of words such as:

> "let PERSON denote the set of all persons, and let CITY denote the set of all cities -- we need not go into the internal structure of these sets any further".

This form of words can be interpreted as meaning "we could give constructive definitions of PERSON and CITY, but such definitions would simply be a distraction at present". The symbols PERSON and CITY are said to denote "given sets" under these circumstances, and a document with such a preface characterises a family of specifications, one for each possible constructive definition of the "given sets".

2.1.2 For any set T, the term $\mathbb{P}\ T$ denotes the set of all sets whose members are drawn from T, ie the set of all subsets of T. Nothing can be a member both of T and of $\mathbb{P}\ T$.

$$\vdash T_1 \in (\mathbb{P}\ T_2) \quad \Leftrightarrow \quad T_1 \subseteq T_2$$

$$\vdash (T_1 \subseteq T_2) \wedge (T_2 \subseteq T_1) \quad \Leftrightarrow \quad (T_1 = T_2)$$

2.1.3 For any set T, the term $\mathbb{F}\ T$ denotes the set of all <u>finite</u> subsets of T. A set of elements is finite if there is a one-one correspondence between its members and an initial segment of the natural numbers.

2.1.4 For any sets $T_1 \ldots T_n$, the term $T_1 \times \ldots \times T_n$ denotes the set of all n-tuples, such as:

$$(t_1, \ldots, t_n)$$

These are characterised by

$$\vdash (t_1, \ldots, t_n) \in (T_1 \times \ldots \times T_n) \quad \Leftrightarrow \quad t_1 \in T_1 \wedge \ldots \wedge t_n \in T_n$$

Nothing can be a member both of one of the sets T_i and of $T_1 \times \ldots \times T_n$.

2.1.5 For any sets $T_1 \ldots T_n$, and any distinct symbols $id_1 \ldots id_n$ the term

$$id_1\langle\langle T_1\rangle\rangle \mid \ldots \mid id_n\langle\langle Tn\rangle\rangle$$

denotes the "labelled disjoint union" of $T_1 \ldots T_n$. Nothing can be a member both of one of the sets T_i and of the disjoint union. This is a set which contains exactly one element for each of the elements in T_1, one for each of the elements of T_2, ... and one for each of the elements of T_n <u>but no more</u>. The symbols $id_1 \ldots id_n$ denote injective functions which map to and from the disjoint union (see section 5 of [Sufrin] for more details).

2.1.5 For any distinct symbols $id_1 \ldots id_n$ the term $id_1 \mid id_2 \mid \ldots id_n$ denotes the "enumerated type" which contains exactly the elements $id_1 \ldots id_n$.

§2.2 Relations

For any sets T_1 and T_2, the set of all binary relations between T_1 and T_2 is defined to be the set of all sets of ordered pairs from $T_1 \times T_2$. When elements t_1 and t_2 are related by R, we sometimes say that R maps t_1 to t_2. The domain of a relation R is the set of all elements which are mapped to something by R; the range of R is the set of all elements to which R maps elements of its domain. The inverse of a relation R, written R^{-1}, has the ordered pairs of R reversed.

Definition:

$$[T_1, T_2]$$
$$T_1 \leftrightarrow T_2 \triangleq \mathbb{P}(T_1 \times T_2)$$

Syntactic Equivalences:

$$t_1 \mapsto t_2 \mathrel{\widehat{=}} (t_1, t_2)$$

Definitions:

$$[T_1, T_2]\ R:(T_1 \leftrightarrow T_2) \Rrightarrow \qquad (1)$$

$$\begin{array}{ll} \text{dom } R & \triangleq \{\ x:T_1 \mid (\exists y:T_2 \cdot xRy)\ \} \\ \text{ran } R & \triangleq \{\ y:T_2 \mid (\exists x:T_1 \cdot xRy)\ \} \\ R^{-1} & \triangleq \{\ (y \mapsto x) \cdot y:T_2;\ x:T_1 \mid xRy\ \} \\ t_1\ R\ t_2 & \triangleq (t_1, t_2) \in R \end{array}$$

Note:
(1) This is a context-dependent set of definitions; it signifies that for any sets T_1 and T_2 the four syntactic equivalences hold for all binary relations R between sets T_1 and T_2.

Examples:
1. For each set S of elements of T, the identity relation between pairs of elements of S is defined by:

$$\boxed{\begin{array}{l} \text{id}: (\mathbb{P}\ T) \rightarrow (T \leftrightarrow T) \\ \hline \forall\ S:\mathbb{P}\ T \cdot \\ \quad \text{id}\ S = \{\ (\ x \mapsto x\ ) \cdot x:T \mid x \in S\ \} \end{array}}$$

so, for example,

$$\text{id}\ (3,4,5) = \{\ 3 \mapsto 3,\ 4 \mapsto 4,\ 5 \mapsto 5\ \}$$

Notice that

$$S:\mathbb{P}\ T \vdash (\text{id}\ S) = (\text{id}\ S)^{-1}$$

2. $\mathbb{N} \leftrightarrow \mathbb{N}$ is the set of all relations between pairs of numbers. An example of such a relation is $\_<\_$.

$$\boxed{\_ < \_ : \mathbb{N} \leftrightarrow \mathbb{N}}$$

Which can be characterised by any of the following, logically equivalent axioms:

$$\_<\_ = \{\ n_1, n_2:\mathbb{N} \mid (\exists\ d:\mathbb{N} \cdot n_1 + d = n_2) \cdot (n_1 \mapsto n_2)\ \}$$

$$\_\leq\_ = \{\, n_1, n_2 : \mathbb{N} \mid (\exists\, d : \mathbb{N} \cdot n_1 + d = n_2) \,\}$$

$$\forall\, n_1, n_2 : \mathbb{N} \cdot \quad n_1 \leq n_2 \Leftrightarrow \exists\, d : \mathbb{N} \cdot n_1 + d = n_2$$

3. Another binary relation betwen numbers is the finite binary relation:

$$\{\, 0 \mapsto 1,\ 1 \mapsto 2,\ 2 \mapsto 3,\ 3 \mapsto 3,\ 0 \mapsto 0 \,\}$$

Theorems:

$$R : T_1 \leftrightarrow T_2 \vdash (R^{-1})^{-1} = R$$
$$R : T_1 \leftrightarrow T_2 \vdash \mathrm{dom}(R^{-1}) = \mathrm{ran}\ R$$
$$R : T_1 \leftrightarrow T_2 \vdash \mathrm{ran}(R^{-1}) = \mathrm{dom}\ R$$

§2.3 Functions

2.3.1 Notation for Functions

The set of partial functions from T_1 to T_2 is the set of relations between T_1 and T_2 which map elements of their domain to a single element of their range. A partial function is said to be total from T_1 if its domain is all of T_1. A function is said to be a finite mapping from T_1 if its domain is a finite subset of T_1.

$$
\begin{array}{l}
[T_1, T_2] \\
\quad T_1 \pfun T_2 \mathrel{\widehat=} \{ R : T_1 \rel T_2 \mid (\forall x : T_1;\ y_1, y_2 : T_2 \cdot xRy_1 \land xRy_2 \Rightarrow y_1 = y_2) \} \\
\quad T_1 \fun T_2 \mathrel{\widehat=} \{ f : T_1 \pfun T_2 \mid \mathrm{dom}\ f = T_1 \} \\
\quad T_1 \ffun T_2 \mathrel{\widehat=} \{ f : T_1 \pfun T_2 \mid \mathrm{dom}\ f \in \mathbb{F}\ T_1 \}
\end{array}
$$

An injective function is one whose inverse is also a function. We use the following symbols to denote the partial, total and finite injections

$$
\begin{array}{l}
[T_1, T_2] \\
\quad T_1 \pinj T_2 \mathrel{\widehat=} \{ f : T_1 \pfun T_2 \mid f^{-1} \in T_2 \pfun T_1 \} \\
\quad T_1 \inj T_2 \mathrel{\widehat=} (T_1 \pinj T_2) \cap (T_1 \fun T_2) \\
\quad T_1 \finj T_2 \mathrel{\widehat=} (T_1 \pinj T_2) \cap (T_1 \ffun T_2)
\end{array}
$$

A function is said to be (a surjection) onto T_2 if its range is the whole of T_2. The arrow-symbols used to denote the surjective functions are derived from the usual function arrows by adding an extra "head". The most important are those used to denote the partial surjections, the total surjections, and the one-one functions, namely:

$$
\begin{array}{l}
[T_1, T_2] \\
\quad T_1 \psur T_2 \mathrel{\widehat=} \{ f : T_1 \pfun T_2 \mid \mathrm{ran}\ f = T_2 \} \\
\quad T_1 \sur T_2 \mathrel{\widehat=} (T_1 \psur T_2) \cap (T_1 \fun T_2) \\
\quad T_1 \bij T_2 \mathrel{\widehat=} (T_1 \psur T_2) \cap (T_1 \inj T_2)
\end{array}
$$

2.3.2 Notation for Function Applications

If F is a function from T_1 to T_2, and if x is a member of the domain of F, then the term $F\ x$ denotes the unique element y of T_2 which stands in the relation F to x, in other words the value of F at x. The simplest way to formalise this is:

$$
\begin{array}{l}
[T_1, T_2] \\
\quad F : T_1 \pfun T_2;\ x : T_1 \mid x \in \mathrm{dom}\ F \vdash (x \mapsto F\ x) \in F
\end{array}
$$

In fact we may not deduce anything interesting about the term $F\ x$ if F doesn't denote a function or if x is not in the domain of F. (see Appendix 2 for further details)

§2.4 Simple Operators

2.4.1 Difference, union, and intersection functions.

$$
\begin{array}{l}
[T] \\
\hline
\quad \_ - \_, \\
\quad \_ \cup \_, \\
\quad \_ \cap \_ : (\mathbb{P}\ T) \times (\mathbb{P}\ T) \rightarrow (\mathbb{P}\ T) \\
\hline
\quad \forall S_1, S_2 : \mathbb{P}\ T \cdot \\
\qquad S_1 - S_2 = \{ x : T \mid x \in S_1 \land x \notin S_2 \} \\
\qquad S_1 \cup S_2 = \{ x : T \mid x \in S_1 \lor x \in S_2 \} \\
\qquad S_1 \cap S_2 = \{ x : T \mid x \in S_1 \land x \in S_2 \}
\end{array}
$$

2.4.2 Subset and proper subset relations.

[T]

$$\_ \subseteq \_,\quad \_ \subset \_ : (\mathbb{P}\ T) \leftrightarrow (\mathbb{P}\ T)$$

$$\forall S_1, S_2 : \mathbb{P}\ T \bullet$$
$$S_1 \subseteq S_2 \Leftrightarrow (\forall x:T \bullet x \in S_1 \Rightarrow x \in S_2)$$
$$S_1 \subset S_2 \Leftrightarrow (S_1 \neq S_2 \wedge S_1 \subseteq S_2)$$

2.4.3 Natural Numbers

The natural numbers have an element, 0. There is an injective total function, succ, on the natural numbers, whose range does not contain 0.

$$0 : \mathbb{N}$$
$$succ : \mathbb{N} \rightarrowtail \mathbb{N}$$

$$0 \notin ran\ succ$$

Notation:

$$\mathbb{N}_1 \triangleq \mathbb{N} - \{0\}$$
$$pred \triangleq succ^{-1}$$

Theorems:

Recursion Principle

[T]

$$x:T;\ g:T \rightarrow T \vdash \exists_1 f:\mathbb{N} \rightarrow T \bullet f\ 0 = x \wedge succ \circ f = f \circ g \qquad (1)$$

Induction Principle

$$S:\mathbb{P}\ \mathbb{N} \vdash 0 \in S \wedge (\forall n:S \bullet succ\ n \in S) \Rightarrow S = \mathbb{N}$$

Note:

(1) This theorem is the justification for our using the "recursion equation" form of specification for the iterative composition function iter defined in §2.7.

2.4.4 Numerals

The Arabic digits denote numbers defined by

$$1 \triangleq suc\ 0,\quad 2 \triangleq suc\ 1,\quad \ldots\quad 9 \triangleq suc\ 8$$

The usual conventions concerning the numbers denoted by multi-digit sequences apply.

2.4.5 Arithmetic Operators and Relations

```
_ + _,
_ × _: N × N → N                                                    (1)

_ - _: N × N ⇸ N                                                    (1)

_ ≤ _: N ↔ N
```

```
_ + _ =
    λ m, n:N . succ^ m                                              (2)

_ × _ =
    λ m, n:N . (_ + m)^ m                                           (3)(5)(7)
  where
    add: N → N → N
    ├──────────────
    add m n = m+n

_ - _ = λ m, n:N . pred^ m                                          (4)(5)

_ ≤ _ = succ^*                                                     (6)
```

Notes:

(1) The operator symbols × and - are given additional meanings here. In any context the particular meaning of one of these symbols can be determined from the types of its operands.

(2) Addition is defined as the repeated taking of successors.

(3) Multiplication is defined by repeated addition.

(4) Notice that the subtraction operator is <u>partial</u>, since no Natural number precedes 0.

(5) Repetition is denoted by R^* and explained in §2.7.

(6) $R^\bullet$ is a relation formed by taking the union of all repetitions of the relation R, as explained in §2.7.

(7) The term (_ + m) denotes a function which adds m to its argument. More generally, the following conditional syntactic equivalences hold:

$$f: T_1 \times T_2 \nrightarrow T_3 \;;\; s{:}T_1;\; t{:}T_2 \;\Rrightarrow\; (s\ f\ \_) \triangleq \lambda\, t{:}T_2 \,.\, f(s, t)$$

$$f: T_1 \times T_2 \nrightarrow T_3 \;;\; s{:}T_1;\; t{:}T_2 \;\Rrightarrow\; (\_\ f\ t) \triangleq \lambda\, s{:}T_1 \,.\, f(s, t)$$

2.4.6 Segments of the natural numbers

```
_ .. _ : N × N → N
```

```
∀ m, n:N . m..n = { i:N | m≤i≤n }
```

2.4.7 Finite Sets and their Cardinality

A finite set of elements of T is a subset of T which can be put into a one-one correspondence with some initial segment of the natural numbers.

```
[T]
    F T ≙ ( S:P T | ∃ n:N; f:T⇸N . f ∈ S⤖(1..n) )
```

In fact it turns out that there is only one such segment; its length is called the cardinality (or size) of the finite set. The size operator # is therefore defined by:

```
# _ : F T → N
─────────────────────────────────────
∀ S: F T .
  #S = μ ( n:N | (∃ f:T⇸N . f ∈ S⤖(1..n)) )
```

Theorem:
The size of the union of two disjoint finite sets is the sum of the sizes of the sets.

```
[T]
   S₁,S₂: F T | S₁∩S₂=()   ⊢   #(S₁ ∪ S₂) = #S₁ + #S₂
                           ⊢   # () = 0
```

2.4.5 Generalised Union and Intersection
The generalised union of a set SS of sets of T is the set which contains those members of T which are in at least one of the sets in SS. The generalised intersection of SS is the set which contains those members of T which are in all the sets of SS.

```
∪ _,
∩ _: P(P T) → (P T)
─────────────────────────────────────
∀ SS:P(P T) .
  ∪ SS = { x:T | (∃ S:SS . x∈S) }
  ∩ SS = { x:T | (∀ S:SS . x∈S) }
```

Theorems:

```
SS:P(P T)  ⊢ SS=() ⇒ ∪SS = ()
SS:P(P T)  ⊢ SS=() ⇒ ∩SS = T
```

§2.5 Operators on Relations

In this section we define relational composition, relational restrictions, relational overriding and relational image. Since functions are simply specialised kinds of relation, the relational operators defined below may also be applied to them.

$[T_1, T_2, T_3]$

2.5.1 Composition

$$\_ \fatsemi \_ : (T_1 \leftrightarrow T_2) \times (T_2 \leftrightarrow T_3) \rightarrow (T_1 \leftrightarrow T_3)$$

$$\forall R_1 : T_1 \leftrightarrow T_2;\ R_2 : T_2 \leftrightarrow T_3;\ x : T_1;\ z : T_3 \bullet (x, z) \in (R_1 \fatsemi R_2) \Leftrightarrow \exists y : T_2 \bullet (x, y) \in R_1 \land (y, z) \in R_2$$

Some authors also use the sign $\circ$ defined by: $R_2 \circ R_1 \triangleq R_1 \fatsemi R_2$

2.5.2 Domain and Range Restriction

$$\_ \upharpoonright \_ : (T_1 \leftrightarrow T_2) \times (\mathbb{P}\ T_1) \rightarrow (T_1 \leftrightarrow T_2)$$
$$\_ \setminus \_ : (T_1 \leftrightarrow T_2) \times (\mathbb{P}\ T_1) \rightarrow (T_1 \leftrightarrow T_2)$$
$$\_ \downharpoonright \_ : (T_1 \leftrightarrow T_2) \times (\mathbb{P}\ T_2) \rightarrow (T_1 \leftrightarrow T_2)$$
$$\_ / \_ : (T_1 \leftrightarrow T_2) \times (\mathbb{P}\ T_2) \rightarrow (T_1 \leftrightarrow T_2)$$

$$\forall R : T_1 \leftrightarrow T_2;\ S_1 : \mathbb{P}\ T_1;\ S_2 : \mathbb{P}\ T_2 \bullet$$
$$R \upharpoonright S_1 = (\mathrm{id}\ S_1) \fatsemi R$$
$$R \setminus S_1 = R \upharpoonright (T_1 - S_1)$$
$$R \downharpoonright S_2 = R \fatsemi (\mathrm{id}\ S_2)$$
$$R / S_2 = R \downharpoonright (T_2 - S_2)$$

It is often convenient, particularly when performing algebraic manipulations, to place domain restrictions to the left of the relation; we therefore define the following variants of the domain restriction operators:

$$\_ \upharpoonright \_ : (\mathbb{P}\ T_1) \times (T_1 \leftrightarrow T_2) \rightarrow (T_1 \leftrightarrow T_2)$$
$$\_ \setminus \_ : (\mathbb{P}\ T_1) \times (T_1 \leftrightarrow T_2) \rightarrow (T_1 \leftrightarrow T_2)$$

$$\forall R : T_1 \leftrightarrow T_2;\ S_1 : \mathbb{P}\ T_1 \bullet$$
$$S \upharpoonright R = (\mathrm{id}\ S) \fatsemi R$$
$$S \setminus R = (T_1 - S) \upharpoonright R$$

2.5.3 Relational Overriding

$$\_ \oplus \_ : (T_1 \leftrightarrow T_2) \times (T_1 \leftrightarrow T_2) \rightarrow (T_1 \leftrightarrow T_2)$$

$$\forall R_1,\ R_2 : T_1 \leftrightarrow T_2 \bullet$$
$$R_1 \oplus R_2 = (R_1 \setminus (\mathrm{dom}\ R_2)) \cup R_2$$

2.5.4 Generalised Application: Relational Image

The image of a set S through a relation R (sometimes called the R-image of S) is the set of elements of the range of R to which R maps elements of S.

$$\_ [\ \_\ ] : (T_1 \leftrightarrow T_2) \times (\mathbb{P}\ T_1) \rightarrow (\mathbb{P}\ T_2)$$

$$\forall\ R:T_1 \leftrightarrow T_2;\ S:\mathbb{P}\ T_1\ .\ R\ [\ S\ ] = \{\ y:T_2 \mid (\exists\ x:S\ .\ xRy)\ \}$$

Syntactic Equivalence:

$$R:T_1 \leftrightarrow T_2;\ t:T_1 \Rrightarrow R[\ t\ ] \triangleq R[\ (t)\ ]$$

Theorems:

$[T, T_1, T_2, T_3, T_4]$

$$R:T_1 \leftrightarrow T_2;\ S_1:\mathbb{P}\ T_1;\ S_2:\mathbb{P}\ T_2 \vdash R\ [\ \mathrm{dom}\ R\ ] = \mathrm{ran}\ R$$
$$R:T_1 \leftrightarrow T_2;\ S_1:\mathbb{P}\ T_1;\ S_2:\mathbb{P}\ T_2 \vdash R^{-1}\ [\ \mathrm{ran}\ R\ ] = \mathrm{dom}\ R$$
$$R:T_1 \leftrightarrow T_2;\ S_1, S_2:T_2 \vdash R[\ S_1 \cup S_2\ ] = R[S_1] \cup R[S_2]$$
$$R:T_1 \leftrightarrow T_2;\ S_1, S_2:T_2 \vdash R[\ S_1 \cap S_2\ ] \subseteq R[S_1] \cap R[S_2]$$
$$F:T_1 \nrightarrow T_2;\ S_1, S_2:T_2 \vdash F[\ S_1 \cap S_2\ ] = F[S_1] \cap F[S_2]$$
$$R:T_1 \leftrightarrow T_2;\ S_1, S_2:T_2 \vdash S_1 \subseteq S_2 \Rightarrow R[S_1] \subseteq R[S_2]$$

$$R:T_1 \leftrightarrow T_2 \vdash \mathrm{id}(\mathrm{dom}\ R) \subseteq R \fatsemi R^{-1}$$
$$R:T_1 \leftrightarrow T_2 \vdash \mathrm{id}(\mathrm{ran}\ R) \subseteq R^{-1} \fatsemi R$$

$$R:T_1 \leftrightarrow T_2 \vdash R^{-1} \in T_2 \nrightarrow T_1 \Rightarrow \mathrm{id}(\mathrm{dom}\ R) = R \fatsemi R^{-1}$$
$$F:T_1 \nrightarrow T_2 \vdash \mathrm{id}(\mathrm{ran}\ F) = F^{-1} \fatsemi F$$

$$R_1:T_1 \leftrightarrow T_2;\ R_2:T_2 \leftrightarrow T_3 \vdash (R_1 \fatsemi R_2)^{-1} = R_2^{-1} \fatsemi R_1^{-1}$$
$$R_1:T_1 \leftrightarrow T_2;\ R_2:T_2 \leftrightarrow T_3 \vdash \mathrm{dom}(R_1 \fatsemi R_2) = R_1^{-1}[\ \mathrm{dom}\ R_2\ ]$$
$$R_1:T_1 \leftrightarrow T_2;\ R_2:T_2 \leftrightarrow T_3 \vdash \mathrm{ran}(R_1 \fatsemi R_2) = R_2[\ \mathrm{ran}\ R_1\ ]$$

$$R_1:T_1 \leftrightarrow T_2;\ R_2:T_2 \leftrightarrow T_3;\ R_3:T_3 \leftrightarrow T_4 \vdash R_1 \fatsemi (R_2 \fatsemi R_3) = (R_1 \fatsemi R_2) \fatsemi R_3$$
$$R:T_1 \leftrightarrow T_2 \vdash R \fatsemi (\mathrm{id}\ T_2) = R = (\mathrm{id}\ T_1) \fatsemi R$$

$$R_1, R_2:T_1 \leftrightarrow T_2;\ R_3:T_2 \leftrightarrow T_3 \vdash (R_1 \cup R_2) \fatsemi R_3 = (R_1 \fatsemi R_3) \cup (R_2 \fatsemi R_3)$$
$$R_1, R_2:T_1 \leftrightarrow T_2;\ R_3:T_2 \leftrightarrow T_3 \vdash (R_1 \cap R_2) \fatsemi R_3 \subseteq (R_1 \fatsemi R_3) \cap (R_2 \fatsemi R_3)$$
$$R_1, R_2:T_1 \leftrightarrow T_2;\ F:T_2 \nrightarrow T_3 \vdash (R_1 \cap R_2) \fatsemi F = (R_1 \fatsemi F) \cap (R_2 \fatsemi F)$$

$$R_1, R_2:T_1 \leftrightarrow T_2;\ R_3:T_2 \leftrightarrow T_3 \vdash R_1 \subseteq R_2 \Rightarrow (R_1 \fatsemi R_3) \subseteq (R_2 \fatsemi R_3)$$
$$R_1:T_1 \leftrightarrow T_2;\ R_2, R_3:T_2 \leftrightarrow T_3 \vdash R_2 \subseteq R_3 \Rightarrow (R_1 \fatsemi R_2) \subseteq (R_1 \fatsemi R_3)$$

$$R_1:T_1 \leftrightarrow T_2;\ R_2, R_3:T_2 \leftrightarrow T_3 \vdash R_1 \fatsemi (R_2 \cup R_3) = (R_1 \fatsemi R_2) \cup (R_1 \fatsemi R_3)$$
$$R_1:T_1 \leftrightarrow T_2;\ R_2, R_3:T_2 \leftrightarrow T_3 \vdash R_1 \fatsemi (R_2 \cap R_3) \subseteq (R_1 \fatsemi R_2) \cap (R_1 \fatsemi R_3)$$
$$F:T_1 \nrightarrow T_2;\ R_2, R_3:T_2 \leftrightarrow T_3 \vdash F \fatsemi (R_2 \cap R_3) = (F \fatsemi R_2) \cap (F \fatsemi R_3)$$

$$R_1, R_2:T_1 \leftrightarrow T_2 \vdash (R_1 \cup R_2)^{-1} = R_1^{-1} \cup R_2^{-1}$$
$$R_1, R_2:T_1 \leftrightarrow T_2 \vdash (R_1 \cap R_2)^{-1} = R_1^{-1} \cap R_2^{-1}$$
$$R_1, R_2:T_1 \leftrightarrow T_2 \vdash (R_1 - R_2)^{-1} = R_1^{-1} - R_2^{-1}$$
$$R_1, R_2:T_1 \leftrightarrow T_2 \vdash R_1 \subseteq R_2 \Leftrightarrow R_1^{-1} \subseteq R_2^{-1}$$

$$R_1, R_2, R_3:T_1 \leftrightarrow T_2 \vdash R_1 \bullet (R_2 \bullet R_3) = (R_1 \bullet R_2) \bullet R_3$$
$$R:T_1 \leftrightarrow T_2 \vdash (R \bullet ()) = R = (() \bullet R)$$
$$R_1, R_2:T_1 \leftrightarrow T_2 \vdash \mathrm{dom}(R_1 \bullet R_2) = \mathrm{dom}\ R_1 \cup \mathrm{dom}\ R_2$$
$$R_1, R_2:T_1 \leftrightarrow T_2 \vdash \mathrm{ran}(R_1 \bullet R_2) = R_1[T_1 - \mathrm{dom}\ R_2] \cup \mathrm{ran}\ R_2$$

§2.6 Finite Sequences

2.6.1 Given a set T, the finite sequences of T are the finite partial functions from N to T whose domains are initial segments of the natural numbers.

[T]

```
seq T
{ f: N ⇸ T | dom f = 1..#f }
```

```
seq1 T
seq T - { {} }
```

For example:

$\{ 1\mapsto 5,\ 2\mapsto 6,\ 3\mapsto 77 \} \in \mathbb{N}^*$

$\{ 1\mapsto \{\},\ 2\mapsto primes \} \in \mathbb{P}(\mathbb{P}\,\mathbb{N})$

In general the following syntactic sugar is used for extensional specifications of sequences:

$\langle\rangle \triangleq \{\}$

$\langle a_1 \rangle \triangleq \{ 1\mapsto a_1 \}$

$\langle a_1 \ldots a_n \rangle \triangleq \{ 1\mapsto a_1, \ldots n\mapsto a_n \}$

Theorem:

$s:\mathbb{N}^* T_1;\ f:T_1\to T_2 \vdash s;f \in \mathbb{N}^* T_2$

2.6.2 Sequence Construction Operators

One way of building a new sequence is to push a new element onto the front of an existing sequence, thus

```
cons: (T × (seq T)) → seq T
```

$\forall\ t:T;\ s:\mathbb{N}^* T \cdot$
$\quad t\ \underline{cons}\ s = \{ 1\mapsto t \} \cup (succ^{-1};s)$

Another way is to push the new element onto the end of the sequence, thus

```
snoc: ((seq T) × T) → seq[T]
```

$\forall\ t:T;\ s:\mathbb{N}^* T \cdot$
$\quad t\ \underline{snoc}\ s = s \cup \{ succ\ \#s \mapsto x \}$

Syntactic Equivalence:

$t_1 \leftarrow t_2 \triangleq t_1\ \underline{cons}\ t_2$

$t_2 \rightarrow t_1 \triangleq t_2\ \underline{snoc}\ t_1$

Theorems:

1. Primitive Recursion for Sequences

$$
\begin{array}{l}
[T_1,\ T_2] \\
\quad x\colon T_2;\ g\colon T_1 \times T_2 \to T_2 \\
\vdash \\
\quad \exists_1\ f\colon (\mathrm{seq}\ T_1) \to T_2\ .\ f\langle\rangle = x \\
\qquad \forall\ t\colon T_1;\ s\colon \mathrm{seq}\ T_1\ .\ f(t\ \underline{\mathrm{cons}}\ s) = g(t,\ f\ s)
\end{array}
$$

2. Sequence Induction

$$
\begin{array}{l}
[T] \\
\quad S\colon \mathbb{P}(\mathrm{seq}\ T) \vdash (\langle\rangle \in S) \wedge (\forall s\colon S;\ t\colon T\ .\ (t\ \underline{\mathrm{cons}}\ s) \in S) \Rightarrow S = (\mathrm{seq}\ T)
\end{array}
$$

2.6.3 Sequence Selection Operators

$$
\begin{array}{l}
\mathrm{hd},\ \mathrm{first},\ \mathrm{last}\colon\ \mathrm{seq}_1\ T \to T \\
\mathrm{tl},\ \mathrm{back},\ \mathrm{front}\colon\ \mathrm{seq}_1\ T \to \mathrm{seq}\ T \\
\hline
\mathrm{first} = \lambda\ s\colon \mathrm{seq}\ T \mid s \neq \langle\rangle\ .\ (s\ 1) \\
\mathrm{front} = \lambda\ s\colon \mathrm{seq}\ T \mid s \neq \langle\rangle\ .\ s\lceil(1\,..\,(\mathrm{pred}\ \#s)) \\
\mathrm{back} = \lambda\ s\colon \mathrm{seq}\ T \mid s \neq \langle\rangle\ .\ \mathrm{succ}\rfloor s \setminus \{0\} \\
\mathrm{last} = \lambda\ s\colon \mathrm{seq}\ T \mid s \neq \langle\rangle\ .\ s(\#s) \\
\mathrm{hd} = \mathrm{first} \\
\mathrm{tl} = \mathrm{back}
\end{array}
$$

2.6.4 Sequence Operators — Append, Reverse

$$
\begin{array}{l}
\_ * \_ : (\mathrm{seq}\ T) \times (\mathrm{seq}\ T) \to (\mathrm{seq}\ T) \\
\mathrm{rev} : (\mathrm{seq}\ T) \to (\mathrm{seq}\ T) \\
\hline
\_ * \_ = \\
\quad \lambda\ s_1, s_2\colon \mathrm{seq}\ T\ .\ s_1 \cup (\_ - \#s_1)\rfloor s_2 \qquad (1) \\
\\
\mathrm{rev} = \\
\quad \lambda\ s\colon \mathrm{seq}\ T\ .\ (\mathrm{revseq}\ \#s)\rfloor \mathrm{succ}\rfloor s \\
\textit{where} \\
\quad \mathrm{revseq}\colon \mathbb{N} \to (\mathrm{seq}\ \mathbb{N}) \\
\quad \vdash \\
\quad \forall\ n\colon \mathbb{N}\ .\ \mathrm{revseq}\ n = (\ i \mapsto (n-i)\ .\ i\colon \mathbb{N} \mid i \in 1\,..\,n\ )
\end{array}
$$

Notes:

(1) The term $(\_ - \#s_1)$ denotes an $\mathbb{N} \to \mathbb{N}$ function which maps a number n into the number $n - \#s_1$. See note (7) of §2.4.5.

Syntactic Equivalence:

$$s_1,\ s_2\colon \mathrm{seq}\ X \Rrightarrow s_1\ \hat{}\ s_2 \triangleq s_1 * s_2$$

Theorems:

$$
\begin{array}{ll}
s\colon \mathrm{seq}\ T & \vdash \langle\rangle * s = s * \langle\rangle = s \\
s_1, s_2, s_3\colon \mathrm{seq}\ T & \vdash s_1 * (s_2 * s_3) = (s_1 * s_2) * s_3 \\
s_1, s_2\colon \mathrm{seq}\ T;\ t\colon T & \vdash (t\ \underline{\mathrm{cons}}\ s_1) * s_2 = t\ \underline{\mathrm{cons}}\ (s_1 * s_2)
\end{array}
$$

```
s1, s2: seq T          ⊢ rev( s1 * s2 ) = (rev s2) * (rev s1)
s: seq T; t: T         ⊢ rev( t cons s ) = (rev s) snoc t
                       ⊢ rev <> = <>
t: T                   ⊢ rev <t> = <t>
                       ⊢ rev ; rev = id(seq T)
```

2.6.5 Subsequences

```
after, .
for:  (seq T) × T → (seq T)
```

```
∀ s: seq T; n: N .
  s after n = succ* ; s
  s for n   = s ↾ (1..n)
```

Theorems:

```
s: seq T; n: N  ⊢ (s for n) * (s after n) = s
s: seq T        ⊢ (s for #s) = s
s: seq T        ⊢ (s after 0) = s
```

2.6.6 Sequences of Relations

Distributed composition and override of sequences of homogeneous functions are defined by primitive recursion over sequences.

```
[T]
```

```
; _,
⊛ _: seq (T↔T) → (T↔T)
```

```
⊛ <> = id T
; <> = id T

∀ s: seq(T↔T); R: T↔T .
   ⊛(R cons s) = R ⊛ (⊛S)
   ;(R cons s) = R ; (⊛S)
```

Theorems:

```
s1, s2: seq(T↔T)  ⊢ ;(s1 * s2) = (;s1) ; (;s2)
s1, s2: seq(T↔T)  ⊢ ⊛(s1 * s2) = (⊛s1) ⊛ (⊛s2)
```

§2.7 Iteration and Transitive Closure

The nth iterate of a homogeneous relation is its n-fold composition with itself. This definition is justified by the Recursion Principle for natural numbers (see §2.4.3)

```
iter: N × (T↔T) → (T↔T)
─────────────────────────────────
∀ R:T↔T; n:N .
        0 iter R = id T
  (suc n) iter R = R ; (n iter R)
```

Syntactic Equivalence:

$$R^{\iota} \quad \triangleq \quad t \ \underline{\text{iter}} \ R$$

The (reflexive) transitive closure of a homogeneous relation is the union of all its iterates, its irreflexive transitive closure is the union of all its iterates but the zeroth.

```
_* ,
_+ : (T↔T) → (T↔T)
─────────────────────────────────
∀ R:T↔T .
   R* = ∪ ( R^n . n:N )
   R+ = ∪ ( R^n . n:N₁ )
```

Theorems:

$R:(T\leftrightarrow T);\ m,n:\mathbb{N}$	$\vdash (R^m)^n = (R^n)^m = R^{(m \cdot n)}$
$R:(T\leftrightarrow T);\ m,n:\mathbb{N}$	$\vdash (R^m) ; (R^n) = R^{(m+n)}$
$R:(T\leftrightarrow T);\ n:\mathbb{N}$	$\vdash (R^n)^{-1} = (R^{-1})^n$
$R_1,R_2:T\leftrightarrow T;\ n:\mathbb{N}$	$\vdash (R_1;R_2) = (R_2;R_1) \Rightarrow (R_1 ; R_2)^n = R_1^n ; R_2^n$
$R:(T\leftrightarrow T)$	$\vdash R^* ; R = R^*$
$R:(T\leftrightarrow T)$	$\vdash R^* ; R = R^* = R ; R^*$
$R:(T\leftrightarrow T)$	$\vdash (R^*)^* = R^*$
$R:(T\leftrightarrow T)$	$\vdash (R^*)^{-1} = (R^{-1})^*$
$R:(T\leftrightarrow T)$	$\vdash (R^+)^{-1} = (R^{-1})^+$
$R_1,R_2:T\leftrightarrow T$	$\vdash (R_1;R_2) = (R_2;R_1) \Rightarrow (R_1 ; R_2)^+ = R_1^+ ; R_2^+$
$R_1,R_2:T\leftrightarrow T$	$\vdash R_1^+ \cup R_2^+ \subseteq (R_1 \cup R_2)^+$
	$\vdash succ^+ = \_<\_$
	$\vdash succ^* = \_\leq\_$

Appendix 1
The Z Type System

(omitted in Draft)

Appendix 2
Rules of Reasoning for Z

(omitted in Draft)

Acknowledgements

As it stands today, Z is the product of many persons' efforts and patience. Our source of inspiration, and (so to speak) spiritual co-author is Jean-Raymond Abrial. Those who have worked closely with us on material related to Z include: Tim Clement, Tony Hoare, Cliff Jones, Stefan Sokolowski, Mike Spivey. Those who have and influenced us over the years include: Rod Burstall, John Hughes, Roger Gimson, David Gries, Lockwood Morris, Steve Schuman, Philip Wadler. Many thanks to the MSc students who have provided a more or less tolerant audience for our ideas since 1979.

References

[Abrial]

J-R. Abrial
The Specification Language Z: Syntax and "Semantics"
Software Engineering Project Working Paper,
Programming Research Group, Oxford. April 1980. (out of print)

[Morgan]

Carroll Morgan
Schemas in Z: A preliminary reference manual
Programming Research Group, Oxford. March 1984.

[Morgan&Sufrin]

Carroll Morgan & Bernard Sufrin
Specification of the Unix file system
IEEE Transactions on Software Engineering. March 1984.

[Sufrin]

Bernard Sufrin
Mathematics for System Specification
Lecture Notes 1983/84
Programming Research Group, Oxford. September 1983.

Bibliography

(omitted in Draft).

(3)

The Schema Language

Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

July 1984

Abstract

One of the more prominent features of the Z specification technique is its use of schemas. This document gives a compact description of what schemas are and how they are manipulated. Some of the notations introduced are still considered preliminary.

Contents.

1. Introduction

Schemas are a device for organising the presentation of the mathematical text of Z specifications. Specifications in Z are generally given as predicates relating observations of the object specified; for example, the following specification is satisfied by any right-angled triangle:

> Let the positive real numbers a, b, and c be the lengths of the given triangle's three sides. Then
>
> $$a^2 + b^2 = c^2$$

In this specification, it is the English text which associates the names a, b, and c with the sides of triangles. In Z, the mathematical text would in addition introduce the variable's types (suppose R^* is the set of positive real numbers):

$$a,\ b,\ c:\ R^*$$

This pattern of declaration and predicate occurs so frequently in Z specifications -

in sets	$(a, b, c:\ R^* \mid a^2 + b^2 = c^2)$
in functions	$\lambda a, b, c:\ R^* \mid a^2 + b^2 = c^2 .\ {}^1/_2\, ab$
in predicates	$\forall a, b, c:\ R^* \mid a^2 + b^2 = c^2 .\ c>a \wedge c>b$ $\exists a, b, c:\ R^* \mid a^2 + b^2 = c^2 .\ a = b$

that it has taken on a life of its own; it has become the schema:

Pythagorean

$$a,\ b,\ c:\ R^*$$

$$a^2 + b^2 = c^2$$

The above is a named schema (Pythagorean) expressing the relationship holding among the sides of a right-angled triangle.

The advantage of recognising and naming schemas is that it simplifies the presentation of large but shallow mathematical text (which is typical of specification).

2. Definition, notation, and naming of schemas

2.1. Definition of a schema

A schema comprises a signature part and a predicate part, either of which may be empty. The signature part is a list of variable declarations, and each declaration consists of the variable's name and its type. For example, the following signature declares two variables:

```
chief:    PERSON
indians:  ℙ PERSON
```

chief is of type PERSON (that is, it may take values from that set), and indians is of type ℙ PERSON (the powerset of the set PERSON). The set PERSON itself is assumed to be defined elsewhere.

The predicate part of a schema consists of a single predicate - for example:

```
chief ∉ indians
```

The type constructors (ℙ etc.) and the predicate syntax are given in Part I of the Z handbook.

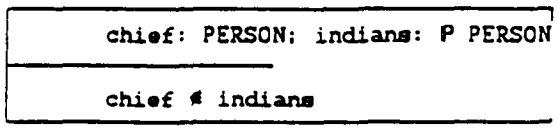
2.2. How schemas are written

Schemas may be written in either a horizontal or vertical form:

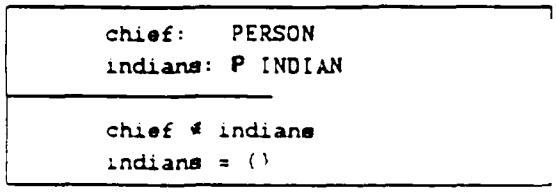
Horizontal

```
chief: PERSON; indians: ℙ PERSON | chief ∉ indians
```

Vertical



The above schema describes the relationship between the chief of an indian tribe and his indians. In the horizontal form, the signature and predicate are separated by a vertical bar (pronounced "such that"). In the vertical form, the signature and predicate are separated by a horizontal line (again "such that"), and the schema itself is enclosed in a box. As a convenience, declarations may be broken at semicolons, and long predicates may be broken at conjunctions (∧), and written on several lines with the ; or ∧ elided - for example,

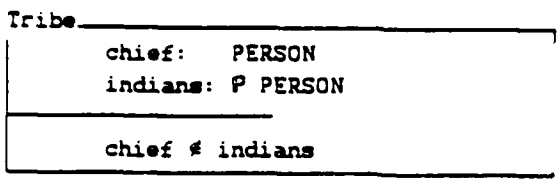


2.3. How schemas are named

Naming a schema introduces a syntactic equivalence between the name and the schema itself. In the horizontal form, a schema is named by introducing the name and the schema together, separated by a ("is syntactically equivalent to"):

$$Tribe \mathrel{\widehat=} chief: PERSON; indians: \mathbb{P}\, PERSON \mid chief \notin indians$$

In the vertical form, the schema is named by labelling its surrounding box:



3. Schemas within mathematical text

Section 1 above noted that many of the mathematical notations of set theory have a schema-like syntax. Set comprehension is one example; here is the set of all tribes:

```
{chief: PERSON; indians: ℙ PERSON |
      chief ∉ indians . (chief, indians)}.
```

Using the syntactic equivalence defined in section 2.3 above, one could write this as just

```
{Tribe . (chief, indians)},
```

or, using the convention introduced in section 1.2 of the handbook, as simply `{Tribe}`. This macro-like use of schemas was in fact their original application, and the "raw" mathematics could always be recovered by substituting a schema's body wherever its name occurred.

3.1. Rules of syntactic equivalence

Given below are the contexts in which schemas may appear directly in mathematical text.

Set comprehension

A schema enclosed in set-braces "{}" is syntactically equivalent to the corresponding set comprehension.

```
{Tribe}
```

is equivalent to

```
{chief: PERSON; indians: ℙ PERSON | chief ∉ indians}
```

In some circumstances, the brackets can be omitted; see 3.2 below. When we use this form to denote a set, we choose not to know the ordering of components in elements of the set. Hence we follow the practice of using this form only where the ordering does not matter. We disallow sentences like:

```
{Tribe} ⊆ PERSON × (ℙ PERSON),
(c, s) ∉ {Tribe} where c: PERSON; s: ℙ PERSON.
```

We allow sentences like

```
{Tribe | #indians < 120} ⊆ {Tribe}.
```

[Here, # is the cardinality operator on sets].

Lambda abstraction

A schema preceded by "λ" and followed by "." is syntactically equivalent to the corresponding lambda abstraction.

λTribe. #indians

is equivalent to

λchief: PERSON; indians: ℙ PERSON | chief ∉ indians. #indians.

The projection function which for a given tribe gives the set of its indians is

λTribe. indians..

If tr: Tribe, then the application of this projection function to tr,

(λTribe. indians)(tr),

is often written indians(tr) or tr.indians for convenience.

Quantification

A schema preceded by "∀" or "∃", and followed by "." is syntactically equivalent to the corresponding quantification.

∃Tribe.indians = {}

is equivalent to

∃chief: PERSON; indians: ℙ PERSON | chief ∉ indians. indians = {}

Tuple

A schema preceded by the symbol "tuple" is syntactically equivalent to the ordered tuple of its variable names in some undetermined order. For example,

tuple Tribe

is an ordered pair containing the names chief and indians, i.e. it might be

(chief, indians)

In some circumstances, the tuple can be omitted; see 3.2 below. As we choose not to know the ordering of components, we follow the practice of using this form only where the ordering does not matter. Hence we disallow

tuple Tribe ∈ PERSON × (ℙ PERSON),

while we allow

tuple Tribe ∈ (Tribe).

Furthermore, we use tuple only where the context agrees with the signature of the schema.

Pred

A schema preceded by the symbol "pred" is syntactically equivalent to its predicate part.

pred Tribe

is equivalent to

chief ≠ indians

In some circumstances, the pred can be omitted; see 3.2 below. To avoid ambiguity, pred should be used only where the context agrees with the schema's signature.

3.2. Omission of (), tuple, and pred

The set braces "()" can be omitted when the schema appears as part of a type; set comprehension is assumed. For example,

size: (Tribe) → N

may be abbreviated

size: Tribe → N

"tuple" can be omitted where syntax requires a term, and "pred" where syntax requires a predicate. For example,

chief ≠ indians → (chief, indians) ∈ (Tribe)

is equivalent to

pred Tribe → tuple Tribe ∈ (Tribe)

which can be abbreviated (but inadvisedly)

Tribe → Tribe ∈ (Tribe)

Note, however, that here, the braces () cannot be dropped.

4. Basic schema operations

The basic operations of renaming, decoration, inclusion, and extension allow schemas to be constructed directly from other schemas. (Some of these operations are special cases of more general operations introduced in section 6.)

4.1. Renaming variables

Renaming a schema variable changes its name in the signature and in the predicate part (where it may be necessary to further rename bound variables in order to avoid clashes). The notation is

$$S\,[new/old]$$

For example,

$$Tribe\,[PM/chief]\,[cabinet/indians]$$

$$\begin{array}{|l} PM: \quad PERSON \\ cabinet: \mathbb{P}\ PERSON \\ \hline PM \notin cabinet \end{array}$$

As usual,

$$[new_1/old_1]\,[new_2/old_2]\ \ldots$$

may be written

$$[new_1/old_1,\ new_2/old_2\ \ldots]$$

4.2. Schema decoration

Schema decoration is a special case of variable renaming: decorating a schema is equivalent to so decorating each of its variables. Typical decorations are superscripts and subscripts; for example

$$Tribe'$$

$$\begin{array}{|l} chief': \quad PERSON \\ indians': \mathbb{P}\ PERSON \\ \hline chief' \notin indians' \end{array}$$

For a decorated schema, it is guaranteed that the ordering of components obtained with tuple and set comprehension agrees with that obtained for the original schema. This allows us to write for example,

$$tuple\ Tribe' = tuple\ Tribe.$$

4.3. Schema inclusion

A (super-)schema can be built from other (sub-)schemas by including the sub-schemas in the signature of the super-schema. Each sub-schema adds its variables to the super-signature, and the sub-predicates are conjoined with the super-predicate. Duplication of variable names is allowed (and in fact is common) as long as the duplicated variables agree in type. For example, let

```
Squaws ≙ indians, squaws: P PERSON | squaws ⊆ indians
```

Then the following four schemas are equivalent:

```
Tribe                                   ≙
Squaws
```

```
Tribe                                   ≙
squaws: P PERSON
─────────────────
squaws ⊆ indians
```

```
chief: PERSON                           ≙
Squaws
─────────────────
chief ∉ indians
```

```
chief:      PERSON
indians,
squaws:     P PERSON
─────────────────
chief  ∉ indians
squaws ⊆ indians
```

4.4. Schema extension

A new declaration can be added to the signature part of a schema by the notation

```
schema: newdeclaration
```

For example,

```
Tribeadults ≙ Tribe: squaws: ℙ PERSON
```

is equivalent to

```
Tribeadults
chief:   PERSON
indians: ℙ PERSON
squaws:  ℙ PERSON
─────────────────
chief ∉ indians
```

A new predicate can be conjoined to the predicate part of a schema by the notation

```
schema | newpredicate
```

For example,

```
SmallTribe ≙  Tribe | indians = {}
```

is equivalent to

```
SmallTribe
chief:   PERSON
indians: ℙ PERSON
─────────────────
chief ∉ indians
indians = {}
```

and

```
∀Tribe | indians = {}. #indians = 0
```

is equivalent to

```
∀chief: PERSON; indians: ℙ PERSON |
        chief ∉ indians ∧ indians = {}. #indians = 0
```

5. Conventions for using the basic operations

The following is a mathematical description of the event of electing a new chief. In it, the variables chief and indians represent observations before that event; chief' and indians' represent observations after it. candidates is the set of indians from which a new chief will be drawn.

As a first step, let the schema ΔTribe describe all events which do not change the membership of the tribe:

```
ΔTribe
        Tribe
        Tribe'
-------------------------
        {chief } ∪ indians  =
        {chief'} ∪ indians'
```

or, in full,

```
ΔTribe
        chief, chief':      PERSON
        indians, indians':  ℙ PERSON
-------------------------
        chief  ∉ indians
        chief' ∉ indians'

        {chief } ∪ indians  =
        {chief'} ∪ indians'
```

Then the schema NewChief is the definition of the event of electing a new chief:

```
NewChief
        ΔTribe
        candidates: ℙ PERSON
-------------------------
        candidates ⊆ indians
        chief' ∈ candidates
```

The above is equivalent to

NewChief

$$
\begin{array}{l}
chief, chief' : PERSON \\
indians, indians' , \\
candidates : \mathbb{P}\ PERSON \\
\hline
chief \notin indians \\
chief' \notin indians' \\
\\
(chief\ ) \cup indians = \\
(chief') \cup indians' \\
\\
candidates \subseteq indians \\
chief' \in candidates
\end{array}
$$

Although the two forms above are equivalent, the choice in the former of appropriate (sub-)schemas, and their names, has allowed a more effective presentation - as a result, it is clear that NewChief describes a tribal event (ΔTribe) which depends on one parameter (candidates).

6. Logical schema operations

Section 3.1 above introduced pred, which allows expressions such as

```
pred A ∧ pred B
```

for schemas A and B. And where allowed by section 3.2, the pred can be omitted, so that the above predicate can be written (remembering that A and B are schemas):

```
A ∧ B
```

This suggests that ∧, for example, could be defined as a schema operator directly; that is, A ∧ B would be a schema, in the appropriate context. But because of the possible confusion between schemas and predicates, it is essential that such a definition would satisfy (schema ∧ on the left):

```
pred (A ∧ B) ⇔ (pred A) ∧ (pred B)
```

In fact there is a simple definition which has this property. The logical schema operations of ∧, ∨, ⇒, ⇔, ¬, and quantification, are introduced below.

6.1. Binary operations

Binary logical operations applied to schemas form their result by:

1. merging the operands' signatures (duplicated variables are identified - but their types must agree), and
2. joining the predicate parts with the logical operator itself.

For example,

```
Tribe ∧ Squaws ⇔
```

```
┌──────────────────────────┐
│ chief:    PERSON         │
│ indians,                 │
│ squaws:   ℙ PERSON       │
├──────────────────────────┤
│ chief ∉ indians          │
│ squaws ⊆ indians         │
└──────────────────────────┘
```

This is of course equivalent also to

```
┌──────────────────────────┐
│ Tribe                    │
│ Squaws                   │
└──────────────────────────┘
```

And given the definition

NoChange

```
ΔTribe
─────────
chief' = chief
```

then

NewChief ∨ NoChange ≙

```
ΔTribe
candidates: ℙ PERSON
─────────
(candidates ⊆ indians ∧
 chief' ∈ candidates)

∨ (chief' = chief)
```

The above schema covers the contingency that the incumbent might remain.

6.2. Unary operations

A unary operator applied to a schema is applied to the predicate part directly; the signature is unaffected. For example,

¬Tribe ≙

```
chief:   PERSON
indians: ℙ PERSON
─────────
¬(chief ∉ indians)
```

6.3. Quantifiers

Both universal and existential quantification can be applied to schemas. The quantified variable must occur in the signature of the schema, and it must agree with its type in the quantification. The resulting schema is formed by removing the quantified component from the signature and so quantifying it in the predicate part. For example,

∀indians: ℙ PERSON | indians = {}. Tribe ≙

```
chief: PERSON
─────────
∀indians: ℙ PERSON | indians = {}. chief ∉ indians
```

and

$$\exists chief: PERSON.\ Tribe \ \hat{=}$$

indians: $\mathbb{P}$ PERSON
$\exists chief: PERSON.\ chief \notin indians$

6.4. Predicates as schemas

Just as a schema can be written where a predicate is required (with pred perhaps implied), a predicate can be written in the place of a schema. The implied signature is formed by declaring each free variable of the predicate, where each variable's type is in agreement with the current context. Thus

$$BigTribe \ \hat{=}\ Tribe \wedge indians \neq \{\}$$

is equivalent to

$$BigTribe \ \hat{=}\ Tribe \wedge (indians: \mathbb{P} PERSON \mid indians \neq \{\})$$

(the context is supplied by the schema Tribe), and is finally

BigTribe

chief: PERSON
indians: $\mathbb{P}$ PERSON
$chief \notin indians$
$indians \neq \{\}$

For a further example, see 7.6 below.

6.5. Conventions for using logical schema operations.

A description of the state of a practical system will often involve a large number of components, and many of the operations will leave all but a few of the components unchanged.

For example, in practice, a tribe will have some non-eligible members:

NonEligible

children, squaws: $\mathbb{P}$ PERSON
$children \cap squaws = \{\}$

and we know that election of a new chief will not affect these non-eligible members, so the election event must conform to

```
=NonEligible
    ΔNonEligible
    NonEligible = NonEligible'
```

where

ΔNonEligible ≙ NonEligible ∧ NonEligible'

The practical tribe is described by

```
PTribe
    Tribe
    NonEligible
    disjoint <indians, children, squaws>
```

where

```
[I, X]
    disjoint: P (I ⇸ P X)
    S ∈ disjoint ⇔
        ∀i, j: dom S. S i ∩ S j = {}
```

The event of electing a new chief is described by

ElectChief ≙ ΔPTribe ∧ NewChief ∧ =NonEligible.

where

ΔPTribe ≙ PTribe ∧ PTribe'.

7. Special-purpose schema operations

This section describes further schema operations which have been developed from time to time for use in particular specifications. Some of these operations have become part of the standard repertoire; some have not. It is part of the Z approach to specification that special tools can and should be developed if necessary; the following list of operations is intended to serve as an example of how this has been done in the past.

7.1. Hiding

The notation

$$schema \backslash variable$$

is syntactically equivalent to the schema (see 6.3 above)

$$\exists variable: type. \ schema$$

where the type of variable is taken from the signature of schema. As a further convenience,

$$schema \backslash variable_1 \backslash variable_2 \ ...$$

can be written

$$schema \backslash (variable_1, variable_2 \ ...)$$

Finally, the list of to-be-hidden variables $variable_1, variable_2 \ ...$ can be taken directly from the signature of another schema. That is,

$$schema_1 \backslash schema_2$$

is equivalent to

$$schema_1 \backslash (\text{"all the variables of } schema_1 \text{ which are also in } schema_2\text{"})$$

(in which the predicate part of $schema_2$ is ignored).

7.2. Projection

Projection hides all variables except those mentioned. Thus the notation

$$\text{schema} \upharpoonright (\text{variable}_1, \text{variable}_2, \ldots)$$

hides all variables of schema except variable_1, variable_2, ... And similarly to hiding,

$$\text{schema}_1 \upharpoonright \text{schema}_2$$

is syntactically equivalent to the schema

$$\text{schema}_1 \setminus (\text{"all the variables of schema}_1 \text{ which are not in schema}_2\text{"})$$

(Again the predicate part of schema_2 is ignored). Projection of schema_1 onto schema_2 retains only those variables also in the signature of schema_2; all others are hidden.

7.3. Consistency

The notation

$$\text{schema}_1 \triangleright \text{schema}_2$$
$$\text{or} \quad \text{schema}_2 \triangleleft \text{schema}_1$$

is read "schema_1 consistent with schema_2"; it is syntactically equivalent to the schema

$$(\text{schema}_1 \upharpoonright \text{schema}_2) \rightarrowtail (\text{schema}_2 \upharpoonright \text{schema}_1)$$

And the notation

$$\text{schema}_1 \diamond \text{schema}_2$$

is equivalent to the schema

$$(\text{schema}_1 \triangleleft \text{schema}_2) \ \wedge \ (\text{schema}_1 \triangleright \text{schema}_2)$$

An example of consistent with is given in section 8 below.

7.4. Forward relational composition

The notation

$$A \fatsemi B$$

denotes the forward relational composition of the two schemas A and B. It is used in specifications where A and B describe events, and follow the "undashed before/ dashed after" convention (NewChief in section 5 above is such a schema). The composition A;B describes the event "A followed by B".

Assuming for illustration the definitions

A

s, s' : S a! : Alpha
P(s, s', a')

B

s, s' : S b? : Beta
Q(b, s, s')

(in which a! is an output from event A, and b? is an input to event B) the forward relational composition is formed as follows:

1. All after variables of A which match before variables of B are identified by renaming both variables of each matching pair to a single fresh variable (variable' in A matches variable in B):

A1 ≙ A [s°/s'] ≙

s, s° : S a! : Alpha
P(s, s°, a')

B1 ≙ B [s°/s] ≙

s°, s' : S b? : Beta
Q(b, s°, s')

2. The renamed schemas are conjoined, and the fresh variable(s) hidden:

A ⨟ B ≙ (A1 ∧ B1) \ s° ≙

s, s' : S a! : Alpha b? : Beta
∃s° : S. P(s, s°, a') ∧ Q(b, s°, s')

The notation

C >> D

denotes an operation performed as a two-stage pipeline: an operation satisfying specification C is performed at the first stage, and an operation satisfying D at the second stage. The output from the first stage is used as input to the second stage if required. The 'pipe' operator >> is used in specifications where C and D describe events and follow the convention that the names of inputs and outputs end in ? and ! respectively.

Assuming for illustration the definitions

C

s1, s1': S1 a!: Alpha
P(s1, s1', a!)

D

s2, s2': S2 a?: Alpha
Q(s2, s2', a?)

then

```
C >> D ≙ (C[a/a!] ∧ D[a/a?]) \ a.
```

7.5. Domain and range

The operations dom and ran are used on schemas following the before/after' convention as in 7.4 above. Taking A and B as before,

```
dom A ≙ A ↾ ("undashed variables of A") ≙
```

s: S
∃ s': S; a': Alpha. P(s, s', a')

ran is formed by projecting onto the dashed variables, and then undashing them (a special case of renaming). Thus

```
(ran B)' ≙ B ↾ ("dashed components of B")
```

and so

```
ran B ≙
```

s: S
∃ s°: S; b: Beta. Q(b, s°, s)

Notice that the variables of ran B are undashed, and that this renaming has forced a change of bound variable (s to s°).

7.6. Application

If schema A follows the before/after' convention, and schema S is undashed, then the application of A to S is written

$$A\ [S]$$

and is syntactically equivalent to the schema

$$\mathrm{ran}\ (S' \fatsemi A)$$

(It is necessary to decorate S so that the composition operator ; properly identifies its now dashed variables with the undashed variables of A.)

This operation is very like forming the image of a set through a relation. For example, given

S

$s: \mathbb{N}$
$s = 5$

and

A

$s, s': \mathbb{N}$
$s' = s + 1$

then

$A\ [S]$ =

$s: \mathbb{N}$
$s = 6$

Following section 6.4 above, this could be written

$A\ [s = 5]$ =

$s: \mathbb{N}$
$s = 6$

7.7. Overriding

If schemas A and B follow the before/after' convention, then A overridden by B is written

$$A \oplus B$$

and is equivalent to the schema

$$(A \wedge \neg \mathrm{dom}\ B) \vee B$$

This operator is very like the overriding of functions or relations. For example, given

A

$$s, s' : \mathbb{N} \quad \vdash \quad s' = s + 1$$

and

B

$$s, s' : \mathbb{N} \quad \vdash \quad s = 5 \;\; s' = 7$$

then $A \bullet B$ =

$$s, s' : \mathbb{N} \quad \vdash \quad (s' = s + 1 \land s \neq 5) \lor (s' = 7 \land s = 5)$$

8. Schemas and theorems

Having used schemas to present a specification, one can also use schemas to construct hypotheses and state theorems about it. If a schema describes the static properties of some object, one may wish to ask if there can be such an object. For example, if the set PERSON is non-empty, then there exist an individual and a set of people which together form an instance of the schema Tribe:

$$PERSON \neq \{\} \vdash \exists Tribe$$

which is syntactically equivalent to

$$PERSON \neq \{\} \vdash \exists chief: PERSON; indians: \mathbb{P}\ PERSON \mid chief \notin indians.$$

Also, we may wish to ask if the objects described by some schema have certain properties. As a first example, when a new chief is elected, the new chief is not a child. This can be formulated

$$\vdash \forall ElectChief.\ chief' \notin children$$

(ElectChief is defined in section 6.5). The theorem can also be formulated

$$ElectChief \vdash chief' \notin children$$

which is syntactically equivalent to

$$\Delta PTribe \wedge NewChief \wedge \neg NonEligible \vdash chief' \notin children.$$

The proof is based on the axioms for the three schemas on the left hand side:

(1)	$chief' \in candidates$	from NewChief
(2)	$candidates \subseteq indians$	from NewChief
(3)	$chief' \in indians$	(1), (2)
(4)	$indians \cap children = \{\}$	from PTribe in ΔPTribe
(5)	$chief' \notin children$	(3), (4).

This completes the proof.

A second example is the following: given a non-empty tribe, it is always possible to elect a new chief, that is,

$$\vdash \forall Tribe \mid indians \neq \{\}.\ \exists Tribe'.\ NewChief.$$

or alternatively,

$$Tribe \mid indians \neq \{\} \vdash \exists Tribe'.\ NewChief.$$

An example of data refinement:

Implementing a two-dimensional array as a one-dimensional vector

Carroll Morgan

Programming Research Group
8-11 Keble Road
Oxford OX1 3QD

Prepared for the
Mathematics for Software Engineering Course
Oxford University
3rd–14th September 1984

1. The abstract state

The abstract state consists simply of a two-dimensional array:

ABS
```
array: ROW × COL → VALUE
```

where

ROW ≙ 0..(Rows - 1)

COL ≙ 0..(Cols - 1)

for some positive integers Rows and Cols.

2. The abstract operations

There are two abstract operations - one for reading from the array, and one for writing to it (ReadA for *read abstract*, WriteA for *write abstract*):

$$\Delta ABS \triangleq ABS \wedge ABS'$$

ReadA

ΔABS
r?: ROW
c?: COL
v!: VALUE

---

array' = array
v! = array (r?,c?)

WriteA

ΔABS
r?: ROW
c?: COL
v?: VALUE

---

array' = array ⊕ [(r?,c?) ↦ v?]

3. The concrete state

The concrete state consists of a one-dimensional vector; it is just large enough to accomodate all of the values in the abstract array:

CON

vector: CELL → VALUE

CELL ≙ 0..(Cells - 1) where Cells = Rows * Cols

4. The abstraction

The abstraction functions lay out the array row-by-row in the vector. IN and OUT are inverses:

IN

ABS
CON

[∀r: ROW; c: COL. array (r,c) = vector ((Cols * r) + c)]

OUT
ABS'
CON'

[∀r: ROW; c: COL. array' (r,c) = vector' ((Cols * r) + c)]

OUT could have been defined as follows, with the same effect:

OUT ≙ IN'

5. The concrete operations

There are two concrete operations, corresponding to the two abstract operations:

ΔCON ≙ CON ∧ CON'

ReadC
ΔCON
r?: ROW
c?: COL
v!: VALUE
---
vector' = vector
v! = vector((Cols * r?) + c?)

WriteC
ΔCON
r?: ROW
c?: COL
v?: VALUE
---
vector' = vector ⊕ [((Cols * r?) + c?) ↦ v?]

6. Proof of refinement

The proof of refinement is in two parts. The first part proves properties of the abstraction itself, independent of the operations that are being refined (subsections 6.1 and 6.2). The second part proves properties of the abstract and concrete operations, in their corresponding pairs (6.3 for ReadA and ReadC, 6.4 for WriteA and WriteC).

6.1 IN is total

We must show [∀ABS. [∃CON. IN]]; that is (we expand the predicate), we must show

```
[∀ array: ROW × COL → VALUE.

   [∃ vector: CELL → VALUE.

      [∀r: ROW; c: COL. array (r,c) = vector ((Cols * r) + c)]]]
```

We do this by constructing the required vector ([∃ vector ...]) explicitly:

(1) `[∀ e: CELL. vector (e) = array (e div Cols, e mod Cols)]`

But to show that this vector exists, we require

```
[∀ e: Cell.

      e div Cols  ∈  ROW
   ∧  e mod Cols  ∈  COL]
```

We need this because array is defined only for arguments in the appropriate sets - and we must show that the arguments *are* in the appropriate sets. In fact, it follows from from the definitions of CELL, ROW, and COL, and the properties of div and mod.

Now we know there *is* a vector with property (1), we must show it's the right one. For this, we need

```
[∀ array: ROW × COL → VALUE.

    [∀ e: CELL. vector (e) = array (e div Cols, e mod Cols)]

→   [∀r: ROW; c: COL. array (r,c) = vector ((Cols * r) + c)]]
```

and this follows from

```
[∀r: ROW; c: COL.

(2)        (Cols * r) + c  ∈ CELL

       ∧  ((Cols * r) + c) div Cols  = r
       ∧  ((Cols * r) + c) mod Cols  = c]
```

6.2 OUT is total

We must show [∀CON'. [∃ABS'. OUT]]; that is, we must show

```
[∀ vector': CELL → VALUE.

   [∃ array: ROW × COL → VALUE.

      [∀r: ROW; c: COL. array' (r,c) = vector' ((Cols * r) + c)]]]
```

But to show this, we need only that

```
[∀r: ROW; c: COL. (Cols * r) + c  ∈  CELL]
```

and this has already been shown ((2) above).

6.3 ReadC is a refinement of ReadA

Showing that a concrete operation is a refinement of an abstract operation is done in two stages. In one stage, we must show that whenever the abstract operation can be applied, then so can the concrete one; this is done for ReadA and ReadC in section 6.3.1. In the other stage, we must show that anything the concrete operation does is acceptable in the sense that the abstract operation could have done it also (section 6.3.2).

6.3.1 Liveness - If the abstract operation can be applied, then so can the concrete one.

We must show that (ReadA ∧ IN) ⇒ [∃CON'; v!: VALUE. ReadC]; that is, we must show that for

```
array, array': ROW × COL → VALUE
vector       : CELL → VALUE
r?           : ROW
c?           : COL
v!           : VALUE
```

the following holds:

```
   array'  = array
∧  v!      = array (r?,c?)
∧  [∀r: ROW; c: COL. array (r,c) = vector ((Cols * r) + c)

⇒

   [∃ vector': Cell → VALUE; v!: VALUE.

         vector' = vector
         v!      = vector ((Cols * r?) + c?)]
```

This implication is easy to prove, because the consequent is *always* true; the antecedent is unnecessary in the proof. The consequent is true because ReadC can always be applied, and this follows from

```
(Cols * r?) + c?  ∈  CELL
```

which we have already shown ((2) above, with change of variable).

6.3.2 Safety - The concrete operation does only what the abstract operation allows.

We must show that (IN ∧ ReadC ∧ OUT ∧ [∃ABS'; v!: VALUE. ReadA]) ⇒ ReadA; that is, we must show that for

```
array, array'  : ROW × COL → VALUE
vector, vector': CELL → VALUE
r?             : ROW
c?             : COL
v!             : VALUE
```

the following holds

```
   [∀r: ROW; c: COL. array (r,c) = vector ((Cols * r) + c)]

∧  vector' = vector
∧  v!      = vector ((Cols * r?) + c?)]

∧  [∀r: ROW; c: COL. array' (r,c) = vector' ((Cols * r) + c)]

∧  [∃ array': ROW × COL → VALUE; v!: VALUE.

        array' = array
        v!     = array (r?,c?)]

⇒

   array' = array
∧  v!     = array (r?,c?)
```

This is trivial.

6.4 WriteC is a refinement of WriteA

6.4.1 Liveness - If the abstract operation can be applied, then so can the concrete one.

We must show that (WriteA ∧ IN) ⇒ [∃CON'. WriteC]; that is, we must show that for

```
array, array': ROW × COL → VALUE
vector       : CELL → VALUE
r?           : ROW
c?           : COL
v?           : VALUE
```

the following holds

```
    array' = array ⊕ [(r?,c?) ↦ v?]
^ [∀r: ROW; c: COL. array (r,c) = vector ((Cols * r) + c)

⇒

  [∃ vector': Cell → VALUE.

        vector' = vector ⊕ [((Cols * r?) + c?) ↦ v?]
```

As usual, this is guaranteed by

```
(Cols * r?) + c?  ∈  CELL
```

6.4.2 Safety - The concrete operation does only what the abstract operation allows.

We must show that (IN ^ WriteC ^ OUT ^ [∃ABS'. WriteA]) ⇒ WriteA; that is, we must show that for

```
array, array'   : ROW × COL → VALUE
vector, vector': CELL → VALUE
r?              : ROW
c?              : COL
v?              : VALUE
```

the following holds

```
    [∀r: ROW; c: COL. array (r,c) = vector ((Cols * r) + c)]
^ vector' = vector ⊕ [((Cols * r?) + c?) ↦ v?]
^ [∀r: ROW; c: COL. array' (r,c) = vector' ((Cols * r) + c)]
^ [∃ array': ROW × COL → VALUE.
        array' = array ⊕ [(r?,c?) ↦ v?]]
⇒

  array' = array ⊕ [(r?,c?) ↦ v?]
```

We show this by considering two cases:

Case 1

We show that

```
[∀r: ROW; c: COL.
        (r,c) ≠ (r?,c?)  ⇒  array' (r,c) = array (r,c)]
```

This is a consequence of

```
[∀r: ROW; c: COL.

        (r,c) ≠ (r?,c?)

    ⟹  (Cols * r) + c  ≠  (Cols * r?) + c?]
```

That is, it is a consequence of the non-overlapping of the row-by-row representation.

Case 2

We show that

```
array (r?,c?)  =  v?
```

This is trivial.

7. Conclusion - what was important

The representation of two-dimensional arrays as one-dimensional vectors is hardly a startling refinement; the mathematics above is a lot of work for something so trivial! Even so, the exercise has not been entirely pointless. We discovered by doing it that the validity of this particular refinement depends on the following facts:

The mapping function (Cols * r) + c is guaranteed to return a result in the set CELL:

```
[∀r: ROW; c: COL. (Cols * r) + c  ∈  CELL]
```

it is an injection:

```
r?: ROW
c?: COL

[∀r: ROW; c: COL.

        (r,c) ≠ (r?,c?)  ⟹  (Cols * r) + c  ≠  (Cols * r?) + c?]
```

and it is onto (that is, it has an inverse which is total):

```
[∀r: ROW; c: COL.

        ((Cols * r) + c) div Cols  = r
    ∧   ((Cols * r) + c) mod Cols  = c]
```

In larger examples, it might be harder to "guess" just what the crucial points of the refinement are - that's why it is important to be able to be systematic. And it should be remembered that such *proofs* of refinement are necessary only once for each refinement. Any subsequent development which uses the refinement does so for free.

Examples of Specification Using Mathematics

Ian Hayes
Programming Research Group,
Oxford University,
8-11 Keble Road,
Oxford,
U. K.
OX1 3QD

Abstract

A number of specification examples are developed in a notation which is based on typed set theory.

31 Aug 84

A Symbol Table

The first example specifies a simple symbol table. It demonstrates using a mathematical function to specify an abstract data type. We will specify a symbol table with operations to update, lookup and delete entries in the symbol table. We will describe our table by a partial function from symbols (SYM) to values (VAL):

$$st : SYM \rightarrow\!\!\!\!\!\!\rightarrow VAL$$

The arrow $\rightarrow\!\!\!\!\!\!\rightarrow$ indicates a function from SYM to VAL that is not necessarily defined for all elements of SYM (hence "partial"). The subset of SYM for which it is defined is its domain of definition:

$$dom(st)$$

If a symbol s is in the domain of definition of st ($s \in dom(st)$) then st(s) is the unique value associated with s ($st(s) \in VAL$). The notation $\langle\, s \mapsto v \,\rangle$ describes a function which is only defined for that particular s

$$dom(\langle\, s \mapsto v \,\rangle) = \langle\, s \,\rangle$$

and maps that s onto v

$$\langle\, s \mapsto v \,\rangle(s) = v$$

More generally we can use the notation:

$$\langle\, x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots, x_n \mapsto y_n \,\rangle$$

where all the x_i's are distinct to define a function whose domain is $\langle\, x_1, x_2, \ldots, x_n \,\rangle$ and whose value for each x_i is the corresponding y_i. For example, if we let our symbols be names and values be ages we have the following mapping:

$$st = \langle\, \text{"Fred"} \mapsto 23, \text{"Mary"} \mapsto 19 \,\rangle$$

which maps "Fred" onto 23 and "Mary" onto 19, then the domain of st is the set:

$$dom(st) = \langle\, \text{"Fred"}, \text{"Mary"} \,\rangle$$

and

$$st(\text{"Fred"}) = 23$$
$$st(\text{"Mary"}) = 19$$

The notation $\langle\rangle$ is used to denote the empty function whose domain of definition is the empty set. Initially the symbol table will be empty:

$$st = \langle\rangle$$

We are describing a symbol table by modelling it as a partial function. This use of a function is quite different to the normal use of functions in computing where an algorithm is given to compute the value of the function for a given argument. Here we use it to describe a data structure. There may be many possible models that we can use to describe the same object. Other models of a symbol table could be a list of pairs of symbol and value, or a binary tree containing a symbol and value in each node. These other models are not as abstract because many different lists (or trees) can represent the same function. We would like two symbol tables to be equal if they give the same values for the same symbols. However, it is possible to distinguish between two unordered list representations that as symbol tables are equal; on the other hand, for the function representation different functions represent different symbol tables. The list and tree models of a symbol table tend to bias an implementor working from the specification towards a particular implementation. In fact, both lists and trees could be used to implement such a symbol table. However, any reasoning we wish to perform involving symbol tables is far easier using the partial function model than either the list or tree model.

As some operations can change the symbol table we represent the effect of an operation by the relationship between the symbol table before the operation and the symbol table after the operation. We use:

$$st, st' : SYM \pfun VAL$$

where by convention we use the undecorated symbol table (st) to represent the state before the operation and the dashed symbol table (st') the state after. The operation to update an entry in the table is described by the following schema:

Update

$$st, st' : SYM \pfun VAL$$
$$s? : SYM$$
$$v? : VAL$$

$$st' = st \oplus \{ s? \mapsto v? \}$$

A schema consists of two parts: the declarations (above the centre line) in which variables to be used in the schema are declared, and a predicate (below the centre line) containing predicates giving properties of and relating those variables. In the schema Update the second line declares a variable with name "s?" which is the symbol to be updated. The third line declares a variable with name "v?" to be the value to be associated with s? in the symbol table. By convention names in the declarations ending in "?" are inputs and names ending in "!" will be outputs; the "?" and "!" are otherwise just part of the name.

The predicate part of the schema states that it updates the symbol table (st) to give a new symbol table (st') in which the symbol s? is associated with the value v?. Any previous value associated with s? (if there was one) is lost.

The operator ⊕ (function overriding) combines two functions of the same type to give a new function. The new function $f \oplus g$ is defined at x if either f or g are defined at x, and will have value $g(x)$ if g is defined at x, otherwise it will have value $f(x)$:

$$dom(f \oplus g) = dom(f) \cup dom(g)$$

$$x \in dom(g) \Rightarrow (f \oplus g)(x) = g(x)$$

$$x \notin dom(g) \wedge x \in dom(f) \Rightarrow (f \oplus g)(x) = f(x)$$

For example:

$$\{ \text{"Mary"} \mapsto 19, \text{"Fred"} \mapsto 23 \} \oplus \{ \text{"Fred"} \mapsto 25, \text{"George"} \mapsto 62 \}$$
$$= \{ \text{"Mary"} \mapsto 19, \text{"Fred"} \mapsto 25, \text{"George"} \mapsto 62 \}$$

For the operation Update above the value of $st'(x)$ is $v?$ if $x = s?$, otherwise it is $st(x)$ provided x is in the domain of st. In our example we are only using ⊕ to override one value in our symbol table function; the operator ⊕ is, however, more general: its arguments may both be any functions of the same type.

The following schema describes the operation to look up an identifier in the symbol table:

LookUp

$st, st' : SYM \pfun VAL$
$s? : SYM$
$v! : VAL$

$s? \in dom(st) \wedge$
$v! = st(s?) \wedge$
$st' = st$

The second line of the signature declares a variable with name "s?" which is the symbol to be looked up. The third line of the signature declares a variable with name "v!" which is the value that is associated with s? in the symbol table.

The first line of the predicate states that the identifier being looked up should be in the symbol table before the operation is performed; the above schema does not define the effect of looking up an identifier which is not in the table. The second line states that the output value is the value associated with s? in the symbol table st. The final line states that the contents of the symbol table is not changed by a LookUp operation.

The operation to delete an entry in the symbol table is given by:

Delete

$$st, st' : SYM \pfun VAL$$
$$s? : SYM$$

$$s? \in dom(st) \land$$
$$st' = st \setminus \{ s? \}$$

To delete the entry for $s?$ from the symbol table it must be in the table to start with ($s? \in dom(st)$). The resultant symbol table st' is the symbol table st with $s?$ deleted from its domain. We use the domain subtraction operator $\setminus$ where:

$$dom(f \setminus s) = dom(f) - s$$

$$x \in dom(f \setminus s) \implies (f \setminus s)(x) = f(x)$$

where f is a function and s is a set of elements of the same type as the domain of f. For example:

$$\{ \text{"Mary"} \mapsto 19, \text{"Fred"} \mapsto 25, \text{"George"} \mapsto 62 \} \setminus \{ \text{"Mary"}, \text{"Fred"} \}$$
$$= \{ \text{"George"} \mapsto 62 \}$$

Exercise 1: Specify an operation to find all (the set of) identifiers that have a given value, $v?$, in the symbol table. □

File Update

The second example is a specification of a simple file update. It uses sets and functions to model the file update operation.

Each record in the file is indexed by a particular key. We will model the file as a partial function from keys to records:

$$f : Key \pfun Record$$

A transaction may either delete an existing record or provide a new record which either replaces an existing record or is added to the file. The transactions for an update of a file will be specified as a set of keys $d?$ which are to be deleted from the file, and a partial function $u?$ giving the keys to be updated and their corresponding new records. We add the further restriction that we cannot both delete a record with a given key and provide a new record for that key. For example, if:

$$f = \{ k_1 \mapsto r_1, k_2 \mapsto r_2, k_3 \mapsto r_3, k_4 \mapsto r_4 \}$$

$$d? = \{ k_2, k_4 \}$$

$$u? = \{ k_3 \mapsto r_5, k_5 \mapsto r_6 \}$$

then the resultant file f' will be:

$$f' = \{ k_1 \mapsto r_1, k_3 \mapsto r_5, k_5 \mapsto r_6 \}$$

Our specification is:

File Update

$$f, f' : Key \pfun Record$$
$$d? : \mathbb{P}\, Key$$
$$u? : Key \pfun Record$$

$$d? \subseteq dom(f) \land$$
$$d? \cap dom(u?) = \{\} \land$$
$$f' = (f \backslash d?) \oplus u?$$

The original file f and the updated file f' are modelled by partial functions from keys to records. The keys to be deleted ($d?$) are a subset of Key. Hence $d?$ is an element of the powerset of Key (the set of all subsets of Key); the notation $\mathbb{P}$ Key is used to denote the powerset of Key. The updates $u?$ are specified as a partial function from Key to Record.

We can only delete records already in the file f. Hence the set of keys to be deleted $d?$ must be a subset of the domain of the original file ($d? \subseteq dom(f)$). We are precluded from trying to both delete a key and add a new record for the same key as the intersection of the deletions with the domain of the updates must be empty ($d? \cap dom(u?) = \{\}$). The resultant file f' is the original file f with all records corresponding to keys in $d?$ deleted ($f \backslash d?$), overridden by the new records $u?$.

The last line of File Update could have equivalently been written:

$$f' = (f \oplus u?) \setminus d?$$

Although it is not always the case that these two lines are equivalent, the extra condition that the intersection of $d?$ and $dom(u?)$ is empty ensures their equivalence in this case.

Lemma: Given $d? \cap dom(u?) = \{\}$ the following identity holds:

$$(f \oplus u?) \setminus d? = (f \setminus d?) \oplus u?$$

Proof: Firstly we show the domains of the two sides are equal:

$$\begin{aligned} dom((f \oplus u?) \setminus d?) &= dom(f \oplus u?) - d? \\ &= (dom(f) \cup dom(u?)) - d? \\ &= (dom(f) - d?) \cup (dom(u?) - d?) \\ &= (dom(f) - d?) \cup dom(u?) \\ &\qquad \text{as } d? \cap dom(u?) = \{\} \\ &= dom(f \setminus d?) \cup dom(u?) \\ &= dom((f \setminus d?) \oplus u?) \end{aligned}$$

Secondly, for any key k in the domain, the two sides are equal. We prove this for the two cases: $k \in dom(u?)$ and $k \notin dom(u?)$:

(a) If $k \in dom(u?)$ then

$k \notin d?$ as $dom(u?) \cap d? = \{\}$

$((f \oplus u?) \setminus d?)(k) = u?(k)$ as $k \in dom(u?) \wedge k \notin d?$

and $((f \setminus d?) \oplus u?)(k) = u?(k)$ as $k \in dom(u?)$

(b) If $k \notin dom(u?)$ then

$((f \oplus u?) \setminus d?)(k) = (f \oplus u?)(k)$ as we assumed $k \in dom((f \oplus u?) \setminus d?)$

$= f(k)$ as $k \notin dom(u?)$

and $((f \setminus d?) \oplus u?)(k) = (f \setminus d?)(k)$ as $k \notin dom(u?)$

$= f(k)$ as $k \in dom((f \oplus u?) \setminus d?)$ □

In the specification of File Update if we were not given the extra restriction then, as specified in the last line, updated records would have precedence over deletions. If the alternative specification were used then deletions would have precedence over updates. It is sensible to include the extra restriction in the specification as it allows the most freedom in implementation without any real loss of generality.

Exercise 2: In the version of File Update given above each key has (at most) a single record associated with it. Define a new data type for a file that allows multiple records for a single key, and a new file update operation; the inputs to the operation will have to take a different form from those given above. (Hint: Use relations.) □

Virtual Memory

Virtual memory can provide a much larger apparent memory to the user than the physical main memory available. A virtual memory (VM) is implemented by a combination of main memory (MM) which stores part of the current virtual memory, a memory map (MMap) which maps those virtual addresses currently resident in the main memory into the corresponding main memory address, and secondary memory (SM) which stores that part of the virtual memory that cannot fit into main memory.

If we let Virtual_Addr be the set of virtual memory addresses, Main_Addr be the set of main memory addresses, and MU be the set of values that can be stored in a unit of memory, then we can model a virtual memory system by:

Virtual_Memory

$$VM : Virtual\_Addr \rightarrow MU$$
$$MM : Main\_Addr \rightarrow MU$$
$$SM : Virtual\_Addr \nrightarrow MU$$
$$MMap : Virtual\_Addr \rightarrowtail\!\!\!\!\!\!\rightarrow Main\_Addr$$

$$ran(MMap) = Main\_Addr \wedge$$
$$VM = SM \oplus (MM \circ MMap)$$

Both VM and MM are total functions: they are defined for all values in their respective sources, Virtual_Addr and Main_Addr. SM is not necessarily defined for all values in its source and hence is a partial function. The uncrossed arrow ($\rightarrow$) is used to indicate a total function and the crossed arrow ($\nrightarrow$) to indicate a partial function.

MMap is a partial function that is also a one-to-one correspondence: for each element in its range there is a unique corresponding element in its domain:

$$\forall y : ran(MMap) \cdot \forall x_1, x_2 : dom(MMap) \cdot (MMap(x_1) = y \wedge MMap(x_2) = y) \Rightarrow (x_1 = x_2)$$

We use the notation "$\rightarrowtail$" for a total one-to-one correspondence and the notation "$\rightarrowtail\!\!\!\!\!\!\rightarrow$" for a partial one-to-one correspondence. The term "injection" is commonly used in mathematics for a one-to-one correspondence.

The first predicate in the schema Virtual_Memory states that the range of MMap is the whole of Main_Addr. This means that for every main memory address there is a corresponding virtual memory address; such corresponding virtual memory addresses are unique because MMap is one-to-one. The memory map is only defined for those virtual addresses currently corresponding

to main memory addresses. In mathematics a function

$$f : X \twoheadrightarrow Y$$

whose range is equal to the whole of its destination (Y) is called a "surjection". We say f maps X onto Y.

In order to understand the second predicate in Virtual_Memory we need the definition of relational (and hence functional) composition "∘". If we have two relations:

$$f : X \leftrightarrow Y$$

$$g : Y \leftrightarrow Z$$

then we can compose these two relations to give a relation

$$g \circ f : X \leftrightarrow Z$$

defined by

$$x \; (g \circ f) \; y \;\equiv\; \exists y : Y \;.\; x \; f \; y \wedge y \; g \; z$$

The domain of $g \circ f$ is given by

$$dom(g \circ f) \;=\; \{ \; x : dom(f) \mid \exists y : dom(g) \;.\; x \; f \; y \; \}$$

which is not necessarily the whole of dom(f).

Properties:

$$ran(f) \subseteq dom(g) \;\Rightarrow\; dom(g \circ f) = dom(f) \qquad (a)$$

$$ran(h) \subseteq dom(g) \;\Rightarrow\; (f \bullet g) \circ h = g \circ h \qquad (b)$$

Another way of writing composition is to use the forward relational composition operator ";" where

$$f \; ; \; g \;=\; g \circ f$$

The second predicate of Virtual_Memory is:

$$VM \;=\; SM \bullet (MM \circ MMap)$$

The virtual memory is equal to the secondary memory except where virtual addresses are in the domain of the memory map, in which case the contents of the virtual memory locations are given by the contents of the corresponding (according to MMap) main memory locations. For an address addr the contents of the virtual memory is given by:

$$\begin{aligned} VM(addr) &= SM(addr) && \text{if } addr \notin dom(MM \circ MMap) \\ &= MM(MMap(addr)) && \text{if } addr \in dom(MM \circ MMap) \end{aligned}$$

Note that the specification does not require that dom(SM) and dom(MM∘MMap) are disjoint. If an address is in both domains then when the virtual memory is used the contents of SM are ignored.

Lemma: dom(MM∘MMap) = dom(MMap)

Proof:

dom(MM) = Main_Addr = ran(MMap)
as MM is total and MMap is onto.
dom(MM∘MMap) = dom(MMap)
by property (a) above □

In order to state a simple theorem about Virtual_Memory we need to introduce the concepts of the inverse of a relation (or function) and the identity function on a set. The inverse of a relation R is the relation R^{-1} defined by:

$$y\ R^{-1}\ x \quad \text{if and only if} \quad x\ R\ y$$

For a function f (a function is a relation with the additional constraint that for any x in its domain there is a unique y related to it by f) its inverse f^{-1} is not necessarily a function. For example, if

$$f = \langle a \mapsto 1,\ b \mapsto 1 \rangle$$

then

$$f^{-1} = \langle 1 \mapsto a,\ 1 \mapsto b \rangle$$

which is not a function as 1 does not map to a unique value.

The identity function on a set S is given by:

$$id(S) \triangleq \langle s : S \cdot s \mapsto s \rangle$$

It maps every element of S onto itself.

Properties: We have the following useful properties of inverses and identity functions. If

$$R : X \leftrightarrow Y$$

$$f : X \rightarrowtail Y$$

(that is, R is a relation and f is a one-to-one correspondence) then:

$$(r^{-1})^{-1} = r \qquad (c)$$

$$f^{-1} \circ f = id(dom(f)) \qquad (d)$$

$$r \circ id(dom(r)) = r \qquad (e)\ \square$$

Exercise 3: Prove the properties (a) - (e) given above. □

Theorem:

$$MM = VM \circ MMap^{-1}$$

Proof:

$VM = SM \bullet (MM \circ MMap)$

$VM \circ MMap^{-1} = (SM \bullet (MM \circ MMap)) \circ MMap^{-1}$

$= MM \circ MMap \circ MMap^{-1}$

by property (b)

as $ran(MMap^{-1}) = dom(MMap) = dom(MM \circ MMap)$ by lemma

$= MM \circ id(dom(MMap^{-1}))$

as MMap is one to one and property (d)

$= MM$

as $dom(MMap^{-1}) = ran(MMap) = Main\_Addr = dom(MM)$

and property (e) □

Exercise 4: Show that:

$$dom(SM) \cup dom(MMap) = dom(VM)$$ □

Exercise 5: If the memory units in the description given are pages of 4K bytes give a definition of MU and operations to read and write single bytes in the virtual memory given a byte address. (Ignore MM, SM, and MM for this exercise.) □

Sorting

The third example specifies sorting a sequence into non-decreasing order; it uses bags (multi-sets) and sequences.

The input and the output to Sort are sequences of items of some base type X. We model a sequence as a partial function from the positive natural numbers ($\mathbb{N}^*$) to the base type X as follows:

$$seq\ X \triangleq \{\ s : \mathbb{N}^* \pfun X \mid dom(s) = 1..\#s\ \}$$

where #s is the number of entries in the mapping s (which is also the length of the sequence s). The notation of enclosing a list of items in angle brackets can be used to construct a sequence consisting of the list of items. For example:

$$t = \langle a, b, c \rangle$$

$$= \{\ 1 \mapsto a, 2 \mapsto b, 3 \mapsto c\ \}$$

We can select an item in a sequence by indexing the sequence with the position of the item:

$$t(2) = b$$

$$s = \langle s(1), s(2), \ldots, s(\#s) \rangle$$

The empty sequence is denoted by <>.

The output of Sort must be in non-decreasing order. We define:

Non-Decreasing(s : seq X)

$$\forall i, j : dom(s) \bullet i < j \Rightarrow \neg(s(j) < s(i))$$

where > is a total ordering on the base type X.

The output of Sort must contain the same values as the input, with the same frequency. We can state this property using bags. A bag is similar to a set except that multiple occurrences of an element in a bag are significant. We can model a bag as a partial function from the base type X of the bag to the positive integers where for each element in the bag the value of the function is the number of times that element occurs in the bag:

$$bag\ X \triangleq X \pfun \mathbb{N}^*$$

We use the notation [...] to construct a bag. For example:

$$[\,1,\,2,\,2,\,2\,] \;=\; \langle\, 1 \mapsto 1,\; 2 \mapsto 3 \,\rangle$$

The following gives some examples of how sets, bags, and sequences (in this case, of natural numbers) are related:

$$\langle 1,2,2,2\rangle = \langle 1,2,2\rangle = \langle 2,1,2\rangle = \langle 1,2\rangle = \langle 2,1\rangle$$

$$[1,2,2,2] \neq [1,2,2] = [2,1,2] \neq [1,2] = [2,1]\ .$$

$$\langle 1,2,2,2\rangle \neq \langle 1,2,2\rangle \neq \langle 2,1,2\rangle \neq \langle 1,2\rangle \neq \langle 2,1\rangle$$

In specifying Sort we would like to say that the bag formed from all the items in the output sequence is the same as the bag of items in the input sequence. We introduce the function items which forms the bag of all the elements in a sequence. For example:

$$\mathrm{items}(\langle\rangle) \;=\; [\,]$$

$$\mathrm{items}(\langle 1\rangle) \;=\; [1]$$

$$\mathrm{items}(\langle 1,2,2\rangle) = \mathrm{items}(\langle 2,1,2\rangle) = [1,2,2]$$

$$\mathrm{items}(\langle 1,2,3\rangle) = \mathrm{items}(\langle 2,1,3\rangle) = [1,2,3]$$

More precisely:

$$\mathrm{items} : \mathrm{seq}\ X \rightarrow \mathrm{bag}\ X$$

$$\mathrm{items}(s) = \{\, x : \mathrm{ran}(s) \;.\; x \mapsto \#\{\, i : \mathrm{dom}(s) \mid s(i) = x \,\} \,\}$$

Each element of the base type X is mapped onto its frequency of occurrence in the sequence. The function items is more concisely given by the equation:

$$\mathrm{items}(s) \;=\; \# \circ s^{-1}$$

Finally, the specification of sorting is given by:

Sort

in?, out! : seq X

Non-Decreasing(out!) ∧
items(out!) = items(in?)

Sort is an example of a non-algorithmic specification. It specifies what Sort should achieve but not how to go about achieving it. The advantage of a non-algorithmic specification is that its meaning may be more obvious than one which contains the extra detail necessary to be algorithmic. The specification is given in terms of the (defining) properties of the problem without biasing the implementor towards a particular form of algorithm. There are many possible sorting algorithms. The implementor should be allowed the freedom to choose the most appropriate.

Exercise 6: Rewrite the sort specification for the case of sorting a sequence with no duplicates into strictly ascending order. □

A Message System

Ian Hayes
Programming Research Group,
Oxford University Computing Laboratory,
8-11 Keble Road,
Oxford,
U. K.
OX1 3QD

Abstract

The following message system is based on the message handling in CICS. The specification itself is an interesting example: it combines states (of input and output devices), and gives a number of examples of the use of the "$\gg$" operator on schemas.

2 Sep 84

Message Output

We can represent a set of output devices by a mapping from a device name to a sequence of messages that have been output to that device:

NOUT

$$noq : Name \pfun seq\ Message$$

The operations on output that we will discuss here neither create nor destroy devices:

$$\Delta NOUT \mathrel{\widehat=} NOUT \land NOUT' \mid dom\ noq' = dom\ noq$$

Sending a message to a device simply appends the message to the queue for that device:

$NSend_0$

$$\Delta NOUT$$
$$n? : Name$$
$$m? : Message$$

$$noq' = noq \oplus \{ n? \mapsto noq(n?) \frown \langle m? \rangle \}$$

Multiple Destinations

A message may be sent to a set of destinations:

$NSendM_0$

$\Delta NOUT$
$ns? : \mathbb{P}\ Name$
$m? : Message$

$ns? \subseteq dom\ noq \land$
$noq' = noq \oplus \{ n : ns? \bullet n \mapsto noq(n) \frown \langle m? \rangle \}$

All the names in ns? must correspond to valid output devices. Each device in n? is sent the message.

Conjecture

Given:

$ToSet \triangleq [n? : Name; ns! : \mathbb{P}\ Name \mid ns! = \{ n? \}]$

the following equality holds:

$NSend_0 = ToSet \gg NSendM_0$

The schema operator ">>" identifies the outputs (variables ending in "!") of its left operand with the inputs (variables ending in "?") of its right operand; these variables are hidden in the result. All other components are combined together as per schema conjunction (∧).

Message Input

We can represent a set of input devices by a mapping from a device name to a sequence of messages yet to be input from that device:

NIN

$$niq : Name \pfun seq\ Message$$

The operations on input described here will neither create nor destroy devices:

$$\Delta NIN \mathrel{\widehat=} NIN \land NIN' \mid dom\ niq' = dom\ niq$$

Receiving a message from a device simply removes it from the head of the input queue for that device:

$NReceive_0$

$$n? : Name$$
$$m! : Message$$

$$m! = hd(niq(n?)) \land$$
$$niq' = niq \oplus \{ n? \mapsto tl(niq(n?)) \}$$

Send and Receive

We can define an operation that both sends a message to a device and receives a message from that device:

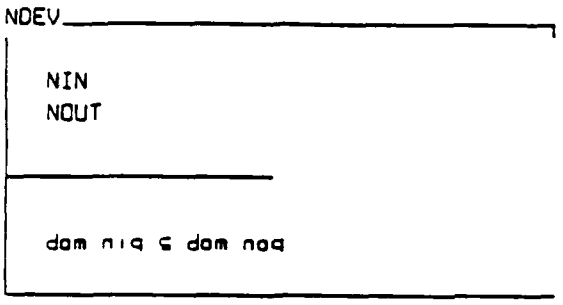
$$NSendReceive_0 \mathrel{\widehat=} NSend_0 \land NReceive_0$$

Conjecture

$$NSendReceive_0 = NSend_0 \fatsemi NReceive_0$$

Combining Input and Output

We will introduce NDEV to describe the combined input and output state for all the devices. If a device can be used for input then it must be able to be used for output:



An input operation will preserve the output state and an output operation will preserve the input state:

$$\equiv NIN \triangleq \Delta NDEV \mid NOUT' = NOUT$$

$$\equiv NOUT \triangleq \Delta NDEV \mid NIN' = NIN$$

where $\Delta NDEV \triangleq NDEV \wedge NDEV'$

The operations on the combined state are:

$$NSend \triangleq NSend_0 \wedge \equiv NIN$$

$$NSendM \triangleq NSendM_0 \wedge \equiv NIN$$

$$NReceive \triangleq NReceive_0 \wedge \equiv NOUT$$

$$NSendReceive \triangleq NSendReceive_0 \wedge \Delta NDEV$$

Conjecture

$$NSendReceive = NSend \; ; \; NReceive$$

Logical Names

Rather than work with actual (physical) device names, as we have up until this point, we would like to work with logical names that are mapped into physical device names. We use the following mapping from logical names to physical names:

LtoP

$$ltop : LName \pfun Name$$

None of the operations discussed here modify the mapping from logical names to physical names hence we will use:

$$\Xi LtoP \mathrel{\widehat=} LtoP \land LtoP' \mid LtoP' = LtoP$$

If a logical name actually corresponds to a device we perform the operation on that device, otherwise we use the device with physical name console:

MapName

$$\Xi LtoP$$
$$dev : Name \pfun seq\ Message$$
$$ln? : LName$$
$$n! : Name$$

$$n! = (\ ln? \in dom(ltop;dev) \fun ltop(ln?),\ console\ )$$

The operations on a single device become:

$$LSend \mathrel{\widehat=} MapName[nog/dev] \gg NSend$$

$$LReceive \mathrel{\widehat=} MapName[nig/dev] \gg NReceive$$

$$LSendReceive \mathrel{\widehat=} MapName[nig/dev] \gg NSendReceive$$

Conjecture

$$NOEV \mid dom\ nig = dom\ nog \vdash LSendReceive = LSend ; LReceive$$

Multiple Logical Destinations

To send a message to a set of logical names we need to map the set of logical names into physical names. If none of the logical names correspond to a device we send the message to the device with physical name console:

MapSet

```
≡LtoP
lns? : ℙ Name
ns!  : ℙ Name
NOUT

ns! = ( (ltop(lns?) ∩ dom noq = {}) → { console },
                                      ltop(lns?) ∩ dom noq
      )
```

The operation to send a message to a set of logical devices is:

```
LSendM  ≙  MapSet >> NSendM
```

Conjecture

Given:

```
ToSetL  ≙  ln? : LName; lns! : ℙ LName  |  lns! = { ln? }
```

the following equality holds:

```
LSend  =  ToSetL >> LSendM
```

Domains of the Operations

In practice we would like all the operations to be total (defined for all inputs). Unfortunately the operations as defined are not total. If a name (or a set of names) does not correspond to an actual device then the name will be translated to the special device console; if the console does not exist the operation is not defined. For the output operations ensuring that the console exists is a sufficient pre-condition for the operation to be defined. (We will also need this pre-condition for input.)

```
Pre  ≙  NDEV; LtoP; m? : Message  |  console ∈ dom nmq
```

Remember that dom nmq ⊆ dom noq so console ∈ dom noq.

Conjectures

```
Pre; ln?  : LName    ⊢  dom LSend

Pre; lns? : ℙ LName  ⊢  dom LSendM
```

For the input operations we need the additional requirement that the queue of messages yet to be input on the device is not empty:

```
PreIn  ≙  Pre; n? : Name  |  nmq(n?) ≠ <>
```

Conjectures

```
MapName[nmq/dev] >> PreIn  ⊢  dom LReceive

MapName[nmq/dev] >> PreIn  ⊢  dom LSendReceive
```

CICS TEMPORARY STORAGE

Ian Hayes
Programming Research Group,
Oxford University,
8-11 Keble Rd.,
Oxford,
U. K.
OX1 3QD

Acknowledgements

The work reported in this paper was supported by a grant from IBM. The starting point for this specification was an earlier specification done by Tim Clement. This specification has benefited greatly from the detailed comments of Carroll Morgan and Ib Holm Sorensen.

Specification

Temporary storage provides facilities for storage of information in named "queues". The operations that can be performed on an individual queue are either the standard queue-like operations (append to the end and remove from the beginning), or array-like random access read and write operations.

A Single Queue

An element of a queue is a sequence of bytes:

$$TSElem = seq[Byte]$$

A single queue may be defined by:

TSQ

$$ar : seq[TSElem]$$
$$p : \mathbb{N}$$

$$p \leq \#ar$$

The array ar contains the items in the queue. The size of the array is always equal to the number of append operations that have been performed on the queue since its creation - independently of the number of other (remove, read, or write) operations. The pointer p keeps track of the position of the item which was last removed or read from the queue.

The initial state of a queue is given by an empty array and a zero pointer:

$$TSQ\_Initial \triangleq TSQ \mid (ar = \langle\rangle) \wedge (p = 0)$$

We will define four operations on a single TSQ. The definitions of these operations will use the schema:

$$\Delta TSQ \triangleq TSQ \wedge TSQ'$$

ΔTSQ (Δ for change) defines a before state TSQ, with components ar and p (satisfying $p \leq \#ar$), and an after state TSQ', with components ar' and p' (satisfying $p' \leq \#ar'$). The schemas for the operations follow.

Append$_0$

ΔTSQ
from? : TSElem
item! : integer

$ar' = ar \frown \langle from? \rangle$
$item! = \#ar'$
$p' = p$

The new element from? (a "?" at the end of a name indicates an input) is appended to the end of ar to give the new value of the array. The position of the new item is returned in item! (a "!" at the end of a name indicates an output). The pointer position is unchanged.

Remove$_0$

ΔTSQ
into! : TSElem

$p < \#ar$
$p' = p + 1$
$into! = ar(p')$
$ar' = ar$

The pointer must not have already reached the end of the array. The pointer is incremented to the next item in the queue and the value of that item is returned in into!. The contents of the array is unchanged.

Write$_0$

ΔTSQ
item? : integer
from? : TSElem

$item? \in 1..\#ar$
$ar' = ar \oplus \{item? \mapsto from?\}$
$p' = p$

The position item? must lie within the bounds of the current array. The item at that position in ar is overridden by the value of from? to give the new value of the array. The pointer position is unchanged.

```
Read0
  ΔTSQ
  item? : integer
  into! : TSElem
  ---------------
  item? ∈ 1..#ar
  into! = ar(item?)
  p' = item?
  ar' = ar
```

The value of the item at position item?, which must lie within the bounds of the array, is returned in into!. The pointer position is updated to be item?. The array is unchanged.

In the above, all the operations have been specified in terms of the array ar and pointer p. While this is reasonable for the Read and Write operations it does not show the queue-like nature of the Append and Remove operations. Let us now show that the queue-like operations are the familiar ones. We can define a standard queue by:

```
Q
  q : seq[TSElem]
```

The standard append to the end of a queue operation is given by:

```
Standard_Append
  ΔQ
  from? : TSElem
  --------------
  q' = q ⌢ <from?>
```

where $\Delta Q \triangleq Q \wedge Q'$.

The standard remove from the front of the queue operation is given by:

```
Standard_Remove
  ΔQ
  into! : TSElem
  --------------
  q = <into!> ⌢ q'
```

The predicate in the above specification may be unconventional to some readers. It states that the value of the queue before the operation is equal to the value returned in into! catenated with the value of the queue after the operation. This form of specification more closely reflects the symmetry between Standard_Append and Standard_Remove than the more conventional:

```
q' = tail(q)
into! = head(q)
```

To see the relationship between standard queues and temporary storage queues we need to formulate the correspondence between the respective states:

```
QLike
  Q
  TSQ
  ─────────────
  q = tail^p(ar)
```

A standard q corresponds to the array ar with the first p elements removed. Given this relationship between states we will now show the relationship between $Append_0$ and Standard_Append. What we will show is that if we perform an $Append_0$ with initial state TSQ and final state TSQ' then the corresponding standard queue states Q and Q' (as determined by QLike and QLike' respectively) are related by Standard_Append. This can be formalised by the following theorem:

$$Append_0 \wedge QLike \wedge QLike' \vdash Standard\_Append$$

Proof:

1.	q,q' : seq[TSElem]; from? : TSElem	from QLike, QLike' and $Append_0$
2.	q' = tail^p(ar')	from QLike'
3.	= tail^p(ar ⁀ <from?>)	from $Append_0$
4.	= (tail^p(ar)) ⁀ <from?>	as p ≤ #ar from TSQ
5.	= q ⁀ <from?>	from QLike
6.	Standard_Append	from (1), (5) □

We can now do the same for Remove. Again we would like to show:

$$Remove_0 \wedge QLike \wedge QLike' \vdash Standard\_Remove$$

Proof:

1.	q,q' : seq[TSElem]; into! : TSElem	from QLike, QLike' and $Remove_0$
2.	p < #ar	from $Remove_0$
3.	q = tail^p(ar)	from QLike
4.	= <ar(p+1)> ⁀ (tail^(p+1)(ar))	from (2) and property of tail
5.	= <into!> ⁀ (tail^p(ar'))	from $Remove_0$
6.	= <into!> ⁀ q'	from QLike'
7.	Standard_Remove	from (1), (6) □

Errors

In allowing for errors we can introduce a report to indicate success or failure of an operation. If an error occurs we would like the TSQ to remain unchanged. This can be encapsulated by:

```
ERROR
  ΔTSQ
  report! : CONDITION
  ─────────────
  TSQ' = TSQ
```

In the operations described above there are three errors that can occur: trying to remove an item from a TSQ that is empty, trying to read or write at a position outside the array, and running out of space to store an item.

NoneLeft!
ERROR
---
p = #ar
report! = ItemErr

OutofBounds!
ERROR
item? : integer
---
item? ∉ 1..#ar
report! = ItemErr

NoSpace!
ERROR
---
report! = NoSpace

If the operations work correctly the report will indicate Success:

Successful
report! : CONDITION
---
report! = Success

The operations given previously can now be combined with the erroneous situations. We will redefine the operations in terms of their previous definitions.

$$\begin{aligned}
\text{Append} &\triangleq \text{Append}_0 \wedge \text{Successful} \vee \text{NoSpace!} \\
\text{Remove} &\triangleq \text{Remove}_0 \wedge \text{Successful} \vee \text{NoneLeft!} \\
\text{Write} &\triangleq \text{Write}_0 \wedge \text{Successful} \vee \text{OutofBounds!} \vee \text{NoSpace!} \\
\text{Read} &\triangleq \text{Read}_0 \wedge \text{Successful} \vee \text{OutofBounds!}
\end{aligned}$$

Note that NoSpace! does not specify under what conditions it occurs. The specifications of Append and Write do not allow us to determine whether or not the operation will be successful from the initial state and inputs to an operation. This is an example of a non-deterministic specification. It is left to the implementor to determine when a NoSpace! report will be returned (we hope it will not be on every call).

Named Queues

We now want to specify a system with more than one queue. A particular TSQ can be specified by name and the above operations performed on it. We will use a mapping ts from queue names (TSQName) to queues. The state of our system of queues is given by:

```
TS
  ts : TSQName ⇸ TSQ
```

The initial state of the system of queues is given by an empty mapping:

```
TS_Initial ≙ TS | ts = {}
```

Our operations require updating of a particular named TSQ. We can introduce a schema to encapsulate the common part of updating for operations on queues that already exist:

```
UpdateQ
  ΔTS
  queue? : TSQName
  ΔTSQ
  ---
  queue? ∈ dom(ts)
  TSQ = ts(queue?)
  ts' = ts ⊕ {queue? ↦ TSQ'}
```

where ΔTS ≙ TS ∧ TS'. Note that UpdateQ specifies that the named queue (alone) is updated but does not specify in what way it is updated. This is achieved by combining it with the single queue operations to get the operation on named queues.

In adding named queues we have added the possibility of a new error: trying to perform operations on non-existent queues. This error is given by:

```
NonExistent!
  ΔTS
  queue? : TSQName
  report! : CONDITION
  ---
  queue? ∉ dom(ts)
  TS' = TS
  report! = QIdErr
```

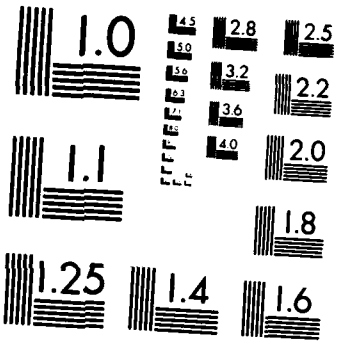
Our operations, except AppendQ which is allowed on a non-existent queue, can now be redefined in terms of our previous definitions:

```
RemoveQ ≙ (UpdateQ ∧ Remove)\ΔTSQ ∨ NonExistent!
WriteQ  ≙ (UpdateQ ∧ Write)\ΔTSQ  ∨ Nonexistent!
ReadQ   ≙ (UpdateQ ∧ Read)\ΔTSQ   ∨ NonExistent!
```

The \ΔTSQ hides the temporary variables (ar, p, ar', p') from the signature of the final operation. These operations all inherit the errors from the equivalent single queue operations.

AD-A171 671 FORMAL TECHNIQUES FOR SPECIFICATION AND VALIDATION OF TACTICAL SYSTEMS(U) MASSACHUSETTS COMPUTER ASSOCIATES INC WAKEFIELD 02 JUN 86 CADD-8606-0203 1/3

UNCLASSIFIED DAAK80-81-C-0072 F/G 9/2 NL



A queue is created by performing an AppendQ operation on a queue that does not yet exist. The following schema describes the creation of a queue:

```
CreateQ
ΔTS
queue? : TSQName
TSQ_Initial
TSQ'
---
queue? ∉ dom(ts)
ts' = ts ∪ {queue? ↦ TSQ'}
```

Again the relationship between TSQ_Initial (ar, p) and TSQ' (ar', p') is not defined within this schema. This is supplied by Append in the following definition:

$$AppendQ \triangleq ((UpdateQ \vee CreateQ) \wedge Append) \backslash \Delta TSQ$$

Note that for a non-existent queue, if an error occurs at the Append level (i.e. a NoSpace condition), then an empty queue will be created.

In addition to these promoted operations on named queues we have an operation to delete a named queue:

```
DeleteQ_0
ΔTS
queue? : TSQName
report! : CONDITION
---
queue? ∈ dom(ts)
ts' = ts \ {queue?}
report! = Success
```

An exception occurs if the queue to be deleted does not exist so the definition of DeleteQ becomes:

$$DeleteQ \triangleq DeleteQ_0 \vee NonExistent!$$

A Network of Systems

Temporary storage queues may be located on more than one system. Let us call the set of all possible system identifiers SysId. We can represent temporary storage queues on a network of systems by:

NTS
$$nts : SysId \pfun TS$$

where dom(nts) is the set of systems that share temporary storage queues and for a system with identity sysid such that sysid ∈ dom(nts), nts(sysid) is the temporary storage state of that system. The operations on temporary storage queues may be promoted to operate for a network of systems by the following schema:

Network
$$\Delta NTS$$
$$sysid? : SysId$$
$$\Delta TS$$

$$sysid? \in dom(nts)$$
$$TS = nts(sysid?)$$
$$nts' = nts \oplus \{ sysid? \mapsto TS' \}$$

where $\Delta NTS \triangleq NTS \wedge NTS'$. As with promoting the operations to work on named queues the above schema only specifies which system is updated but not how it is updated. This will be supplied when this schema is combined with the definitions of the operations on a single system. Network operation also introduces the possibility of an error if the given system does not exist:

NoSystem!
$$\Delta NTS$$
$$sysid? : SysId$$
$$report! : CONDITION$$

$$sysid? \notin dom(nts)$$
$$NTS' = NTS$$
$$report! = SysIdErr$$

The operations on a multiple system are given by:

$$AppendQN_0 \triangleq (AppendQ \wedge Network) \setminus \Delta TS \vee NoSystem!$$
$$RemoveQN_0 \triangleq (RemoveQ \wedge Network) \setminus \Delta TS \vee NoSystem!$$
$$ReadQN_0 \triangleq (ReadQ \wedge Network) \setminus \Delta TS \vee NoSystem!$$
$$WriteQN_0 \triangleq (WriteQ \wedge Network) \setminus \Delta TS \vee NoSystem!$$

The sysid? and queue? name supplied as inputs are not necessarily the ones on which an operation takes place. A queue name on a given system may be marked as actually being located on another (remote) system, possibly with a different name on that remote system. We will model this by the following function which takes the input pair of sysid? and queue? name and gives the corresponding actual sysid! and queue! name on which the operation will be performed:

$$remote : (SysId \times TSQName) \rightarrow (SysId \times TSQName)$$

In many cases the input sysid? and queue? name are the actual system and queue name; in these cases remote will behave as the identity.

We will use the following schema to incorporate remote into the operations:

TSRemote

$$sysid?, sysid! : SysId$$
$$queue?, queue! : TSQName$$

$$(sysid!, queue!) = remote(sysid?, queue?)$$

The outputs, sysid! and queue!, of TSRemote form the inputs to the operations. If a sysid? parameter is supplied then the operations on temporary storage queues are defined by:

$$AppendQN_1 \triangleq TSRemote >> AppendQN_0$$
$$RemoveQN_1 \triangleq TSRemote >> RemoveQN_0$$
$$ReadQN_1 \triangleq TSRemote >> ReadQN_0$$
$$WriteQN_1 \triangleq TSRemote >> WriteQN_0$$

If no sysid? parameter is given then the operations are given by:

$$AppendQN_2 \triangleq AppendQN_1[cursysid?/sysid?]$$
$$RemoveQN_2 \triangleq RemoveQN_1[cursysid?/sysid?]$$
$$ReadQN_2 \triangleq ReadQN_1[cursysid?/sysid?]$$
$$WriteQN_2 \triangleq WriteQN_1[cursysid?/sysid?]$$

That is, the sysid? parameter is replaced by a parameter giving the identity of the current system (the system on which the operation was initiated).

A note on the current implementation

Each system keeps track of the names of queues that are located on other (remote) systems and for each remote queue the identity of the remote system and the name of the queue on that system. It is possible that the referred request could be for a queue name that is also remote to the referred system, in which case the request will be referred on to yet another system. To find the system on which the queue actually resides we need to follow through a chain of systems until we get to a system on which the queue name is considered local. We can model the implementation by the function:

$$rem : (SysId \times TSQName) \pfun (SysId \times TSQName)$$

which for a sysid and queue name gives the sysid and queue name of the next link in the chain; if a sysid and queue name pair is not in the domain of rem then the chain is finished. The correspondence between rem and $remote$ is given by:

$$remote = rem^*$$

where rem^* is the transitive reflexive closure of rem, defined by:

$$rem^* \triangleq (id \setminus dom\ rem) \cup (rem^* \circ rem)$$

That is:

$$\begin{aligned} rem^*(s, q) &= (s, q) && \text{if } (s, q) \notin dom\ rem \\ &= rem^*\ rem\ (s, q) && \text{if } (s, q) \in dom\ rem \end{aligned}$$

As $remote$ is a total function the equality of $remote$ and rem^* requires that no chain of rem contains any loop (so that rem^* is also total).

Given the function rem if we take the corresponding (curried) function with the following shape:

$$r : SysId \to (TSQName \pfun (SysId \times TSQName))$$

so that:

$$\begin{aligned} r(s)(q) &= rem(s, q) \\ dom(r(s)) &= \{ q : TSQName \mid (s, q) \in dom\ rem \} \end{aligned}$$

The mapping that needs to be stored on a system s is given by $r(s)$, which is of type:

$$TSQName \pfun (SysId \times TSQName)$$

Formal Specification of a Simple Assembler

(Draft 1: March 1984)

Bernard Sufrin
Oxford University Programming Research Group
8 Keble Road
Oxford OX1 3QD

Abstract

In the first part of this paper we show how to construct an abstract specification of requirements for a simple assembler, so as to illustrate a typical way of using the language of set theory outlined in[Sufrin84]. Both procedural and representational abstraction are employed in order to capture the essence of the requirements without overwhelming the reader with details of possible implementations. In the second part of the paper we give the outline of a high level design for a program and indicate how to prove that it is a realisation of the requirements formalised in the first part.

Introduction

In this paper we first show how to use the language of set theory to construct a simple formalisation of requirements for an assembler, then we outline the design of a simple "in-core" assembler, show that this design meets the specification, and indicate how the design might be further developed towards an implementation. Both specification and design are presented at a rather abstract level, and are therefore "unreal". It is this very high level of abstraction which allows the specification to be simply explained and easily understood, and allows the design to be easily proven to meet the specification.

An Assembler is a program which translates a sequence of "assembly language instructions" into a sequence of "machine language instructions" ready to place in the store of the machine. In this paper we shall assume that the machine for which we are going to specify our assembler is a "one and a half address" computer; in other words each machine instruction will reside at a certain location in the store of the machine, and will have an opcode field, a register field, and an address field. We shall also assume an assembly language instruction to be divided into several "fields" — an optional symbolic label field, a symbolic opcode field, a symbolic register field, and a symbolic operand field. Each assembly language instruction will determine the content of the opcode field, the content of the register field and the content of the address field of the corresponding computer instruction. Sometimes the opcode field will contain a "directive" — perhaps indicating that the radix in which subsequent numbers are to be interpreted should change.

In order to simplify what follows we shall consider the symbolic opcode and register fields as one, and consider the machine opcode field to include the register information. A typical translation performed by the assembler, might be

Assembly Language				Machine Store		
Label	Opcode		Operand	Loc'n	Opcode	Address(octal)
	.radix		10	|		
v1:	.const		1024	|1:		2000
v2:	.const		4095	|2:		7777
	.radix		8	|		
loop:	move	r2,	v2	|3:	22	2
	addi	r2,	23	|4:	12	23
	movem	r4,	v1	|5:	34	1
	comp	r2,	v2	|6:	52	2
	jump		exit	|7:	70	11
	jumpe	r2,	loop	|8:	72	3
exit:	return			|9:	77	
	.end			|		

Primitive Data Types

Applying the principle of representational abstraction, the first thing that we decide is that in order to characterise an assembly language instruction, we do not need to know the exact details of its representation as a sequence of characters or bit strings. By the same principle, neither do we need to know the exact details of the representation of a machine instruction. We will therefore denote the entire set of assembly language instructions by:

A

and the entire set of machine language instructions by

M

The essence of an assembler can be characterised by the relationship between its input (a sequence of assembly language instructions) and its output (a sequence of machine language instructions). We will find it far easier to investigate this essence if for the time being we abstain from considering things like error listings, and relocateable binary files. This is not to say that such things will not be important in a more complete specification of requirements, but the general rubric under which we sail is "essence first, decorations later".

Formally, then, we will derive a relation of type

$$\text{seq } A \leftrightarrow \text{seq } M$$

The next step in our formalisation is to further characterise the structure of the assembly language and of the machine language. In doing so we shall denote the set of (symbolic) label identifiers by

SYM

and the set of opcode symbols by

OPSYM

Structure of Instructions

The abstract structure of assembly language instructions can now be formalised by the introduction of four functions, corresponding to the fields of the instruction and related by two axioms:

$$\begin{aligned} &lab: A \nrightarrow SYM \\ &op: A \nrightarrow OPSYM \\ &ref: A \nrightarrow SYM \\ &num: A \nrightarrow N \\ &\hline \\ &\text{dom ref} \cap \text{dom num} = \{\} \\ &\text{dom ref} \cup \text{dom num} = A \end{aligned}$$

Taken together, these formalise the fact that an assembly language instruction may have a label, and an opcode field, and must have a reference or a number field, but not both. Now in characterising the abstract structure of the language we do not care whether the Number in the operand field arose from the interpretation of a string of decimal digits, binary digits, or unary digits, so that is has been possible here to suppress radix directives. Indeed the structure presented here assumes the suppresssion of all directives.

The abstract structure of machine instructions may be characterised similarly: assuming that the instruction and address fields of such instructions are represented in our specification by natural

numbers, we have:

```
inst: M ⇸ N
addr: M → N
```

A machine language instruction may have an instruction field, and must have an address field; this allows us to use the assembler to preload numeric or symbolic values.

We shall assume that we have been given a way of translating symbolic opcodes to numbers, that is, a function:

```
mnem: OPSYM ⇸ N
```

The set of valid mnemonic opcode symbols is the domain of this function.

Part I: Requirements

We require that the assembler translate symbolic opcodes to their corresponding numeric opcodes, translate symbolic addresses, where they appear, to numbers representing the corresponding address, and translate numeric fields where they appear. In what follows we shall derive predicates corresponding to each of these requirements in turn.

Symbol Definitions

Suppose that the input sequence of assembly instructions is denoted by:

```
in: seq A
```

Exploiting the fact that a sequence is just a special kind of function from the natural numbers, we note that the composition:

```
in ; lab
```

is a function of type

```
N ⇸ SYM
```

which maps the number of each instruction in which a symbolic label is defined, to the label which is defined there. In the case of our example we have:

```
in ; lab  = ( 1↦V1   2↦V2   3↦loop   9↦exit )
```

The inverse of this function is in general a relation which maps symbols to all the places in the input where they appear as labels. For this reason we define:

```
symbtab ≙ (in ; lab)⁻¹
```

In order to formalise the idea that there should be no multiply defined symbols, we require that the inverse of symbtab be a function. Remember that in general the inverse of a function may be a one-to-many relation; requiring that it be a function is the same as requiring that it map each element of its domain to just one element of its range. Later we will be able to give additional justification for this rather obvious requirement, which is expressed formally by:

```
symbtab ∈ SYM ⇸ N
```

Symbolic and Numeric References

Once more exploiting the definition of sequences, the composition

```
in ; ref
```

is a function of type

```
N ⇸ SYM
```

which maps assembler instruction numbers in the input to the symbols which are referenced at those instructions. In the case of our example we have:

```
( 3↦v2  5↦v1  6↦v2  7↦exit  8↦loop )
```

The term

```
ran( in ; ref )
```

denotes the finite set of symbols which are referenced in the input, so to formalise the requirement that all symbols which are referenced by assembly language instructions are defined in the input, we write:

```
ran( in ; ref ) ⊆ dom symbtab
```

The function

```
in ; num
```

of type

```
N ⇸ N
```

likewise maps assembler instruction numbers in the input to the numbers which are referenced by those instructions.

Exercise 1. show that by virtue of the axioms for ref and num the two functions we have just discussed have disjoint domains.

Opcode References

The function

```
in ; op
```

of type

$N \rightarrow\!\!\!\!\!\!\rightarrow OPSYM$

maps assembler instruction numbers to the opcode symbols which are referenced by them. To formalise the requirement that all referenced opcode symbols be valid mnemonics, we write:

$ran(in \fatsemi op) \subseteq dom\ mnem$

Address Fields

Suppose that the output sequence of machine instructions is denoted by

$out: seq\ M$

then the function

$out \fatsemi addr \qquad \text{of type} \qquad N \rightarrow\!\!\!\!\!\!\rightarrow N$

maps machine addresses to their corresponding address fields. We want the address field of the instruction at location n to have the value of the symbol at

$(in \fatsemi ref)\ n$

if assembler instruction n had a symbolic operand. The corresponding value is

$(in \fatsemi ref \fatsemi symbtab)\ n$

and we want it to have the value

$(in \fatsemi num)\ n$

if the corresponding assembler instruction has a numeric operand. Since every assembler instruction must have either a numeric or a symbolic operand, we can express this formally in a single line, namely:

$(out \fatsemi addr) = (in \fatsemi ref \fatsemi symbtab) \cup (in \fatsemi num)$

In order to check that our formalisation is sensible, we should ensure that the right hand side of this equality denotes a function (since we have already established that the left hand side does. so), and (because we have stated that each assembler instruction corresponds to a single machine instruction) that the domain of this function is the same as the domain of in. Of course these needn't always be true, but the conditions under which they are true will be the preconditions for a successful assembly.

Let us first examine the conditions under which the RHS of the equality denotes a function. Since (exercise 1) $in \fatsemi ref$ and $in \fatsemi num$ must by virtue of the structure of the assembly language be functions with disjoint domains, all that remains necessary for us to articulate explicitly is that symbtab itself be a function; this condition corresponds to the "no multiply defined symbols" condition which we discussed in detail earlier.

Next we examine the conditions under which the function has the same domain as in. Since the union of the domains of

```
(in ; ref)   and   (in ; num)
```

is the domain of in, all we must articulate explicitly is the condition under which the domain of

```
(in ; ref ; symbtab)
```

is no smaller than the domain of (in ; ref). This is precisely when

```
ran(in ; ref) ⊆ dom symbtab
```

which corresponds to the "all referenced symbols are defined" condition discussed earlier.

Opcode Fields

All that remains is for us to formalise the relationship we require between the opcode fields of the input and the instruction fields of the output. This is simply:

```
out ; inst = in ; op ; mnem
```

Ensuring that every assembler instruction with an opcode field gives rise to a machine instruction with a corresponding field is just a question of ensuring that the domain of the right hand side is equal to the domain of in;op. This is ensured providing that the range of in;op is a subset of the domain of mnem, corresponding to the "all referenced opcodes must be valid mnemonics" condition discussed above.

Specification Summary

In this section we summarise the discussion so far by defining the relation assembleato which we wish to hold between the inputs and outputs of the assembler.

Context:

```
lab:  A ⇸ SYM
op:   A ⇸ OPSYM
ref:  A ⇸ SYM
num:  A ⇸ N
-----------------
dom ref ∩ dom num = {}
dom ref ∪ dom num = A
```

```
inst: M ⇸ N
addr: M → N
```

```
mnem: OPSYM ⇸ N
```

Specification:

```
assembleato: seq A ↔ seq M
-----------------------------------------------------------
∀ in:seq A; out: seq M .

  in assembleato out ⇔

     ran( in ⨾ ref ) ⊆ dom symbtab  ∧
     ran( in ⨾ op  ) ⊆ dom mnem     ∧
     symbtab ∈ SYM ⇸ N              ∧
     (out ⨾ addr)  =  (in ⨾ ref ⨾ symbtab) ∪ (in ⨾ num) ∧
     (out ⨾ inst)  =  (in ⨾ op ⨾ mnem)

where
    symbtab ≙ (in ⨾ lab)⁻¹
```

Discussion

We have illustrated two important techniques, namely procedural abstraction and representational abstraction, by formalising the essence of the relationship required between the input and output of a simple assembler. By procedural abstraction we mean statement of input-output relationships without statement of the computational structures used to achieve them; by representational abstraction we mean the statement of essential structural or semantic qualities of data, without statement of the computational structures used to store them.

In one sense we might be said to have established the basis for outlining a small "theory" of simple assemblers. Such a theory, however simple and abstract, gives us an intellectual handle by which we may grasp much more complicated machine and assembly languages, such as those outlined in the exercises below.

Any program which can be proved to satisfy the relationship defined here is, for us, an assembler. Now we haven't given any clues about how to go about constructing such a program, but that enterprise is the subject of the next section of our paper.

Exercises

2. How would you specify the appearance of a listing on which errors, such as multiply defined and undefined symbolic references, are noted.

3. What should the output sequence of instructions look like for erroneous input? Is it important?

4. How could we extend the specification to cover radix directives in the input language?

5. How would you extend the specification so as to treat register symbols properly.

6. Specify an assembler for a Vax-like machine, whose machine instructions don't all occupy the same number of addressable units.

Part 2: High Level Design of an In-Store Assembler

In this section of the paper we outline the design of a simple in-core assembler and show that it meets the specification defined above. The assembler will operate in two phases: during the first phase it will build a symbol table, place numeric operands and opcodes in store, and build structures which represent the positions of symbolic references; in the second phase it will use the symbol table and structure reference information to place the correct values in the remaining unfilled address fields.

In order to construct a model of a two phase program, we will need to define three things: a set, IS, of intermediate states (to model the state of the assembler between phases), a function, phase1, to model the first phase, and a function, phase2, to model the second phase. In the language of set theory, the way to model "first this, then that" is to compose the functions "this" and "that". More formally we aim to define

```
IS
phase1: seq A ⇸ IS
phase2: IS ⇸ seq M
```

In order to prove that the model satisfies the specification, we will have to prove that the composition of the two functions has at least the same domain as the relation assembleSto, and, moreover, that it agrees with assembleSto on its domain. More formally

```
dom(phase1 ; phase2) = dom assembleSto ∧
   (phase1 ; phase2) ⊆ assembleSto
```

We call this relationship "satisfies", and write:

```
phase1 ; phase2 ⊑ assembleto
```

The theory of "satisfaction" is presented in Appendix L. The main result allows us to give specifications spec1 and spec2 of the two phases, such that

```
spec1 ; spec2 ⊑ assembleto
```

knowing that if we can find phase1 and phase2 such that:

```
phase1 ⊑ spec1 ∧
phase2 ⊑ spec2 ∧
(ran phase1) ⊆ (dom phase2)
```

then

```
phase1 ; phase2 ⊑ assembleto
```

The Design

At the end of the first phase we shall be left with an intermediate state in which information about symbolic definitions (st), and information about symbolic references (rt) is available, and in which there is a sequence of partly-filled machine instructions (anachronistically but evocatively called core). We model this information as abstractly as possible at this first stage of development. More formally,

```
IS
  st: SYM ↔ N
  rt: N ⇸ SYM
core: seq M
```

We now define spec1

```
spec1: seq A ↔ IS

in spec1 (st, rt, core) ⇔

   st = (in ; lab)⁻¹    ∧
   rt = (in ; ref)      ∧
   (core ; addr) \ (dom rt) = (in ; num) ∧
   (core ; inst) = (in ; mnem)           ∧
   st ∈ SYM ⇸ N
```

The first line of the predicate formalises the statement that the symbol table records all definitions of each label. The second line formalises the statement that the reference table records the symbol referenced at each location whose assembly instruction had a symbolic operand. The third line formalises the statement that the in-store values of address fields corresponding to locations whose assembly instruction had numeric operands are in place. The

fourth line formalises the statement that all opcode fields are in place. The address fields of the instructions with symbolic operands are allowed, by this specification, to take any values at all.

The second phase should leave all opcode fields in place, should leave numeric address fields in place, and should "fix-up" the values of address fields corresponding to symbolic operands. Since the "source text" is no longer accessible, the only way to tell the difference between symbolic and numeric address fields is by inspecting the domain of the reference table.

Formally, we have:

```
spec2: IS ↔ seq M

(st, rt, core) spec2 out ⇔

    st ∈ SYM ⇸ N     ∧
    ran rt ⊆ dom st  ∧
    (out ; inst) = (core ; inst)  ∧
    (out ; addr) \ (dom rt) = (core ; addr) \ (dom rt) ∧
    (out ; addr) ↾ (dom rt) = (rt ; st)
```

Exercise:

Prove that the composition of spec1 and spec2 satisfies the specification assemblesto.

Conclusion

We have constructed specifications for the two phases of an in-store assembler. In each case we have captured the essence of the information processed by the phase, but in neither case have we specified the order in which the information is processed, nor have we specified the final form in which this information will be stored in a computer. This leaves a number of possibilities open to those who will define structurally and algorithmically more explicit realisations of the two phases.

Appendix 1: Satisfaction

A relation r is said to satisfy a (relational) specification s if its domain is identical to that of s and if it agrees "pointwise" with s for each element of their common domain.

```
⊴ : (X↔Y) ↔ (X↔Y)

r ⊴ s ⇔ (dom r)=(dom s) ∧ (r ⊆ s)
```

For example, the successor function on the natural numbers is a relation which satisfies the specification: "greater than"; indeed any number of iterations of suc satisfies "greater than".

```
suc ⊴ ">"      ∀ n:N . n>1 ⇒ suc^n ⊴ ">"
```

The predecessor function on the natural numbers satisfies the specification "less than", but may

not be iterated more than once without failing to satisfy it.

$$\text{pred} \sqsubseteq \text{"<"} \quad \forall\, n{:}\mathbb{N} \;.\; n{>}1 \Rightarrow \text{pred}^{n} \sqsubseteq \text{"<"}$$

The first two results are rather obvious: satisfaction is both reflexive and transitive.

$$r{:}\ X \leftrightarrow Y \vdash\ r \sqsubseteq r$$

$$r1,\ r2,\ r3{:}\ X \leftrightarrow Y \vdash (r1 \sqsubseteq r2) \wedge (r2 \sqsubseteq r3) \Rightarrow (r1 \sqsubseteq r3)$$

The main result used in this paper concerns the relationship between specifications and relational composition.

$$s1,\ r1{:}\ X \leftrightarrow Y;\ s2,\ r2{:}\ Y \leftrightarrow Z$$

$$\vdash$$

$$r1 \sqsubseteq s1 \wedge r2 \sqsubseteq s2 \wedge (\text{ran } r1) \subseteq (\text{dom } r2) \Rightarrow (r1;r2) \sqsubseteq (s1;s2)$$

These results allow us, when searching for a relation which satisfies a certain specification, to search instead for two relations which satisfy the specification when composed. These two relations can serve in turn as specifications for a pair of relations whose composition will satisfy the original specification, provided that the second implementation relation is prepared to "process" everything "output" by the first.

Acknowledgements

The author has benefited from much discussion over the years with colleagues and students at the Programming Research Group. Jean-Raymond Abrial, who first showed us how to put set theory to productive use as a Software Engineering tool, remains a continuing source of inspiration. The specification is a much altered version of [Sorensen82]; any mistakes herein are the author's own.

Bibliography

Sorensen82

Specification of a Simple Assembler
CICS Project Working Paper
Ib Holm Sorensen
Oxford University Programming Research Group, 1982

Sufrin84

Mathematics for System Specification
Lecture Notes (MSc. in Computation)
Bernard Sufrin
Oxford University Programming Research Group, 1983/84

Case Studies
in
Formal System Specification

Bernard Sufrin
Oxford
Hilary 1982/83

Abstract Algorithms

Introduction

The correctness criterion for our simple asssembler was a relation between input and output in which we used functions such as *inverse* which are not part of the repertoire of everyday (or even functional) programming languages. We used these functions in the *specification* because they were easy to *reason with*; as we develop a *program* from our specification we will be introducing functions and structures which are ever-closer to those provided by our target programming language(s).

In the case of the assembler, the *representational abstraction* which we made was to model the input and output as sequences of abstract objects rather than characters, and to model the symbol table as a binary relation between symbols and numbers. The *procedural abstraction* we made was to specify the assembly operation by using *composition on entire sequences* and *inversion*. In a later section of the course we will show how we can begin to use more machine-oriented *representations*; in this section we show how to begin to use machine-oriented *control-structures* in the realisation of what we will call Abstract Algorithms.

As our first example we take the problem of constructing the inverse of a function such as:

$$in \fatsemi f$$

where

$$in: seq[X]$$
$$f: X \pfun Y$$

This is clearly part of a solution to the problem of constructing an assembly algorithm.

We shall first give a *procedurally abstract*, state-oriented specification of the problem. Our abstract algorithm will have state which is characterised by the input which remains to be read, and by the inverse relation which has so far been constructed; more formally we have...

ST
$$in: seq[X]$$
$$rel: Y \rel N$$

For a given input and function, in_0 and f say, the abstract algorithm must transform a state in which the relation is empty and none of the input has yet been read...

$$INITIAL \triangleq (ST \mid rel=(); in=in_0)$$

into one in which there is no more input to read and in which the relation is the required inverse, namely:

$$FINAL \triangleq (ST \mid in=\langle\rangle: rel=(in_0 \fatsemi f)^{-1})$$

Perhaps it will come as no surprise that our plan for building a less procedurally abstract program will involve a loop each iteration of which will be (modelled by) the function step ...

$$step: ST \rightarrow ST$$

we want to choose this so that the function

$$INVALG\ a\ (repeat\ step)$$

satisfies the specification. In other words so that

$$INITIAL \subseteq dom\ INVALG \wedge$$
$$INVALG\ [\ INITIAL\ ] \subseteq FINAL$$

Note: *we have included a small appendix in which we define the function* repeat *and develop a small mathematical analogue of the theory of iterative control structures. Although the rest of this document is relatively independent of the appendix, it provides a more formal justification for some of the informal arguments we use below.*

Design of Step

We will define step so that it decreases the length of in whilst always resulting in a state which is in the set:

$$INVAR\ a\ (\ ST \mid rel=(sofar;f)^{-1}\ \text{where}\ sofar * in = in_0\ )$$

our definition is...

$$\begin{array}{l} step = \\ \quad \pi ST \mid in \neq <> \\ \qquad in' = tail\ in; \\ \qquad first\ in \in dom\ f \rightarrow \\ \qquad\quad rel' = rel \cup (\ f{\circ}first\ in \mapsto 1+\#in_0-\#in\ ) \\ \qquad first\ in \notin dom\ f \rightarrow \\ \qquad\quad rel' = rel \end{array}$$

... which can be proven to satisfy the above requirements.

Because step decreases the length of in, and maintains the INVARiant, the theory of iteration now allows us to conclude that:

$$(repeat\ step) \in (ST \rightarrow ST)$$
$$ran(repeat\ step) \subseteq (\ INVAR \cap\ (ST - dom\ step)\ )$$

Procedurally interpreted this means that the loop terminates in state...

$$(\ ST \mid in=<>\ ) \cap INVAR$$

that is...

$$(\ ST \mid in=<>;\ rel=(sofar;f)^{-1}\ \text{where}\ sofar=in_0\ )$$

... which by a few manipulations transforms to FINAL.

Making Step less Abstract

The specification of step is still procedurally abstract in the sense that it involves two implications:

$$\text{first in} \in \text{dom } f \rightarrow \ldots$$
$$\text{first in} \notin \text{dom } f \rightarrow \ldots$$

We can continue the process of algorithmic development by splitting step into two parts...

$$\text{change}: \quad ST \nrightarrow ST$$
$$\text{nochange}: ST \nrightarrow ST$$

which are defined so that

$$\text{step} = (\text{change} \cup \text{nochange})$$

as follows:

$$\text{change} = \mu ST \mid in \neq \langle\rangle : \text{first in} \in (\text{dom } f)$$
$$in' = \text{tail in};$$
$$rel' = rel \cup \{ f[.o]\text{first in} \mapsto 1 + \#in_0 - \#in \}$$

$$\text{nochange} = \mu ST \mid in \neq \langle\rangle : \text{first in} \notin (\text{dom } f)$$
$$in' = \text{tail in};$$
$$rel' = rel$$

The domains of these functions are disjoint, so their union is still a function. We claim that this function behaves exactly like step, leaving the proof as an exercise.

In fact taking the union of functions with disjoint domains is the closest mathematical analogue we have to the *alternation* construct of programming languages. (We will not here consider *nondeterministic alternation* of the kind modelled by unions of functions with nondisjoint domains, although this is of great theoretical and practical interest).

So far we have convinced ourselves that -- in a context where f is known -- the function

$$(\text{repeat step})$$

takes us from an initial state in which the entire input sequence remains to be processed and the relation is empty, to one in which there is no input left to be processed and the relation is the required inverse.

We can achieve this initial state by a function

$$\text{init}: \text{seq}[X] \rightarrow \text{INITIAL}$$

$$\text{init} = \lambda\, in_0 : \text{seq}[X] \,.\, \mu ST \,.\, rel = \langle\rangle;\ in = in_0$$

The overall structure of the function which specifies our abstract algorithm is now:

```
init ; (repeat (change ∪ nochange))
```

The "control constructs" used have been emphasised; they are all mathematical functions.

We now ask a rhetorical question: is it possible to convince ourselves that the following *program* implements the above abstract algorithm?

```
let in:seq[X] = in0
    rel:Y ↔ N = {} in
do
  in≠<> →
     let x:X = first in
         n:N = 1+#in0-#in in
          if
             x∈(dom f) →
                rel, in := rel ∪ { x⟼n }, tail in
          []
             x∉(dom f) →
                in := tail in
          fi
od
```

The answer, as you've probably guessed, is supposed to be "yes". What you may not have guessed is just *how* we are going to do the convincing.

What we shall do is to give the semantics of the *programming language* in terms of the mathematical tools with which we are already familiar. In order to make the presentation simple, we will not give a *formal denotational semantics* -- a mapping from phrases of the language to phrases of the mathematical toolkit; instead we will develop the language as if it were a *syntactic sugaring* of the toolkit. Once we have done this it will simply be obvious that the program and the algorithm are the same mathematical function expressed in different notations.

By showing how the control constructs of a programming language can be modelled by the "algorithmic" combinators of our specification language we hope to convince the reader that "specifications" and "programs" are both mathematical objects, subject to mathematical reasoning methods, and that there is no unbridgeable gulf between a procedurally-abstract function specification and a program which computes that function.

The Abstract Algorithmic Language

In this section we define the Abstract Algorithmic Language (AAL). The *full* language is not implemented anywhere; it is merely a notation which allows the program designer to express "procedurally" strategies for computing functions (and relations) which have been specified nonprocedurally.

An *abstract algorithm* is just a partial homogeneous relation (in this paper we will only deal with *deterministic* abstract algorithms, which are partial homogeneous functions). As we illustrated earlier, the development of an *abstract algorithm* starts from a procedurally-abstract specification; in successive steps we try to decompose the original specification into smaller "more computable" specifications, which when assembled using the "algorithmic" combinators:

∪	alternation
;	sequencing
repeat	iteration
↾	guarding

are equivalent to the original specification.

Assignment

Here is one of the simplest of the abstract algorithms:

$$\lambda\ n{:}\mathbb{N}\ .\ n \times n$$

It is the algorithm which squares its single, numeric, variable. Another way of writing it, which makes its status as an "algorithm" a little clearer is:

$$\pi\ n{:}\mathbb{N}\ .\ n' = n \times n$$

Another way of writing the *same* algorithm is provided in the AAL notation, namely:

$$\mathbf{prog}\ n{:}\mathbb{N}\ .\ n := n \times n$$

AAL Programs

In general, programs in AAL have the form:

$$\mathbf{prog}\ v_1{:}T_1\ \ldots\ v_n{:}T_n\ .\ St$$

where St is an AAL Statement in which the variables v1 ... v_n appear. We are going to explain the meaning of the AAL by giving the syntax of each kind of statement and then showing what *partial homogeneous function* the program

$$\mathbf{prog}\ v_1{:}T_1\ \ldots\ v_n{:}T_n\ .\ St$$

stands for. For the moment, however, we will continue to consider the meanings of programs of just one variable and of the forms by which such programs are cobined to produce more complex programs; later we will give the rules by which these can be generalised.

Even simple assignment programs may not always "work"; for example consider (program 1):

$$\mathbf{prog}\ n{:}\mathbb{N}\ .\ n := n-1$$

In fact this is the function

$$\pi\ n{:}\mathbb{N}\ .\ n' = n-1$$

whose domain is $\mathbb{N}-(0)$. If we give a procedural interpretation to this (that is to say if we think of it as something that is going to "run") then one explanation for this is that program 1 does not terminate for all possible starting states; an alternative explanation is to say that this program is going to cause an "exception" to occur if started with n=0.

Let us consider how we might "totalise" this program, *ie* make it run for all possible starting values of n. The simplest way is to see if n is zero and in that case do nothing at all. The *hybrid* (because expressed in a mixture of the AAL and mathematical notation) relation below describes just this behaviour:

$$(\mathbf{prog}\ n{:}\mathbb{N}\ .\ n := n)\ \upharpoonright\ (\ n{:}\mathbb{N}\ \mid\ n=0\ )\ \cup$$
$$(\mathbf{prog}\ n{:}\mathbb{N}\ .\ n := n-1)$$

The domains of its two component functions are disjoint, so it is in fact a function. It maps nonzero n to n-1 and maps zero to zero.

We can write the first of the unioned functions in a more compact way in AAL using the "guarded" form of statement.

```
prog n:N . n=0 → n := n
```

In general, if St is an AAL statement and P is a (mathematical) predicate, then the AAL program:

```
prog x:X . P → St
```

means the (now hybrid) function:

```
(prog x:X . St) ↑ ( x:X | P )
```

Even if we use this new notation for restriction, our function is still expressed in hybrid form, namely:

```
(prog n:N . n=0 → n := n) ∪
(prog n:N . n := n-1)
```

We would like to have an AAL form for taking the union of two programs.

In general, if St1 and St2 are AAL statements, then the AAL program:

```
prog x:X . St1 [] St2
```

means the same as the hybrid program:

```
(prog x:X . St1) ∪ (prog x:X . St2)
```

Sometimes, to make it clear that there is more than one possible "control path" within the statement St we will write:

```
if St fi
```

This means the same as St.

How do we say "do nothing?". The AAL program:

```
prog x:X . skip
```

means the same as the abstract program

```
π x:X . x'=x
```

that is

```
(id X)
```

At last we can write our abstract "exception proof" predecessor program entirely in AAL, (program Z):

```
prog n:N .
  if
    n := n-1
  []
    n=0 → skip
  fi
```

Healthiness of Alternations

If we try think of the AAL notation as a genuine programming language then we notice something rather strange about program 2: there is an alternation clause, one of whose arms is not "guarded"! As mathematicians (at least temporarily) this doesn't bother us; after all, AAL is just a notation for describing mathematical relations. But as (potentially) implementers, we might quake in our boots (or foam at the mouth) at the prospect of being asked to implement a programming language with assignments in which "failure" of a statement causes the system to "backtrack" to the last [] *undoing all assignments on the way!*

In order to get around this difficulty we shall impose a **healthiness constraint** on alternation statements. The rule will be that *all statements combined with* [] *must be guarded;* that is, alternatives should take the form:

$$P_1 \rightarrow St_1$$
$$[]$$
$$P_2 \rightarrow St_2$$
$$[]$$
$$\vdots$$
$$[]$$
$$P_n \rightarrow St_n$$

Moreover, we should prove for each of the guarded statements,

$$P_i \rightarrow St_i$$

that the guard is stronger than the "weakest precondition" of the statement, *ie*

$$(x{:}X \mid P_i) \subseteq \mathrm{dom}(\mathbf{prog}\ x{:}X\ .\ ST_i)$$

Finally, for an alternative statement to be "deterministic", the "guard sets" must be proven to be disjoint (this is a sufficient not a necessary condition, but we will voluntarily burden ourselves with it).

Iteration

If St is an AAL statement, then the AAL program:

prog x:X . **do** St **od**

means the same as the hybrid program:

repeat(**prog** x:X . St)

Our theory of iteration (in which **repeat** is defined) tells us that if we are to prove anything about such a loop then

prog x:X . St

had better denote a (strictly) partial decreasing function.

In general then the Statement of a loop will be an alternation of several guarded statements. For example:

```
prog n: N .
  do
     n < 3 → n := n+1
  ]
     n > 4 → n := n-1
  od
```

(Which, no matter what the value of n has originally, leaves it either 3 or 4 on "termination").

Sequencing
If St1 and St2 are statements, then the AAL program:

$$\text{prog } x{:}X \;.\; St_1 \;;\; St_2$$

means the hybrid program:

$$(\text{ prog } x{:}X \;.\; St_1 \;) \;;\; (\text{ prog } x{:}X \;.\; St_2 \;)$$

Generalisation to Several Variables
For the most part the generalisation of the above definitions to programs of many variables is completely straightforward. The only exception to this is the *assignment statement*, which we now consider. The meaning of the AAL program:

$$\text{prog } v_1{:}T_1 \;\ldots\; v_n{:}T_n \;.\; v_j := E$$

-- where E is a term in which v1 ... vn appear free -- is the function:

$$\pi\; v_1{:}T_1 \;\ldots\; v_n{:}T_n \;.$$

$$v_1' = v_1;$$
$$\vdots$$
$$v_j' = E;$$
$$\vdots$$
$$v_n' = v_n$$

In other words, the function which leaves all variables unchanged except for the one on the left-hand side of the assignment sign.

It is convenient to have a notation for *simultaneous assignment*, exemplified by:

$$\text{prog } v_1{:}T_1 \;\ldots\; v_n{:}T_n \;.\; v_i, v_j, \ldots v_k := E_1, E_2, \ldots E_3$$

$$\text{prog } v_1{:}T_1 \;\ldots\; v_n{:}T_n \;.$$

$$v_i := E_1,$$
$$v_j := E_2,$$
$$\ldots$$
$$v_k := E_3$$

which both mean the function

$$\lambda\ v_1{:}T_1 \ldots v_n{:}T_n\ .$$
$$v_i' = E_1;$$
$$v_j' = E_2;$$
$$v_k' = E_3;$$
$$v_l' = V_l$$

The final predicate is intended to convey that the values of all variables other than vi vj vk remain the same.

Notice that the two programs

```
prog a,b:N . a := b, b := a
```

```
prog a,b:N . a := b; b := a
```

mean different functions; the first means:

```
λ a,b:N . (b, a)
```

whilst the second means:

```
λ a,b:N . (b, b)
```

Assignment to Mappings

In the same spirit, we introduce a form of assignment to variables of a special type, namely mappings. Within the scope of a variable

```
M: X ↠ Y
```

(a mapping from X to Y) the statement:

```
M E1 := E2
```

means the same as the statement:

```
M := M ⊕ ( E1 ↦ E2 )
```

Naturally the terms E1 and E2 had better be well-typed!

Declarations

Next we define an AAL construct which introduces "initialised" variables. If St is a statement and E is a term of type Y then the AAL program:

```
prog x:X .
  let y:Y = E in St
```

means the function

```
(λ x:X . (x, E))  ; ( prog x:X; y:Y . St ) ; (λ x:X; y:Y . x)
```

Similarly, the AAL program:

```
prog x:X .
  let y:Y = E1 and z:Z = E2 in St
```

means the function

```
(λ x:X . (x, E1, E2) ;
 ( prog x:X; y:Y; z:Z . St ) ;
(λ x:X; y:Y; z:Z . x)
```

This concludes our definition of the Abstract Algorithmic Language.

Summary

We have given rules which allow AAL "programs" to be transformed into mathematical relations (functions). In principle it is enough to refer the person who wishes to reason about AAL programs to this (albeit only semi-formal) semantics, and suggest that any reasoning be done using only the rules of mathematical language. In practice it will prove useful to derive some proof rules for AAL constructs and combining forms from their mathematical translations. Indeed the constructs we have described so far were chosen precisely because the derived proof rules for them are simple to understand and work with. In the next section of this document we will present derived proof rules for AAL which will be familiar to students of Dijkstra's or Hoare's systems of reasoning about programs. There is nothing arbitrary or synthetic about the rules we will present, they are simply consequences of (and proved using) the definitions as mathematical functions of the AAL constructs.

Z Reference Card

1. Definitions.

LHS ≙ RHS Definition by syntactic equivalence.

2. Logical symbols.

P ∧ Q	Conjunction: "P and Q".
P ∨ Q	Disjunction: "P or Q".
P ⟹ Q	Implication: "P implies Q" or "if P then Q".
¬ P	Negation: "not P".
∀	Universal quantification: "for all . . .".
∃	Existential quantification: "there exists . . .".

3. Sets.

∈	Set membership
⊆	Set inclusion: S ⊆ T ≙ (∀ x: S. x ∈ T).
⊂	Strict set inclusion: S ⊂ T ≙ S ⊆ T ∧ S ≠ T.

(sig | pred . term) The set of term such that pred given sig:
x ∈ (sig | pred . term) ⟺ (∃ sig | pred . x = term)

∪	Set union: S ∪ T ≙ (x: X | x ∈ S ∨ x ∈ T).
∩	Set intersection: S ∩ T ≙ (x: X | x ∈ S ∧ x ∈ T).
-	Set difference: S - T ≙ (x: X | x ∈ S ∧ ¬ x ∈ T).
()	The empty set
(a, b)	Ordered pair.
×	Cartesian product: X × Y ≙ (x: X; y: Y).
ℙ	Powerset: ℙ X is the set of all subsets of X.
𝔽	Set of finite subsets: 𝔽 X ≙ (S: ℙ X | S is finite).
#	Size (number of elements) of a finite set.

∪, ∩ Generalised set union and intersection: for SS: ℙ (ℙ X),
∪ SS ≙ (x: X | (∃ S: SS . x ∈ S)),
∩ SS ≙ (x: X | (∀ S: SS . x ∈ S)).

disjoint Pairwise disjointness: for SS: ℙ (ℙ X),
disjoint SS ≙ ∀ S, T: SS. S ∩ T ≠ () ⟹ S = T.

4. Relations and Functions.

A ↔ B	The set of relations from A to B: A ↔ B ≙ ℙ (A × B).
A ⇸ B	The set of partial functions from A to B: A ⇸ B ≙ (f: A ↔ B | (∀ a: A; b, b': B . a f b ∧ a f b' ⟹ b = b')).
A → B	The set of total functions from A to B: A → B ≙ (f: A ⇸ B | (∀ a: A . ∃ b: B . a f b)).

$\{ a \mapsto b, c \mapsto d, \dots \}$	The relation $\{ (a, b), (c, d), \dots \}$ relating a and b, c and d . . .
λ	Lambda-abstraction: $\lambda\ a: A \mid pred \,.\, term \ \hat{=}\ \{ a: A \mid pred \,.\, (a, term) \}$
$f\ x$	The function f applied to x
dom	The domain of a relation or function: for $R: A \leftrightarrow B$, $\mathrm{dom}\ R \ \hat{=}\ \{ a: A \mid (\exists\, b: B \,.\, a\ R\ b) \}$.
ran	The range of a relation or function: for $R: A \leftrightarrow B$, $\mathrm{ran}\ R \ \hat{=}\ \{ b: B \mid (\exists\, a: A \,.\, a\ R\ b) \}$.
$\circ$	Relational or functional composition: for $R: A \leftrightarrow B$; $S: B \leftrightarrow C$, $S \circ R \ \hat{=}\ \{ a: A; c: C \mid (\exists\, b: B \,.\, a\ R\ b \wedge b\ S\ c) \}$.
$;$	Forward relational (or functional) composition: $R\ ;\ S \ \hat{=}\ S \circ R$.
id	Identity function: $\mathrm{id}\ A \ \hat{=}\ \lambda\ a: A.\ a$.
R^{-1}	The inverse of relation R: for $R: A \leftrightarrow B$, $R^{-1} \ \hat{=}\ \{ b: B; a: A \mid a\ R\ b \}$.
f^{k}	The relation (or function) f composed with itself k times: for $f: A \leftrightarrow A$, $f^0 = \mathrm{id}\ A,\ f^1 = f,\ f^2 = f \circ f,\ f^3 = f \circ f \circ f, \dots$
$[\ ]$	Image: for $R: A \leftrightarrow B$; $S: \mathbb{P}\ A$, $R[S] \ \hat{=}\ \{ b: B \mid (\exists\, a: S \,.\, a\ R\ b) \}$
$\upharpoonleft$	Domain restriction: for $R: A \leftrightarrow B$; $S: \mathbb{P}\ A$, $R \upharpoonleft S \ \hat{=}\ \{ a: A; b: B \mid a\ R\ b \wedge a \in S \}$.
$\setminus$	Domain co-restriction: $R \setminus S \ \hat{=}\ R \upharpoonleft (A - S)$.
$\downharpoonright$	Range restriction: for $R: A \leftrightarrow B$; $T: \mathbb{P}\ B$, $R \downharpoonright T \ \hat{=}\ \{ a: A; b: B \mid a\ R\ b \wedge b \in T \}$.
$/$	Range co-restriction: $R\ /\ T \ \hat{=}\ R \downharpoonright (B - T)$.
$\oplus$	Relational or functional overriding: for $f, g: A \leftrightarrow B$, $f \oplus g \ \hat{=}\ (f \setminus \mathrm{dom}\ g) \cup g$.

5. Numbers.

$\mathbb{N}$	The set of natural numbers (non-negative integers)
succ	Successor function: $\mathrm{succ} \ \hat{=}\ \lambda\ n: \mathbb{N}\ .\ n+1$.
m..n	The set of natural numbers between m and n inclusive: $m..n \ \hat{=}\ \{ k: \mathbb{N} \mid m \leq k \leq n \}$.
max(m, n)	The greater of m and n: $m \geq n \Rightarrow \max(m, n) = m$; $m \leq n \Rightarrow \max(m, n) = n$

6. Sequences.

seq A	The set of sequences whose elements are drawn from A: $\mathrm{seq}\ A \ \hat{=}\ \{ s: \mathbb{N} \rightarrow A \mid (\exists\, n: \mathbb{N} \,.\, \mathrm{dom}\ s = 1..n) \}$
#s	The length of sequence s: $\mathrm{dom}\ s = 1..\#s$.
$\langle\rangle$	The empty sequence $\{\}$.
$\langle a, b, c \rangle$	The sequence $\{ 1 \mapsto a, 2 \mapsto b, 3 \mapsto c \}$, etc.

head	The first element of a sequence.
last	The last element of a sequence.
tail	All but the first element of a sequence.
front	All but the last element of a sequence.
	For s: seq A | s ≠ <>,
	head s ≙ s (1),
	last s ≙ s (#s),
	tail s ≙ succ ; (s \ (1)),
	front s ≙ s \ (#s).
^	Adding new head and tail: for s: seq A; x: A,
	x ^ s ≙ (1 ↦ x) ∪ succ⁻¹ ; s,
	s ^ x ≙ s ∪ (succ #s ↦ x).
*	Concatenation: <> * t = t, (x ^ s) * t = x ^ (s * t).
rev	Reversing: rev <> = <>, rev (x ^ s) = (rev s) ^ x.

7. Schema Notation.

[For details see "Schemas in Z"].

Schema definition:

SCH
a: A
b: B
axioms

Use in signatures after ∀, ∃, λ, (...), etc.:

(SCH \| predicate) ≙ (a: A; b: B \| axioms ∧ predicate).

∀ SCH . predicate ≙ ∀ a: A; b: B \| axioms ⇒ predicate.

tuple	The tuple formed of a schema's variables: tuple SCH ≙ (a, b).
pred	The predicate part of a schema: pred SCH ≙ axioms.
[new/old]	Renaming of components.
S', S_2	Decoration; systematic renaming.

Use in definition of other schemas: inclusion, extension.

∧, ∨, ¬, etc.	Logical operations.
\	Hiding.
↾	Projection.
▷	Relative consistency.
;	Relational composition.
dom, ran	Domain and range.
( )	Application.
⊕	Overriding.

END

DTIC

11-86