THE UNIVERSITY *of* EDINBURGH

SEMANTICS, IMPLEMENTATION AND PRAGMATICS OF CLEAR, A PROGRAM SPECIFICATION LANGUAGE

by

Donald Theodore Sannella

Doctor of Philosophy
University of Edinburgh
1982

ABSTRACT

Specifications are necessary for communicating decisions and intentions and for documenting results at many stages of the program development process. Informal specifications are typically used today, but they are imprecise and often ambiguous. Formal specifications are precise and exact but are more difficult to write and understand. We present work aimed toward enabling the practical use of formal specifications in program development, concentrating on the Clear language for structured algebraic specification.

Two different but equivalent denotational semantics for Clear are given. One is a version of a semantics due to Burstall and Goguen with a few corrections, in which the category-theoretic notion of a colimit is used to define Clear's structuring operations independently of the underlying 'institution' (logical formalism). The other semantics defines the same operations by means of straightforward set-theoretic constructions; it is not institution-independent but it can be modified to handle all institutions of apparent interest.

Both versions of the semantics have been implemented. The set-theoretic implementation is by far the more useful of the two, and includes a parser and typechecker. An implementation is useful for detecting syntax and type errors in specifications, and can be used as a front end for systems which manipulate specifications. Several large specifications which have been processed by the set-theoretic implementation are presented.

A semi-automatic theorem prover for Clear built on top of the Edinburgh LCF system is described. It takes advantage of the structure of Clear specifications to restrict the available information to that which seems relevant to proving the theorem at hand. If the system is unable to prove a theorem automatically the user can attempt the proof interactively using the high-level primitives and inference rules provided.

We lay a theoretical foundation for the use of Clear in systematic program development by investigating a new notion of the implementation of a specification by a lower-level specification. This notion extends to handle parameterised specifications. We show that this implementation relation is transitive and commutes with Clear's structuring operations under certain conditions. This means that a large specification can be refined to a program in a gradual and modular fashion, where the correctness of the individual refinements guarantees the correctness of the resulting program.

CONTENTS

Acknowledgements

I would like most of all to express my gratitude to my supervisor Rod Burstall for many inspiring talks, some gentle prodding, constant guidance and encouragement, and for patiently reading multiple drafts of this thesis. I am also grateful to Robin Milner for acting as supervisor when Rod was away.

My thanks to David Rydeheard for category-theoretic expertise (both electronic and otherwise) and collaboration, to Alan Mycroft for help with the stubborn DEC-10, to Martin Wirsing for mighty efforts in the course of our work on implementations and generous hospitality, to David MacQueen for the HOPE system, to all of the above and Luca Cardelli, Wei Li, Brian Monahan and Oliver Schoett for interesting discussions and arguments, and to Monika and friends for moral support and distraction.

This work was supported by a studentship from the University of Edinburgh, and by the Science and Engineering Research Council.

Statement

This thesis builds upon previous work on the Clear specification language, invented by Burstall and Goguen [1977]. The description of HOPE in section I.2 and appendix 1 is a revised version of [Burstall, MacQueen and Sannella 1980]. The definitions in chapter II and the semantic equations of section III.4 are adapted from [Burstall and Goguen 1980]. The semantics described in chapter V is a corrected version of [Burstall and Goguen 1980]; the implementation was a joint effort with David Rydeheard, building on a improved version of a program to compute colimits due to Burstall [1980]. Section VI.5 is abridged from [MacQueen and Sannella 1982]. Chapter VII is adapted from [Sannella and Wirsing 1982].

The remainder is my own work and this thesis was composed by myself. Chapter II and most of chapter III have been published in a different form as [Sannella 1981].

INTRODUCTION

Specifications play a part in every phase of program development. The construction of a program cannot commence without some kind of specification of what it is intended to do. Every program is written to solve some problem, and the problem must be known before work on the program is begun. In the course of a large programming project specifications serve as a means of communication. Each programmer is responsible for a certain component of the program which may use facilities provided by several 'foreign' components. Precise specifications of those components are required before any program which relies on them can be written. These specifications are produced during the design phase when a way of decomposing the task is decided upon and the component subtasks recorded. It is important that the specifications of the components avoid giving away unnecessary details of the implementation. If nobody is able to depend on the idiosyncratic features of a particular solution to a subtask, then another solution may be easily substituted without affecting the correctness of the program.

Once a program has been written some attempt is normally made to check that it is correct. This check may be an informal test of the program on a few values, or a formal proof of correctness. In any case, some specification is needed to compare the program against; a program is only correct with respect to some specification of its expected behaviour. Finally, documentation is required, both for the use of the customer and to aid the future maintenance and modification of the program. This documentation is also a specification of the program.

Up to now the word "specification" has been used in the broadest possible sense. Every means of describing a program and its behaviour is included, from informal English documentation to a precise description in a formal specification language. Even the text of a program itself is a specification, as is a vague idea in the head of a programmer. Some kinds of specifications are useless for certain purposes; for example, vague ideas are only useful if

the roles of customer, designer and programmer are all played by the same person, and even then they are not enough for a formal proof of correctness. The text of a program is not usually considered to be a good specification because it is too detailed to be easily understood and is not sufficiently abstract for some uses (e.g. specification of the modules in a large program, as already remarked), but a very simple and straightforward program may be useful as a specification of an equivalent program which must be complex in order to be efficient.

Informal specifications suffer from imprecision. This is a serious fault because (for example) the use of a specification as the basis of a formal proof of program correctness demands precision, and heavy penalties can be incurred if a specification used as a means of communication is misunderstood. Specifications written in a formal language are necessarily precise, since the meaning of each specification is given by the semantics of the specification language and accordingly there is no room for confusion. Various formal specification techniques and languages exist; Liskov and Berzins [1977] survey those available in 1977. A great deal of attention has recently been devoted to algebraic methods of specification, which seem to yield specifications which are both concise and easy to understand. Prominent in this area is work by Guttag and his colleagues [Guttag, Horowitz and Musser 1978] and by the ADJ group [Goguen, Thatcher and Wagner 1978], of which the latter is the most mathematically rigorous. In this framework, a specification consists of a signature -- a set of sorts (kinds of data) and some operators (for constructing and manipulating data) -- together with axioms (normally equations) describing constraints on the results produced by operators. Such a specification is called a theory, and it describes a set of algebras (a set of data objects for each sort, and a function on those sets for each operator), where each algebra in the set is a model of the theory (it satisfies the axioms). Programs can be considered to be algebras, so all programs satisfying a theory are in its set of models.

Most workers in algebraic specification concentrate on the specification of abstract data types, for which the method is

particularly well suited. Although an algebraic specification could be written for a large system, such a theory would be impossible to understand because it would contain so many axioms. The value of a specification depends on the ease with which it was written and can be understood; a large number of pages full of axioms are not of much use to anybody.

The Clear specification language was invented by Burstall and Goguen [1977] to combat just this problem. Clear is a language for writing structured algebraic specifications; that is, it provides facilities for combining small theories in various ways to make large theories. With a tool such as this, the specification of a large real-world system can be built from small, easy to understand and (in many cases) reusable bits.

An obvious way to combine theories is to simply add them together, giving a theory which includes the sorts, operators and axioms of each component. Clear also provides a facility for parameterising a theory to give a theory procedure, which can be applied to various different theories to produce new theories which have been systematically enriched in some way. A typical example is a parameterised theory of sorting, which would produce a theory of sorting lists of numbers if applied to the theory of natural numbers together with the usual $\leq$ order relation. An operation called data can be applied when adding new sorts and operators to a theory; this constrains the set of models to a small number of 'best' ones. Finally, some of the operators and sorts of a theory can be 'hidden' to yield a less elaborate theory. Clear is described in detail in chapter I.

With an intuitive understanding of Clear it is possible to begin to write structured specifications which can be used in the development of programs. Clear should be better than most specification languages for this purpose because specifications have structure, and the structure of a program will normally be similar to the structure of the specification from which it was developed. But in order to rigorously prove that a program implements a Clear specification or to build a system incorporating Clear to aid in any phase of program development, it is necessary to have a formal

semantics which gives the precise meaning of any Clear specification (i.e. the theory described by the specification). A language which has a formal syntax but no semantics gives an illusion of precision but none of its benefits.

A formal semantics of Clear is given in chapter III, following definitions of the underlying mathematical entities in chapter II. This semantics defines the meanings of Clear's theory-building operations using simple set-theoretic constructions. A denotational semantics is then used to attach a syntax to these operations and to provide for an environment of named theories. An earlier semantics given by Burstall and Goguen [1980] relies heavily on a number of ideas from category theory to define the meanings of Clear's theory-building operations. This semantics is described in chapter V; this is the only chapter which requires any knowledge of category theory, and it is not necessary for the sequel.

Why is it necessary to give two separate versions of the semantics? Surely one version is sufficient to define the meaning of Clear. The answer is that although both versions of the semantics are equivalent, each has its advantages over the other. The category-theoretic semantics was developed at the same time as the Clear language itself. The requirement that new features be expressed using simple concepts of category theory acted as a powerful filter for ideas, screening out some bad ideas and suggesting non-obvious generalisations of others. Moreover, the category-theoretic definitions are very elegant to those who understand them. The advantage of the set-theoretic semantics is that it is concrete and easy to understand, and is therefore more useful for practical applications. The category-theoretic semantics abstracts away from any particular definitions of the fundamental elements of Clear (signatures, axioms and models) using the notion of an institution, defining all at once the semantics of a large class of Clear-like languages. But at the end of chapter III it is shown that the set-theoretic semantics appears to be capable of straightforward modifications to cover all cases of interest.

An important step on the way to the practical use of formal specifications in program development is an implementation of the

specification language. But what does it mean to implement a specification language? It is helpful to first consider the relation of semantics to implementation in a more familiar context, that of a programming language.

The denotational semantics of a programming language describes a mapping between the syntax of the language (expressions, statements, programs) and the mathematical objects they represent. In a typical language, an expression maps to a function from states to numbers (or to lists, or to some other domain of values); a statement maps to a state-transforming function; and a program maps to a function taking (for example) an input file to an output file. The denotation of a program tells what the answer will be for any input. The implementation of a programming language transforms a string of characters representing a program into the function denoted by that program.

The denotation of a Clear specification is a theory. This is still only a specification; it specifies a set of algebras. The transformation from a character string (representing a specification) to a theory is complex but mechanical. This is what an implementation of Clear does. Going from a theory to a model is a much more formidable task -- this is the problem of program synthesis.

It is easy to make mistakes when writing specifications in Clear or in any other specification language, just as it is easy to make errors when writing programs. An implementation of Clear -- a parser together with an implementation of (a version of) the semantics and a typechecker -- could be used to check specifications for syntactic and semantic errors. Such an implementation could also act as a front end to any system which requires specifications as input (such as a program verification system). An implementation of Clear using the set-theoretic version of its semantics is discussed in chapter IV along with some specifications it has been used to process. An implementation of the category-theoretic semantics (without parser or typechecker) is discussed in chapter V. A comparison of these implementations exposes another advantage of the set-theoretic semantics -- its implementation is by far the

faster of the two.

A practical implementation of Clear opens the door for systems to aid program development using Clear specifications. Already mentioned was the possibility of a system for verifying programs; another possibility is a high-level programming system like the one envisioned by Winograd [1979], which is essentially a sophisticated database containing the components of a large software project and their specifications. A handy facility to begin with (and an essential prerequisite for the construction of almost any system making serious use of specifications) would be a system for proving that a theorem follows from the axioms of a theory. A theorem prover is in fact required by the Clear implementation to check the semantic validity of specifications. In chapter VI a semi-automatic theorem prover for Clear is described. This system takes advantage of the structure of Clear specifications to restrict the information available at any time (axioms and previously proved theorems) to that which is relevant to the theorem at hand. This is an important feature, for theorem provers easily get irretrievably bogged down in exploring the large number of blind alleys made available by an overabundance of (mostly irrelevant) information.

Some Clear specifications are actually executable; a sufficient condition is that all data be anarchic (no axioms on data 'constructors') and that the axioms which define other operators be equations with simple left-hand sides, enabling their use as rewrite rules. This executable sublanguage is in fact HOPE [Burstall, MacQueen and Sannella 1980], with slightly different notation (except that HOPE has no equivalent to Clear's theory procedures). Call specifications of this kind programs. Now, a program can be evolved from a specification entirely in Clear by repeatedly rewriting (refining) the theories in the specification until the entire specification is in the executable sublanguage. This will normally involve the introduction of auxiliary functions, particular data representations and so on. This approach to program development is related to the well-known programming discipline of stepwise refinement advocated by Wirth [1971] and Dijkstra [1972] in

which the specifications are nonexecutable 'abstract programs'. In chapter VII a theoretical foundation is laid for the use of Clear in systematic program development. An adequate notion of the implementation of one theory by another 'lower level' theory is first established; a refinement is only correct if the new theory is an implementation of the old. Unlike most previous notions, this generalises to handle parameterised theories as well as loose theories (having an assortment of different models). It is then shown that implementations of several theories can be put together in the same way as the theories themselves are put together, the result being an implementation of the composite theory. This allows a large specification to be refined in a modular fashion, one theory at a time.

Systems have been constructed which support systematic program development in a manner similar to that just discussed. Examples are ZAP [Feather 1982] and CIP [Bauer et al 1981]. In these systems the programmer provides the insight, deciding which direction the development will take, while the system performs the routine clerical work and checks that the programmer's decisions are valid. Fully automatic program synthesis is also possible (for small examples) as demonstrated by Manna and Waldinger [1980,1981]. A feature of each of these systems is that the finished program is guaranteed to satisfy the original specification, since the system checks every step in its development. A similar but more ambitious system called CAT [Goguen and Burstall 1980] has been proposed to support systematic program development using Clear. The results in chapter VII are a first step toward the implementation of CAT.

It remains to be seen if writing a specification and carefully refining it step by step to a program is easier than simply writing a correct program in the first place. However, construction of correct programs is well-known to be a very difficult endeavor. And although some have claimed that writing specifications is more difficult than writing programs, experience with Clear indicates that the main barrier to easy specification is the computer scientist's natural inclination towards algorithms rather than descriptions. Precision and formality are crucial (as in a

programming language) but the most important feature of a specification language like Clear is that it permits problems to be described in a natural way.

CHAPTER ONE

CLEAR AND HOPE

This chapter is devoted to a brief review of Clear and HOPE, two languages which figure greatly in the research reported in subsequent chapters. Although they have been discussed in more detail elsewhere, an outline of their features is given here in order to make this work self-contained.

Clear is a specification language which is particularly suitable for specifying large programs. It provides facilities for building large theories in a structured fashion from small bits. Constructing and understanding large specifications is made much easier by this approach, since the small component theories may be contemplated in isolation. A brief discussion of theories and their models in section 1.1 is followed by an informal presentation of the theory-building operations of Clear. The formal semantics of these operations will be given in later chapters. An important feature of Clear is that the definitions of the theory-building operations are independent of the precise nature of the theories themselves; any notion of signature, axiom, algebra and satisfaction will do (provided they meet certain basic requirements). In section 1.3 examples of some different and possibly useful kinds of theories are given. This is followed by an example of a small but complete Clear specification in section 1.4. Finally, Clear is compared briefly with some other specification approaches.

HOPE is a very high-level applicative programming language which was used as an implementation tool for most of the programs described here. It has the advantage of being sufficiently close to the language of denotational semantics that semantic definitions can be quickly and easily translated into an executable form. This fact enabled the construction of the programs discussed in chapters IV and V. Although HOPE is not so close to the language of some other branches of mathematics, it contains high-level features which permit the relatively painless expression of definitions and constructions from category theory as described by Burstall [1980]

and Rydeheard [1981]; this provides the foundation for the program in chapter V. HOPE is not so different from ML (see [Gordon, Milner and Wadsworth 1979]).

HOPE (without polymorphism) can be considered as a notational variant of a subset of Clear. This is very convenient for the work on stepwise implementation of specifications in chapter VII. A refinement step takes a Clear specification to a 'lower level' Clear specification, with a HOPE program as the eventual goal. Thus the problem of translation into the target language can be neatly ignored, and full attention can be devoted to the more interesting problems of developing programs from specifications.

A third section of this chapter might have been devoted to a brief description of Edinburgh LCF, an interactive theorem-proving system upon which the Clear theorem prover described in chapter VI is built. But since the remaining chapters are entirely independent of LCF, the description has been relegated to the beginning of that chapter.

1. Clear

This is a brief and non-technical account of Clear as a specification language. It is intended to give the reader an idea of nearly all the features of Clear and to convince him with an example that the language can be put to use. The utility of specification languages in general and the advantages of Clear over other specification languages have already been detailed in the introduction. More detailed informal descriptions of Clear appear in [Burstall and Goguen 1977] and [Burstall and Goguen 1981]; see also chapter IV for a few more examples.

1.1. Theories and their models

Clear is a language for describing theories; in turn, each theory describes (or denotes) a class of algebras. A theory is a set of sorts (names of data types), a set of (possibly nullary) operators for constructing and manipulating data, and a set of axioms (in the form of equations) describing constraints on the results produced by operators. The sorts and operators alone are called the signature. For example, here is a theory of truth values:

```
const Bool =
  theory
    sorts bool
    opns true, false : bool
         not : bool -> bool
    eqns not(true) = false
         not(not(p)) = p    endth
```

The equations are implicitly universally quantified over all variables; the equations here would be more properly written

```
all p:bool. not(not(p)) = p
```

and so on. The examples in the sequel will leave variable declarations out of equations in the understanding that they could easily be supplied by a mechanical typechecker.

An algebra is a family of named carriers (sets of data values) and some named total functions on those carriers. An algebra is a model of a theory if it satisfies all the equations in the theory

for any assignment of values to variables; this is provided that the names of the carriers and functions in the algebra match the names in the signature of the theory, of course.

Here are some models of the theory Bool:

M_1 : bool={0,1}; true=1; false=0; not(0)=1, not(1)=0
M_2 : bool={no,yes}; true=no; false=yes; not(no)=yes, not(yes)=no
M_3 : bool={42}; true=42; false=42; not(42)=42

But something is wrong; we do not want M_3 to be a model for Bool, yet it does satisfy all the necessary equations. We need some way of excluding models like M_3.

The problem with M_3 is that it satisfies too many equations, including ones like true = false which are not in Bool. We really want as models of Bool only those algebras which satisfy exactly the equations of Bool (and all of the equations which these entail), and no others. In addition, we want each element in the carrier to be the value of some (ground) term; this avoids models with useless extra elements. We can rewrite Bool to indicate that this is the class of models we want, using Clear's data operation:

```
const Bool =
  theory
    data sorts bool
         opns true, false : bool
              not : bool -> bool
         eqns not(true) = false
              not(not(p)) = p    endth
```

The new Bool has the class of models we want (including M_1 and M_2, but not M_3). These are called the initial models, and they have the property that any two initial models are the same up to isomorphism (i.e. except for renaming of data values -- compare M_1 and M_2). As pointed out by ADJ in [Goguen, Thatcher and Wagner 1978], the notion of an isomorphism class of algebras captures precisely the meaning of the word "abstract" in "abstract data type" -- we are not committed to any particular representation of data, but only to the behaviour shared by all members of the class. Furthermore, the isomorphism class containing the initial models of a theory seems to be the one we want, although this position is not

universally accepted (see for example [Wand 1979]).

Initial models seem so great that it may be hard to think of an example where the full class of models is appropriate. But such theories do exist; see Equiv in the next section, for example. Since sometimes we want all models and sometimes we want only initial models, the data operation is provided to allow the two cases to be distinguished. See section II.5 for a more detailed discussion of this aspect of Clear.

The data operation does a little bit more than specify that we want the class of initial models. It also adds an extra operator, an equality predicate ==:s,s->bool for each 'data' sort s. For any pair of terms p and q, p==q = true if and only if p = q is entailed by the equations (i.e. it holds for the initial models). Note that the data operation can therefore only be used in theories which include Bool, but this is not really much of a restriction.

1.2. Theory-building operations

Bool and its models (in the last section) were easy to understand, and similar little theories like natural numbers, sets of numbers, and stacks and arrays of truth values present no difficulties. But what about a theory to specify a compiler for a programming language like Pascal? This would have many sorts, hundreds of operators and perhaps a thousand axioms.

Clear provides a set of simple theory-building operations which allow a large theory like this to be built out of many small and comprehensible component theories. For example, the theory of a compiler for Pascal might be built from separate theories of the semantics of Pascal and the semantics of the target machine:

```
const Pascal_compiler =
  enrich Pascal_semantics + VAX_semantics by
    opns compile : pascal_program -> VAX_program
    eqns VAX_meaning(compile(p)) = pascal_meaning(p)    enden
```

The theories Pascal_semantics and VAX_semantics are in turn built separately from many smaller theories. But the difficulty of understanding the specification has already been roughly halved by

this simple decomposition, since Pascal_semantics may be constructed and contemplated entirely independently of VAX_semantics (although they will share some common subtheories like Bool and Nat).

Enrich

A theory can be enriched by some new sorts, operators and/or axioms. The new material is just added to the existing theory. For example, we could add some boolean operators to Bool:

```
const Boolopns =
  enrich Bool by
    opns and, or, --> : bool,bool -> bool
    eqns p and true = p
         p and false = false
         p or true = true
         p or false = p
         p --> q = not(p and not(q))    enden
```

Or, we could add natural numbers:

```
const Nat =
  enrich Bool by
    data sorts nat
         opns 0 : nat
              succ : nat -> nat
              + : nat,nat -> nat
         eqns 0 + m = m
              succ(n) + m = succ(n + m)    enden
```

Note that infixed operators like or and + are allowed. Also note that names (like Bool, Boolopns and Nat) can be given to theories using the notation const Name = ... (const means constant). For local declarations the syntax let Name = ... in ... is used (see the example in section 1.4).

The data operation is associated with an enrichment as in Nat above and not just with a theory. In fact, data does not in general restrict to initial models but to models which are free extensions of the models of the theory being enriched; see section II.5 for details. Observe that

theory ... endth

as used in the last section is equivalent to

enrich Empty by ... enden

where Empty is the theory with no sorts or operators.

Here is an example of a theory in which we do not want to use data:

```
const Equiv =
  enrich Boolopns by
    sorts element
    opns ■ : element,element -> bool
    eqns m ■ m = true
         m ■ n = n ■ m
         m ■ n and n ■ p --> m ■ p = true    enden
```

If we use data for this enrichment then we get only trivial models (apart from the portion associated with the sort and operators of Boolopns); the carrier associated with the sort element is empty, because there are no ground terms of sort element. But this is not because the specification is silly; it is just not very specific. It is intended to specify the set of algebras with one sort and an equivalence relation. Equiv is called a loose theory, since its models do not form an isomorphism class. It is also called a non-data theory because it contains a sort which was added in a non-data enrichment.

It is important to distinguish between the very similar notions of loose and non-data theories. Non-data implies loose (except in the case of a theory which is unsatisfiable or has only trivial models) but not vice versa. Here is a theory which is loose but is not non-data:

```
const Natx =
  enrich Natord by
    opns x : nat
    eqns x ≤ 2 = true    enden
```

(Natord is Nat with an order relation, as given below.) This is a simple example of the way that Clear can be used to write specifications which are purposefully vague so as to allow some freedom to the implementor.

Combine

The combination of two theories is (roughly speaking) just the

union of the sorts, operators and axioms. For example, the combination of Boolopns and Nat (written Boolopns + Nat) has the following sorts, operators and axioms:

```
sorts bool, nat
opns true, false : bool
     not : bool -> bool
     and, or, --> : bool,bool -> bool
     0 : nat
     succ : nat -> nat
     + : nat,nat -> nat
eqns not(true) = false
     . . .
```

Note that we get only one copy of the sorts and operators of Bool, although Bool is included in both Boolopns and Nat (Bool is called a shared subtheory in this case). This is important; we do not want several kinds of truth values rattling around in a large specification (or several kinds of anything else, for that matter). But different (separately defined) operators with the same names are not combined; for example, if we add an operator 'and' to Nat

```
const Nat1 =
  enrich Nat by
    opns and : nat,nat -> nat
    eqns n and m = n + m    enden
```

then Boolopns + Nat1 will have two operators called and (and even Boolopns + Nat will have two == operators). If there are two sorts or operators with the same name there should be a way of distinguishing between them (although a typechecker can often determine the appropriate one); for this Clear provides the notation "== of Nat".

Derive

The derive operation is used to 'forget' some of the sorts and operators of a theory, possibly renaming those which remain. While enrich and combine build elaborate theories from simple components, derive takes a complex theory and reduces it to a more modest subtheory. This turns out to be necessary in cases where it is easier to define something by construction than by description; the construction is built using enrich and combine, and then the

irrelevant details can be forgotten using derive.

For example, suppose we have a theory of natural numbers with an order relation:

```
const Natord =
  enrich Nat by
    opns ≤ : nat,nat -> bool
    eqns 0 ≤ m = true
         succ(n) ≤ 0 = false
         succ(n) ≤ succ(m) = n ≤ m    enden
```

Then we can use this to construct a theory of characters with the usual lexical ordering:

```
const Char =
  derive sorts char
         opns 'A', ..., 'Z' : char
              ≤, == : char,char -> bool
    using Bool
    from Natord
    by char is nat,
       'A' is 0,
         . . .
       'Z' is 25    endde
```

Char inherits the order on numbers and the data equality, but the operators succ and + are forgotten, as well as all numbers greater than 25. Bool is a shared subtheory of Char and Natord. The correspondence between the signature of the result and the signature of the original theory is given by a signature morphism: char is nat, 'A' is 0, ... (This example assumes that the numbers 1-25 have been defined as operators in Natord; these definitions were omitted above but they are easy to add.) Sorts and operators which have the same name in both signatures may be omitted (≤ and == in this case).

It is very convenient to be able to specify an order relation on characters in this way; a direct specification would require hundreds of axioms. In some cases a direct specification is not even possible without 'hidden' operators (see [Thatcher, Wagner and Wright 1978] for an example).

Apply

In Clear procedures can be defined and applied, just as in a programming language (actually, more like functions in a programming language). But since Clear is a language for describing theories, the arguments and result of a procedure are theories.

Here is an example of a theory procedure (usually called a parameterised theory):

```
meta Ident =
  enrich Boolopns by
    sorts element
    opns eq : element,element -> bool
    eqns eq(n,n) = true
         eq(n,m) = eq(m,n)
         eq(n,m) and eq(m,p) --> eq(n,p) = true    enden


proc Set(X:Ident) =
  let Set0 =
    enrich X by
      data sorts set
           opns Ø : set
                singleton : element -> set
                U : set,set -> set
           eqns Ø U S = S
                S U S = S
                S U T = T U S
                S U (T U V) = (S U T) U V    enden in
  enrich Set0 by
    opns is_in : element,set -> bool
         choose : set -> element
    eqns a is_in Ø = false
         a is_in singleton(b) = eq(a,b)
         a is_in S U T = a is_in S or a is_in T
         choose(singleton(a) U S) is_in (singleton(a) U S) = true
                                                          enden
```

Ident is a metatheory; it describes a class of theories rather than a class of algebras. Ident describes those theories having at least one sort together with an operator which satisfies the laws for an equivalence relation on that sort. A metatheory will ordinarily be a non-data theory (as is Ident).

Ident is used to give the 'type' of the parameter for the procedure Set. The idea is that Set can be applied to any theory which matches Ident. Ident is called the metasort or requirement of

Set. The declaration of Set can use the formal parameter X and the sorts and operators of Ident. When Set is supplied with an appropriate actual parameter theory, it gives the theory of sets over the sort which matches element in Ident. For example

Set(Boolopns[element is bool, eq is ==])

gives the theory of sets of truth values and

Set(Nat[element is nat, eq is ==])

gives the theory of sets of natural numbers. Notice that a signature morphism (called the fitting morphism) must be provided to match the signature of Ident with the signature of the actual parameter. Of course, procedures may have more than one parameter if desired.

Metatheories are subtly different from ordinary constant theories; see section III.3 for details. Pragmatically, the difference is unimportant as long as metatheories are always used for giving the requirements of theory procedures, and for no other purpose.

Note that for any actual parameter A and fitting morphism ρ, Set(A[ρ]) will be a loose theory, even when A is not itself a loose theory. The choose operator is loosely specified as selecting an arbitrary element from a non-empty set. This is not allowed by most other notions of parameterised theory (see section 1.5).

Copy

Clear provides an operation called copy which makes a fresh copy of a theory with the exception of a specified set of subtheories (which are left as they are).

For example, here is a specification of the class of algebras having two sorts (each with an equivalence relation) and a function between them:

```
const Funct =
  let CopyEquiv = copy Equiv using Boolopns in
  enrich Equiv + CopyEquiv by
    opns f : element of Equiv -> element of CopyEquiv    enden
```

Copy is used so that the two sorts named element and the two = operators will remain distinct in the combined theory Equiv + CopyEquiv. But there is only one sort named bool in the result because of the using clause. The same result could be accomplished by explicitly writing out the definition of Equiv again; copy simply saves the trouble.

1.3. Error theories and more

Sometimes when specifying the action of an operator we find values for which it should not yield a result but instead should return some kind of error. Division by zero is an example. It is not sufficient to just leave cases like this unspecified; if a division by zero is attempted, we want an error message and not just any old result. We can extend the notion of theory given in section 1.1 to allow specification of errors; the new theories are called error theories. Details of this approach are given by Goguen [1978].

The idea is to add error elements to each sort which behave differently from the ordinary (OK) elements. Error elements are produced by error operators, and also by OK operators when applied to exceptional arguments. We add error equations to specify how errors are generated and manipulated.

Here is an example -- a specification of lists:

```
meta Triv =
  theory sorts element    endth


proc List(X:Triv) =
  enrich X + Bool by
    data sorts list
         opns nil : list
              cons : element,list -> list
              hd : list -> element
              tl : list -> list
         erroropns nohead : element
                   notail : list
         eqns hd(cons(a,l)) = a
              tl(cons(a,l)) = l
         erroreqns hd(nil) = nohead
                   tl(nil) = notail    enden
```

The models of such a theory are error algebras, in which each carrier contains distinguished error elements as well as OK elements. To be a model it need not satisfy all the equations for all variable assignments; it need only satisfy the OK equations for assignments in which both sides of the equation evaluate to an OK element, and the error equations for assignments in which either side evaluates to an error element. Furthermore, error algebras are restricted so as to propagate errors; that is, error operators always produce error elements, and OK operators applied to error elements produce error elements.

Another way we could extend the notion of theory is to add conditional equations, such as

```
put(i,v,a)[j] = v      if i==j
put(i,v,a)[j] = a[j]   if not(i==j)
```

to specify indexing on arrays (see [Thatcher, Wagner and Wright 1976]). We could regard

```
a = b  if c
```

as an abbreviation for

```
cond(c,a,b) = b
```

where cond : bool,s,s -> s is defined for each sort s by

```
cond(true,m,n) = m
cond(false,m,n) = n
```

But this means that all theories will contain a lot of extra operators, which is untidy.

Another way would be to simply extend theories to include conditional equations, calling the result a conditional theory. The notion of satisfaction would have to be changed slightly to deal with conditional equations.

Two ways of extending Clear have been mentioned. For error theories we defined a new kind of signature (with sorts, OK operators and error operators); a new notion of axiom (OK equations and error equations); a new kind of algebra (error algebras, with error elements); and a new notion of satisfaction. For conditional theories we only needed to define a new kind of axiom and a new notion of satisfaction; the signatures and algebras remain the same.

It is possible to define the theory-building operations of section 1.2 without reference to any particular notions of signature, signature morphism, axiom, algebra or satisfaction. This approach was taken in [Burstall and Goguen 1980], and is explained less formally in [Burstall and Goguen 1981]. Any choice for these five notions is appropriate as long as a few conditions hold. Briefly and very roughly, it must be possible to 'put together' signatures (the category of signatures and their morphisms must be cocomplete) and the various definitions must satisfy certain natural consistency conditions. Any such choice of notions is called an institution (or sometimes a language) and gives rise to a specification language like the one described in sections 1.1 and 1.2. The precise syntax of the language must be defined anew for each institution, since arbitrary signatures and axioms will not fit into the notation used above. For the data operation to work something more than an institution is needed; an enrichment must give rise to free extensions for the models of the enriched theory. Call an institution with this extra property a data institution (Goguen and Burstall [1980a] call this a liberal institution).

So Clear is not a specific language but instead a large family of languages (although references to Clear in the sequel will usually be to the particular language described in the last two sections -- this will be called ordinary Clear, or simply Clear). We are free to use 'error Clear' or 'conditional Clear' once we verify that our definitions describe a data institution. Other possibilities are: polymorphic Clear (section III.6), higher-order Clear (see [Dybjer 1981]), continuous Clear (see [Goguen, Thatcher, Wagner and Wright 1977]), order-sorted Clear (see [Goguen 1978a]) and predicate-calculus Clear (see [Burstall and Goguen 1981]).

1.4. An example

Here is a Clear specification (from scratch) of a larger and more interesting example than those which have appeared up to now. It specifies the problem of determining if a sequence of natural numbers is in ascending order.

```
const Bool =
  let Bool0 =
    theory
      data sorts bool
           opns true, false : bool    endth in
  enrich Bool0 by
    opns not : bool -> bool
         and, or, --> : bool,bool -> bool
    eqns not(true) = false
         not(not(p)) = p
         p and true = p
         p and false = false
         p or true = true
         p or false = p
         p --> q = not(p and not(q))    enden

meta Triv = theory sorts element    endth

proc Sequence(X:Triv) =
  enrich X + Bool by
    data sorts sequence
         opns empty : sequence
              unit : element -> sequence
              . : sequence,sequence -> sequence
         eqns empty.s = s
              s.empty = s
              (s.t).v = s.(t.v)    enden

meta Ident =
  enrich Bool + Triv by
    opns ≡ : element,element -> bool
    eqns m≡m = true
         m≡n = n≡m
         m≡n and n≡p --> m≡p = true    enden

meta POSet =
  enrich Ident by
    opns ≤ : element,element -> bool
    eqns m≤m = true
         m≤n and n≤m --> m≡n = true
         m≤n and n≤p --> m≤p = true    enden

proc Ordered(X:POSet) =
  enrich Sequence(X) by
    opns isordered : sequence -> bool
    eqns isordered(empty) = true
         isordered(unit(m)) = true
         isordered(s.unit(m).unit(n).t) = isordered(s.unit(m))
                    and isordered(unit(n).t) and m≤n    enden
```

```
const Nat =
  enrich Bool by
    data sorts nat
         opns 0 : nat
              succ : nat -> nat    enden

const Natord =
  enrich Nat by
    opns ≤ : nat,nat -> bool
    eqns 0≤n = true
         succ(n)≤0 = false
         succ(n)≤succ(m) = n≤m    enden

Ordered(Natord[element is nat, ≡ is ==])
```

1.5. Comparison with other approaches

We now briefly compare Clear with a variety of other approaches to specification. The features which set Clear apart from the myriad of alternative approaches seem to be:

- Clear provides theory-building operations (enrich, combine, derive, apply and copy) for constructing specifications in a structured fashion.
- Use of the data operation yields theories containing data constraints (section II.5), permitting loose specifications where some details are left unspecified. An example is the specification of sets with a choose operator in section 1.2.
- Clear is a complete language with a precise formal semantics.
- Clear is not dependent on any particular institution, so the notions of axiom, algebra, etc. may be easily changed.
- The theory-building operations respect shared subtheories.

It will be instructive to keep these features in mind when comparing Clear with the approaches described below.

Guttag, Horowitz and Musser [1978] present algebraic abstract data type specifications in an informal way, stressing the practical

application of specifications in programming (for proofs of correctness, program testing and program development). Guttag and Horning [1978] give a more formal treatment oriented toward providing guidelines for the construction of correct specifications. They distinguish a single type of interest in any specification, in contrast to Clear and many other approaches. Any algebra which satisfies the axioms of a specification and is finitely generated (every carrier element is the value of some term) with true≠false is acceptable as a model, although they seem to favour the 'final algebra' view that two terms should have the same value unless they are demonstrably different (see the notes on [Wand 1979] below).

The ADJ group [Goguen, Thatcher and Wagner 1978] is responsible for the first rigorous approach to the semantics of algebraic specifications. An equational theory specifies the isomorphism class of its initial models. Errors are discussed, but the approach is more primitive than that of Goguen [1978] which is adopted by Clear.

The ADJ approach to parameterised theories has evolved from a CLU-style view where application of a parameterised theory required only the presence of certain sorts and operators in the actual parameter [Goguen, Thatcher and Wagner 1978]. Starting with [Thatcher, Wagner and Wright 1978], a parameterised theory P with metasort theory R is seen as specifying a functor F taking any model M of R to a single model of P (in fact, to the P-model freely generated by M) — this is a special case of parameterised theories in Clear, where the theory P(A) may have 'more' models than the theory A. If M is the initial model of A, then F(M) is the initial model of P(A) provided that P is well-behaved (i.e. persistent — F(M) restricted to A is isomorphic to M — see section VII.4). In the absence of data constraints, R is allowed to include conditional axioms of the form

$$e_1 \text{ and } \ldots \text{ and } e_n \Rightarrow e_{n+1}$$

where the e_i may be equations or inequations. This work was a significant influence on the design of Clear. Later in [Ehrig, Kreowski, Thatcher, Wagner and Wright 1980] these were restricted to universal Horn sentences [Grätzer 1979] where $e_1,\ldots,e_n$ must be

equations, and application was defined as the pushout of R⟶P with the fitting morphism R⟶A as in Clear -- see section V.3.4. Application was generalised to allow composition of parameterised theories (it would be easy to extend Clear to permit this).

Continuing along the same line, Ehrig [1981] permits R to contain requirements of a general kind; anything having a well-defined set of algebras satisfying it is allowed (this flexibility is very reminiscent of Clear's institutional approach). Possible kinds of requirements include functor image restrictions, a generalisation of data constraints where any persistent functor is allowed in place of the free functor (see [Burstall and Goguen 1980] for the category-theoretic approach to data constraints). He suggests that this approach to parameterised theories can be used to solve the problems attacked by Clear of combining theories with shared subtheories in an easier way, but this remark does not seem to be justified. Ehrig and Fey [1981] allow theories (not just parameterised theories) to include requirements; such a theory may have an assortment of nonisomorphic models. Such a requirements specification is seen as a step between an informal specification and a design specification (which does not include requirements and specifies the initial model). Requirements in parameterised theories are still restricted to the metasort R, and a parameterised theory is viewed as specifying a functor taking any model of R to a model of the parameterised theory P. This rules out specifications such as the parameterised theory of sets with an operator choose:set->element loosely defined to select an arbitrary element of a set (see section 1.2).

Ganzinger [1980] discusses parameterised theories from a purely syntactic point of view (without considering models at all). The metasort of a parameterised theory includes all primitive subtheories (such as Bool and Nat); this is important for his notion of implementation, and it also has the consequence that if A and P share a common primitive subtheory T, P(A) will contain only one copy of T (again as a primitive subtheory). This idea resembles Clear's notion of a based theory (section III.1). All theories are parameterised, and all parameterised theories are required to be

persistent. The example of sets with a choose function is not a parameterised theory according to his definition of persistence. Application of parameterised theories is defined by a construction. The main emphasis is on proving that persistency guarantees correct parameter passing (i.e. that A is 'protected' in P(A)).

Ehrich [1982] presents an approach to parameterised theories building on earlier work by Ehrich and Lohberger [1978] which is similar in many ways to that of Clear. A metasort theory R is associated with each parameterised theory P, and a fitting morphism from R to an actual parameter theory A is needed to produce the application P(A) (as in Clear, this is defined using pushouts). A theory is viewed as specifying its initial model, and consequently a parameterised theory denotes a functor as in the ADJ approach. No analogue to data constraints is considered (so loose specifications are not permitted) and the problem of combining theories having shared subtheories is not treated.

Hupbach, Kaphengst and Reichel [1980] present a specification language and define its semantics. Theories may specify partial functions and may include conditional equations. Canons are theories which include initial restrictions (data constraints as we call them) and may be loose, specifying any model satisfying the axioms and initial restrictions. An operation like enrich is defined (actually, two separate operations for data and non-data enrichment) as well as a combine operation which is just union. Identification of common sorts and operators is therefore entirely by name, so overloading of identifiers is not permitted. Parameterised theories are as in Clear, and application is defined by means of a construction. The language also includes a construct for specifying that one theory is an implementation of another (see chapter VII).

Wand [1979] presents an alternative to ADJ's initial algebra approach, using the framework of Lawvere theories [Lawvere 1963]. He argues that the initial model of a theory often retains too much information. For example, consider the theory of sets of integers with operators Ø, add and is_in, and the following equations:

```
n is_in ∅ = false
n is_in add(n,S) = true
n is_in add(m,S) = n is_in S  if not(n==m)
```

The equation add(1,add(2,∅))=add(2,add(1,∅)) is not satisfied in the initial model since we have forgotten equations like

```
add(n,add(m,S)) = add(m,add(n,S))
add(n,add(n,S)) = add(n,S)
```

But even without these extra equations the two sets are behaviourally equivalent (with respect to the sort bool); any 'computation' involving the given operators yielding a boolean value will give the same result for both sets. This notion of behavioural equivalence is captured by Wand's final algebra approach. In the final algebra of a theory two terms have the same value unless they are demonstrably different. In order for this approach to work it is necessary to start with some primitive sort (e.g. bool) with some values which are known to be unequal (true, false); otherwise no two values will ever be demonstrably different in the absence of inequations.

Another alternative to the initial algebra approach is advocated by Lehmann and Smyth [1981] based on work by Scott [1976]. A data type is specified by a recursive domain equation which defines an endofunctor on a special category of complete partial orders; the data type is regarded as the initial fixpoint of this functor. For example, (finite) binary trees with labels from the domain A are specified by the following equation:

$$BtreeA \cong 2 + A \bullet BtreeA \bullet BtreeA$$

(where $2=\{\bot,\top\}$ with $\bot \sqsubseteq \top$, and $\bullet$ is coalesced (smash) product). A parameterised data type is a functor as well. This approach seems to work well for simple data types and has the advantage of automatically extending to higher-order types, but there seems to be no way of imposing equations on types so it is difficult to see how to specify sets (for example).

Nakajima, Honda and Nakahara [1980] describe a language called ι (iota) for building specifications and implementing them with programs. A theory can be either a type (Clear data theory) or a sype (combining Clear non-data and meta theories). As the approach is rather syntactic models are not discussed, but it seems from the

examples given that any finitely generated model satisfying the axioms (which are in first-order logic) would be acceptable. Specifications may include operations (returning results via arguments) for specifying procedures, but these are viewed as functions as well (an operation f:array (var),array,int is like the function f:array,array,int->array). A type can be implemented by writing a realisation as an ALGOL-like program, and a method for proving correctness of realisations is given. A theory-building operation which combines + and enrich is provided, and parameterised theories are allowed as in Clear (the requirement theory is a sype). These operations take proper account of shared subtheories, using 'tags' as in chapter III (but only operator names may be overloaded). The notion of fitting morphism is somewhat more restricted than in Clear (it must be an inclusion with the exception of the name of the 'principle' sort) and building a sype by enriching another sype is not allowed; no reason is given for either restriction.

Bauer et al [1981] describe and give a semantics for CIP-L, a 'wide-spectrum' language including constructs suitable for programming as well as specification. CIP-L is intended for use in a program development system, and allows a program to be expressed at every stage of its evolution from a specification to an efficient program. Abstract data type specifications allow hidden sorts and functions, partial functions and first-order axioms. Operations similar to enrich, combine and apply in Clear are defined but name clashes are forbidden. Parameterised types are viewed as type schemes, and application is by textual substitution. When the enrich operation is used, the theory being enriched is regarded as a primitive subtheory of the result of the enrichment. Models are required to be hierarchy preserving, meaning that all values of primitive sorts must be generated by primitive operators. Models must also be finitely generated and must satisfy the axioms. There is no way to restrict consideration to the set of initial or freely generated models, but because the axioms may include inequations and because of the finite generation requirement this is not a problem, although specifications tend to be longer than in Clear.

Meta-IV [Jones 1978], the meta-language of the Vienna development method, is a notation for describing the denotational semantics of large programming languages and systems. It has been used to specify a subset of PL/I [Bekić et al 1974]. The abstract syntax of the object language is described using a BNF-like notation which provides constructors, recognisers and selectors for use in the rest of the definition. Context conditions are then given to specify for each syntax class which objects are well-formed. Next the semantic domains are defined using combinators such as $\rightarrow$ (continuous function). Meaning functions then provide denotations for all well-formed objects. The meaning functions (and the context conditions) are mutually recursive functions written in a language similar to HOPE (section 2) but with a few nonalgorithmic constructs such as "let var be s.t. condition". Meta-IV is not restricted to specifying the semantics of programming languages; the abstract syntax is merely a signature in disguise (or vice versa) and meaning functions provide a (more constructive) substitute for equational definitions, so specifications of abstract data types and programs are also possible.

Abrial, Schuman and Meyer [1979] describe Z, a specification language based on axiomatic set theory, and give a number of large and interesting specification examples. Z is essentially a formal (and rather verbose) language for describing sets. The natural numbers, relations, sequences etc. can be viewed as sets using the classical constructions. Since everything is a set (the elements of a sets are themselves sets) there is no notion of type. The set union function works equally well on sets, natural numbers and relations; it is not clear what happens if a sequence is subtracted from a number. Second- and higher-order functions can be specified in the same way as ordinary functions. Definitions may be parameterised (generic), but any set is accepted as an actual parameter; there is no equivalent to Clear's metasort theory. Structures (classes) consisting of a tuple of sets and some axioms about them may be defined (examples are groups and rings). Specifications are structured into chapters, and new chapters may be built by enriching previous chapters. Theorems which the definitions are expected to satisfy may be included, but these have

no effect on the specification itself.

SPECIAL [Roubine and Robinson 1977] is the specification language for HDM [Spitzen, Levitt and Robinson 1978], [Levitt, Robinson and Silverberg 1979], a design methodology which is based on suggestions of Parnas [1972, 1972a] concerning the decomposition of large systems into hierarchical collections of modules. A module implements an abstract machine which is realised by a collection of programs running on a lower-level abstract machine. A module can also be viewed as an abstract data type. Modules are described in SPECIAL by specifying how O- (operation) functions change the internal state of the module as visible through the use of V- (value) functions. That is, the specifications of V-functions describe the initial state of the module, and the specification of an O-function describes what changes a use of the O-function causes in the results returned by V-functions. Modules can be parameterised, where the parameters are functions or values. A module may reference the functions of other modules, and apparently a call of an O-function may even result in a change in the state of another module. A feature is included for specifying that the execution of a function will be delayed until some event takes place; this permits the specification of systems of parallel processes. Mapping functions which describe how a module is implemented in terms of a lower-level module may also be specified. An operational semantics of a subset of SPECIAL has been given by Boyer and Moore [1978].

ORDINARY [Goguen and Burstall 1980a] is an attempt to combine the rigorous theoretical foundation and theory-building ideas of Clear with the state-machine specification approach of SPECIAL. Its semantics will be given by translation into Clear, although the translation has not yet been defined. ORDINARY provides facilities for specifying clusters (like Clear theories) and modules (as in SPECIAL), and both clusters and modules may be parameterised as in Clear. The specification of modules is different from in SPECIAL. The state is defined explicitly, rather than implicitly through the collection of available V-functions. A function like add:int in a set module (add the given integer to the set which forms the state)

is specified as add:[set],int->[set] (bracketed arguments and results are invisible); the state is thus passed around as a secret argument and result of appropriate functions. So although module specifications are superficially different from cluster specifications (with effects on the state defined using a SPECIAL-like syntax rather than using ordinary equations) they are essentially the same. In contrast to SPECIAL, states of modules are never accessible from outside. Higher-order operators like lambda are (tentatively) handled as macros. All the theory-building operations of Clear are available (including data), albeit with a more convenient syntax in some cases. Application of parameterised clusters and modules is nonprolific (see section III.5) in contrast to Clear. Like Clear, ORDINARY is independent of any particular institution; a different application (such as specification of concurrent systems) will be handled by switching to a different institution (such as temporal logic).

2. HOPE

This section contains only a very brief glimpse into the features and nature of HOPE. A full description appears in Appendix 1.

The underlying goal in the design of HOPE was to produce by a judicious selection of well-understood ideas a very simple programming language which would encourage the construction of clear and manipulable programs. HOPE is a purely applicative language without an assignment statement or destructive operators. This was felt to be an important simplification, encouraging a transparent and less error-prone style of programming. Backus [1978] makes this case strongly.

The user may freely define his own data types. A type is the sum of a set of disjoint subtypes, each having its own data construction function. There is no need to devise a complicated encoding of a new type in terms of low-level types, since data constructors are uninterpreted; this leads to inefficient use of space in some cases but it make programs much easier to write. The type system is strongly enforced but at the same time very flexible, allowing the definition of polymorphic types and the free use of higher-order types and overloaded operators.

Functions are defined by a set of recursion equations. The left-hand side of each equation includes a pattern built from data constructors and variables; the pattern is used both to select which equation to use for a given argument and to bind the variables in that equation to the appropriate parts of the argument. For example:

```
--- reverse nil <= nil
--- reverse(a::l) <= reverse l <> [a]
```

The availability of arbitrary higher-order types allows functions to be defined which 'package' recursion over data structures to save writing it explicitly. These functions can be used to write programs in a concise style similar to that of APL [Iverson 1962]. Lazily-evaluated lists (streams) are provided, allowing the use of infinite lists which could be used to provide interactive input/output and concurrency.

HOPE also includes a simple modularisation facility which allows programs to be constructed as a collection of small self-contained pieces communicating with each other in a disciplined and explicit manner. A module may be used to protect the implementation of an abstract data type, for example. Careful modular development is felt to be the main trick in the construction of large bug-free programs.

CHAPTER TWO

PREREQUISITES -- BASIC CONCEPTS AND NOTATION

The basic concepts which underlie the semantics of Clear will now be defined. The notions of signature, algebra and equation are similar to those used by most authors (see for example [Goguen, Thatcher and Wagner 1978]), but theories in Clear are different from the usual notion of theory elsewhere (which corresponds to a simple theory presentation in Clear). The definitions themselves are taken (with minor changes) from [Burstall and Goguen 1980].

In order to define the meaning of the data operation of Clear we need the notion of a data constraint discussed in section 5. Essentially the same concept is described by Reichel [1980] (cf. Kaphengst and Reichel [1971]) and by Wirsing and Broy [1981] (cf. Broy et al [1979]). Data constraints for Clear were defined very technically in [Burstall and Goguen 1980] and then discussed informally in [Burstall and Goguen 1981]; the presentation here is precise but avoids the use of category theory, although this necessarily restricts the discussion to data constraints in ordinary Clear.

The data operation is used in Clear to specify that only the initial models of a specification are desired (more precisely, models which are free extensions of models of the theory which is enriched using data). In contrast to this 'initial algebra approach' is the final algebra approach of Wand (see [Wand 1979], also [Guttag, Horowitz and Musser 1978]). This seems to offer a viable alternative (which is even better in some respects) by considering a different class of distinguished models. In this thesis (apart from chapter VII) only the initial algebra approach to specification will be discussed. The choice is irrelevant to the bulk of the material presented; initial models are used in order to avoid departing from previous work on Clear.

Although many of the definitions below (those concerning signatures, algebras and equations) are special to ordinary Clear, the definitions concerning theories and data constraints could be

generalised to the case of an arbitrary institution. In that event, all the results given below would remain valid.

1. Signatures

A signature is a set of sorts (data type names) together with a set of operators (operation names), where each operator has an arity (such as s,t -> t where s and t are sorts). A signature morphism maps the sorts and operators of one signature to sorts and operators in another in such a way that arities are preserved.

Def: A (many-sorted) signature Σ is a pair $\langle S,\Sigma\rangle$ where S is a set (of sorts) and Σ is a family of sets (of operators) indexed by $S^+=S^*xS$. The index of a set $O\in\Sigma$ is the arity of every element of O.

Def: A signature morphism σ is a pair $\langle f,g\rangle : \langle S,\Sigma\rangle \rightarrow \langle S',\Sigma'\rangle$ where $f:S\rightarrow S'$ and g is a family of maps $g_{us}:\Sigma_{us}\rightarrow\Sigma'_{f^*(u)f(s)}$, where $u\in S^*$, $s\in S$ and $f^*:S^*\rightarrow S'^*$ is the extension of f to strings of sorts. We write $\sigma(s)$ for f(s), $\sigma(u)$ for $f^*(u)$ and $\sigma(\omega)$ for $g_{us}(\omega)$, where $\omega\in\Sigma_{us}$.

2. Algebras

A Σ-algebra has a set (the elements of a data type) for each sort of Σ and a function (operation) on those sets for each operator of Σ. A Σ-homomorphism maps the 'data types' of one Σ-algebra to those of another in such a way that the operations are preserved. Given a Σ'-algebra A and a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$, we can recover the Σ-algebra buried inside A (since A is just an extension of this algebra).

Let Σ be a signature.

Def: A Σ-algebra A is a pair $\langle A,\alpha\rangle$, where A is an S-indexed family of sets (the carriers of A) and α is an S^*xS-indexed family of maps $\alpha_{us}:\Sigma_{us}\rightarrow(A_u\rightarrow A_s)$ where $u\in S^*$, $s\in S$ and $A_{u_1\ldots u_n} = A_{u_1}x\ldots xA_{u_n}$. If $\omega\in\Sigma_{us}$ then the map $\alpha_{us}(\omega):A_u\rightarrow A_s$ is called the operation associated with ω, and is referred to by the name ω when there is no ambiguity.

Def: A Σ-homomorphism $f : \langle A,\alpha\rangle \rightarrow \langle A',\alpha'\rangle$, where $\langle A,\alpha\rangle$ and $\langle A',\alpha'\rangle$ are Σ-algebras, is a map $f:A\rightarrow A'$ (actually an S-indexed family of maps $f_s:A_s\rightarrow A'_s$) such that for each $\omega\in\Sigma$ and each $a_1\in A_{s_1},\ldots,a_n\in A_{s_n}$ $f_s(\alpha(\omega)(a_1,\ldots,a_n)) = \alpha'(\omega)(f_{s_1}(a_1),\ldots,f_{s_n}(a_n))$.

Def: If $\sigma=\langle f,g\rangle$ is a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ and $\underline{A}'=\langle A',\alpha'\rangle$ is a Σ'-algebra, then the Σ-restriction of $\underline{A}'$ (along σ), written $\underline{A}'\big|^{\sigma}_{\Sigma}$ is the Σ-algebra $\langle A,\alpha\rangle$ where $A_s=A'_{f(s)}$ and $\alpha(\omega)=\alpha'(g(\omega))$. Normally σ is obvious from context, in which case the notation $\underline{A}'\big|_{\Sigma}$ may be used.

3. Equations

The definition of Σ-equations and the meaning of applying a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ to a Σ-equation are the obvious ones. A Σ-algebra satisfies a Σ-equation if the equation is 'true' (both sides evaluate to the same thing) for all assignments to the variables.

Def: A Σ-equation e is a triple $\langle X,\tau_1,\tau_2\rangle$ where X is an S-indexed set (of variables) and τ_1, τ_2 are Σ-terms on X of the same sort. The equation $\langle X,\tau_1,\tau_2\rangle$ is written 'for all X. $\tau_1 = \tau_2$'.

If σ is a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ then the extension of σ to Σ-terms, $\sigma^{\#}$: Σ-terms $\rightarrow$ Σ'-terms may be applied to a Σ-equation e; this application is written simply $\sigma(e)$.

Def: A Σ-algebra $\underline{A}=\langle A,\alpha\rangle$ satisfies a Σ-equation $\langle X,\tau_1,\tau_2\rangle$ if for all maps $f:X\rightarrow A$, $f^{\#}(\tau_1) = f^{\#}(\tau_2)$ where $f^{\#}$: Σ-terms $\rightarrow$ A is the extension of f to Σ-terms on X ($f^{\#}$ evaluates a term using the assignment of values to variables given by f). $\underline{A}$ satisfies e is written $\underline{A}\models e$. A Σ-algebra satisfies a set of Σ-equations if it satisfies every equation in the set.

Satisfaction Lemma: If $\sigma:\Sigma\rightarrow\Sigma'$ is a signature morphism, e is a Σ-equation and $\underline{A}'$ is a Σ'-algebra, then $\underline{A}'\models\sigma(e)$ iff $\underline{A}'\big|^{\sigma}_{\Sigma}\models e$.

Proof: See [Burstall and Goguen 1980].

4. Simple theories

A simple theory presentation is a signature together with a set of equations on that signature. The closure of a set of equations is that set together with all its (model-theoretic) logical consequences. A simple theory is then a signature together with a closed set of equations. This is a simple theory because no data constraints (section 5) are included.

Def: A simple Σ-theory presentation is a pair $\langle\Sigma,E\rangle$ where Σ is a signature and E is a set of Σ-equations.

Def: A Σ-algebra A satisfies a simple theory presentation $\langle\Sigma,E\rangle$ if A satisfies E. Then A is called a model of $\langle\Sigma,E\rangle$.

Def: If E is a set of Σ-equations, let E^* be the set of all Σ-algebras which satisfy E.

Def: If M is a set of Σ-algebras, let M^* be the set of all Σ-equations which are satisfied by each algebra in M.

Fact: For any set E of equations (and dually replacing E by any set M of algebras):

(i) $E \subseteq E^{**}$

(ii) If $E \subseteq E'$ then $E'^* \subseteq E^*$

This is called a Galois connection (see [Birkhoff 1948]). The laws (i) and (ii) together imply

(iii) $E^* = E^{***}$

Def: The closure of a set E of Σ-equations is the set E^{**}, written $\overline{E}$. E is closed if $E=\overline{E}$.

Def: A simple Σ-theory T is a simple theory presentation $\langle\Sigma,E\rangle$ where E is closed. The simple Σ-theory presented by the presentation $\langle\Sigma,E\rangle$ is $\langle\Sigma,\overline{E}\rangle$. A simple theory $\langle\Sigma,\overline{\emptyset}\rangle$ is called anarchic. A theory is called satisfiable if it has at least one model.

Def: A simple theory morphism $\sigma : \langle \Sigma, E \rangle \rightarrow \langle \Sigma', E' \rangle$ is a signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ such that $\sigma(e) \in E'$ for each $e \in E$.

Closure Lemma: $\sigma(\overline{E}) \subseteq \overline{\sigma(E)}$

Proof: See [Burstall and Goguen 1980]; uses the Satisfaction Lemma.

Presentation Lemma: If $\sigma: \Sigma \rightarrow \Sigma'$ is a signature morphism and $\langle \Sigma, E \rangle$, $\langle \Sigma', E' \rangle$ are simple theory presentations then $\sigma : \langle \Sigma, \overline{E} \rangle \rightarrow \langle \Sigma', \overline{E}' \rangle$ is a simple theory morphism iff $\sigma(E) \subseteq \overline{E}'$.

Proof: See [Burstall and Goguen 1980]; uses the Closure Lemma.

The Presentation Lemma gives a shortcut for checking if a signature morphism σ is a simple theory morphism -- one must only check, for each equation e of the source presentation, that $\sigma(e)$ can be proved from the equations in the target presentation.

Theorem: The category of simple theories and simple theory morphisms is finitely cocomplete (has finite colimits).

Proof: See section V.2.

The category-theoretic semantics of Clear given in chapter V relies on this theorem. In that semantics the theory-building operations of Clear are defined in terms of certain colimits in the category of theories.

5. Data constraints and data theories

In the last section a definition was given for the meaning of an algebra satisfying a simple theory (presentation). If an algebra satisfies a theory, it is called a model for that theory. The theory specifies a set of algebras, namely the set of all its models.

Unfortunately, this notion of specification is too simple for most uses. The problem is that a theory has far too many models, some of which have trivial carriers. It turns out that in many cases (for example, when a theory is written to specify an abstract data type) the model which is really intended is easily characterised; it is the initial model of the theory. See section I.1.1 for some examples.

The word 'initial' comes from category theory; however, it is not necessary to know about category theory to understand initial models.

Def: An initial model of a theory presentation $\langle \Sigma, E \rangle$ is a Σ-algebra $\underline{A}$ which is a model of $\langle \Sigma, E \rangle$ such that

- $\underline{A}$ does not satisfy any ground (i.e. variable-free) Σ-equation which is not in $\overline{E}$.
- Every element in $\underline{A}$ is the value of some ground Σ-term.

In the last section the closure of a set of equations, $\overline{E}$, was defined as the set of equations satisfied by every model of E. One may think of $\overline{E}$ as the set E together with all equations provable from E using purely equational logic -- that is, using substitution and the reflexive, symmetric and transitive properties of equality (but without use of induction). This aids intuition but is slightly inaccurate because of the incompleteness result to be given in section VI.5. The set of equations satisfied by an initial model correspond to the equations provable by equational deduction together with induction, since the second extra condition above amounts to an induction rule on each sort of Σ.

Fact: An initial model of a theory presentation $\langle \Sigma, E \rangle$ is $T_\Sigma / \equiv_E$ where T_Σ is the 'initial' Σ-algebra, consisting of ground Σ-terms, and $\equiv_E$ is the Σ-congruence on T_Σ generated by E.

Proof: See [Goguen, Thatcher and Wagner 1978].

But in Clear the situation is more complicated than this. Smaller theories are put together to make larger theories; if a loose or non-data theory is put together with an 'initial' theory, then what is the result? The models of the result should be all models of the combined theory which satisfy the initiality constraint for the appropriate sorts, operators and equations.

Consider the case where a non-data theory (Equiv from section I.1.2) is extended by adding some data, as in the following:

```
const Set =
  enrich Equiv by
    data sorts set
         opns Ø : set
              singleton : element -> set
              U : set,set -> set
         eqns Ø U S = S
              S U S = S
              S U T = T U S
              (S U T) U V = S U (T U V)    enden
```

In this example the interpretation of the extension must depend on the interpretation of Equiv, which can be any algebra having a sort together with an equivalence relation. But given a particular algebra for Equiv, Set should be interpreted initially based on that algebra; that is, Set specifies an initial algebra relative to the interpretation of Equiv. Set is a data extension of Equiv; each Set-model is the free extension of the included interpretation of Equiv.

It is necessary to keep track of more than just a signature and a set of equations to determine the set of algebras specified by a Clear specification; of equal importance are the details concerning which enrichments are data extensions of which subtheories. The constraint that an enrichment is to be interpreted as a data extension is called a data constraint (or constraint for short).

Each application of the data operator contributes a data constraint.

Def: A Σ-constraint c is a pair $\langle i,\sigma\rangle$ where $i:\underline{T}\hookrightarrow\underline{T}'$ is a simple theory inclusion and σ:signature($\underline{T}'$)$\rightarrow\Sigma$ is a signature morphism.

A constraint is a description of an enrichment (the theory inclusion goes from the theory to be enriched to the enriched theory) together with a signature morphism 'translating' the constraint to the signature Σ.

A signature morphism from Σ to another signature Σ' can be applied to a Σ-constraint, translating it to a Σ'-constraint, just as it can be applied to a Σ-equation to give a Σ'-equation.

Def: If $\sigma':\Sigma\rightarrow\Sigma'$ is a signature morphism and $\langle i,\sigma\rangle$ is a Σ-constraint, then σ' applied to $\langle i,\sigma\rangle$ gives the Σ'-constraint $\langle i,\sigma.\sigma'\rangle$.

A data constraint imposes a restriction on a set of algebras, just as an equation does. In [Burstall and Goguen 1980] this restriction was defined category-theoretically. Here is the same definition from a different point of view:

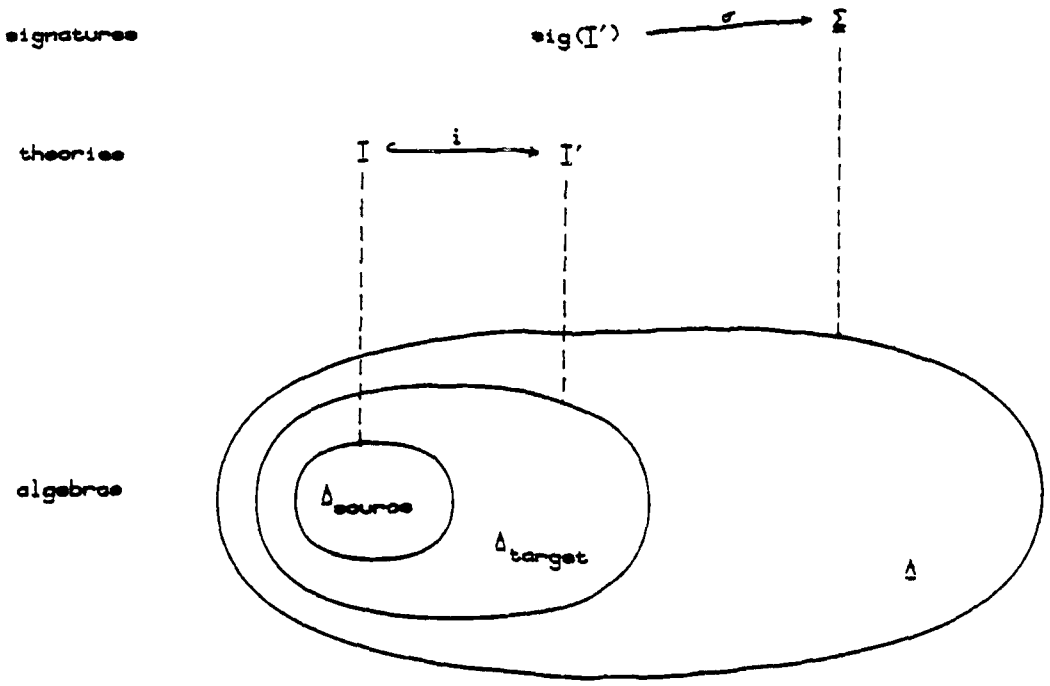
Def: A Σ-algebra $\underline{A}$ satisfies a Σ-constraint $\langle i:\underline{T}\hookrightarrow\underline{T}',\sigma:\text{signature}(\underline{T}')\rightarrow\Sigma\rangle$ if

[letting $\underline{A}_{\text{target}} = \underline{A}|^{\sigma}_{\text{signature}(\underline{T}')}$
and $\underline{A}_{\text{source}} = \underline{A}|^{i.\sigma}_{\text{signature}(\underline{T})}$]

$\underline{A}_{\text{target}}$ is a model of $\underline{T}'$ and

- "No confusion": $\underline{A}_{\text{target}}$ does not satisfy any signature($\underline{T}'$)-equation e with variables only in sorts of $\underline{T}$ for any injective assignment of variables to $\underline{A}_{\text{source}}$ values unless e is in $\overline{\text{eqns}(\underline{T}')\cup\underline{A}_{\text{source}}}^{*}$.
- "No junk": Every element in $\underline{A}_{\text{target}}$ is the value of a $\underline{T}'$-term which has variables only in sorts of $\underline{T}$, for some assignment of $\underline{A}_{\text{source}}$ values.

The diagram of the situation below may help make the notation easier to understand.



The "no confusion" condition requires that no two terms have the same value in $\underline{A}_{target}$ unless this is forced by the equations of $\underline{T}'$ or by previous identification of the values of terms in $\underline{A}_{source}$ (such identifications are recorded in $\underline{A}^{*}_{source}$). The assignment is restricted to be injective because (for example) the equation x=y will always be satisfied under *some* (noninjective) assignment, but this equation will almost never be in $\overline{eqns(\underline{T}') \cup \underline{A}^{*}_{source}}$. The "no junk" condition requires that all values in $\underline{A}_{target}$ be 'generated' by constants or by the application of functions to values in $\underline{A}_{source}$. The slogans are from [Burstall and Goguen 1981].

An alternative "no confusion" condition which may be slightly easier to understand requires that $\underline{A}_{source}$ be countable. If a signature $\underline{\Sigma}$ includes the signature of $\underline{T}$, then let $\underline{\Sigma}^{\dagger}$ be $\underline{\Sigma}$ together with a (constant) operator c_a for every value a in $\underline{A}_{source}$. Similarly, if $\underline{B}$ is a $\underline{\Sigma}$-algebra and $\underline{A}_{source}$ is a subalgebra of $\underline{B}$, let $\underline{B}^{\dagger}$ be the $\underline{\Sigma}^{\dagger}$-algebra obtained from $\underline{B}$ by associating each new operator c_a with the value a in $\underline{B}$. Then:

- "No confusion": $\underline{A}^{\dagger}_{target}$ does not satisfy any ground equation which is not in $\overline{eqns(\underline{T}') \cup \underline{A}^{\dagger *}_{source}}$.

The new constants give names to the values which we previously could only refer to using variables under an injective assignment.

Since data constraints behave just like equations, they can be added to the equation set in a simple theory presentation to give a data theory presentation (or theory presentation for short).

Def: A (data) Σ-theory presentation is a pair $\langle\Sigma,EC\rangle$ where Σ is a signature and EC is a set of Σ-equations and Σ-constraints.

The notions of (data) theory, satisfaction (of a data theory), closure and (data) theory morphism follow as in the 'simple' case. The Satisfaction Lemma (section 3) holds for constraints as well as equations, and all the results in section 4 still hold.

Note what happens if an attempt is made to tamper with a theory in a way which violates one of its constraints:

```
const Bool =
  theory
    data sorts bool
         opns true, false : bool
              not : bool -> bool
         eqns not(true) = false
              not(not(p)) = p    endth

const FunnyBool =
  enrich Bool by
    eqns not(p) = p    enden
```

The new equation in FunnyBool is inconsistent with the data constraint produced by the application of data in Bool. FunnyBool has no models, since no algebra exists which satisfies both the constraint and the new equation.

For other presentations of this material, consult [Burstall and Goguen 1980] (technical) or [Burstall and Goguen 1981] (informal). The data constraints described here are a special case of those discussed in [Burstall and Goguen 1980]; general data constraints never arise in ordinary Clear, but they are necessary for describing the semantics of Clear under an arbitrary (data) institution. In its more general form, a data constraint consists of an arbitrary simple theory morphism (not necessarily an inclusion) together with a signature morphism, and satisfaction of a data constraint is

defined using the category-theoretic notion of an adjunction. The definition of data constraint satisfaction given above is an attempt to capture, in this special case, the definition of Burstall and Goguen [1980] using a different approach.

CHAPTER THREE

A SET-THEORETIC SEMANTICS OF CLEAR

In the Introduction we argued for the necessity of supplying a specification language with a precise and formal semantics. A specification language like Clear can be useful on an informal level as a tool for the development of programs, providing a notation for elaborating the theory behind and surrounding a problem. But without a semantics the connection between specifications and programs is tenuous at best, giving no possibility of proving that a program is correct with respect to its specification (for example).

A semantics of Clear is presented here which uses the language of set theory. The theory-building operations presented in chapter I are described by means of elementary set-theoretic constructions. In order to properly treat the problem of shared subtheories, a tag is attached to every sort and operator to indicate its theory of origin; this trick allows the combine operation to be expressed as little more than the set-theoretic union of 'tagged' theories. The remaining operations are only a little bit more difficult to describe. A denotational semantics is then given which attaches a syntax to these operations and provides for an environment of named theories. An additional section gives the semantics for an improved version of Clear, identical to ordinary Clear except for the absence of an annoying characteristic. This shows how easily the semantics can be changed to accommodate new features.

Burstall and Goguen [1980] have described a semantics of Clear which relies heavily on a number of ideas from category theory to describe the underlying concepts and operations of the language. Their semantics is presented in chapter V. The semantics in the present chapter was invented after Burstall and Goguen's semantics as an equivalent but more accessible alternative. The category-theoretic semantics, by abstracting away from any particular notion of signature, model or axiom (using the concept of an institution mentioned in section I.1.3), is able to describe all at once the meaning of a large class of Clear-like specification languages. But

in the special case of ordinary Clear (the language described in chapter I) this highly abstract treatment can be simplified to give the semantics described here; this has the advantage of being concrete and constructive and therefore more useful for practical applications. And even this semantics can be generalised to give the semantics of Clear under all institutions which have been suggested up to now (see section 6).

1. Dealing with shared subtheories

Consider the following specification, defining the theory of natural numbers with an order relation and the theory of upper case alphabetic characters (it is assumed that the theory Bool of boolean values has been previously defined):

```
const Nat =
  enrich Bool by
    data sorts nat
         opns 0 : nat
              succ : nat -> nat
              ≤ : nat,nat -> bool
         eqns 0≤n = true
              succ(n)≤0 = false
              succ(n)≤succ(m) = n≤m    enden
```

```
const Char =
  enrich Bool by
    data sorts char
         opns A, B, ..., Z : char
              is_vowel : char -> bool
         eqns is_vowel(c) = c==A or c==E
                          or c==I or c==O or c==U    enden
```

Notice that both Nat and Char 'include' the theory Bool; Bool is a shared subtheory of Nat and Char. What does this mean formally? And, how does the semantics of Clear define the theory-combining operations so that the theory Nat + Char includes only one copy of Bool?

In [Burstall and Goguen 1977], shared subtheories are explained by analogy with the EQ predicate of LISP. The EQUAL function in LISP tests whether two lists look the same (i.e. whether they contain the same elements in the same order), while EQ tests whether two lists are the same (occupy the same list cells in storage -- note that EQ(a,b) implies EQUAL(a,b) but not vice versa). The important features of EQ are given by the following examples (a, b and c are arbitrary lists):

i. EQ(CONS(a,b),CONS(a,b)) = false (but EQUAL(..,..) = true)

ii. (EQ(l,l) where l=CONS(a,b)) = true

iii. EQ(CAR(CONS(a,b)),CDR(CONS(c,a))) = true

These examples show that

i. Writing down a CONS expression twice gives two different lists.

ii. Two uses of the same variable refer to the same list.

iii. Two different lists can share a common sublist.

Now to complete the analogy, the theory-building operations of Clear act like CONS and the behaviour of EQ indicates what is meant by "identical" in the following:

Requirement: The theory-building operations should be defined in such a way that a theory can never contain two identical subtheories.

This leads (for example) to the following informal constraint on the combine (+) operation:

Constraint: If B is a subtheory of A and D is a subtheory of C, then B and D should be identified when forming A + C iff they are identical.

In order to write a semantics for Clear we must devise some representation of theories which makes it easy (or at least possible) to determine if two theories are identical, so that the above constraint can be satisfied. The category-theoretic semantics of Burstall and Goguen [1980] uses a rather complicated representation of a theory (called a <u>based theory</u> -- see section V.2 for details) which shows explicitly how the theory is related to every one of its subtheories. In the special case of ordinary Clear a much simpler representation can be used because the only way that a theory and one of its subtheories can be related is by an inclusion.

An important observation is the fact that the requirement above is inherited by the sorts and operators of a theory (where identity is again by analogy to EQ in LISP), giving:

Requirement: The theory-building operations should be defined in such a way that a theory can never contain two identical sorts or operators.

Moreover, if this low-level requirement is satisfied (and the operations are defined in a reasonable way) then the previous requirement will be satisfied as well. The above contraint on combine also has a low-level equivalent.

Referring to our LISP analogy, the obvious way to define the semantics of EQ (see [McCarthy et al 1962]) is to use a model of storage where lists are stored in addressable cells and EQ simply checks whether its arguments begin at the same address (although the semantics of EQ can be defined in other ways -- see [Levy 1980] for example). By associating a unique address with each non-EQ list cell, the meaning of EQ is reduced to equality of addresses.

Sorts, operators and theories normally have nothing to do with anything as mundane as storage and addresses. But by associating an appropriate tag with each sort and operator we can easily determine whether two tagged sorts or tagged operators are identical in the sense given by analogy with EQ. If the name of the theory of origin of a sort or operator is used as a tag, then the sort or operator name together with the tag forms a unique and precise name for the object (sort or operator). Then if (for example) ω is an operator belonging to both A and B, ω will appear once in A + B if ω has the same tag (theory of origin) in both A and B; otherwise ω of A and ω of B are really different operators which just happen to have the same name, and A + B should include both. The language ι (Iota) [Nakajima, Honda and Nakahara 1980] also uses tags (to qualify operator names).

Each theory is therefore represented in the semantics as a tagged theory (a theory where the names are all tagged). The tagged theories Nat and Char look like this, where tags appear as subscripts (assuming that Bool contains the operators true, false, not and ==):

$$
\begin{array}{l}
\text{Nat} = \ \underline{\text{sorts}}\ \text{nat}_{\text{Nat}},\ \text{bool}_{\text{Bool}} \\
\qquad \underline{\text{opns}}\ 0_{\text{Nat}} : \text{nat}_{\text{Nat}} \\
\qquad\qquad \text{succ}_{\text{Nat}} : \text{nat}_{\text{Nat}} \rightarrow \text{nat}_{\text{Nat}} \\
\qquad\qquad \leq_{\text{Nat}},\ =\!=_{\text{Nat}} : \text{nat}_{\text{Nat}}, \text{nat}_{\text{Nat}} \rightarrow \text{bool}_{\text{Bool}} \\
\qquad\qquad \text{true}_{\text{Bool}},\ \text{false}_{\text{Bool}} : \text{bool}_{\text{Bool}} \\
\qquad\qquad \text{not}_{\text{Bool}} : \text{bool}_{\text{Bool}} \rightarrow \text{bool}_{\text{Bool}} \\
\qquad\qquad =\!=_{\text{Bool}} : \text{bool}_{\text{Bool}}, \text{bool}_{\text{Bool}} \rightarrow \text{bool}_{\text{Bool}} \\
\qquad \underline{\text{eqns}}\ \ldots
\end{array}
$$

Char = sorts $char_{Char}$, $bool_{Bool}$
opns A_{Char}, ..., Z_{Char} : $char_{Char}$
is_vowel_{Char} : $char_{Char}$ -> $bool_{Bool}$
$==_{Char}$: $char_{Char}$,$char_{Char}$ -> $bool_{Bool}$
$true_{Bool}$,$false_{Bool}$: $bool_{Bool}$
not_{Bool} : $bool_{Bool}$ -> $bool_{Bool}$
$==_{Bool}$: $bool_{Bool}$,$bool_{Bool}$ -> $bool_{Bool}$
eqns . . .

Nat + Char is simply the set-theoretic union of these two tagged theories:

sorts nat_{Nat}, $char_{Char}$, $bool_{Bool}$
opns 0_{Nat} : nat_{Nat}
$succ_{Nat}$: nat_{Nat} -> nat_{Nat}
$\leq_{Nat}$, $==_{Nat}$: nat_{Nat},nat_{Nat} -> $bool_{Bool}$
A_{Char}, ..., Z_{Char} : $char_{Char}$
is_vowel_{Char} : $char_{Char}$ -> $bool_{Bool}$
$==_{Char}$: $char_{Char}$,$char_{Char}$ -> $bool_{Bool}$
$true_{Bool}$,$false_{Bool}$: $bool_{Bool}$
not_{Bool} : $bool_{Bool}$ -> $bool_{Bool}$
$==_{Bool}$: $bool_{Bool}$,$bool_{Bool}$ -> $bool_{Bool}$
eqns . . .

The remaining semantic operations are fairly simple and straightforward set-theoretic constructions.

It is necessary to keep track of the names of all subtheories of a theory; the apply operation and Clear's 's of T' notation (to refer to a sort or operation s in a subtheory T of the current theory) both require it. Adding this information to a tagged theory gives a based theory. The base is a subset of the global theory environment, mapping each subtheory name to the theory bound to that name. The addition of a base does not complicate the definition of the sum of two theories; the base of the sum is simply the union of the bases.

Def: A based theory is a pair <T,B> where T is a theory with tagged sorts and operators and B:theory-name⟶theory (the base) is a map containing the subtheories of T. <T,B> is normally written T_B.

Def: A based theory morphism $\sigma:\underline{T}_B \rightarrow \underline{T}'_{B'}$ (where $B \subseteq B'$) is a theory morphism $\sigma:\underline{T} \rightarrow \underline{T}'$ such that σ restricted to theories in B is the identity.

This notion of based theory should not be confused with Burstall and Goguen's [1980] notion, discussed in section V.2. Although the definitions are different, both kinds of based theories serve the same purpose (and in fact the two representations are isomorphic) so we use the same name to draw attention to this similarity.

The particular tags used are not important; all that matters is that the tags for two different sorts (or operators) which have the same name, are different. Thus, X146 and Y27 would serve as well as Bool and Nat above. Also (for example) succ and $\leq$ need not have the same tag. This fact will be useful in the semantics; it turns out to be inconvenient to tag sorts and operators with the name of their theory of origin.

2. Semantic operations

In this section the semantic operations which 'implement' the theory-building operations of Clear are defined. This forms the quintessence of Clear's semantics; the semantic equations given in section 4 serve only to attach a syntax to the operations defined here. The definitions depend heavily upon the special representation of based theories described in section 1; the objects defined in chapter II are used as well (signatures, equations, constraints) but their representations are not important. The definitions assume that the based theories to be put together are compatible. This will always be the case in practice because all available theories have been constructed using Clear.

Def: If $\underline{\Sigma}=\langle S,\Sigma\rangle$ and $\underline{\Sigma}'=\langle S',\Sigma'\rangle$ are tagged signatures then the union of $\underline{\Sigma}$ and $\underline{\Sigma}'$, written $\underline{\Sigma}\cup\underline{\Sigma}'$, is $\langle S\cup S',\tilde{\Sigma}\cup\tilde{\Sigma}'\rangle$ (where $\tilde{\Sigma}$ and $\tilde{\Sigma}'$ are the extensions of Σ and Σ' to indexed sets of operators over $S\cup S'$).

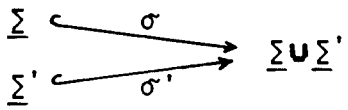
2.1. Combine

This implements the '+' theory-building operation of Clear.

combine : based-theory x based-theory $\longrightarrow$ based-theory

combine($\langle\underline{\Sigma},EC\rangle_B$, $\langle\underline{\Sigma}',EC'\rangle_{B'}$) = $\langle\underline{\Sigma}\cup\underline{\Sigma}',\overline{\sigma(EC)\cup\sigma'(EC')}\rangle_{B\cup B'}$

where σ and σ' are the signature inclusions



We will sometimes use '+' in the sequel rather than combine; this should cause no confusion.

The result has the sorts and operators of both theories, the closed union of the axioms (translated to give $\underline{\Sigma}\cup\underline{\Sigma}'$-equations and constraints), and the union of the two bases. Since $\underline{\Sigma}$ and $\underline{\Sigma}'$ are tagged signatures, $\underline{\Sigma}\cup\underline{\Sigma}'$ will treat shared sorts and operators properly.

2.2. Enrich

An enrichment consists of some new sorts, operators and equations. The enrich operation takes a based theory and an enrichment and produces the enriched based theory. Each new sort and operator must be given a unique tag, according to the discussion in the preceding chapter. This tagging is not done by the enrich operation itself; we require that new sorts and operators be given unique tags before they are used to enrich a theory. This is necessary because the arity of a new operator may include one of the new sorts, and this requires that the new sort be tagged. The tags are attached by the semantic equations (as part of the semantics of sort and operator declarations -- section 4.3).

enrich : based-theory x sort-set x operator-set x equation-set
$\longrightarrow$ based-theory

$$\text{enrich}(\ \langle \underline{\Sigma}, EC\rangle_B,\ S',\ \Sigma',\ E'\ ) = \langle \underline{\Sigma} \cup \langle S', \Sigma'\rangle, \overline{\sigma(EC) \cup E'}\rangle_B$$

where Σ' is indexed over sorts($\underline{\Sigma}$) $\cup$ S'
E' is a set of $\underline{\Sigma} \cup \langle S', \Sigma'\rangle$ -equations
and σ is the signature inclusion

$$\underline{\Sigma} \xhookrightarrow{\ \sigma\ } \underline{\Sigma} \cup \langle S', \Sigma'\rangle$$

As mentioned above, it is understood that S' and Σ' have already been given unique tags before enrich is applied.

2.3. Data enrich

When a theory is enriched by some new data, the axioms of the resulting theory contain the constraint that the enrichment is to be interpreted freely. Moreover, an equality predicate ==:s,s->bool for each new sort s is included. Otherwise the result is the same as for ordinary (non-data) enrich. We employ a model-theoretic approach to obtain the equations which specify the meaning of the new equality predicates.

Def: Suppose $\underline{\Sigma}$ is a tagged signature which includes the sort $bool_{Bool}$ and the operators $true_{Bool}, false_{Bool}: bool_{Bool}$, $\underline{A}$ is a $\underline{\Sigma}$-algebra, EC is a set of $\underline{\Sigma}$-equations and constraints, x is a new tag, S is a subset of the sorts of $\underline{\Sigma}$, and $s \in S$. Then:

- $\underline{\Sigma}^S_x$ is $\underline{\Sigma}$ with an additional operator $==_x: s,s \rightarrow bool_{Bool}$. $\underline{\Sigma}^S$ is defined similarly (i.e., an additional == operator for each sort in S).

- $\underline{A}^S_x$ is a $\underline{\Sigma}^S_x$-algebra just like $\underline{A}$ but with an operation == satisfying $==(a,b)$ = true iff $a=b$, for all $a,b \in A_s$. $\underline{A}^S_x$ is defined similarly.

- EC^S_x is the set of $\underline{\Sigma}^S_x$-equations and constraints given by M^*, where $M = \{\underline{A}^S_x \mid \underline{A} \in EC^*\}$.

If S is the set of new sorts and EC is the set of equations and constraints already in a theory, then EC^S_x includes EC as well as all the equations needed to define the new equality predicates on sorts in S.

data-enrich : based-theory x sort-set x operator-set
x equation-set x tag $\rightarrow$ based-theory

$$\text{data-enrich}(\ \langle \underline{\Sigma}, EC \rangle_B,\ S',\ \Sigma',\ E',\ x\ ) = \langle\ (\underline{\Sigma}enr)^{S'}_x,\ (\overline{ECenr \cup \langle F, 1_{\underline{\Sigma}enr} \rangle})^{S'}_x\ \rangle_{Benr}$$

where $\langle \underline{\Sigma}enr, ECenr \rangle_{Benr} = enrich(\langle \underline{\Sigma}, EC \rangle_B, S', \Sigma', E')$
and F is the theory inclusion

$$\langle \underline{\Sigma}, \emptyset \rangle \xhookrightarrow{F} \langle \underline{\Sigma}enr, \overline{E'} \rangle$$

data-enrich gives an error if $\underline{\Sigma}enr$ does not include the sort $bool_{Bool}$ and the operators $true_{Bool}, false_{Bool}: bool_{Bool}$.

The result is the same as the result of enrich, with the addition of an operator == for each new sort, the equations concerning those operators, and the data constraint $\langle F, 1_{\underline{\Sigma}enr} \rangle$ where F is the theory morphism describing the enrichment.

2.4. Derive

The derive operation is used to 'forget' sorts and operators of a theory, possibly renaming the ones remaining. The renaming is accomplished by a signature morphism which takes the new names into the old names. Given a $\underline{\Sigma}$-theory, a $\underline{\Sigma}'$-theory and a signature morphism $\sigma:\underline{\Sigma}\rightarrow\underline{\Sigma}'$, derive produces a theory with the signature and base of the $\underline{\Sigma}$-theory, and all the $\underline{\Sigma}$-equations and constraints which are satisfied in all models of the $\underline{\Sigma}'$-theory -- this turns out to be the inverse image under σ of the equations and constraints of the $\underline{\Sigma}'$-theory.

derive : based-theory x signature-morphism x based-theory
$\rightarrow$ based-theory

derive($\langle\underline{\Sigma},EC\rangle_B$, σ, $\langle\underline{\Sigma}',EC'\rangle_{B'}$) = $\langle\underline{\Sigma},\sigma^{-1}(EC')\rangle_B$

where $\sigma^{-1}(EC') = \{e \mid \sigma(e)\in EC'\}$

derive gives an error if σ is not a based theory morphism.

The result is a theory because of the following fact:

Fact: If EC is closed then $\sigma^{-1}(EC)$ is closed.

Proof: (outline of the proof in [Burstall and Goguen 1980])
$\sigma^{-1}(EC) = (EC^*|_{\underline{\Sigma}})^* = \sigma^{-1}(EC)^{**}$ via two applications of the Satisfaction Lemma.

Also, $EC \subseteq \sigma^{-1}(EC')$ since σ is a theory morphism.

Intuitively, the derive operation should satisfy the following law:

$\underline{A}'$ is a model of $\underline{T}'$ iff $\underline{A}'|_{sig(\underline{T})}$ is a model of derive($\underline{T}$,σ,$\underline{T}'$)

The 'forward' implication (=>) follows by the proof of the previous fact ($\sigma^{-1}(EC) = (EC^*|_{sig(\underline{T})})^*$, so $EC^*|_{sig(\underline{T})} \subseteq \sigma^{-1}(EC)^*$). Unfortunately, the reverse (<=) does not hold. Consider the example:

```
const AB =
  enrich Bool by
    data sorts ab
         opns a, b : ab    enden
```

```
const ABC =
  enrich AB by
    opns c : ab    enden

const AC =
  derive sorts ac
         opns a, c : ac
    using Bool
    from ABC
    by ac is ab    endde
```

The theory ABC has two models (up to isomorphism). Both models have two elements in the carrier for sort ab; one model satisfies a=c≠b and the other satisfies a≠c=b. But AC has an infinity of non-isomorphic models. The problem is that the inverse image of the data constraint on sort ab of ABC is empty, so sort ac of AC is unconstrained. It seems that this slightly unpleasant situation can be put right by giving a somewhat more elaborate definition of data constraints. But it is not yet clear that this is the right way to handle the problem, and data constraints are already complex enough. So we ignore this complication for now; although derive does not have all the properties we want, in most cases this will not be a problem.

2.5. Apply

Apply defines the meaning of applying a theory procedure to its arguments. A procedure is represented as a based theory (the procedure) together with a list of based theories (the metasorts). This is the first argument of apply; the second is a list of (based-theory x signature-morphism) -pairs (actual parameter x fitting morphism). The third argument is the tag to be attached to the 'new' sorts and operators.

$$\text{apply} : (\text{based-theory} \times \text{based-theory}^*) \times (\text{based-theory} \times \text{signature-morphism})^* \times \text{tag} \longrightarrow \text{based-theory}$$

$$\begin{bmatrix}\text{procedure}\\ \text{parameters}\end{bmatrix}$$

The definition of apply uses two auxiliary functions. The first applies a signature morphism $\sigma:\underline{\Sigma}\longrightarrow\underline{\Sigma}'$ to a theory $\underline{T}$ with a signature which includes $\underline{\Sigma}$; the sorts and operators in $\underline{T}$ but not in $\underline{\Sigma}$ are not

affected. This is used to apply a fitting morphism to a procedure, and is also useful in defining the second auxiliary function.

_ **altered by** _ : theory x signature-morphism $\longrightarrow$ theory

Suppose $\underline{\Sigma} = \langle S,\Sigma\rangle$, $\underline{\Sigma}A = \langle SA,\Sigma A\rangle$, $\underline{\Sigma}B = \langle SB,\Sigma B\rangle$, $\langle f,g\rangle = \sigma:\underline{\Sigma}A\longrightarrow\underline{\Sigma}B$ and $\underline{\Sigma}A \subseteq \underline{\Sigma}$. Then:

$\langle\underline{\Sigma},EC\rangle$ **altered by** $\sigma = \langle\underline{\Sigma}',\overline{\sigma'(EC)}\rangle$

where $\underline{\Sigma}'$ and σ' are constructed as follows:

for $s\in S$, let $f'(s) = \begin{cases} f(s) & \text{if } s\in SA \\ s & \text{otherwise} \end{cases}$

let $S' = \{f'(s) \mid s\in S\}$

for $u\in S'^{*}$, $v\in S$ and $\omega\in\Sigma_{uv}$,
let $g'_{uv}(\omega) = \begin{cases} g_{uv}(\omega) & \text{if } \omega\in\Sigma A_{uv} \\ \omega & \text{otherwise} \end{cases}$

for $u'\in S'^{*}$ and $v'\in S'$,

let $\Sigma'_{u'v'} = \bigcup_{u,v\in I} \{g'_{uv}(\omega) \mid \omega\in\Sigma_{uv}\}$
where $I = \{u,v\in S \mid f'^{*}(uv)=u'v'\}$

then $\underline{\Sigma}' = \langle S',\Sigma'\rangle$
and $\sigma':\underline{\Sigma}\longrightarrow\underline{\Sigma}' = \langle f',g'\rangle$

Informally, $\langle\underline{\Sigma},EC\rangle$ **altered by** σ just replaces the sorts and operators of $\underline{\Sigma}$ which are in $\underline{\Sigma}A$ by their images in $\underline{\Sigma}B$.

The second auxiliary function attaches a given new tag to all of the sorts and operators in a theory, excluding those sorts and operators which belong to a distinguished subsignature.

_ **retagged with** _ **preserving** _ : theory x tag x signature $\longrightarrow$ theory

$\langle\underline{\Sigma},EC\rangle$ **retagged with** x **preserving** $\underline{\Sigma}' = \langle\underline{\Sigma},EC\rangle$ **altered by** mtag

where mtag is a signature morphism which gives each of the sorts and operators in $\underline{\Sigma} - \underline{\Sigma}'$ the tag x

an error results if $\underline{\Sigma}' \not\subseteq \underline{\Sigma}$

Apply is now defined with the help of these two functions. The idea is to first attach the given new tag to each sort and operator in the procedure, excluding those belonging to a metatheory or base

theory. This is necessary so that (for example) the sort 'list' in the theory List(Bool) will always remain distinct from the sort 'list' in List(Nat). The fitting morphisms are then applied to change each reference to the metasort signature into the corresponding reference to a sort or operator in the signature of the actual parameter, and the base of the procedure is attached. Finally, the actual parameters are added using combine to give the result.

$$\text{apply}(\langle \underline{P}_{BP}, \langle \underline{\Sigma}M_1, ECM_1\rangle_{BM_1} \ldots \langle \underline{\Sigma}M_n, ECM_n\rangle_{BM_n}\rangle, \langle \underline{A}_1, m_1\rangle \ldots \langle \underline{A}_n, m_n\rangle, x)$$

$$= \underline{A}_1 + \ldots + \underline{A}_n + ((\underline{P} \textbf{ retagged with } x \textbf{ preserving } \underline{\Sigma}\text{old}) \textbf{ altered by } m_1 \cup \ldots \cup m_n)_{BP}$$

$$\text{where } \underline{\Sigma}\text{old} = \underline{\Sigma}M_1 \cup \ldots \cup \underline{\Sigma}M_n \cup \bigcup_{\langle N, TN\rangle \in BP} \text{signature}(TN)$$

apply gives an error if some $m_i : \langle \underline{\Sigma}M_i, ECM_i\rangle_{BM_i} \rightarrow \underline{A}_i$ is not a based theory morphism.

This construction is rather more elaborate than any of those given previously. In order to understand it, consider first the simple case in which all theories contain only sorts (no operators or equations) and the procedure has only one argument. For example:

P = sorts $bool_{Bool}$, m_M, nat_{Nat}, p_P base Bool, Nat

M = sorts $bool_{Bool}$, m_M base Bool

A = sorts $bool_{Bool}$, $char_{Char}$, a_A, a'_A base Bool, Char

σ = [$bool_{Bool} \mapsto bool_{Bool}$, $m_M \mapsto a_A$]

Now let us evaluate apply(<P,M>,<A,σ>,'J36'). The 'old' sorts upon which P was built ($\underline{\Sigma}$old) is:

sorts $bool_{Bool}$, m_M, nat_{Nat}

Retagging P while preserving $\underline{\Sigma}$old gives:

sorts $bool_{Bool}$, m_M, nat_{Nat}, p_{J36}

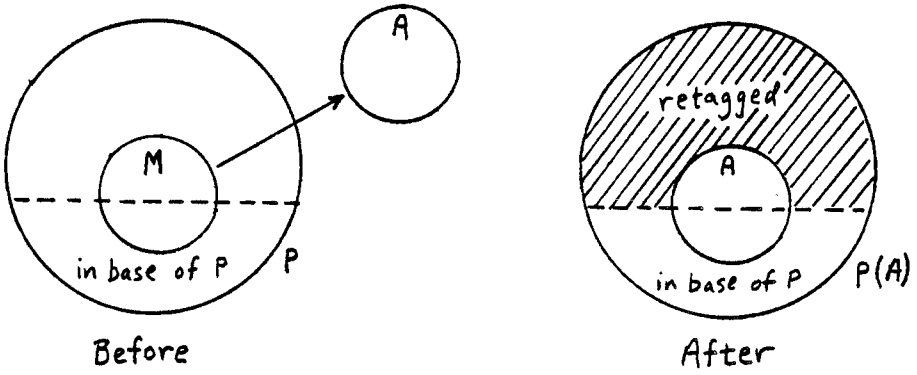
This is exactly P except that the sort p (which is 'new' in P) is tagged with J36 to ensure that it remains distinct from the sort p in the application of P to some other parameter. Applying the fitting morphism σ and reattaching the base of P gives:

sorts $bool_{Bool}$, a_A, nat_{Nat}, p_{J36} base Bool, Nat

and combining this with the actual parameter A gives the final

result:

sorts $bool_{Bool}$, a_{A}, nat_{Nat}, p_{J36}, $char_{Char}$, a'_{A}
base Bool, Nat, Char



For a more difficult example, consider the following Clear specification (assuming the usual specification of Bool):

```
const Natmod2 =
  enrich Bool by
    data sorts natmod2
         opns 0 : natmod2
              succ : natmod2 -> natmod2
         eqns succ(succ(n)) = n    enden

meta Triv = theory sorts element    endth

proc Pair(X:Triv) =
  enrich X + Bool by
    data sorts pair
         opns <_,_> : element,element -> pair
         eqns <a,b> = <b,a>    enden
```

Now the expression

Pair(Natmod2[element is natmod2])

should give the theory of (unordered) pairs of natural numbers modulo 2.

The denotation of Natmod2 is the following based theory (ignoring equations):

```
sorts bool_Bool, natmod2_Natmod2
opns 0_Natmod2, succ_Natmod2, ==_Natmod2, true_Bool, ...
eqns succ(succ(n)) = n
     not(true) = false
     . . .
base Bool, Natmod2
```

Triv gives just sorts $\text{element}_{\text{Triv}}$ with the empty base. Metatheories will be explained in section 3; briefly, the special thing about a metatheory is that its base excludes metatheories, itself included.

The procedure Pair has a denotation consisting of the following based theory together with Triv:

```
sorts element_Triv, pair_Pair, bool_Bool
opns <_,_>_Pair, ==_Pair, true_Bool, ...
eqns <a,b> = <b,a>
     not(true) = false
     . . .
base Bool
```

The environment contains Bool and Natmod2 (Triv and Pair are in the metatheory and procedure environments, respectively).

Referring to the definition of apply, the value of $\underline{\Sigma}$old is:

```
Σold = sorts element_Triv, bool_Bool
       opns true_Bool, false_Bool, not_Bool, ==_Bool
```

Retagging P (i.e., Pair without its base) with the new tag J37 while preserving $\underline{\Sigma}$old gives:

```
sorts element_Triv, pair_J37, bool_Bool
opns <_,_>_J37, ==_J37, true_Bool, ...
eqns . . .
```

Applying the fitting morphism $[\text{element}_{\text{Triv}} \mapsto \text{natmod2}_{\text{Natmod2}}]$ to this theory and reattaching the base of Pair yields:

```
sorts natmod2Natmod2, pairJ37, boolBool
opns <_,_>J37, ==J37, trueBool, ...
eqns . . .
base Bool
```

Finally, this is combined with the actual parameter Natmod2 to give the answer:

```
sorts natmod2Natmod2, pairJ37, boolBool
opns <_,_>J37, ==J37, trueBool, ..., 0Natmod2, ...
eqns succ(succ(n)) = n
     not(true) = false
     . . .
     <a,b> = <b,a>
base Bool, Natmod2
```

Note that applying a procedure P with formal parameter X and metasort M to an argument A using signature morphism σ is the same as rewriting the text of the procedure, with A substituted for X and all occurrences of sorts and operators in M translated using σ to the matching bits of A. For example, Pair(Natmod2[element is natmod2]) is equivalent to:

```
enrich Natmod2 + Bool by
  data sorts pair
       opns <_,_> : natmod2,natmod2 -> pair
       eqns <a,b> = <b,a>    enden
```

The definition of apply simulates this rewriting, using the trick of attaching fresh tags to the sorts and operators which are 'new in P' (i.e., not included in the base or metasorts) to distinguish them from the corresponding objects produced in a different application of the same procedure.

2.6. Copy

The copy operation is used to make a fresh copy of a theory, preserving a given set of subtheories.

copy : based-theory x based-theory x tag $\longrightarrow$ based-theory

copy($\underline{T}_B$, $\langle \underline{\Sigma}', EC' \rangle_{B'}$, x) = ($\underline{T}$ **retagged with x preserving** $\underline{\Sigma}'$)$_{B \cap B'}$

Given two based theories (the second theory is the combination of the subtheories to be shared), copy simply gives the new tag x to the sorts and operators of the first theory which are not in the second theory. The base of the result is the intersection of the bases of the argument theories.

3. Metatheories

In section I.1.2 the notion of a metatheory was informally introduced as a way of describing a class of theories (while an 'ordinary' theory describes a class of algebras). Metatheories are used to give the metasorts (requirements) of theory procedures. For example:

```
proc Set(X:Ident) =
  enrich X by . . .
```

Here, Ident is a metatheory (from section I.1.2) 'describing' all theories having at least one sort and an equivalence relation on that sort. Any such theory can be used as an argument of Set. In this section the relation between metatheories and ordinary theories is discussed. The semantics of [Burstall and Goguen 1980] did not treat this issue correctly, using ordinary theories to give procedure requirements (this error was only discovered during testing of the implementation of that semantics presented in chapter V).

It turns out that a metatheory is not a new kind of theory, but only an ordinary (based) theory used in a special way. The class of theories described by a metatheory M is the class containing only those based theories T for which a based theory morphism σ:M$\rightarrow$T exists. The definition of the apply operation in the last section uses the 'fitting morphism' (supplied by the user) to construct the result of a procedure application. But in order for this to work the metatheory M must be handled in a slightly different way from an ordinary theory; this is the reason why the meta construct is used to define a metatheory.

It is helpful to compare a sample metatheory with a similar ordinary theory. A typical metatheory is Ident, used above; this will be called Idmeta for now:

```
meta Idmeta =
  enrich Bool by
    sorts element
    opns ≡ : element,element -> bool
    eqns m≡m = true
         . . .              enden
```

This gives the following based theory:

```
sorts element_Idmeta, bool_Bool
opns =_Idmeta
     true_Bool, false_Bool, ...
eqns . . .
base Bool
```

Now consider a similar ordinary theory. Idconst 'loosely' specifies the set of algebras having one sort and an equivalence relation (see the theory Equiv in section I.1.2):

```
const Idconst =
  enrich Bool by
    sorts element
    opns ≡ : element,element -> bool
    eqns m≡m = true
         . . .              enden
```

which yields the based theory:

```
sorts element_Idconst, bool_Bool
opns =_Idconst
     true_Bool, false_Bool, ...
eqns . . .
base Idconst, Bool
```

The only apparent difference between these two based theories is that while Idmeta has a base consisting only of Bool, the base of Idconst contains Idconst itself as well. Consider the consequences if Idconst is used as the metasort of a theory procedure such as Set (called Setconst for now):

```
proc Setconst(X:Idconst) =
  enrich X by . . . enden
```

What are the possible actual parameter theories to which Setconst can be applied? Recall that a based theory morphism is used to fit an actual parameter to its corresponding metasort; the morphism goes from the metasort to the actual parameter. Since the base of the target of a based theory morphism must include the base of the source (and the morphism restricted to the base must be the

identity), the actual parameter must contain Idconst as a subtheory. In essence, the only theory Setconst can be applied to is Idconst itself. This is clearly neither intended nor desirable.

Now consider what happens if Idmeta is used as the metasort of Set:

```
proc Setmeta(X:Idmeta) =
  enrich X by . . . enden
```

Since the base of Idmeta contains Bool, any actual parameter of Setmeta must include Bool as a subtheory. But it need only match the rest of Idmeta; that is, it must include a sort with an equivalence relation. Suitable actual parameter theories and fitting morphisms are:

```
Nat [element is nat, ■ is ==]
Bool [element is bool, ■ is ==]
```

and many others.

In the example above, an ordinary theory (Bool) was included in a metatheory (Idmeta). In general, metatheories can be put together (with each other and with ordinary theories) using the same operations as for ordinary theories, since they are nothing more than a special kind of ordinary theory. When such a conglomerate is used as a metasort, any matching actual parameter must include all of the ordinary theories in the metasort (not just some theories which happen to resemble them), as well as sorts and operators which match those of the metatheories.

The only difference we have so far encountered between a metatheory and an ordinary theory is that the base of a metatheory does not include the metatheory itself (and thus does not include any other metatheories either). Unexpectedly, this is exactly the result if a parameterless theory procedure is used in place of a metatheory (this observation is due to R.M. Burstall):

```
proc Idproc() =
  enrich Bool by
    sorts element
    opns ■ : element,element -> bool
    eqns m■m = true
         . . .              enden

proc Setproc(X:Idproc()) =
  enrich X by . . . enden
```

Accordingly, a metatheory is treated in this semantics as a parameterless procedure. This is of course invisible to the user of the language. In the category-theoretic semantics to be given in chapter V, metatheories are treated as ordinary theories with altered bases (which gives the same result, since there a sort or operator may only be shared if it appears in a base theory). The semantics of metatheories in both cases is incorporated into the definition of environment operations.

A side-effect of the use of the apply operation to give the semantics of metatheories is that writing a metatheory twice gives two different theories; that is:

Idproc() + Idproc() ≠ Idproc()

This property is actually somewhat desirable for metatheories, since otherwise some extra mechanism must be added to the semantics of procedure declaration (in the next section) to keep separate multiple instances of the same metatheory as metasorts in a single procedure:

proc P(X:Idmeta,Y:Idmeta) = . . .

But this means that the theory-building operations do not respect shared sub-metatheories. It is difficult to decide if this last property (which also holds for metatheories in the category-theoretic semantics) is desirable or not. In section 5 a modification to the semantics is given which (among other things) causes theory-building operations to respect shared sub-metatheories.

The concept of a metatheory in Clear is similar to the notion of a sype in the language ι (Iota) [Nakajima, Honda and Nakahara 1980];

there too, a sype is not very different from an ordinary type, although it can be regarded as a higher order concept.

4. Semantic equations

Now we are ready to give the semantic equations for Clear, providing a 'syntactic dress' for the operations defined in section 2. The equations are divided into several levels. Level I deals with the semantics of sort and operator names, and depends on the notion of a dictionary. Level IIa contains the semantics of enrichments (sort and operator declarations, and equations), and level IIb describes signature changes (used in derive and in application of a theory procedure). Finally, level III gives the semantics of Clear's theory-building operations and procedure declarations, based on the semantic operations defined in section 2. It requires the definition of an environment. Most of the material in this section is taken from [Burstall and Goguen 1980]; differences are recorded in section V.4.

4.1. Dictionaries

In Clear the notation 's of T' (where s is a sort name and T is a theory name) may be used to refer to a sort which is included in a subtheory T of the current theory (similarly 'o of T' for operators). This may be necessary if the sort (or operator) name alone is ambiguous. A dictionary gives the correspondence between such an expression and the tagged sort or operator to which it refers.

Def: A dictionary is a pair of functions <sd,od> where

sd : sort-name x theory-name $\longrightarrow$ sort
od : operator-name x theory-name $\longrightarrow$ operator

The operation dict is used to construct a dictionary from a based theory; the resulting dictionary interprets sort and operator expressions referring to sorts and operators in that theory.

dict : based-theory → dictionary

dict($\langle \underline{\Sigma}, EC \rangle_B$) = <sd,od>

where sd(s,T) = the unique sort with name s in B(T)
and od(o,T) = the unique operator with name o in B(T)

sd(s,T) gives an error if T∉domain(B), or if there is not a unique sort called s in B(T) (similarly for od(o,T)).

Note that this definition means that the notation 's of T' (similarly 'o of T') may only be used to refer to theories which are in the base of the current theory.

4.2. Level I: Sorts, operators, terms

Syntactic categories

s : sort name	(lower case identifier)
o : operator name	(identifier or operator symbol)
T : theory name	(capitalised identifier)
sex : sort expression	
oex : operator expression	
x : variable	(identifier)
tex : term expression	

Syntax

sex ::= s ¦ s of T	e.g. element of X
oex ::= o ¦ o of T	e.g. not of Bool
tex ::= x ¦ oex(tex_1,...,tex_n)	e.g. or(p,q) (infixes etc. also permitted)

Values

d : dictionary
X : sort-indexed variable set
tm : term

Semantic functions

Sex : sort-expression → signature → dictionary → sort
Oex : operator-expression → signature → dictionary → operator
Tex : term-expression → signature → dictionary → sorted-variable-set → term

Semantic equations

$Sex[\![s]\!]\Sigma d$ = the unique sort in sorts(Σ) with name s
$Sex[\![s \text{ of } T]\!]\Sigma d = sd(s,T)$ where $\langle sd,od\rangle = d$

$Oex[\![o]\!]\Sigma d$ = the unique operator in operators(Σ) with name o
$Oex[\![o \text{ of } T]\!]\Sigma d = od(o,T)$ where $\langle sd,od\rangle = d$

$Tex[\![x]\!]\Sigma dX$ = x (a Σ-term on X)
$Tex[\![oex(tex_1,\ldots,tex_n)]\!]\Sigma dX =$
  let $\omega = Oex[\![oex]\!]\Sigma d$ in
  let $tm_1,\ldots,tm_n = Tex[\![tex_1]\!]\Sigma dX,\ldots,Tex[\![tex_n]\!]\Sigma dX$ in
    $\omega(tm_1,\ldots,tm_n)$ (a Σ-term on X)

4.3. Level IIa: Enrichments

Syntactic categories

sd : sort declaration
od : operator declaration
varl : variable list
eq : equation expression
enrb : enrichment body
enr : enrichment

Syntax

sd ::= s	e.g. nat
od ::= o: $sex_1,\ldots,sex_n$->sex	e.g. <: nat,nat->bool
varl ::= $x_{11},\ldots,x_{1n_1}:sex_1,\ldots,x_{m1},\ldots,x_{mn_m}:sex_m$	e.g. i,j:nat, p:bool
eq ::= all varl. tex_1 = tex_2	e.g. all p:nat. p+0=p

enrb ::= sorts $sd_1,\ldots,sd_m$
opns $od_1\ldots od_n$
eqns $eq_1\ldots eq_p$

enr ::= enrb | data enrb

e.g. data sorts bool
opns true: bool
false: bool
not: bool->bool
eqns all. not(true) = false
all p:bool. not(not(p)) = p

The notation

$o_1,\ldots,o_m$: $sex_1,\ldots,sex_n$->sex

is also allowed for operator declarations, defined by the obvious expansion into a sequence of declarations.

Semantic functions

Sd : sort-declaration → tag → sort
Od : operator-declaration → tag → signature → dictionary
→ (operator x arity)
Varl : variable-list → signature → dictionary
→ sorted-variable-set
Eq : equation-expression → signature → dictionary → equation
Enrb : enrichment-body → tag → signature → dictionary
→ (sort-set, (operator x arity) -set, equation-set)
Enr : enrichment → tag → based-theory → based_theory

Semantic equations

$Sd[\![s]\!]x = s_x$

$Od[\![o: sex_1,\ldots,sex_n\text{->}sex]\!]x\underline{\Sigma}d =$
$\underline{\text{let}}\ s_1,\ldots,s_n,s = Sex[\![sex_1]\!]\underline{\Sigma}d,\ldots.Sex[\![sex_n]\!]\underline{\Sigma}d,Sex[\![sex]\!]\underline{\Sigma}d\ \underline{\text{in}}$
$\langle o_x,\langle\langle s_1,\ldots\ s_n\rangle,s\rangle\rangle$

$$\begin{aligned}
&\text{Varl}[\![x_{11},\ldots,x_{1n_1}:\text{sex}_1,\ldots,x_{m1},\ldots,x_{mn_m}:\text{sex}_m]\!]\underline{\Sigma}d = \\
&\quad \underline{\text{let}}\ s_1,\ldots,s_m = \text{Sex}[\![\text{sex}_1]\!]\underline{\Sigma}d,\ldots,\text{Sex}[\![\text{sex}_m]\!]\underline{\Sigma}d\ \underline{\text{in}} \\
&\qquad \{\ \langle x_{11},s_1\rangle,\ldots,\langle x_{1n_1},s_1\rangle, \\
&\qquad\quad \cdot\ \cdot\ \cdot \\
&\qquad\quad \langle x_{m1},s_m\rangle,\ldots,\langle x_{mn_m},s_m\rangle\ \}
\end{aligned}$$

$$\begin{aligned}
&\text{Eq}[\![\underline{\text{all}}\ \text{varl}.\ \text{tex}_1 = \text{tex}_2]\!]\underline{\Sigma}d = \\
&\quad \underline{\text{let}}\ X = \text{Varl}[\![\text{varl}]\!]\underline{\Sigma}d\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ \text{tm}_1,\text{tm}_2 = \text{Tex}[\![\text{tex}_1]\!]\underline{\Sigma}dX,\text{Tex}[\![\text{tex}_2]\!]\underline{\Sigma}dX\ \underline{\text{in}} \\
&\qquad \langle X,\text{tm}_1,\text{tm}_2\rangle
\end{aligned}$$

$$\begin{aligned}
&\text{Enrb}[\![\underline{\text{sorts}}\ \text{sd}_1,\ldots,\text{sd}_m\ \underline{\text{opns}}\ \text{od}_1\ldots\text{od}_n\ \underline{\text{eqns}}\ \text{eq}_1\ldots\text{eq}_p]\!]x\underline{\Sigma}d = \\
&\quad \underline{\text{let}}\ S' = \{\text{Sd}[\![\text{sd}_1]\!]x,\ldots,\text{Sd}[\![\text{sd}_m]\!]x\}\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ \underline{\Sigma}' = \underline{\Sigma}\cup\langle S',\emptyset\rangle\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ \underline{\Sigma}' = \{\text{Od}[\![\text{od}_1]\!]x\underline{\Sigma}'d,\ldots,\text{Od}[\![\text{od}_n]\!]x\underline{\Sigma}'d\}\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ \underline{\Sigma}'' = \underline{\Sigma}'\cup\langle\emptyset,\underline{\Sigma}'\rangle\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ E' = \{\text{Eq}[\![\text{eq}_1]\!]\underline{\Sigma}''d,\ldots,\text{Eq}[\![\text{eq}_p]\!]\underline{\Sigma}''d\}\ \underline{\text{in}} \\
&\qquad \langle S',\underline{\Sigma}',E'\rangle
\end{aligned}$$

$$\text{Enr}[\![\text{enrb}]\!]x\underline{T} = \text{enrich}(\underline{T},\text{Enrb}[\![\text{enrb}]\!]x\,\text{signature}(\underline{T})\text{dict}(\underline{T}))$$

$$\begin{aligned}
&\text{Enr}[\![\underline{\text{data}}\ \text{enrb}]\!]x\underline{T} = \\
&\quad \text{data-enrich}(\underline{T},\text{Enrb}[\![\text{enrb}]\!]x\,\text{signature}(\underline{T})\text{dict}(\underline{T}),x)
\end{aligned}$$

4.4. Level IIb: Signature changes

Syntactic categories

sc : sort change
oc : operator change
sic : signature change

Syntax

$$\begin{aligned}
&\text{sc} ::= s_1\ \underline{\text{is}}\ \text{sex}_1,\ldots,s_n\ \underline{\text{is}}\ \text{sex}_n \\
&\text{oc} ::= o_1\ \underline{\text{is}}\ \text{oex}_1,\ldots,o_n\ \underline{\text{is}}\ \text{oex}_n \\
&\text{sic} ::= \text{sc},\text{oc}
\end{aligned}$$

e.g. element is nat,
order is < of Nat

Semantic functions

Sc : sort-change → signature → signature → dictionary
→ (sort → sort)
Oc : operator-change → signature → signature → dictionary
→ (operator → operator)
Sic : signature-change → signature → signature
→ dictionary → signature-morphism

Semantic equations

$Sc[\![s_1 \text{ is } sex_1,\ldots,s_n \text{ is } sex_n]\!]\Sigma\Sigma'd' =$
$\{ \langle Sex[\![s_1]\!]\Sigma d, Sex[\![sex_1]\!]\Sigma' d'\rangle,$
$\ldots$
$\langle Sex[\![s_n]\!]\Sigma d, Sex[\![sex_n]\!]\Sigma' d'\rangle \}$
where $d = \langle \emptyset,\emptyset\rangle$ (the null dictionary)

$Oc[\![o_1 \text{ is } oex_1,\ldots,o_n \text{ is } oex_n]\!]\Sigma\Sigma'd' =$
$\{ \langle Oex[\![o_1]\!]\Sigma d, Oex[\![oex_1]\!]\Sigma' d'\rangle,$
$\ldots$
$\langle Oex[\![o_n]\!]\Sigma d, Oex[\![oex_n]\!]\Sigma' d'\rangle \}$
where $d = \langle \emptyset,\emptyset\rangle$ (the null dictionary)

$Sic[\![sc,oc]\!]\Sigma\Sigma'd' =$
let $f = Sc[\![sc]\!]\Sigma\Sigma'd'$ in
let $g = Oc[\![oc]\!]\Sigma\Sigma'd'$ in
make_signature_morphism(Σ,f,g,Σ')
(where make_signature_morphism(Σ,f,g,Σ') is the signature morphism $\langle f,g\rangle:\Sigma\rightarrow\Sigma'$ with g_{us} the set of all pairs $\langle\omega,\nu\rangle\in g$ such that $\omega\in\Sigma_{us}$)

4.5. Environments

Reference has already been made in the definition of based theories to an environment of theories. In that case we were referring to the constant theory environment, only one of the three environments we will need. This is simply a map binding names to based theories. The other two environments store metatheory and procedure bindings; the metatheory environment is again a map from names to based theories, while in the procedure environment each name is bound to a value consisting of a based theory (the procedure) together with a list of based theories (the metasorts).

We define several operations on these environments. The

operation

bind : name x value x environment → environment

returns an environment with an added association between the name and value given (the type of value depends on the environment). Similarly,

bind : name-list x value-list x environment → environment

binds a list of names to the corresponding elements in a list of values.

The retrieve operation finds the value bound to a name in the combined constant theory and metatheory environment and constructs the corresponding based theory. Both environments must be checked because there is otherwise no way of telling whether a name refers to a constant theory or a metatheory. In case it refers to a metatheory, a new tag must be provided for use in retagging sorts and operators in the result. The procedure environment is accessed simply as a map, so no retrieve function is needed for it.

retrieve : name x const-environment x meta-environment x tag
→ based-theory

$$\text{retrieve}(N,\rho,\mu,x) = \begin{cases} \underline{T}_B \cup \langle N, T\rangle \text{ where } \underline{T}_B = \rho(N) & \text{if } N \in \text{domain}(\rho) \\ \text{apply}(\langle \mu(N), \langle\rangle, \langle\rangle, x) & \text{if } N \in \text{domain}(\mu) \end{cases}$$

retrieve gives an error if N is in neither or both domains

The apply operation is used to construct the result in the case of a metatheory, as discussed in section 3.

The restrict operation restricts an environment (or the mini-environment found in the base of a theory) to a subset of its domain.

restrict : environment x name-set → environment

This operation is useful for removing locally declared theories at the end of their scope from the bases of theories they have been used to build.

4.6. Level III: Theory building operations

Let $\mathbb{T}$ be a countably infinite list of distinct tags. This is where the tags required by the representation discussed in section 1 come from. The functions

hd : tag-list → tag
tl : tag-list → tag-list
split : tag-list x nat → (tag-list) -sequence

are defined by the following axioms:

$$\begin{aligned}
&\text{hd } [x_1\ x_2\ \ldots\ ] = x_1\\
&\text{tl } [x_1\ x_2\ \ldots\ ] = [x_2\ \ldots\ ]\\
&\text{split } [x_1\ x_2\ \ldots\ ] = [x_1\ x_{n+1}\ x_{2n+1}\ \ldots\ ],\\
&\qquad\qquad\qquad\qquad [x_2\ x_{n+2}\ x_{2n+2}\ \ldots\ ],\\
&\qquad\qquad\qquad\qquad\qquad\ \ldots\\
&\qquad\qquad\qquad\qquad [x_n\ x_{2n}\ x_{3n}\ \ldots\ ]
\end{aligned}$$

Syntactic categories

P : procedure name (capitalised identifier)
e : expression
spec : specification

Syntax

e ::= T | theory enr endth
| e1 + e2
| enrich e by enr enden
| derive enr using $e_1,\ldots,e_n$ from e by sic endde
| $P(e_1[sic_1],\ldots,e_n[sic_n])$
| let T = e_1 in e_2
| copy e using $e_1,\ldots,e_n$

spec ::= e | const T = e spec
| meta M = e spec
| proc $P(T_1:e_1,\ldots,T_n:e_n)$ = e spec

e.g. const Bool = theory ... endth
meta Triv = theory ... endth
proc String(X:Triv) = theory ... endth
String(Bool[element is bool])

Values

T : based theory
ρ : constant theory environment (name → based-theory)
μ : metatheory environment (name → based-theory)
π : procedure environment (name → based-theory x based-theory*)
L : tag-list

Semantic functions

E : expression → environment → metatheory-environment
→ procedure-environment → tag-list → based-theory
Spec : specification → environment → metatheory-environment
→ procedure-environment → tag-list → based-theory

Semantic equations

$E[\![T]\!]\rho\mu w L = \text{retrieve}(T,\rho,\mu,hd(L))$

$E[\![\underline{\text{theory}}\ \text{enr}\ \underline{\text{endth}}]\!]\rho\mu w L = Enr[\![\text{enr}]\!]hd(L)\Phi_I$
(Φ_I is the empty based theory)

$E[\![e_1 + e_2]\!]\rho\mu w L =$
$\quad \underline{\text{let}}\ L_1, L_2 = \text{split}(L,2)\ \underline{\text{in}}$
$\quad\quad E[\![e_1]\!]\rho\mu w L_1 + E[\![e_2]\!]\rho\mu w L_2$

$E[\![\underline{\text{enrich}}\ e\ \underline{\text{by}}\ \text{enr}\ \underline{\text{enden}}]\!]\rho\mu w L = Enr[\![\text{enr}]\!]hd(L)(E[\![e]\!]\rho\mu w tl(L))$

$E[\![\underline{\text{derive}}\ \text{enr}\ \underline{\text{using}}\ e_1,\ldots,e_n\ \underline{\text{from}}\ e\ \underline{\text{by}}\ \text{sic}\ \underline{\text{endde}}]\!]\rho\mu w L =$
$\quad \underline{\text{let}}\ L_1,\ldots,L_{n+1} = \text{split}(L,n+1)\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ T = E[\![e_1]\!]\rho\mu w L_1 + \ldots + E[\![e_n]\!]\rho\mu w L_n\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ T' = Enr[\![\text{enr}]\!]T\, hd(L_{n+1})\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ T'' = E[\![e]\!]\rho\mu w\, tl(L_{n+1})\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ \sigma = Sic[\![\text{sic}]\!]\text{signature}(T')\text{signature}(T'')\text{dict}(T'')\ \underline{\text{in}}$
$\quad\quad \text{derive}(T',\sigma,T'')$

$E[\![P(e_1[\text{sic}_1],\ldots,e_n[\text{sic}_n])]\!]\rho\mu w L =$
$\quad \underline{\text{let}}\ L_1,\ldots,L_{n+1} = \text{split}(L,n+1)\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ T'_1,\ldots,T'_n = E[\![e_1]\!]\rho\mu w L_1,\ldots,E[\![e_n]\!]\rho\mu w L_n\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ \langle T,\langle T_1,\ldots,T_n\rangle\rangle = w(P)\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ \sigma_1,\ldots,\sigma_n =$
$\quad\quad Sic[\![\text{sic}_1]\!]\text{signature}(T_1)\text{signature}(T'_1)\text{dict}(T'_1),$
$\quad\quad \ldots$
$\quad\quad Sic[\![\text{sic}_n]\!]\text{signature}(T_n)\text{signature}(T'_n)\text{dict}(T'_n)\ \underline{\text{in}}$
$\quad\quad \text{apply}(\langle T,\langle T_1,\ldots,T_n\rangle\rangle,\langle\langle T'_1,\sigma_1\rangle,\ldots,\langle T'_n,\sigma_n\rangle\rangle,hd(L_{n+1}))$

$E[\![\underline{\text{let}}\ T = e_1\ \underline{\text{in}}\ e_2]\!]\rho\mu w L =$
$\quad \underline{\text{let}}\ L_1, L_2 = \text{split}(L,2)\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ T = E[\![e_1]\!]\rho\mu w L_1\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ \rho' = \text{bind}(T,T,\rho)\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ T'_B = E[\![e_2]\!]\rho'\mu w L_2\ \underline{\text{in}}$
$\quad\quad T'\text{restrict}(B,\text{domain}(B)-\{T\})$

$E[\![\underline{\text{copy}}\ e\ \underline{\text{using}}\ e_1,\ldots,e_n]\!]\rho\mu w L =$
$\quad \underline{\text{let}}\ L_1,\ldots,L_{n+2} = \text{split}(L,n+2)\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ T = E[\![e]\!]\rho\mu w L_1\ \underline{\text{in}}$
$\quad \underline{\text{let}}\ T' = E[\![e_1]\!]\rho\mu w L_2 + \ldots + E[\![e_n]\!]\rho\mu w L_{n+1}\ \underline{\text{in}}$
$\quad\quad \text{copy}(T,T',hd(L_{n+2}))$

$$\mathrm{Spec}[\![e]\!]\rho\mu\varpi L = E[\![e]\!]\rho\mu\varpi L$$

$$\mathrm{Spec}[\![\underline{\mathrm{const}}\ T = e\ \mathrm{spec}]\!]\rho\mu\varpi L =$$
$$\quad \underline{\mathrm{let}}\ L_1, L_2 = \mathrm{split}(L,2)\ \underline{\mathrm{in}}$$
$$\quad \underline{\mathrm{let}}\ \rho' = \mathrm{bind}(T, E[\![e]\!]\rho\mu\varpi L_1, \rho)\ \underline{\mathrm{in}}$$
$$\quad\quad \mathrm{Spec}[\![\mathrm{spec}]\!]\rho'\mu\varpi L_2$$

$$\mathrm{Spec}[\![\underline{\mathrm{meta}}\ T = e\ \mathrm{spec}]\!]\rho\mu\varpi L =$$
$$\quad \underline{\mathrm{let}}\ L_1, L_2 = \mathrm{split}(L,2)\ \underline{\mathrm{in}}$$
$$\quad \underline{\mathrm{let}}\ \mu' = \mathrm{bind}(T, E[\![e]\!]\rho\mu\varpi L_1, \mu)\ \underline{\mathrm{in}}$$
$$\quad\quad \mathrm{Spec}[\![\mathrm{spec}]\!]\rho\mu'\varpi L_2$$

$$\mathrm{Spec}[\![\underline{\mathrm{proc}}\ P(T_1{:}e_1, \ldots, T_n{:}e_n) = e\ \mathrm{spec}]\!]\rho\mu\varpi L =$$
$$\quad \underline{\mathrm{let}}\ L_1, \ldots, L_{n+2} = \mathrm{split}(L, n+2)\ \underline{\mathrm{in}}$$
$$\quad \underline{\mathrm{let}}\ \underline{T}_1, \ldots, \underline{T}_n = E[\![e_1]\!]\rho\mu\varpi L_1, \ldots, E[\![e_n]\!]\rho\mu\varpi L_n\ \underline{\mathrm{in}}$$
$$\quad \underline{\mathrm{let}}\ \rho' = \mathrm{bind}(\langle T_1, \ldots, T_n\rangle, \langle \underline{T}_1, \ldots, \underline{T}_n\rangle, \rho)\ \underline{\mathrm{in}}$$
$$\quad \underline{\mathrm{let}}\ \underline{T}'_B = E[\![e]\!]\rho'\mu\varpi L_{n+1}\ \underline{\mathrm{in}}$$
$$\quad (\underline{\mathrm{let}}\ \varpi' = \mathrm{bind}(P, \langle \underline{T}'_{\mathrm{restrict}(B, \mathrm{domain}(B) - \{T_1, \ldots, T_n\})},$$
$$\qquad\qquad\qquad \langle \underline{T}_1, \ldots, \underline{T}_n\rangle\rangle, \varpi)\ \underline{\mathrm{in}}$$
$$\quad\quad \mathrm{Spec}[\![\mathrm{spec}]\!]\rho\mu\varpi' L_{n+2})\ \underline{\mathrm{if}}\ \{T_1, \ldots, T_n\} \subseteq \mathrm{domain}(B)\ \underline{\mathrm{else}}\ \mathrm{error}$$

The denotation of a specification spec in the initial environments ρ, μ, ϖ is then given by the value of $\mathrm{Spec}[\![\mathrm{spec}]\!]\rho\mu\varpi\mathbb{T}$ (recall that $\mathbb{T}$ is an infinite supply of distinct tags).

Consider the following procedure declaration:

```
proc Silly(X:Triv) = Bool
```

Because the body of this procedure does not include its metasort, the final equation above yields an error. An earlier version of the semantics (see [Sannella 1981]) did not produce an error in such cases, treating the above declaration as equivalent to:

```
proc Silly(X:Triv) = Bool + X
```

5. A 'nonprolific' semantics

The semantic equations in the last section complete a new semantics for Clear which yields exactly the same denotation for any specification as the semantics given by Burstall and Goguen [1980] (except for corrections to minor errors and the new metatheory notion). Although the language it defines is a convenient tool for writing specifications, it possesses at least one very annoying characteristic, as described below. A revised semantics without this characteristic is described here; only a few changes to the existing semantics are required. This demonstrates how easily the semantics can be changed to accommodate new features, as well as providing the semantics for a useful new version of Clear.

An essential feature of Clear is the fact that different theories (say, A and B) can share subtheories (say Bool) so that the combination A + B has only one copy of Bool. But consider the following specification:

const A = enrich Set(Bool[element is bool]) by ... enden
const B = enrich Set(Bool[element is bool]) by ... enden

Unfortunately, the combination A + B will have two copies of the theory Set(Bool[element is bool]). In general, each application of a procedure will give a fresh copy of the resulting theory and so in the specification above Set(Bool[element is bool]) is not a shared subtheory. This is called 'proliferation' by Burstall and Goguen [1981]. It is due to the definition of the semantic operation 'apply' in section 2; in particular to the use of the retag operation to give each of the new sorts and operators contributed by the procedure a new tag. Proliferation is clearly not desirable and therefore a 'nonprolific' semantics would be an improvement.

At first glance it might seem that the solution is simply to leave out the retagging of new sorts and operators, leaving the tags alone. But this is not quite right; the theory

Set(Bool[element is bool]) + Set(Nat[element is nat])

would then have just one copy of the sort 'set' (this would be in effect the theory of sets containing both bool and nat, so the term

{true} U {3} would be well-typed). The proper modification is to have apply change the tags of new sorts and operators in the procedure to a value which describes the application in question; this requires that tags like Set(Bool[element is bool]) be permitted as well as the usual names like Bool and J37. Here is the appropriate modification to the definition of apply (in section 2.5):

apply : (based-theory x based-theory*)
x (based-theory x signature-morphism)* $\rightarrow$ based-theory

apply($\langle \underline{P}_{BP}, \langle \underline{\Sigma}M_1, ECM_1 \rangle_{BM_1} \ldots \langle \underline{\Sigma}M_n, ECM_n \rangle_{BM_n} \rangle, \langle \underline{A}_1, m_1 \rangle \ldots \langle \underline{A}_n, m_n \rangle$)

$= \underline{A}_1 + \ldots + \underline{A}_n +$ ((P **retagged with** Ptag **preserving** $\underline{\Sigma}$old)
altered by $m_1 \cup \ldots \cup m_n)_{BP'}$

where $\underline{\Sigma}\text{old} = \underline{\Sigma}M_1 \cup \ldots \cup \underline{\Sigma}M_n \cup \bigcup_{\langle N, TN \rangle \in BP} \text{signature}(TN)$
$BP' = \{\langle N, TN \text{ **altered by** } m_1 \cup \ldots \cup m_n \rangle \mid \langle N, TN \rangle \in BP\}$
and Ptag is the tag
$\langle \underline{P}_{BP}, \langle \underline{\Sigma}M_1, ECM_1 \rangle_{BM_1} \ldots \langle \underline{\Sigma}M_n, ECM_n \rangle_{BM_n} \rangle, \langle \underline{A}_1, m_1 \rangle \ldots \langle \underline{A}_n, m_n \rangle$

This tag looks alarming, but it is simply the parameter list of the apply operation. Consequently, the result of apply will be the same when (and only when) it is applied to the same parameters. The above definition includes a modification to alter the theories in the base of the result according to the fitting morphisms. This is necessary for cases where the procedure includes an application of another procedure to the formal parameter, as changes below cause the result of that application to appear in BP.

The level III equation which gives the semantics of procedure application must now be altered to include the application in the base of the result (see section 4.6 — only the final line of that definition has been changed):

$$
\begin{aligned}
&\mathcal{E}[\![P(e_1[sic_1],\ldots,e_n[sic_n])]\!]\rho\mu\varpi L = \\
&\quad \underline{\text{let}}\ L_1,\ldots,L_{n+1} = \text{split}(L,n+1)\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ \underline{T}'_1,\ldots,\underline{T}'_n = \mathcal{E}[\![e_1]\!]\rho\mu\varpi L_1,\ldots,\mathcal{E}[\![e_n]\!]\rho\mu\varpi L_n\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ \langle \underline{T}, \langle \underline{T}_1,\ldots,\underline{T}_n \rangle \rangle = \varpi(P)\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ \sigma_1,\ldots,\sigma_n = \\
&\qquad\qquad \mathcal{S}ic[\![sic_1]\!]\text{signature}(\underline{T}_1)\text{signature}(\underline{T}'_1)\text{dict}(\underline{T}'_1), \\
&\qquad\qquad \ldots \\
&\qquad\qquad \mathcal{S}ic[\![sic_n]\!]\text{signature}(\underline{T}_n)\text{signature}(\underline{T}'_n)\text{dict}(\underline{T}'_n)\ \underline{\text{in}} \\
&\quad \underline{\text{let}}\ \underline{T}''_B = \text{apply}(\langle \underline{T}, \langle \underline{T}_1,\ldots,\underline{T}_n \rangle \rangle, \langle \langle \underline{T}'_1, \sigma_1 \rangle,\ldots,\langle \underline{T}'_n, \sigma_n \rangle \rangle)\ \underline{\text{in}} \\
&\quad \underline{T}''_B \cup \langle P(e_1[sic_1],\ldots,e_n[sic_n]), \underline{T}''_B \rangle
\end{aligned}
$$

This change is necessary because the 'apply' semantic operation requires that all shareable subtheories of a theory be recorded in the base of that theory (they are needed to form Σold), and the theory which results from application of a theory procedure to some arguments is shareable because of the previous changes.

A fortunate by-product of the above change is that metatheories automatically become nonprolific along with theory procedures, since the semantics of both use the same apparatus (recall that metatheories can be thought of as parameterless procedures). Because of this, the semantics of procedures must be changed slightly; the problem is that in a theory procedure such as the following:

proc P(X:Ident,Y:Ident) = . . .

the two metasorts merge into a single copy of Ident. The solution is to make a new copy of each metasort (excluding the subtheories in their bases) when a procedure is declared. The semantics of procedure declaration becomes:

$$
\begin{array}{l}
\mathrm{Spec}[\![\underline{\mathrm{proc}}\ P(T_1{:}e_1,\ldots,T_n{:}e_n) = e\ \mathrm{spec}]\!]\rho\mu w L = \\
\quad \underline{\mathrm{let}}\ L_1,\ldots,L_{n+2} = \mathrm{split}(L,n+2)\ \underline{\mathrm{in}} \\
\quad \underline{\mathrm{let}}\ \underline{T}_1,\ldots,\underline{T}_n = \mathrm{copy\_meta}(E[\![e_1]\!]\rho\mu w \mathrm{tl}(L_1),\mathrm{hd}(L_1)),\ldots, \\
\qquad\qquad \mathrm{copy\_meta}(E[\![e_n]\!]\rho\mu w \mathrm{tl}(L_n),\mathrm{hd}(L_n))\ \underline{\mathrm{in}} \\
\quad \underline{\mathrm{let}}\ \rho' = \mathrm{bind}(\langle T_1,\ldots,T_n\rangle,\langle \underline{T}_1,\ldots,\underline{T}_n\rangle,\rho)\ \underline{\mathrm{in}} \\
\quad \underline{\mathrm{let}}\ \underline{T}'_B = E[\![e]\!]\rho'\mu w L_{n+1}\ \underline{\mathrm{in}} \\
\quad (\underline{\mathrm{let}}\ w' = \mathrm{bind}(P,\langle \underline{T}'_B \mathrm{restrict}(B,\mathrm{domain}(B)-\{T_1,\ldots,T_n\}), \\
\qquad\qquad \langle \underline{T}_1,\ldots,\underline{T}_n\rangle\rangle,w)\ \underline{\mathrm{in}} \\
\qquad \mathrm{Spec}[\![\mathrm{spec}]\!]\rho\mu w' L_{n+2})\ \underline{\mathrm{if}}\ \{T_1,\ldots,T_n\}\ \ \mathrm{domain}(B)\ \underline{\mathrm{else}}\ \mathrm{error} \\
\\
\quad \underline{\mathrm{where}}\ \mathrm{copy\_meta}(\underline{T}_B,x) = \\
\qquad (\underline{T}\ \textbf{retagged with}\ x\ \textbf{preserving}\ \underline{\Sigma}\mathrm{old})_B \\
\\
\qquad \underline{\mathrm{where}}\ \underline{\Sigma}\mathrm{old} = \bigcup_{\langle N,TN\rangle \in B} \mathrm{signature}(TN)
\end{array}
$$

Level I of the semantic equations is concerned with providing a meaning for sort and operator expressions such as 's of T'. Only a slight modification is now necessary to extend the semantics to expressions like 's of P(A)'. To extend sort expressions (operator expressions are handled in exactly the same way) the level I BNF syntax must be augmented:

sex ::= s | s of T | s of $P(e_1[sic_1],\ldots,e_n[sic_n])$

The semantic equation for the new alternative is nearly identical to the one which handles 's of T':

Sex⟦s of $P(e_1[sic_1],\ldots,e_n[sic_n])$⟧⟦Σ⟧d =
sd(s,$P(e_1[sic_1],\ldots,e_n[sic_n])$) where <sd,od> = d

The notion of dictionary needs no change, provided that expressions of the form $P(e_1[sic_1],\ldots,e_n[sic_n])$ are permitted as theory names. The base of the result of a procedure application already includes bindings to such names, as a result of the earlier change to the level III equation giving the semantics of procedure application.

The modification just described has the defect that the procedure application in an expression 's of $P(e_1[sic_1],\ldots,e_n[sic_n])$' must be syntactically identical to the expression $P(e_1[sic_1],\ldots,e_n[sic_n])$ which originally 'generated' the required sort (and similarly for operators). Slightly better would be to bind the appropriate theory in the base of the result of a procedure application to a semantic object combining the denotations of the procedure P and each of the theory expressions e_j and signature changes sic_j. The semantic equations for sort and operator expressions would then need to determine the denotations of procedures, theory expressions and signature changes, requiring them to be supplied with the current environment of procedures and theories (which in turn requires these environments to be made available to all the semantic equations of levels IIa and IIb). The necessary changes are not given here; they are routine although widespread, affecting nearly all of the semantic equations.

6. A generalisation

In section I.1.3 two extensions to Clear were discussed (error Clear with error operators and error equations in addition to the usual (OK) operators and equations; and conditional Clear with conditional equations) and several more such extensions were mentioned briefly. It was revealed that Clear can be regarded as a family of languages, where the notions of signature, signature morphism, axiom, algebra and satisfaction are not necessarily as defined in chapter II but vary from one language to another. Any choice for these five notions is satisfactory as long as a few conditions hold (it must be possible to 'put together' signatures and the definitions must satisfy certain consistency conditions). Any such collection of notions is called an institution, and the semantics of (most of) Clear can be described without reference to a particular institution. This will be done in chapter V, where the notion of an institution will be formalised.

The semantics just described does not work under an arbitrary institution; it is a semantics of ordinary Clear (the language described in chapter I). Its advantage lies in being very concrete and easy to understand. But it is easy to see that the semantics does not depend at all on the definition of:

- Axioms: We require only the existence of a map $\sigma_{ax}:\Sigma\text{-axioms}\rightarrow\Sigma'\text{-axioms}$ for every signature morphism $\sigma:\Sigma\rightarrow\Sigma'$. The discussion of data constraints in section II.5 relies on axioms being equations, but the more abstract discussion in [Burstall and Goguen 1980] is equivalent and does not rely on the form of axioms.

- Algebras and satisfaction: The only references to algebras and satisfaction in the semantics are in the definition of the closure operation (on sets of equations and constraints), in the definition of data-enrich (where EC^S is EC together with all the equations which are true about the equality operators on sorts in the set S) and in the discussion which justifies the definition of derive. These depend not on the particular notion of algebra and satisfaction but only on the validity of the Satisfaction Lemma (section II.3).

So far, this is the same freedom as allowed by an institution; there the Satisfaction Lemma must hold as well. The difference is

that the semantics presented in this chapter does depend on the notions of signature and signature morphism, while an institution permits use of any cocomplete category of signatures. This dependency is a consequence of our use of the tagging trick in representing theories. But in fact the semantics does not rely on the exact definitions of signature and signature morphism, but only on the following features of their definitions:

- Signatures must be sets (or collections of sets). The definition of enrich and data-enrich here are dependent on the exact structure of signatures, but it would be easy to give appropriate definitions for any notion of signature. The equality operators added by data rely on the existence of sorts, but these are not a vital feature of the semantics and cannot be included for an arbitrary signature in the 'institutional approach' either.

- Signature morphisms must be functions (maps) between the source and target signatures.

In addition, the tagging trick for representing theories with sharing depends on the following:

- Enrichments must be inclusions (in the institutional approach enrichments may be arbitrary theory morphisms).

Section 2 of this chapter (defining the semantic operations) could easily be rewritten for Clear under any institution satisfying these restrictions. The result would not be very much different from what appears here; only the definitions of enrich and data-enrich would change noticeably (since the remaining definitions are in terms of operations like signature union and the image of a signature under a signature morphism). Sections 1 and 3 would remain unchanged, being independent of the definitions of signatures and their morphisms. The semantic equations of section 4 would need to be changed substantially, for the syntax of a language is naturally very dependent on the entities it manipulates. But the level III equations and the definition of environments would survive intact. The semantics is given here for the special case of ordinary Clear in order to make it easy to understand.

It is enlightening to see how restrictive the extra conditions on signatures, signature morphisms and enrichments are. Perhaps surprisingly, every one of the institutions which has ever (to my knowledge) been proposed for Clear satisfies these extra conditions.

- Error Clear: Signatures include an extra set containing error operators. Signature morphisms map sorts to sorts, OK-operators to OK-operators, and error-operators to error-operators. Error equations must be distinguished from OK-equations. Algebras and satisfaction are as described in section I.1.3; see also [Goguen 1978].

- Order-sorted Clear: The sort and operator sets of signatures have extra structure -- the sort set is a strict lower semilattice, and the operator set respects coercions between sorts. Signature morphisms must preserve this structure. Equations, algebras and satisfaction are as defined in [Goguen 1978a].

- Polymorphic Clear: The sort set of a signature contains sort generators -- a normal sort like nat is a nullary sort generator; list is a unary sort generator. These generators give rise to a (possibly infinite) set of sorts (sort terms, e.g. {nat, list(nat), list(list(nat)), ...}). Operators may be polymorphic, so their arities are tuples of sort terms (which may contain variables). Signature morphisms map sort generators to sort generators and operators to operators; they must preserve the structure of signatures. A polymorphic algebra has a carrier for every (variable-free) sort term and a function for every instance of a polymorphic operator. Equations may be polymorphic, in which case an algebra satisfies an equation if the equation is satisfied for every type instance.

Other examples are conditional Clear (section I.1.3), higher-order Clear (see [Dybjer 1981]), continuous Clear (see [Goguen, Thatcher, Wagner and Wright 1977]), and predicate-calculus Clear (see [Burstall and Goguen 1981]). The version of Clear whose implementation is described in chapter IV is a combination of error Clear, conditional Clear and predicate-calculus Clear, with some further extensions.

Are there any useful institutions which do not satisfy the extra conditions? It is not difficult to think of a cocomplete category which does not satisfy the conditions -- for example, the natural numbers form a cocomplete category, where there is a (unique) morphism $n \longrightarrow m$ iff $n \leq m$ (we ignore the fact that natural numbers can be represented as sets so that the extra conditions are satisfied) -- but it is hard to imagine a useful specification language using natural numbers for signatures. It may be that the greater generality of an institution is not useful in practice, but it is also possible that there is some undiscovered useful version of

Clear in which signatures and their morphisms do not satisfy our extra conditions.

There is at least one useful non-institution which satisfies our conditions. If signature morphisms in polymorphic Clear are generalised so that sort generators can map to sort terms containing variables (not just other sort generators) then signatures cannot be 'put together' in the required way (that is, the resulting category of signatures is not cocomplete) although all the conditions given above are still satisfied. This 'extended polymorphic Clear' seems more natural than ordinary polymorphic Clear. For example, if $\underline{T}$ is a theory of polymorphic lists (including the nullary sort generator 'nat' and the unary sort generator 'list') and $\underline{\Sigma}$ is a signature for stacks of natural numbers (including the nullary sort generator 'stacknat'), then in extended polymorphic Clear we can write:

$\underline{\text{derive}}\ \underline{\Sigma}\ \underline{\text{from}}\ \underline{T}\ \underline{\text{by}}\ \sigma$

where σ = [stacknat$\mapsto$list(nat), ...]. This is not allowed in ordinary polymorphic Clear. Burstall and Goguen's [1980] semantics could be modified to permit generalisation to extended polymorphic Clear (the category of signatures really need only have an initial object, coproducts and a funny kind of asymmetric pushout -- arbitrary colimits are not required) but much of its elegance would then be lost.

CHAPTER FOUR

AN IMPLEMENTATION OF CLEAR AND SOME SPECIFICATION EXAMPLES

In this chapter an implementation of Clear is discussed along with some of the specifications it has been used to process. This implementation is somewhat unusual in that it is (with the exception of a parser and a typechecker) a direct translation into HOPE of the denotational semantics of Clear described in the last chapter. This approach to language implementation is similar to that of Mosses [1976] who has developed a system which carries out the translation from denotational semantics to a lower-level language automatically. Although such an approach results in an implementation which may be inefficient (compared with a 'normal' implementation) it is nearly guaranteed to be correct because it is only a short step away from the formal definition of the language.

It is important to stress exactly what is meant by "an implementation of Clear". Before Clear was invented, in order to specify a problem we would have to write down a theory explicitly — for a large problem this is a long list of sorts, operators and axioms. Such a theory can be described in Clear in a highly structured way as the combination (using theory-building operations like combine and apply) of a number of small theories. The semantics of Clear specifies the correspondence between such a structured description and the theory it describes. An implementation of Clear is then a program which takes a Clear specification to the theory it denotes, checking in the process that the syntax and types are correct. Since the set of axioms in the resulting theory may be infinite, the program cannot represent it explicitly; such sets will be described using a very simple language. Although a data constraint gives rise to inequalities and an induction rule (section VI.3), the implementation does not perform the conversion.

An implementation of Clear is useful for a variety of reasons. First, when the implementation is a direct translation of the semantic definitions it can be used to debug the definitions

themselves; the semantics of any real language is large and complex enough that errors are bound to crop up. In fact, several minor errors were discovered in an earlier version of the semantics of chapter III during testing of its implementation, and the implementation of Clear's category-theoretic semantics (chapter V) uncovered a serious error in Burstall and Goguen's [1980] original semantics, as discussed in section III.3.

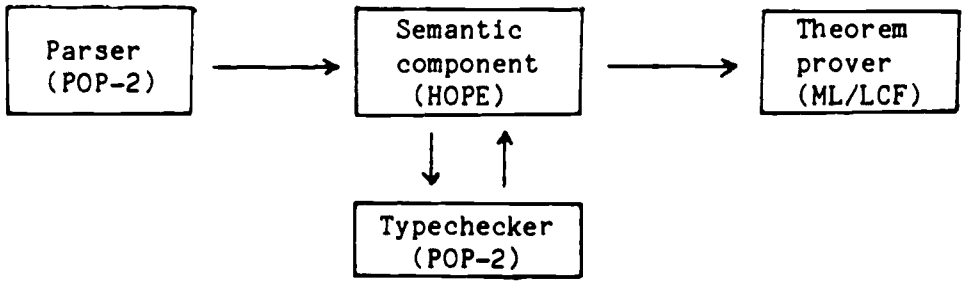
A second use for an implementation would be in checking specifications for syntactic and semantic errors. Although an important goal of any specification language is to permit theories to be easily described, mistakes are always easy to make. Some errors are difficult for an implementation to catch (and of course an implementation of the semantics cannot determine if a specification has the class of models intended by its author), but still it is comforting to know that a specification contains no glaring mistakes — this is similar to the peace of mind a HOPE programmer (or a programmer in any other strongly typed language) has when a program survives the typechecker's inspection without a fault being discovered.

A third use for an implementation is simply to produce denotations of specifications. These can be inspected by the user to find out whether the result is as expected, or else used by a theorem proving system (see chapter VI), a program development system (see chapter VII), a program verification system, or for any other purpose which requires specifications as input.

The Clear implementation is described in section 1. The intent was to provide a practical implementation capable of being used for the purposes described above. Some features are therefore supported which make specifications easier to write but are not mentioned in the semantics (errors, conditional equations, quantifiers and typechecking). These are provided with an informal semantics based on the semantic definitions of chapter III. The remainder of the chapter is devoted to three specification examples, all of them large enough to provide a challenge to the Clear system.

1. Implementation

The Clear implementation is composed of three parts: a parser, a typechecker, and a semantic component (a fourth part -- a theorem prover -- is discussed in chapter VI).



The parser is adapted from David MacQueen's parser for HOPE, written in POP-2. It parses the language described by the grammar of section III.4 (with a minor addition -- "T enriched by Enr" is permitted as an alternative to "enrich T by Enr") and also provides facilities for the declaration and use of 'distributed-fix' operators as in HOPE and OBJ [Goguen and Tardo 1979]. Distributed-fix operators are declared in the same way as normal operators, but with their special syntax displayed (surrounded by parentheses):

```
opns f : nat,nat -> nat
     (_ + _) : nat,nat -> nat
     (if _ then _ else _) : bool,nat,nat -> nat
eqns if n==0 then m+3 else (m+n) = f(n,m)
     . . .
```

As shown, such operators may be used in equations once they are declared. It is not possible to give a distributed-fix operator a special precedence; for this reason the parentheses in the left-hand side of the equation above are unavoidable, as + cannot be given a higher precedence than else. The name of a distributed-fix operator (for use in signature changes in derive's and procedure applications) is the leftmost identifier in its declaration (so + and if are the names of the operators declared above). A comment may appear anywhere in a specification preceeded by an exclamation mark (as in POP-2 and HOPE).

The typechecker is adapted from another piece of the HOPE system -- David MacQueen's polymorphic typechecker with facilities for

resolving occurrences of overloaded operators. Polymorphic types are not allowed in Clear, so the full facilities of the typechecker are not needed. But if the system is ever extended to allow polymorphism (as described in section III.6) the same typechecker can be used without modification. The Clear system does make use of the facilities for resolving overloaded operators; this allows the user to write equations without using qualified operators (such as "o _of_ T") except in the rare cases when the equation would otherwise be truly ambiguous. The user is also not required to supply the types of variables in equations, since the typechecker can determine them automatically (but variable declarations can be given if desired, and they are occasionally needed to help resolve overloading).

The semantic component consists of the semantic definition of Clear in chapter III translated into HOPE. This is the heart of the Clear system — the parser serves as a front end to the semantics, and the typechecker extends the semantics to provide automatic resolution of overloaded operators. Of course, both the parser and the typechecker also report any errors they discover, providing a valuable error-checking facility. Two versions of the system exist; one is prolific and the other nonprolific (incorporating the changes described in section III.5).

The translation from the mathematical definitions of chapter III to HOPE was a straightforward task. A function newname (which produces a unique name each time it is called) was added to HOPE to generate the tags used by the semantics. Strictly speaking, this addition renders HOPE nonapplicative but it is far more convenient than alternative ways of generating unique names. The only major problem to be solved in translating the definitions was how to represent and manipulate closed sets of equations and constraints in HOPE, given that:

- A closed set of equations and constraints will normally be infinite.
- The closure operation is defined model-theoretically.
- No complete proof system exists for Clear (see section VI.5).

Faced with such insurmountable difficulties we are obviously unable to give any explicit representation of a closed set of equations and constraints. Such sets can only be described using some language which must be left uninterpreted for the moment.

This matter is discussed at somewhat greater length in chapter VI, where the problem of interpreting such a representation (determining if a given equation is in the infinite closed set thus described) is addressed. A closed set of equations and constraints may be represented as an agglomerate, a value of a data type with several uninterpreted constructor functions. An examination of the semantics reveals that five constructors suffice for the representation of all necessary values. Two constructors are used to represent the result of the combine operation:

union : agglomerate x agglomerate → agglomerate
translate : signature-morphism x agglomerate → agglomerate

The first produces (an agglomerate representing) the closure of the union of two closed sets, and the second produces (a representation of) the closure of the set which results from applying a signature morphism to each equation and constraint in a set. The enrich operation needs the closure of a (finite) set of equations and constraints:

close : equation-set x constraint-set → agglomerate

Derive requires the inverse image of a set under a signature morphism:

inv-translate : signature-morphism x agglomerate → agglomerate

And data-enrich needs the result of adding to a set all equations which are true about the equality predicates on a set of sorts (see section III.2.3 for details -- for the purposes of the theorem prover described in chapter VI we record the signature inclusion $\sigma:\underline{\Sigma}\hookrightarrow\underline{\Sigma}^{S}_{x}$ rather than the set of sorts S and the tag x):

add-equality : signature-morphism x agglomerate → agglomerate

The Clear semantics program does not use these constructors

directly; instead it uses functions which apply the appropriate constructor and then simplify the result. Only a few simplifications are applied, such as:

$$translate(\sigma,translate(\sigma',A)) = translate(\sigma'.\sigma,A)$$

Care is taken not to disturb the structure of agglomerates, since the theorem prover described in chapter VI employs heuristics which make use of this structure. We postpone the presentation of the formal semantics of agglomerates until then; the informal meaning of each constructor as given above should be enough for now. An alternative name for an agglomerate would be structured theory, because an agglomerate displays (in 'flattened' form, with procedure applications removed) the structure of the original Clear specification. An ordinary (data) theory (chapter II) contains only a set of equations and constraints.

In developing the Clear system the intention was to provide a practical system for writing and checking specifications which could some day be incorporated in a program development or program verification system. It is vitally important that specifications be easy to write and understand, and that the specification language itself possess a well-defined semantics. Clear satisfies the latter goal, but not always the former; its limitations make it rather difficult to write some specifications. The system therefore supports several extensions which make specifications easier to write but are not mentioned in the semantics. As each one of these is described below it is provided with a (sometimes informal) semantics to justify its inclusion and explain its meaning.

Errors

Error operators and error equations are allowed along with ordinary (OK) operators and equations. This extension and its semantics has already been discussed in sections I.1.3 and III.6, and is discussed at greater length in [Goguen 1978].

Conditional equations

Besides the usual equations, conditional equations such as the following are allowed:

```
a is_in singleton(b) = false  if not(a==b)
```

The condition must be a bool-valued term. Semantically, the conditional equation $t_1=t_2$ if c is equivalent to the ordinary equation $cond(c,t_1,t_2)=t_2$, where cond:bool,s,s->s (for any sort s) is a 'hidden' operator defined by the equations:

```
cond(true,a,b) = a          cond(false,a,b) = b
```

Conditional equations have already been discussed in section I.1.3.

Multilevel binding

This is a convenience borrowed from HOPE (section A1.3). A variable may be bound to the value of any term in an equation to save writing the same term a second time, for example:

```
insert(R1 & insert(R,a,b), a, b) = R1
```

or alternatively:

```
insert(R1,a,b) = R1  where R1 = insert(R,a,b)
```

rather than insert(insert(R,a,b), a, b) = insert(R,a,b). This is a purely syntactic feature; the system removes such bindings immediately after parsing an equation containing them by replacing each occurrence of the variable with a copy of the term. A variable can only be so bound once in an equation, and may not itself appear in the term to which it is bound.

Don't care variables

This is another feature borrowed from HOPE. Any variable which appears only once in an equation may be replaced by an underscore to

save thinking of a name. The system replaces each underscore by a uniquely generated variable name. The following two equations are therefore equivalent:

```
isempty(push(_,_)) = false

isempty(push(v291,v292)) = false
```

Quantifiers

Equations may include existential and universal quantifiers, for example:

```
even(n) = exists m. (2 * m) == n

prime(n) = n>1 and forall m, p. (1<m and 1<p) --> not(m*p == n)
```

The condition of a quantifier must be a bool-valued term, and the result of a quantifier has type bool. It is easy to extend the formal notion of equation and satisfaction (section II.3) to equations with quantifiers.

The system does not permit the use of quantifiers within data enrichments. As noted in [Burstall and Goguen 1981], a data enrichment of T in which quantifiers are included does not always give rise to free extensions of models of T. If quantifiers are used only outside data enrichments this is not a problem.

The prohibition on quantifiers within data enrichments could be relaxed. Bergstra, Broy, Tucker and Wirsing [1981] describe a way of coding quantifiers in ordinary Clear with equations using an auxiliary operator. The only restriction is that quantification must be over a previous 'data' sort; that is, a quantifier within a data enrichment is safe as long as the quantified sort is not one of those being added in the current enrichment.

Furthermore, note that the conditional equation:

```
t = t'  if exists x. p(x)
```

is equivalent to:

t = t' if p(x)

if x does not occur in t or t'. Existential quantifiers of this special kind are therefore safe anywhere, even within the data enrichment which adds the quantified sort. Neither of these two exceptions to the exclusion of quantifiers within data enrichments is recognised by the system.

Typechecking

The user is not required to provide variable declarations in equations, or to use unambiguous operator names in equations and signature changes. As already discussed, the typechecker includes facilities for resolving overloaded operators which may be used to disambiguate almost every reference to an overloaded operator. The typechecker can also determine the types of variables automatically, although the user may supply them if desired.

The semantics must be changed slightly to take advantage of the facilities for disambiguation offered by the typechecker. An operator name no longer denotes a single operator; it denotes the set of all operators available with that name. The question of which operator in the set is the right one is postponed until an equation or signature change has been assembled. The typechecker is then applied; it selects the appropriate operator from each set based on the type information available from its context, yielding an unambiguous equation or signature change.

But what if the equation or signature change is truly ambiguous, and the typechecker is unable to select a single appropriate operator from a set of well-typed possibilities? The obvious course is to give an error message, telling the user that he must provide more information (a variable declaration for example). Unfortunately, there are some cases in which Clear does not provide any way of unambiguously referring to a certain operator (or sort). For example, in the theory

Set(Set(Nat[element is nat])[element is set]

there are two sorts called set and two operators

(_ U _):set,set->set. One of these sorts (and one of the operators) may be unambiguously referenced using the expression "set of Set(Nat[element is nat])" (respectively "U of Set(Nat[element is nat])") in nonprolific Clear, but there is no way of referring to the other sort (and operator). Another instance of the same problem occurs in the specification example in section 2.2. The solution adopted by the Clear system (the prolific version only) is to select the operator with the largest tag (that is, the most recently 'created' operator, since each tag includes a number and tags are issued in increasing numerical order) whenever there is a choice between several otherwise identical operators. The same policy is used to select a sort when the reference given is ambiguous. There is some logic in this choice; it should be easier to refer to a recently created object than to an older object with the same name, so in case of ambiguity it is natural to assume that the most recently created object was intended. In the example just discussed, the names set and U will refer to the otherwise unnameable sort and operator. A warning message is produced whenever this policy is applied.

The user is also not required to fully specify signature changes, since in almost every case a signature change is nearly the identity map, with just a few sorts and operators mapping onto different objects. The system will 'fill in' signature changes, mapping each sort and operator left unmentioned in the source signature onto the same object in the target signature; if this fails then it is mapped onto an object in the target signature with the same name but a different tag, using the disambiguation policy mentioned above if there is more than one choice. If there is still no match then the system reports an error.

Theory library

The Clear system includes a library of basic theories which the user may find useful in writing specifications. The library is listed in Appendix 2.

The Clear system occupies 149K words on a DEC KL-10 computer (the HOPE system itself occupies 66K words of this total, and the built-in theory library occupies another 32K words). The timing figures given after each example in the next section provide a measure of the system's performance. Parsing and typechecking typically account for about 6% of the processing time, with the remainder consumed by the semantic component. Specifications may be typed directly into the system or else read from files.

The system could be made much faster and smaller by recoding in a lower-level language (such as BCPL) with some attention paid to efficiency. It should be possible to process specifications at least as rapidly as a typical compiler can process programs, since there is nothing very complex about the computations required. The program is slow because it is written mostly in HOPE; apart from the speed of HOPE itself, the interfaces between the HOPE portions of the program and the remaining portions (parser, typechecker and theorem prover) contribute to its sluggishness.

2. Examples

The following subsections contain three specification examples which have been processed by the Clear implementation described in section 1. The first and third examples were processed by the nonprolific version and the second example by the prolific version of the program (but without the theorem prover discussed in chapter VI) which failed to detect any errors. This does not ensure that the specifications have the intended classes of models, but only that their syntax and types are correct.

The examples are presented only as sample specifications; although the problems are interesting in themselves, the discussion which accompanies each example concentrates on very briefly describing the specification and dealing with the problems of style which arise. The time which was required to process each specification is given to provide some indication of the system's performance.

2.1. Length of the longest upsequence

This problem comes from a set of specification and program development tasks [IFIP WG 2.1 1979] circulated prior to the December 1979 IFIP WG 2.1 meeting in Brussels. The following informal specification is taken from that source:

> Given a sequence of n integers, a_0, a_1, ... a_{n-1}, an upsequence is a subsequence which is ordered in ascending order. A subsequence is any subset of the original sequence where the original order is retained (there are 2^n possible subsequences). Ordered in ascending order means that no element of the upsequence has a right hand neighbor smaller than itself.
>
> Give an algorithm which, given a sequence, computes the length of its longest upsequence.
>
> Note that all subsequences of length 1 are upsequences by this definition.
>
> There may be more than one longest upsequence having the same length, for example the sequence (3,1,1,2,5,3) yields 4 for the maximum length, realised either by (1,1,2,5) or (1,1,2,3).

The statement of the problem asks for an algorithm, but a specification is given instead (an algorithm is given by Dijkstra

[1980]). The specification is quite straightforward; an upsequence is defined as an ordered subsequence, and then a hidden operator producing any of the longest upsequences of a sequence is used to specify the length of the longest upsequence. The operator is hidden because we do not wish to bias the specification toward solutions which generate longest upsequences; it is possible to determine the length of the longest upsequence without explicitly generating the upsequence itself.

```
proc Subsequence(X:Ident) =
  enrich Sequence(X) by
    opns (_ is_subsequence_of _) : sequence,sequence -> bool
    eqns s is_subsequence_of t = exists a, b, x, y.
                                      (a.b==s and x.y==t
                                       and a is_subsequence_of x
                                       and b is_subsequence_of y)
         empty is_subsequence_of _ = true
         s is_subsequence_of empty = s==empty
         unit(a) is_subsequence_of t =
                          exists x, y. (x.unit(a).y==t)    enden

proc Upsequence(X:POSet) =
  enrich Subsequence(X) by
    opns (_ is_ordered) : sequence -> bool
         (_ is_upsequence_of _) : sequence,sequence -> bool
    eqns s is_ordered = forall x, a, y, b, z.
                        (x.unit(a).y.unit(b).z==s --> a=<b)
         s is_upsequence_of t = s is_subsequence_of t
                                and (s is_ordered)    enden


proc LongestUpseqLength(X:POSet) =
  let LongestUS =
    enrich Upsequence(X) by
      opns longest_upseq : sequence -> sequence
      eqns length(p)=<length(longest_upseq(s)) = true
                                          if p is_upsequence_of s
           longest_upseq(s) is_upsequence_of s = true    enden in
  derive opns longest_upseq_length : sequence -> nat
    using Upsequence(X)
    from
      enrich LongestUS by
        opns longest_upseq_length : sequence -> nat
        eqns longest_upseq_length(s) = length(longest_upseq(s))
                                                    enden endde
```

This procedure may now be applied to (for example) the theory of natural numbers (which includes =<) to specify the length of the longest upsequence of a sequence of natural numbers:

```
LongestUpseqLength(Nat[element is nat])
```

Processing time: 1.65 minutes.

2.2. Lexical analysis problem

The following problem comes from the same source as the problem in the last section (see [IFIP WG 2.1 1979]) and the informal specification below is taken from there. The problem bears some resemblance to a part of the well-known 'Telegram problem' due to Henderson and Snowdon [1972] but is slightly simpler.

> A line consists of a sequence of characters composed of letters and blanks only. A word is a sequence of letters delimited by blanks or the end of the line. The parse of a line is the sequence of words, in order, contained in the line. Give the algorithm for obtaining the parse of a line, given the line.

Again, a specification is given for the problem rather than an algorithm. The specification relies heavily on the notion of a regular expression and the set of strings described by a regular expression (see [Hopcroft and Ullman 1979]). Regular expressions are used as a tool to specify the action of the 'parser'.

```
meta Classify =
  enrich Triv + Bool by
    sorts type
    opns (_ isa _) : element,type -> bool    enden
```

Permissible parameters for RegExpr will be theories describing a relation between objects and a set of basic types. The result of applying RegExpr to such a theory is the theory of regular expressions over the given types, providing a way of describing sequences of objects using 'complex' types.

```
proc RegExpr(X:Classify) =
  let RE =
    enrich X by
      data sorts regexpr
           opns empty : regexpr
                (' _ ') : type -> regexpr
                (_ U _), (_ . _) : regexpr,regexpr -> regexpr
                (_ *) : regexpr -> regexpr
           eqns e* = empty U (e.(e*))    enden in
  enrich RE + Sequence(X) by
    opns (_ isa _) : sequence,regexpr -> bool
    eqns s isa empty = s==empty
         unit(a) isa 't' = a isa t
         s isa 't' = false  if not(length(s)==1)
         s isa (e1 U e2) = (s isa e1) or (s isa e2)
         s isa (e1 . e2) =
               exists s1,s2. (s==(s1.s2) and (s1 isa e1)
                              and (s2 isa e2))    enden
```

CharacterClassify describes a classification of characters into two disjoint types: separators (blanks) and letters (everything else). The procedure application:

```
RegExpr(CharacterClassify[element is character])
```

gives the theory of regular expressions over these types. One such regular expression is:

```
('letter' *) U ('separator' *)
```

denoting all sequences which contain either letters or separators but not both. (The operator U is used rather than the usual + because + is a Clear keyword.)

```
const CharacterClassify =
  let Type =
    enrich Bool by
      data sorts type
           opns letter, separator : type    enden in
  enrich Type + Character by
    opns (_ isa _) : character,type -> bool
    eqns c isa separator = c==blank
         c isa letter = not(c==blank)    enden
```

WordsandGaps defines two special regular expressions which will be useful in specifying the parser.

```
const WordsandGaps =
  enrich RegExpr(CharacterClassify[element is character]) by
    opns word, gap : regexpr
    eqns word = 'letter' . ('letter' *)
         gap = 'separator' . ('separator' *)    enden
```

The specification of the parser below is simple and direct. Gaps in a sequence act as separators where the result is the concatenation of the parses of the two halves. A sequence without a gap is either a word (which parses to the unit sequence of words containing the word itself) or empty. This specification was processed by the prolific version of the Clear implementation because it relies on the 'largest-tag' method (discussed earlier) for disambiguation of a reference to the sort 'sequence'.

```
const Parse =
  enrich Sequence(WordsandGaps[element is sequence]) by
    opns parse : sequence of WordsandGaps -> sequence
                    ! result sort resolved by the disambiguation
                    ! method discussed in section 1 (Typechecking)
    eqns parse(x.g.y) = parse(x).parse(y)  if g isa gap
         parse(empty) = empty
         parse(x) = unit(x)  if x isa word    enden
```

Processing time: 1.04 minutes.

Regular expressions seem to be very useful in the specification of problems of this kind as they provide quite a high-level way of describing sequences; this permits very elegant specifications of sequence-manipulation operators (such as parse above). The idea of using regular expressions in Clear specifications is due to R.M. Burstall.

2.3. Polymorphic type checking

The specification below describes a polymorphic typechecker for a simple applicative language. Such typecheckers are used in the implementation of HOPE (appendix 1) and ML [Gordon, Milner and Wadsworth 1979]. Informally, the problem is as follows: given an expression exp in the language Exp generated by the following grammar (where x is any identifier, function application is denoted

by juxtaposition, and fix x.e is the least fixed-point of λx.e):

```
e ::= x | (e e') | if e then e' else e" | lambda x.e
      | fix x.e | let x=e in e'
```

with some predefined identifiers (of predefined types), assign a polymorphic type to every subexpression of exp so that the result is well-typed; if no well-typing exists then return an error. The notion of a well-typed expression is defined in section 3 of [Milner 1978] and depends on the definitions of several subsidiary notions so it is not reproduced here except in the specification itself. A polymorphic type is any of the following:

- a basic type (e.g. bool)
- a type variable
- $\alpha \rightarrow \beta$, where α and β are polymorphic types.

Given the following predefined identifiers:

b : bool f : $\alpha \rightarrow \alpha$ n : num m : num

this Exp expression is well-typed:

```
(let g:α→α = f:(α→α)→(α→α) (f:α→α) : α→α in
    (if g:bool→bool (b:bool) : bool
     then g:num→num (n:num) : num
     else m:num) : num
 ) : num
```

but this expression is not:

```
(if b:bool then f:bool→bool (b:bool) : num
           else m:num) : num
```

(f:bool→bool applied to b:bool gives a result of type bool, not num).

The language Exp and the definition of polymorphic types are rather simpler than a real language and its types would be. There is no provision for tupling (and so functions always have one argument), no type constructors (such as list, which can be used to

construct types like list(α), list(bool), and list(list(α)→β)) and no constants (but we imagine instead that some identifiers are bound in advance to certain constant values). But the language as it stands is large enough to expose the main problems arising in a polymorphic typechecker for a larger language.

The specification is more or less a direct translation into Clear of section 3 of [Milner 1978]. Some explanation of the notions defined by the specification is given below, but the careful reader is encouraged to refer to [Milner 1978] for more background and motivation. This is the largest specification which has so far been processed by the Clear system. Some measure of the usefulness of the system is given by the fact that it found 19 errors in successive versions of the specification. The present version may still contain some semantic errors, but at least it is better than the first version of the specification.

The typechecking function will be defined 'implicitly'; once the notion of a well-typed expression has been specified it is enough to say that for any expression typecheck assigns a well-typing if one exists. The specification of well-typed requires a number of prior notions; the language Exp and a theory of types and typed expressions is followed by the definition of technical notions concerning type variable instantiation.

The specification begins with a definition of the abstract syntax of the language Exp. After giving the theory of identifiers (by a loose specification -- any set with an equivalence relation will do) we specify Exp's syntax using distributed-fix operators for readability. All Exp keywords are capitalised to avoid conflicts with Clear syntax.

```
const Id =
  enrich Bool by
    sorts id
    opns (_ == _) : id,id -> bool
    eqns x==x = true
         x==y = y==x
         x==y and y==z --> x==z = true    enden
```

```
const Expr =
  enrich Id by
    data sorts expr
         opns (VAR _) : id -> expr
              (APPLY _ TO _) : expr,expr -> expr
              (IF _ THEN _ ELSE _) : expr,expr,expr -> expr
              (LAMBDA _ . _), (FIX _ . _) : id,expr -> expr
              (LET _ BE _ IN _) : id,expr,expr -> expr    enden
```

Next we specify polymorphic types. It is assumed that we are given an arbitrary set of basic types which includes BOOL; therefore the theory of basic types is just like the theory of identifiers Id above except for the name of the sort and the addition of a distinguished element called BOOL. Likewise, type variable names are arbitrary and so again we use Id with a change of sort name. Then a type is defined to be either a basic type, or a type constant, or an 'arrow' type $\alpha \longrightarrow \beta$ where α and β are types.

```
const BasicType =

  let T =
    derive sorts basictype
           opns (_ == _) : basictype,basictype -> bool
    using Bool
    from Id
    by basictype is id    endde in

  enrich T by
    opns BOOL : basictype    enden


const TypeVar =
  derive sorts typevar
         opns (_ == _) : typevar,typevar -> bool
  using Bool
  from Id
  by typevar is id    endde


const Type =
  enrich BasicType + TypeVar by
    data sorts type
         opns constant : basictype -> type
              var : typevar -> type
              (_ ---> _) : type,type -> type    enden
```

A typed expression is an expression of Exp with types assigned to

all its subexpressions. The easiest way to define typed expressions is to repeat the specification of Exp syntax, adding slots for the insertion of type information. Initial keywords are prefixed with T to avoid conflict with the distributed-fix operators declared in the theory Expr; the parser does not permit two distributed-fix operators having the same initial keyword but different subsequent syntax. A operator typeof giving the (top-level) type of a typed expression is defined for the convenience of later parts of the specification.

The theory TypedExprEq defines another operator which will be convenient later. It determines if an expression is identical to a typed expression, forgetting about types.

```
const TypedExpr =
  enrich Id + Type by
    data sorts typedexpr
         opns (TVAR _ | _) : id,type -> typedexpr
              (TAPPLY _ TO _ | _) :
                  typedexpr,typedexpr,type -> typedexpr
              (TIF _ THEN _ ELSE _ | _) :
                  typedexpr,typedexpr,typedexpr,type -> typedexpr
              (TLAMBDA _ | _ . _ | _), (TFIX _ | _ . _ | _) :
                  id,type,typedexpr,type -> typedexpr
              (TLET _ | _ BE _ IN _ | _) :
                  id,type,typedexpr,typedexpr,type -> typedexpr
              typeof : typedexpr -> type
         eqns typeof(TVAR _|t) = t
              typeof(TAPPLY _ TO _|t) = t
              typeof(TIF _ THEN _ ELSE _|t) = t
              typeof(TLAMBDA _|_ . _|t) = t
              typeof(TFIX _|_ . _|t) = t
              typeof(TLET _|_ BE _ IN _|t) = t    enden

const TypedExprEq =

  let TEE =
    enrich TypedExpr + Expr by
      opns (_ == _) : expr,typedexpr -> bool
           forget : typedexpr -> expr
      eqns forget(TVAR x|_) = VAR x
           forget(TAPPLY a TO b|_) = APPLY forget(a) TO forget(b)
           forget(TIF a THEN b ELSE c|_) =
                       IF forget(a) THEN forget(b) ELSE forget(c)
           forget(TLAMBDA x|_ . a|_) = LAMBDA x . forget(a)
           forget(TFIX x|_ . a|_) = FIX x . forget(a)
           forget(TLET x|_ BE a IN b|_) =
                       LET x BE forget(a) IN forget(b)
           e==te = e==forget(te)    enden in
```

```
derive opns (_ == _) : expr,typedexpr -> bool
  using TypedExpr, Expr
  from TEE    endde
```

A (typed) prefix is a sequence of items of the form let x|t, fix x|t or lambda x|t where x is a variable and t is a type. Initial keywords are prefixed with P to avoid conflict with Expr and TypedExpr. A prefix can be thought of as a list of bound variables ('most local' bindings are rightmost) which records the way that each variable was bound as well as its type. A prefixed expression (pe) is a prefix together with a typed expression. We include an 'error' pe called illtyped for later use. The typechecker will be defined to take a prefix and an (untyped) expression and return a well-typed pe — illtyped is the result if it is impossible to assigned a well-typing to the input expression.

```
const Prefix =

  let PrefixElement =
    enrich Id + Type by
      data sorts prefixelement
           opns (PLET _ | _), (PLAMBDA _ | _), (PFIX _ | _) :
                              id, type -> prefixelement    enden in

  derive sorts prefix
         opns empty : prefix
              unit : prefixelement -> prefix
              (_ . _ ) : prefix,prefix -> prefix
              (_ == _) : prefix,prefix -> bool
    using PrefixElement
    from Sequence(PrefixElement[element is prefixelement])
    by prefix is sequence    endde


const PrefixExpr =
  enrich Prefix + TypedExpr by
    data sorts prefixexpr
         opns (_ | _) : prefix,typedexpr -> prefixexpr
         erroropns illtyped : prefixexpr    enden
```

Each prefixed expression has a set of sub-pe's given by the following rules, together with their reflexive-transitive closure:

- p|x has no sub-pe's except itself,

- p|(e e') has sub-pe's p|e and p|e',

- p¦(if e then e' else e") has sub-pe's p¦e, p¦e' and p¦e",

- p¦(lambda x.e) has sub-pe (p.lambda x)¦e,

- p¦(fix x.e) has sub-pe (p.fix x)¦e,

- p¦(let x=e in e') has sub-pe's p¦e and (p.let x)¦e.

A sub-pe is thus a subexpression with a prefix consisting of all the variable bindings which enclose it. We define below a operator which yields the set of sub-pe's of a prefixed expression, where the types in the sub-pe's are induced by the types in the pe.

```
const SubPE =
  enrich PrefixExpr + Set(PrefixExpr[element is prefixexpr]) by
    opns subpe : prefixexpr -> set
    eqns subpe(pe & (_ ¦ (TVAR _¦_))) = singleton(pe)
         subpe(pe & (p ¦ (TAPPLY a TO b¦_))) =
                         singleton(pe) U subpe(p¦a) U subpe(p¦b)
         subpe(pe & (p ¦ (TIF a THEN b ELSE c¦_))) =
                         singleton(pe) U subpe(p¦a) U subpe(p¦b)
                         U subpe(p¦c)
         subpe(pe & (p ¦ (TLAMBDA x¦t . a¦_))) = singleton(pe)
                         U subpe(p . unit(PLAMBDA x¦t) ¦ a)
         subpe(pe & (p ¦ (TFIX x¦t . a¦_))) = singleton(pe)
                         U subpe(p . unit(PFIX x¦t) ¦ a)
         subpe(pe & (p ¦ (TLET x¦t BE a IN b¦_))) = singleton(pe)
                         U subpe(p . unit(PLET x¦t) ¦ b)
                         U subpe(p¦a)                    enden
```

An item let x¦t, fix x¦t or lambda x¦t in a prefix p is said to be active in p iff no prefix element containing x occurs to the right of it in p. That is, a binding is active in a prefix if it has not been hidden by a more local binding of the same identifier.

```
const Active =

  let Var =
    enrich Prefix by
      opns var : prefixelement -> id
      eqns var(PLET x¦_) = x
           var(PLAMBDA x¦_) = x
           var(PFIX x¦_) = x    enden in
```

```
let IsActive =
  enrich Var by
    opns (_ is_active_in _) : prefixelement,prefix -> bool
    eqns _ is_active_in empty = false
         p is_active_in (_ . unit(p)) = true
         p is_active_in (s . unit(q)) = false
                        if not(p==q) and var(p)==var(q)
         p is_active_in (s . unit(q)) = p is_active_in s
                        if not(var(p)==var(q))    enden in

derive opns (_ is_active_in _) : prefixelement,prefix -> bool
  using Prefix
  from IsActive    endde
```

Given a prefixed expression p¦e and a binding let x¦t in p, a type variable in t which does not occur in the type of any enclosing lambda or fix binding (that is, in the type of any lambda or fix item to the left of the let in p) is called generic for the binding let x¦t. Only generic type variables are instantiable; other type variables are fixed (at least locally). The operator is_generic_in is defined below to determine if the given type variable is generic for the PLET prefix element at the rightmost extremity of the given prefix. It is not defined for prefixes not ending with a PLET. Milner [1978] also defines what it means for a type variable which is in the expression part of a prefixed expression to be generic. This concept is not needed to characterise well-typed expressions so it is omitted here.

```
const VarsinType =
  enrich Type + Set(TypeVar[element is typevar]) by
    opns varsintype : type -> set
    eqns varsintype(constant(_)) = empty
         varsintype(var(x)) = singleton(x)
         varsintype(t1 ---> t2) = varsintype(t1) U varsintype(t2)
                                                    enden

const GenericVars =

  let NonLetVarsinPrefix =
    enrich Prefix + VarsinType by
      opns nonletvars : prefix -> set
      eqns nonletvars(empty) = empty
           nonletvars(unit(PLET _¦_)) = empty
           nonletvars(unit(PLAMBDA _¦t)) = varsintype(t)
           nonletvars(unit(PFIX _¦t)) = varsintype(t)
           nonletvars(s . t) = nonletvars(s) U nonletvars(t)
                                                  enden in
```

```
let GenVars =
  enrich NonLetVarsinPrefix by
    opns (_ is_generic_in _) : typevar,prefix -> bool
    eqns v is_generic_in (s.unit(PLET _¦t))
                  = (v is_in varsintype(t))
                    and not(v is_in nonletvars(s))   enden in

derive opns (_ is_generic_in _) : typevar,prefix -> bool
  using Prefix, VarsinType
  from GenVars    endde
```

A generic instance of a type t of a prefix element let x¦t is an instance of t in which only generic type variables of t are instantiated. We must first specify what it means for one type to be an instance of another; "t1 is_instance_of t2 wrt S" is defined below to be true iff t1 is an instance of t2 with respect to the type variables in S. Note that any prefix given to is_generic_instance_of must have the appropriate PLET at its rightmost extremity; otherwise the result is not defined.

```
const Instance =

  let Substitution =
    enrich Type + Map(TypeVar[element is typevar],
                      Type[element is type]) by
      opns substitute : type,map -> type
      eqns substitute(constant(b),_) = constant(b)
           substitute(var(x),f) = f[x]  if x is_in domain(f)
           substitute(var(x),f) = var(x)
                                  if not(x is_in domain(f))
           substitute(t1--->t2,f) =
               substitute(t1,f) ---> substitute(t2,f)    enden in

  enrich Substitution by
    opns (_ is_instance_of _ wrt _) : type,type,set -> bool
    eqns t1 is_instance_of t2 wrt S =
              exists f. ((domain(f)==S)
                         and (substitute(t2,f)==t1))    enden


const GenericInstance =
  enrich Instance + GenericVars by
    opns (_ is_generic_instance_of _) : type,prefix -> bool
    eqns t1 is_generic_instance_of (p & (_ . unit(PLET _¦t2)))
              = exists S. (t1 is_instance_of t2 wrt S
                           and forall v. (v is_in S -->
                                          v is_generic_in p))
                                                    enden
```

A prefixed expression p¦e is standard iff for every sub-pe p'¦e' the generic type variables of each let binding in p' occur nowhere else in p'¦e'. For example, the following prefixed expression is standard:

(lambda x¦α . let f¦α--->β) ¦ (f¦α--->§ x¦α) ¦ §)

(only β is generic, and it appears only in the let) but this one is not:

(lambda x¦α . let f¦α--->β) ¦ (f¦α--->β x¦α) ¦ β)

A well-typed prefixed expression is required to be standard for technical reasons; the reader is referred to [Milner 1978].

```
const Standard =

  let VarsinTypedExpr =
    enrich TypedExpr + VarsinType by
      opns varsintypedexpr : typedexpr -> set
      eqns varsintypedexpr(TVAR _¦t) = varsintype(t)
           varsintypedexpr(TAPPLY a TO b¦t) = varsintype(t)
                      U varsintypedexpr(a) U varsintypedexpr(b)
           varsintypedexpr(TIF a THEN b ELSE c¦t) = varsintype(t)
                      U varsintypedexpr(a) U varsintypedexpr(b)
                      U varsintypedexpr(c)
           varsintypedexpr(TLAMBDA _¦t1 . a¦t2) = varsintype(t1)
                      U varsintype(t2) U varsintypedexpr(a)
           varsintypedexpr(TFIX _¦t1 . a¦t2) = varsintype(t1)
                      U varsintype(t2) U varsintypedexpr(a)
           varsintypedexpr(TLET _¦t1 BE a IN b¦t2) =
                        varsintype(t1)
                      U varsintype(t2) U varsintypedexpr(a)
                      U varsintypedexpr(b)              enden in

  let VarsinPrefix =
    enrich Prefix + VarsinType by
      opns varsinprefix : prefix -> set
      eqns varsinprefix(empty) = empty
           varsinprefix(unit(PLET _¦t)) = varsintype(t)
           varsinprefix(unit(PLAMBDA _¦t)) = varsintype(t)
           varsinprefix(unit(PFIX _¦t)) = varsintype(t)
           varsinprefix(s . t) =
                      varsinprefix(s) U varsinprefix(t)
                                                  enden in
```

```
let IsStandard =
  enrich SubPE + GenericVars + VarsinTypedExpr
                             + VarsinPrefix by
    opns (_ isstandard) : prefixexpr -> bool
         genericvarsok : prefix,prefixexpr -> bool
                       ! auxiliary opn -- checks a given PLET
         exposelet : prefix -> prefix
                       ! auxiliary opn -- exposes next PLET
    eqns pe isstandard =
           forall p, e. ((p|e) is_in subpe(pe)) -->
                          genericvarsok(exposelet(p),p|e)
         genericvarsok(empty,_) = true
         genericvarsok(p & (s . unit(_)),p1|e) =
            forall v. ((p=(p.v)) -->
                       forall x. (x is_in (varsintypedexpr(e)
                                           U varsinprefix(s.v))
                                  --> not(x is_generic_in p)))
            and genericvarsok(exposelet(s),p1|e)
         exposelet(empty) = empty
         exposelet(p & (_ . unit(PLET _|_))) = p
         exposelet(s . unit(PLAMBDA _|_)) = exposelet(s)
         exposelet(s . unit(PFIX _|_)) = exposelet(s)  enden in

derive opns (_ isstandard) : prefixexpr -> bool
  using SubPE, GenericVars
  from IsStandard    endde
```

Armed with all the definitions given above, we can finally define what it means for a prefixed expression to be well-typed (wt).

- p|(TVAR x|t) is wt iff it is standard, and either

  . lambda x|t or fix x|t is active in p, or

  . let x|t' is active in p and t is a generic instance of t'.

- p|(TAPPLY e TO e'|t") is wt iff p|e and p|e' are wt and t = t'⟶t", where t and t' are the types assigned to e and e'.

- p|(TIF e THEN e' ELSE e"|t) is wt iff p|e, p|e' and p|e" are wt, the type of e is BOOL and the types of e' and e" are both t.

- p|(TLAMBDA x|t . e'|t") is wt iff (p.PLAMBDA x|t)|e' is wt and t" = t⟶t', where t' is the type of e'.

- p|(TFIX x|t . e'|t") is wt iff (p.PFIX x|t)|e' is wt, t=t" and the type of e' is t.

- p|(TLET x|t BE e IN e'|t') is wt iff p|e and

(p.PLET x|t)|e' are wt, the type of e is t and the type of e' is t'.

See the beginning of this section for examples of well- and ill-typed expressions.

Once the operator is_welltyped has been defined, we specify the typecheck operator by saying that anything typecheck returns is well-typed and identical (except for types) to the prefixed expression it was given, if some well-typing exists then typecheck finds one (not necessarily the same one), and if no well-typing exists then typecheck returns illtyped (the error pe). Note that this specification only requires typecheck to find some type; the type it finds is not necessarily the best (most general) one.

```
const WellTyped =

  ! expose a given (active) prefixelement
  let ExposeActive =
    enrich Prefix by
      opns exposeactive : prefixelement,prefix -> prefix
      eqns exposeactive(pe,p & (_ . unit(pe))) = p
           exposeactive(p,s . unit(q)) = exposeactive(p,s)
                                         if not(p==q)    enden in

  let IsWellTyped =
    enrich Standard + GenericInstance + Active + ExposeActive by
      opns (_ is_welltyped) : prefixexpr -> bool
      eqns (pe & (p | (TVAR x|t))) is_welltyped =
               ( (PLAMBDA x|t) is_active_in p
                 or ((PFIX x|t) is_active_in p)
                 or (exists t1. ((PLET x|t1) is_active_in p and
                       (t is_generic_instance_of
                                 exposeactive(PLET x|t1,p)))) )
               and pe isstandard
           p | (TAPPLY a TO b|t) is_welltyped
               = (p|a) is_welltyped and ((p|b) is_welltyped)
                 and typeof(a)==(typeof(b)--->t)
           p | (TIF a THEN b ELSE c|t) is_welltyped
               = (p|a) is_welltyped and ((p|b) is_welltyped)
                 and ((p|c) is_welltyped) and typeof(b)==t
                 and typeof(c)==t and typeof(a)==constant(BOOL)
           p | (TLAMBDA x|t1 . a|t2) is_welltyped
               = (p . unit(PLAMBDA x|t1) | a) is_welltyped
                 and t2==(t1--->typeof(a))
           p | (TFIX x|t1 . a|t2) is_welltyped
               = (p . unit(PFIX x|t1) | a) is_welltyped
                 and t1==t2 and typeof(a)==t2
           p | (TLET x|t1 BE a IN b|t2) is_welltyped
               = (p . unit(PLET x|t1) | b) is_welltyped
                 and (p|a) is_welltyped and t1==typeof(a)
                 and typeof(b)==t2                enden in
```

```
derive opns (_ is_welltyped) : prefixexpr -> bool
  using Standard, GenericInstance, Active
  from IsWellTyped    endde
```

```
const Typecheck =
  enrich WellTyped + TypedExprEq by
    opns typecheck : prefix,expr -> prefixexpr
    eqns (p|e) is_welltyped and e0==e = true
                        if typecheck(p,e0)==(p|e)
         exists e1. (typecheck(p,e0)==(p|e1)) = true
                       if exists e. ((p|e) is_welltyped and e0==e)
    erroreqns typecheck(p,e0) = illtyped
                       if not(exists e. ((p|e) is_welltyped
                                          and e0==e))    enden
```

Processing time: 15.1 minutes.

If an additional operator is defined which recognises if the type of one prefixed expression is a generic instance of the type of another:

```
(_ is_generic_instance_of _) : prefixexpr,prefixexpr -> bool
```

then adding the following equation to Typecheck specifies that the operator typecheck always finds the most general type:

```
typecheck(p,e0) is_generic_instance_of p|e = true
                                 if (p|e) is_welltyped and e==e0
```

This addition was omitted from the specification in the interests of brevity.

The specification is a straightforward translation of [Milner 1978]; its complexity is due almost entirely to the number and complexity of the notions which must be defined in order to specify which expressions are well-typed. It is of course more difficult to specify concepts precisely in Clear than in English, since a phrase like "... does not occur in any enclosing binding" must necessarily be described as a search of some kind in Clear, probably involving one or more auxiliary operators which for the sake of tidiness must later be hidden using a derive. With higher-order types such

routine searches could largely be expressed using a few special operators (such as

```
occurs : sequence,(element->bool) -> bool
```

for searching a sequence for an element satisfying a certain condition) as in HOPE, but the Clear system does not yet permit such operators. Goguen [1981] indicates that meta-operations (apparently like macros) will be available in the Ordinary specification language for this purpose.

It would be possible to give both a higher-level and a lower-level specification of the same problem. The high-level specification would give a semantics of the language Exp where some expressions yield an error, and then define well-typed expressions as those which do not result in errors. The low-level specification would be an explicit algorithm for computing a well-typing. Theorems in [Milner 1978] state that any expression which is well-typed according to our specification will be well-typed according to the high-level specification (but not the converse), and that any expression accepted by the low-level algorithm will be well-typed according to our specification (the converse is proved by Damas and Milner [1982]).

CHAPTER FIVE

A CATEGORY-THEORETIC SEMANTICS OF CLEAR AND ITS IMPLEMENTATION

In chapter III a semantics of Clear was given using simple set-theoretic constructions to describe the theory-building operations of Clear. This chapter is devoted to a discussion of another semantics of Clear, invented by Burstall and Goguen [1980]. This semantics is intended as a generalisation of the set-theoretic semantics of chapter III (although historically it came first) and uses ideas from category theory to describe the underlying concepts and operations of Clear. Although as remarked in chapter III this results in a description which is rather inaccessible because it is so abstract, there are some benefits to be gained from such an approach. The most important advantage is that category theory acts as a ruthless filter for ideas. If an idea cannot be expressed gracefully using the standard concepts of category theory, then often there is something wrong with the idea. If the idea can be expressed, then its category-theoretic description will often suggest a generalisation which may not have been obvious otherwise. These are advantages for the language designer. But once the design is complete the category-theoretic description will still often be more elegant than an equivalent description in a different style, although it may be more difficult to understand. Without this kind of high-level motivation the set-theoretic constructions of chapter III may seem to come out of thin air, appearing complex and mysterious. And finally, in this case a category-theoretic description makes it possible to abstract away from particular notions of signatures, models or axioms, allowing a description of (most of) Clear under an arbitrary institution. However, in section III.6 we saw that the set-theoretic semantics can be readily altered to accomodate all institutions of apparent interest.

Burstall and Goguen's category-theoretic semantics relies most heavily on the notion of a colimit, which is used to give a meaning to the combine and apply theory-building operations. A HOPE program for computing colimits in arbitrary cocomplete categories and in a kind of 'comma' category has been described by Burstall [1980].

Further developments along these lines are given by Rydeheard [1981], who presents a category-theoretic approach to programming.

Burstall's colimit program provided a basis which allowed an implementation of the category-theoretic semantics of Clear following almost exactly Burstall and Goguen's original presentation (this project was done in collaboration with David Rydeheard). The ease with which this implementation and the original colimit program were carried out can be attributed to the high-level features of HOPE (in particular, the strong yet flexible type system) described in appendix 1.

This chapter combines presentations of the semantics and the implementation; the semantics is explained through descriptions of the programs which implement it. The facilities provided by the colimit program are described in section 1, although an explanation of how the program works is not given (see [Burstall 1980] or [Rydeheard 1981] for details). After a presentation of the semantics of Clear and its implementation in sections 2, 3 and 4 the outcome of the implementation attempt is briefly discussed in section 5. For a less 'algorithmic' explanation of the semantics, refer to [Burstall and Goguen 1980]; for another presentation of the semantics program see [Rydeheard 1981]. The program described here is different from the one discussed in [Rydeheard 1981] for expository reasons.

This program was an experiment in 'categorical programming' as much as an attempt to provide a useful implementation of Clear. We accordingly used category-theoretic ideas whenever possible, insofar as this was practical. For example, the graphs which underlie diagrams are represented as objects in a comma category, even though this is not necessary for any of the algorithms used (see the next section for the meaning of 'diagram' and 'comma category'). Our attempts in this direction are related to the "doctrines" given by ADJ in [Goguen, Thatcher, Wagner and Wright 1973]. Unfortunately, all of these things are computationally expensive, and the resulting program is too large and much too slow for practical use; see section 5 for more on this matter.

This chapter assumes some previous knowledge of elementary category theory. See [Arbib and Manes 1975] for the meanings of the important concepts of *category*, *morphism*, *functor* and *colimit* (especially important are the *initial object*, *coproduct*, *coequaliser* and *pushout* -- these are all special kinds of colimits). See [MacLane 1971] for the definition of a *comma category*.

1. Computing colimits

The facilities provided by Burstall's colimit program (which has since been reorganised and partially rewritten by Rydeheard and myself) are described here only briefly. For a much more detailed description consult [Burstall 1980] or [Rydeheard 1981]. See [Arbib and Manes 1975] and [MacLane 1971] for the elementary category theory which this program encodes.

A category is characterised by two HOPE types (objects and morphisms) and four functions for manipulating morphisms. These functions tell us the source and target objects of a morphism, the identity morphism on an object, and how to compose morphisms. As a HOPE declaration this is simply:

```
typevar o, m                ! objects, morphisms

data Cat(o,m) == cat((m->o),(m->o),(o->m),(m#m->m))
                            ! source, target, identity, composition
```

(Comments in HOPE are preceded by an exclamation mark.) A particular category is a data object of this type. We want equations such as the following to hold in the category cat(source,target,identity,compose):

```
source(identity(o)) = target(identity(o)) = o
source(m1) = source(compose(m1,m2))
target(m2) = target(compose(m1,m2))
```

but there is no convenient way in HOPE (or in other programming languages) for these to be enforced, so the responsibility for ensuring that the functions he supplies describe a legitimate category rests with the user.

An example is the category of (finite) sets (not unrestricted sets, but sets containing elements of a uniform type, as required by the HOPE type system):

```
typevar alpha

data Set_Mor alpha == mor(set alpha,(alpha-->alpha),set alpha)
                            ! source object, map, target object
```

```
dec source, target : Set_Mor alpha -> set alpha
dec identity : set alpha -> Set_Mor alpha
dec comp : Set_mor alpha # Set_Mor alpha -> Set_Mor alpha

--- source(mor(a,_,_)) <= a
--- target(mor(_,_,b)) <= b
--- identity(a) <= mor(a,id_map a,a)
--- comp(mor(a1,m1,b1),mor(a2,m2,b2)) <=
                mor(a1,m1.m2,b2) if b1=a2 else error()

dec cat_of_sets : Cat(set alpha,Set_Mor alpha)

--- cat_of_sets <= cat(source,target,identity,comp)
```

The notation alpha-->alpha in HOPE refers to a map; id_map and (composition) are primitive functions on maps; and error() causes a HOPE error, giving us a (crude) way of implementing the partial function comp.

A functor is a pair of functions mapping objects and morphisms in one category to objects and morphisms in another category. Again, these functions should satisfy certain conditions (e.g. preservation of identities) which the program must ignore.

```
typevar o, m, o1, m1

data Functor(o,m,o1,m1) == functor((o->o1),(m->m1))
                         ! F : Cat(o,m) -> Cat(o1,m1)
```

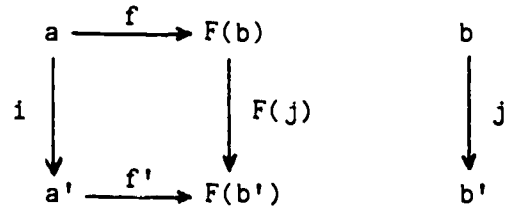
A functor can be applied to an object or a morphism using an (overloaded) infix function called "of".

Given two categories $\underline{\underline{K}}$:Cat(o,m) and $\underline{\underline{L}}$:Cat(o1,m1) and a functor F:Functor(o1,m1,o,m) (i.e. F:$\underline{\underline{K}}\rightarrow\underline{\underline{L}}$) the comma category ($\underline{\underline{K}}$,F) has objects like (a,f,b) of type o#m#o1:

$$a \xrightarrow{\ f\ } F(b) \qquad\qquad b$$

[in $\underline{\underline{K}}$] [in $\underline{\underline{L}}$]

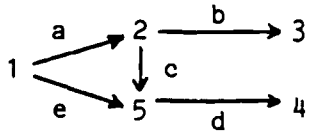
and morphisms like (i,j) of type m#m1 taking (a,f,b) to (a',f',b') such that the following diagram commutes:



[in $\underline{\underline{K}}$] [in $\underline{\underline{L}}$]

More general comma categories than this can be defined, but for our purposes this version (actually a right comma category) is sufficient.

Comma categories are used throughout the entire Clear semantics program; it turns out that many common data types can be represented in this way. Examples will crop up here and there; the first one is the category of (directed) graphs. A graph can be considered to be a map from a set of edges into a set of pairs of nodes:



is $\{a,b,c,d,e\} \xrightarrow{G} \{(1,1),(1,2),\ldots,(5,5)\}$ where $G=[a\mapsto(1,2), b\mapsto(2,3), c\mapsto(2,5), d\mapsto(5,4), e\mapsto(1,5)]$.

A graph morphism from G to G' is a pair of maps. One map associates nodes of G' with the nodes of G, and the other does the same with the edges. The edge map must respect the sources and targets of edges; that is,

$source_{G'}(edgemap(e)) = nodemap(source_G(e))$ and

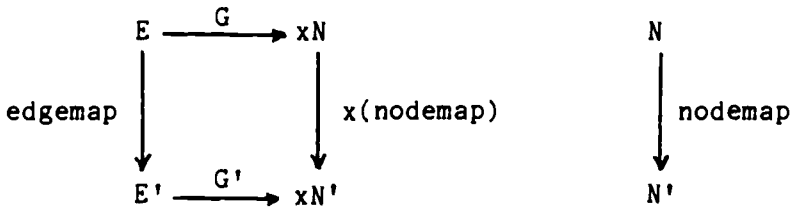
$target_{G'}(edgemap(e)) = nodemap(target_G(e))$.

A graph can thus be seen as an object in the comma category ($\underline{\underline{Set}}$,x) where $x:\underline{\underline{Set}}\rightarrow\underline{\underline{Set}}$ is the crossproduct functor taking a set S to the set SxS. So the graph above is the triple $(\{a,b,c,d,e\},G,\{1,2,3,4,5\})$:

$\{a,b,c,d,e\} \xrightarrow{G} x\{1,2,3,4,5\}$ $\{1,2,3,4,5\}$

[in $\underline{\underline{Set}}$] [in $\underline{\underline{Set}}$]

Similarly, a graph morphism can be viewed as a morphism in the comma category (Set,x):



In the program a slightly more complicated representation of the morphisms in a comma category is used since the source and target objects must be recorded as well as the morphism itself:

```
data FComma_Mor(o,m,o1,m1) ==
                fcomma_mor((o#m#o1),(m#m1),(o#m#o1))
                    ! source object, morphisms, target object
```

Now we can construct the comma category (K,F) given the categories K and L and the functor F:

```
dec functor_comma_cat :
                Cat(o,m) # Cat(o1,m1) # Functor(o1,m1,o,m)
                        -> Cat((o#m#o1),FComma_Mor(o,m,o1,m1))
```

The definition is easy; for example, the 'identity part' of this category is the function:

```
lambda obj & (a,_,b) =>
       fcomma_mor(obj,(idK a,idL b),obj)
```

where idK and idL are the identity parts of the categories K and L.

For the category of graphs we already have the two categories; they are both cat_of_sets defined above. We need only the functor x:Set→Set. This is easy to define except for a snag with HOPE's type system; the problem is that the natural way to define the functor gives the type

```
dec crossprod : Functor(set alpha, set alpha,
                        set alpha#alpha, set alpha#alpha)
```

and the target of this does not match the type of the category we want for K. We need a type which is the disjoint union of alpha and

alpha#alpha:

```
data Tag alpha == just(alpha) ++ pair(Tag alpha,Tag alpha)
```

Now crossprod can be easily defined, with the following type:

```
dec crossprod : Functor(set(Tag alpha), Set_Mor(Tag alpha)
                        set(Tag alpha), Set_Mor(Tag alpha))
```

So we can define the category of graphs, and abbreviations for the types of graphs and their morphisms:

```
type Graph alpha == set(Tag alpha) # Set_Mor(Tag alpha)
                                   # set(Tag alpha)

type Graph_Mor alpha ==
            FComma_Mor(set(Tag alpha), Set_Mor(Tag alpha),
                       set(Tag alpha), Set_Mor(Tag alpha))

dec cat_of_graphs : Cat(Graph alpha, Graph_Mor alpha)

--- cat_of_graphs <=
            functor_comma_cat(cat_of_sets,cat_of_sets,crossprod)
```

The advantage of defining something as an object in a comma category is that colimits on the underlying categories can be automatically 'lifted' to give colimits for the comma category. This will be discussed in slightly more detail at the end of this section.

One more function on comma categories will be helpful in writing the semantics of Clear:

```
dec right_compose : Cat(o,m) # Cat(o1,m1) # Functor(o1,m1,o,m)
                           -> (m1 # (o#m#o1) -> (o#m#o1))

--- right_compose(cat(_,_,_,cmp),cat(_,t1,_,_),F) <=
          lambda g, (a,f,_) => (a, cmp(f,F of g), t1 g)
```

This function modifies an object in a comma category by composition 'on the right':

goes to

$$a \xrightarrow{f.F(g)} F(c) \qquad\qquad c$$

[in $\underline{\underline{K}}$] [in $\underline{\underline{L}}$]

The function left_compose can be defined analogously.

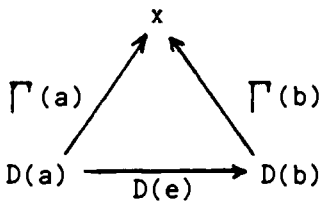
A diagram on a category $\underline{\underline{K}}$ is a graph with objects of $\underline{\underline{K}}$ attached to the nodes and morphisms of $\underline{\underline{K}}$ attached to the edges. A diagram morphism from D to D' is a map f taking nodes of D to nodes of D', together with another map which associates a morphism from the object at n to the object at f(n) to each node n in D. We label the nodes and edges of graphs with strings (character lists).

```
type Name == list char

data Diagram(o,m) == diagram(Graph Name, (Name-->o), (Name-->m))
                          ! diagram on a category of type Cat(o,m)

data Diagram_Mor(o,m) == diagram_mor(Diagram(o,m), (Name-->Name),
                                     (Name-->m), Diagram(o,m))
                         ! source diagram, node-node map,
                         ! node_morphism map, target diagram
```

It is easy to define the category of diagrams

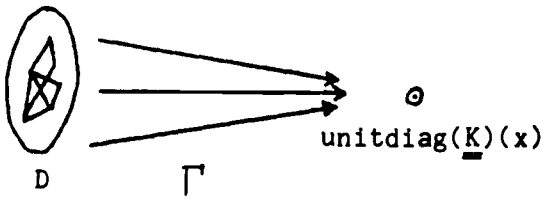
```
dec cat_of_diagrams :
             Cat(o,m) -> Cat(Diagram(o,m),Diagram_Mor(o,m))
```

A cone on a category $\underline{\underline{K}}$ (actually a cocone, but the word "cone" will be used throughout) is a diagram D (the base), an object x of $\underline{\underline{K}}$ (the apex) and a family of morphisms Γ from each node of D to x (the flanks) such that all triangles of morphisms of the following form commute:



where $a \xrightarrow{e} b$ is an edge in the graph of D. A cone morphism from C to C' is a diagram morphism from the base of C to the base of C' and

a $\underline{\underline{K}}$ morphism from the apex of C to the apex of C', satisfying certain commutation conditions. In the program the category of cones on $\underline{\underline{K}}$ is taken to be the comma category ($\underline{\underline{\underline{Diag}}}(\underline{\underline{K}})$,unitdiag($\underline{\underline{K}}$)) where the functor unitdiag($\underline{\underline{K}}$):$\underline{\underline{K}}$⟶$\underline{\underline{\underline{Diag}}}(\underline{\underline{K}})$ takes an object in $\underline{\underline{K}}$ to the diagram consisting of only a single node with that object attached. The flanks ⌈ ⌉ are embodied in the diagram morphism from the base to unitdiag($\underline{\underline{K}}$) of the apex.



[in $\underline{\underline{\underline{Diag}}}(\underline{\underline{K}})$] [in $\underline{\underline{K}}$]

We supply abbreviations for the types of cones and their morphisms, and define the category of cones:

```
type Cone(o,m) == Diagram(o,m) # Diagram_Mor(o,m) # o
                        ! cone on a category of type Cat(o,m)

type Cone_Mor(o,m) ==
            FComma_Mor(Diagram(o,m),Diagram_Mor(o,m),o,m)

dec cat_of_cones : Cat(o,m) -> Cat(Cone(o,m),Cone_Mor(o,m))

--- cat_of_cones K <=
            functor_comma_cat(cat_of_diagrams K,K,unitdiag K)
```

The colimit of a diagram D is a cone C with base D which is 'better' than all other such cones, in the sense that for any cone C' (with base D) there is a unique cone morphism from C to C'. It turns out that it is possible to construct the colimit of any (finite) diagram (on a category $\underline{\underline{K}}$) given only the initial object of $\underline{\underline{K}}$ and functions which compute (binary) coproducts and coequalisers in $\underline{\underline{K}}$. These have the following types:

```
type Initial_Obj(o,m) == o # (o->m)

type Coproduct(o,m) == o#o -> (o#m#m) # (o#m#m -> m)

type Coequaliser(o,m) == m#m -> (o#m) # (o#m -> m)
```

Note that each of these includes a universal part; that is, besides

producing the coproduct (or whatever) a function computing the unique morphism from the coproduct to any other object is also provided.

Now we can define a cocomplete category as a category with initial object, coproducts and coequalisers:

```
data C_Cat(o,m) == c_cat(Cat(o,m), Initial_Obj(o,m),
                         Coproduct(o,m), Coequaliser(o,m))
```

An example of a cocomplete category is the category of sets defined above with appropriate initial object, coproducts and coequalisers. We need a type which is the disjoint union of alpha and alpha; this is accomplished by extending the earlier definition of the type Tag alpha:

```
data Tag alpha == just(alpha) ++ . . . ++ pink(Tag alpha)
                                  ++ blue(Tag alpha)
```

Then (for example) we can define the coproduct as follows:

```
dec coprod : Coproduct(set(Tag alpha),Set_Mor(Tag alpha))

--- coprod(s,t) <=
    let cp == (pink * s) U (blue * t) in
    let univ ==
        (lambda v, mor(a,f,b), mor(a1,f1,b1) =>
            error() if not(s=a and t=a1 and v=b and v=b1)
                    else let fg == (lambda pink x => f of x
                                        | blue x => f1 of x) in
                         mor(cp,fn_to_map(cp,fg),v) ) in
    (cp, mor(s,fn_to_map(s,pink),cp),
         mor(t,fn_to_map(t,blue),cp)), univ
```

Recall from appendix 1 that infix * in HOPE is just like LISP mapcar:

$$f * [a_1, \ldots, a_n] = [f(a_1), \ldots, f(a_n)]$$

The initial object (just the empty set) and coequaliser are not difficult to define. The cocomplete category of sets is then:

```
dec c_cat_of_sets : C_Cat(set(Tag alpha),Set_Mor(Tag alpha))

--- c_cat_of_sets <= c_cat(cat_of_sets,init,coprod,coeq)
```

Now the colimit program takes a cocomplete category and gives it a colimit function. See [Burstall 1980] or [Rydeheard 1981] for the definition; the types are as follows:

```
type Colimit(o,m) ==
       Diagram(o,m) -> Cone(o,m) # (Cone(o,m) -> Cone_Mor(o,m))

dec colimit : C_Cat(o,m) -> Colimit(o,m)
```

We can then define a colimit category; sets provide an example:

```
data Colimit_Cat(o,m) == colimit_cat(Cat(o,m),Colimit(o,m))

dec colim_cat_of_sets ==
             Colimit_Cat(set(Tag alpha),Set_Mor(Tag alpha))

--- colim_cat_of_sets <=
             colimit_cat(cat_of_sets,colimit(c_cat_of_sets))
```

As mentioned earlier, if we have colimits on the categories $\underline{\underline{K}}$ and $\underline{\underline{L}}$ then we can compute colimits on the comma category $(\underline{\underline{K}},F)$ for any functor $F:\underline{\underline{L}}\longrightarrow\underline{\underline{K}}$ (see [Goguen and Burstall 1978]). This is an advantage of using comma category representations, especially since the Clear semantics program makes heavy use of colimits. See [Rydeheard 1981] for the program; the type of the colimit function is as follows:

```
dec lift_colimit :
      Colimit_Cat(o,m) # Colimit_Cat(o1,m1) # Functor(o1,m1,o,m)
           -> Colimit(o#m#o1, FComma_Mor(o,m,o1,m1))
```

We can use this to define the colimit category of graphs (although this is not used by the Clear semantics program):

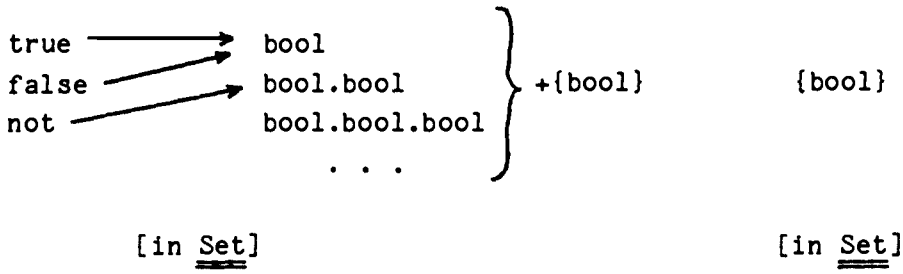
```
dec colim_cat_of_graphs :
              Colimit_Cat(Graph alpha,Graph_Mor alpha)

--- colim_cat_of_graphs <=
        colimit_cat(cat_of_graphs,
              lift_colimit(colim_cat_of_sets,colim_cat_of_sets,
                                          crossprod))
```

2. Signatures, institutions, theories and based objects

In this section a program for computing colimits on the category of based theories based on the programs in section 1 is described. This is the foundation of the semantics of Clear to be given in sections 3 and 4; the denotation of a specification is a based theory, and the theory-building operations of Clear correspond to simple colimits on that category.

We begin by defining signatures. However, they will not actually be used until the end of section 3. All of the programs given until then will be parameterised on an institution (this concept was informally discussed in section I.1.3); that is, they do not depend on particular definitions of signatures or axioms (or algebras or the satisfaction relation, although these do not arise in the program). Thus a general notion of theory can be defined, together with a program for computing colimits in the category of theories. But theories alone are not enough to give the semantics of Clear; we need a notion of theories with sharing. We define based objects (and their colimits), a general notion of objects with sharing. This can be instantiated to give based theories, and is further instantiated in section 4 to give based Clear theories (theories with the 'usual' kinds of signatures, axioms and models).

As already defined, a signature is a set of sorts S together with a family of sets of operators indexed by $S^* \times S$ (or S^+). A signature morphism is a map from the sorts and operators of one signature to those of another which preserves arities. We represent signatures as objects in the comma category $(\underline{\underline{\text{Set}}},+)$ where + is a functor taking a set to the set of nonempty strings over that set (and taking an ordinary function over the set to a function on strings). For example, here is the comma category representation of the signature with the single sort bool and operators true, false and not:



There are two problems in defining the functor +:Set→Set in HOPE. The first problem is the same as the one we met when trying to define the crossprod functor and the function coprod in section 1; the natural type of + is:

```
dec plus : Functor(set alpha,Set_Mor alpha,
                   set(list alpha), Set_Mor(list alpha))
```

and this clashes with the type required by the functor_comma_cat function (for constructing the comma category of signatures). Again, tags are used to solve this problem:

```
data Tag alpha == just(alpha) ++ . . . ++ string(list(Tag alpha))
```

The type of plus is then:

```
dec plus : Functor(set(Tag alpha), Set_Mor(Tag alpha),
                   set(Tag alpha), Set_Mor(Tag alpha))
```

The second problem occurs when we try to define the 'object part' of the functor plus. The result of applying plus to any non-empty set will be infinite. HOPE is equipped to handle infinite sets (lazy lists, see [Burstall, MacQueen and Sannella 1980]) but not infinite sets, although lazy sets could probably be added. For the purposes of the program, we can represent all infinite sets by the constant bigset:

```
dec bigset : set alpha
```

We provide no definition of bigset, and so evaluating it will cause an error. But we will never actually be interested in the value of the object part of the plus functor, so this is sufficient. With a similar 'definition' for bigmap (representing all infinite maps) plus is easy to define, and the category of signatures with colimits

is then defined as follows:

```
type Signature alpha ==
        Set(Tag alpha) # Set_Mor(Tag alpha) # set(Tag alpha)

type Signature_Mor alpha ==
        FComma_Mor(set(Tag alpha), Set_Mor(Tag alpha),
                   set(Tag alpha), Set_Mor(Tag alpha))

dec colim_cat_of_signatures :
        Colimit_Cat(Signature alpha,Signature_Mor alpha)

--- colim_cat_of_signatures <=
    colimit_cat(functor_comma_cat(cat_of_sets,cat_of_sets,plus),
                lift_colimit(colim_cat_of_sets,colim_cat_of_sets,
                                                plus))
```

As mentioned before, this definition will not actually be needed until 'signed' theories are defined at the end of the next section.

Institutions were discussed informally in section I.1.3; they provide a way of giving most of the semantics of Clear independently of any particular definitions of signatures, axioms, algebras or the satisfaction relation. Formally, an institution is any data object of the following type:

```
typevar o, m, alpha, beta      ! signatures, signature morphisms,
                               ! algebras, axioms

data Institution(o,m,alpha,beta) ==
        institution(Colimit_Cat(o,m),
                    Functor(o,m,set alpha,Set_Mor alpha),
                    Functor(o,m,set beta,Set_Mor beta),
                    (o -> (set alpha # set beta -> truval)) )
```

The parts of an institution are:

- An arbitrary cocomplete category $\underline{\underline{Sig}}$ of 'signatures'

- A functor $Mod:\underline{\underline{Sig}}\rightarrow\underline{\underline{Set}}^{op}$ (giving the set of models over a signature). If $\sigma:\underline{\Sigma}\rightarrow\underline{\Sigma}'$ is a morphism in $\underline{\underline{Sig}}$ and M' is in $Mod(\underline{\Sigma}')$ then we write $M'|_{\underline{\Sigma}}$ rather than $Mod(\sigma)(M')$.

- A functor $Sen:\underline{\underline{Sig}}\rightarrow\underline{\underline{Set}}$ (giving the set of axioms over a signature -- e.g. equations and data constraints). If $\sigma:\underline{\Sigma}\rightarrow\underline{\Sigma}'$ is a morphism in $\underline{\underline{Sig}}$ and S is in $Sen(\underline{\Sigma})$ then we write $\sigma(S)$ rather than $Sen(\sigma)(S)$.

- A relation $\models_{\underline{\Sigma}} \subseteq Mod(\underline{\Sigma}) \times Sen(\underline{\Sigma})$ for each object $\underline{\Sigma}$ of

Sig satisfying $M' \models \sigma(S)$ iff $M'|_{\Sigma} \models S$ for each $\sigma:\Sigma \rightarrow \Sigma'$ in Sig, $S \in Sen(\Sigma)$ and $M' \in Mod(\Sigma')$.

All of the functions defined from now until the end of the next section will be parameterised by an institution. By extracting such things as the particular category of signatures from the given institution rather than using a fixed set of definitions, most of the semantics is made orthogonal to the definition of these key concepts. It is only when we come down to writing the semantic equations (attaching a syntax to the mathematical operations we will define) that it will be necessary to decide on a particular institution.

A theory is a signature Σ together with a closed set of Σ-axioms. We can use the agglomerates of chapter IV to represent closed sets of axioms with the same constructors as before. We parameterise the definition by the types of signatures, signature morphisms, and axioms:

```
typevar o, m, beta   ! signatures, signature morphisms, axioms

data Agglomerate(o,m,beta) ==
            close(set beta)
          ++ union(Agglomerate(o,m,beta),Agglomerate(o,m,beta))
          ++ translate(m,Agglomerate(o,m,beta)),
          ++ inv_translate(m,Agglomerate(o,m,beta))
          ++ add_equality(set(Tag Name),Agglomerate(o,m,beta))
                  ! set(Tag Name) is a set of sort names
```

The definitions of theory and theory morphism are parameterised by the same types:

```
data Theory(o,m,beta) == theory(o,Agglomerate(o,m,beta))

data Theory_Mor(o,m,beta) ==
          theory_mor(Theory(o,m,beta),m,Theory(o,m,beta))
```

The category of theories is then easily defined, parameterised on an institution. The identity and composition functions come from the category of signatures contained in the institution.

```
dec cat_of_theories : Institution(o,m,alpha,beta) ->
                      Cat(Theory(o,m,beta),Theory_Mor(o,m,beta))
```

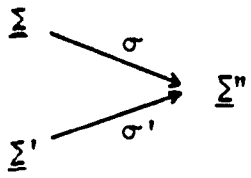
But we will need to compute colimits in this category. As mentioned before, the semantics of Clear is given in terms of colimits in the category of theories (actually, in the category of based theories, defined below -- but colimits for that category depend on colimits in the category of theories). A program for computing colimits follows the (constructive) proof of the following theorem; it depends on the availability of colimits in the category of signatures.

Theorem: The category of theories over any institution has (finite) colimits.

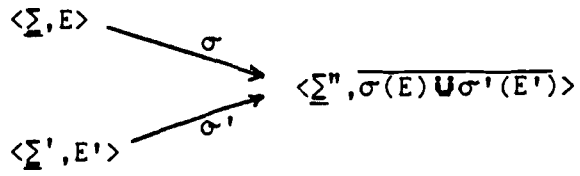
Proof: (outline; from [Burstall and Goguen 1980])
As mentioned in section 1, it suffices to show that the category of theories (over any institution) has an initial object, coproducts and coequalisers. The category Sig of signatures contained in any institution has these, by definition.

If ϕ is the initial object of Sig, then $\langle \phi, \tilde{\emptyset} \rangle$ is the initial object in the category of theories.

If the coproduct of Σ and Σ' in Sig is given by



then the coproduct of the theories $\langle \Sigma, E \rangle$ and $\langle \Sigma', E' \rangle$ is given by



If the coequaliser of $\sigma,\sigma' : \underline{\Sigma} \rightarrow \underline{\Sigma}'$ in $\underline{Sig}$ is given by

$$\underline{\Sigma} \underset{\sigma'}{\overset{\sigma}{\rightrightarrows}} \underline{\Sigma}' \xrightarrow{\sigma''} \underline{\Sigma}''$$

then the coequaliser of $\sigma,\sigma' : \langle\underline{\Sigma},E\rangle \rightarrow \langle\underline{\Sigma}',E'\rangle$ in the category of theories is given by

$$\langle\underline{\Sigma},E\rangle \underset{\sigma'}{\overset{\sigma}{\rightrightarrows}} \langle\underline{\Sigma}',E'\rangle \xrightarrow{\sigma''} \langle\underline{\Sigma}'',\overline{\sigma''(E')}\rangle$$

Programs for computing the initial object, coproducts and coequalisers in the category of theories can be written following the constructions above. Here only the definition of the initial object is given:

```
dec init : Institution(o,m,alpha,beta) ->
              Initial_Obj(Theory(o,m,beta),Theory_Mor(o,m,beta))

--- init(institution(colimit_cat(sigcat,sigcolim),_,_,_)) <=
    let sigcone,siguniv == sigcolim nil_diagram in
    let initsig == apex sigcone in
                           ! the initial signature
    let initth == theory(initsig,close nilset)
                           ! the initial theory
    let univ ==            ! the universal part
        (lambda pth & theory(psig,_) =>
            let univmor ==
              siguniv(cone sigcat (nil_diagram,nil_map,psig)) in
            theory_mor(initth,apex_morphism univmor,pth) ) in
    (initth, univ)
```

The constants nil_diagram, apex (the apex of a cone), apex_morphism (the apex part of a cone morphism) and cone (for constructing a cone as a 'comma object', given the components) are auxiliary functions whose definitions are omitted. The functions coprod and coeq (for the coproduct and coequaliser) are just as easy to write, although a bit longer. Using these we define the cocomplete category of theories, and then the colimit program described in section 1 can be employed to build the category of theories with colimits:

```
dec c_cat_of_theories : Institution(o,m,alpha,beta) ->
                C_Cat(Theory(o,m,beta),Theory_Mor(o,m,beta))

--- c_cat_of_theories I <=
                c_cat(cat_of_theories I, init I, coprod I, coeq I)

dec colim_cat_of_theories : Institution(o,m,alpha,beta) ->
                Colimit_Cat(Theory(o,m,beta),Theory_Mor(o,m,beta))

--- colim_cat_of_theories I <=
                colimit_cat(cat_of_theories I,
                            colimit(c_cat_of_theories I))
```

The function extend_signature_morphism will be used later in the semantics to extend a signature morphism to a theory morphism.

```
dec extend_signature_morphism : Institution(o,m,alpha,beta) ->
                (Theory(o,m,beta) # m # Theory(o,m,beta) ->
                        Theory_Mor(o,m,beta))

--- extend_signature_morphism I <= theory_mor
```

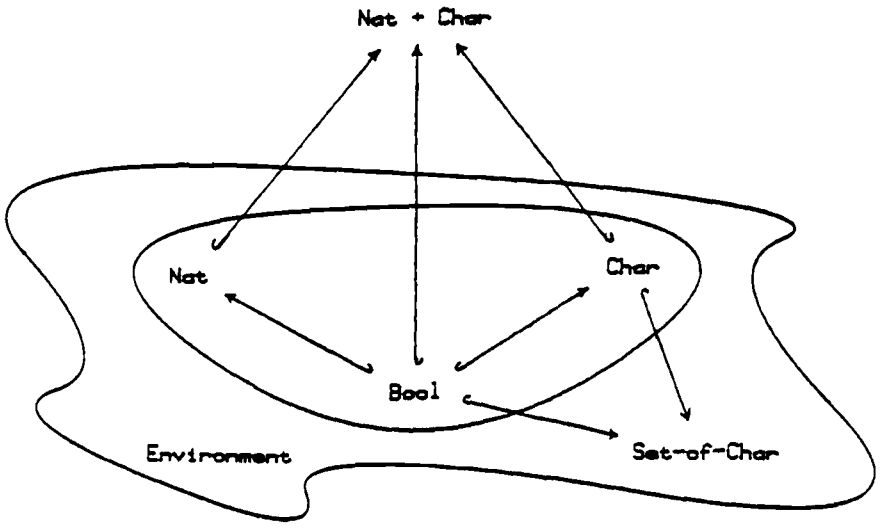
When this function is applied to the arguments $\langle\underline{\Sigma},E\rangle,\sigma,\langle\underline{\Sigma}',E'\rangle$ where $\sigma:\underline{\Sigma}\rightarrow\underline{\Sigma}'$ and $\sigma(E)\not\subseteq E'$ it should fail, since the result will not be a proper theory morphism. This is something which cannot be determined without the help of a theorem prover, so we do not check for it (but see chapter VI).

In Clear, if the theory C has been used to build the theories A and B in such a way that A and B both contain C, then C is called a shared subtheory of A and B and we require that A+B contain only one copy of C. The importance of taking account of shared subtheories when combining theories has already been discussed. The names of sorts and operators alone are not enough to distinguish shared subtheories; we want the freedom to have several different sorts and operators with the same names.

These requirements mean that the semantics of Clear must include a mechanism for keeping track of the genealogy of theories -- it is necessary to know which theories have been put together to produce other theories. The set-theoretic semantics of chapter III used the simple trick of attaching a tag to each sort and operator to record its theory of origin. This will not work here, because the

institutional approach requires signatures to be viewed as indivisible objects. Here the more elaborate notion of a based theory defined in [Burstall and Goguen 1980] must be used. Note that based theories here are not the same as the based theories used in the set-theoretic semantics, although they serve a similar purpose.

A based theory is a theory together with a set of morphisms to it from the theories in the environment from which it was built. The environment associates names with (constant) theories, analogous to environments in the semantics of an ordinary language; however, now the environment must also record the relationships between all the named theories. The environment is therefore represented as a diagram on the category of theories, where the edges describe how theories are shared. (See section 4.5 for more about environments.) A based theory is then a cone on the category of theories with a base which is a subdiagram of the environment. The apex is the theory of interest, and the flanks show how this theory is related to the theories in the base. For example, here is a picture of the based theory representing Nat + Char (these theories were given in chapter I):



We can define based objects analogously, a general notion of objects with sharing. The based objects on a category themselves form a category; this is a subcategory of the category of cones (a cone morphism $f:C \longrightarrow C'$ is a based object morphism iff the base of C

is included in the base of C' and the 'base part' of the cone morphism is the inclusion). The four functions which determine a category in our program (source, target, identity, composition) are the same for both categories and so the category of based objects is the same as the category of cones as far as our program is concerned:

```
type BasedObj(o,m) == Cone(o,m)

type BasedObj_Mor(o,m) == Cone_Mor(o,m)

dec cat_of_based_objects :
                    Cat(o,m) -> Cat(BasedObj(o,m),BasedObj_Mor(o,m))

--- cat_of_based_objects <= cat_of_cones
```

The colimit in the category of based objects is however not the same as the colimit in the category of cones. A different construction must be used:

Theorem: The category of based objects on a category $\underline{\underline{C}}$ has (finite) colimits if $\underline{\underline{C}}$ has.

Proof: (outline; see [Burstall and Goguen 1980] for the full proof)
Let D be a finite diagram in the category of based objects on $\underline{\underline{C}}$ with objects D_i having apices $\hat{D}_i$ and bases β_i. The colimit object of D is the based object with base $\bigcup_i \beta_i$, and with apex the colimit in $\underline{\underline{C}}$ of the diagram which results from 'flattening' the apices and flanks of the based objects D_i into the diagram D. The flanks of the colimit and the universal part are obtained from the colimit in $\underline{\underline{C}}$.

A program which produces the colimit in the category of based theories can be written following the above construction. The program is too long (about 60 lines) and complicated to include here; we give only its type:

```
dec bo_colimit : Colimit_Cat(o,m) ->
                          Colimit(BasedObj(o,m),BasedObj_Mor(o,m))
```

Now the category of based objects with colimits can be defined:

```
dec colim_cat_of_based_objects : Colimit_Cat(o,m) ->
                    Colimit_Cat(BasedObj(o,m),BasedObj_Mor(o,m))

--- colim_cat_of_based_objects(K & colimit_cat(C,_)) <=
                colimit_cat(cat_of_based_objects, bo_colimit K)
```

The careful reader may have observed that our definition of based objects differs slightly from the definition in [Burstall and Goguen 1980]. There the category of based objects over a given diagram (environment) is considered, while our category of based objects makes no reference to a particular diagram. But this makes no difference; the construction of the colimit is identical in both cases.

We can instantiate the category of based objects to give the category of based theories; this is the only instance of based objects which we will need. This category will be used in the next section to define the semantics of the theory-building operations.

```
type Based_Theory(o,m,beta) ==
            BasedObj(Theory(o,m,beta),Theory_Mor(o,m,beta))

type Based_Theory_Mor(o,m,beta) ==
            BasedObj_Mor(Theory(o,m,beta),Theory_Mor(o,m,beta))

dec colim_cat_of_based_theories : Institution(o,m,alpha,beta) ->
            Colimit_Cat(Based_Theory(o,m,beta),
                        Based_Theory_Mor(o,m,beta))

--- colim_cat_of_based_theories I <=
            colim_cat_of_based_objects(colim_cat_of_theories I)
```

3. Semantic operations

In this section the semantics of Clear's theory-building operations will be given. These will then be used in the semantic equations of the next section.

The definitions of these operations depend crucially on the properties of the colimit in the category of based theories defined in the previous section. The denotation of a Clear specification is a based theory, and all of our work until now has been carefully directed so that the combine and apply operations can be elegantly defined as nothing more than simple colimits in this category. The remaining operations (enrich, data and derive) are defined readily but less gracefully in terms of lower-level manipulations of the based theories themselves.

3.1. Combine

This function implements the '+' theory-building operation of Clear.

```
dec combine : Institution(o,m,alpha,beta) ->
              (Based_Theory(o,m,beta) # Based_Theory(o,m,beta) ->
                    Based_Theory(o,m,beta))

--- combine I <=
     lambda t1,t2 =>
         let colimit_cat(_,bthcolim) ==
                  colim_cat_of_based_theories I in
         let cpcone,_ == bthcolim(cpdiagram(t1,t2)) in
         apex cpcone
```

That is, combine I ($\underline{T}_1$,$\underline{T}_2$) is the coproduct of the based theories $\underline{T}_1$ and $\underline{T}_2$ (cpdiagram is an auxiliary function which produces a two-node coproduct diagram, given the objects to be attached to the nodes). Because we are dealing with based theories, combine will treat shared subtheories properly.

3.2. Enrich

The treatment of enrich here is different from that in the set-theoretic semantics. The denotation of an enrichment there was just

some new sorts, operators and axioms; here an enrichment is a theory morphism of the form $\sigma:\langle\underline{\Sigma},\bar{\emptyset}\rangle\rightarrow\langle\underline{\Sigma}',E'\rangle$, where $\underline{\Sigma}$ is the signature of the theory being enriched, $\underline{\Sigma}'$ is the signature of the enriched theory, and E' are the new axioms (closed). Enrich applies this morphism to the based theory being enriched to give the enriched based theory. This approach is necessary in order to define the enrich operation under an arbitrary institution. The theory morphism representing the enrichment must be built differently under each institution, for it requires the manipulation of signatures as something more than impenetrable objects in a category. The enrichment operation is defined later in this section for the usual institution of Clear; it takes the signature to be enriched and the new sorts, operators and axioms, and gives the theory morphism needed here (it will always be an inclusion in this case). The data and add-equality operations (defined later) can be applied to this morphism in the case of a data enrichment, modifying it to include the appropriate new data constraint and equality operators.

```
dec enrich : Institution(o,m,alpha,beta) ->
             (Based_Theory(o,m,beta) # Theory_Mor(o,m,beta) ->
                    Based_Theory(o,m,beta))

--- enrich I <=
     lambda t, theory_mor(_,g,theory(sig1,eq1)) =>
         let th & theory(_,eq) == apex t in
         let th1 == theory(sig1,union(eq1,translate(g,eq))) in
         right_compose ( cat_of_diagrams(cat_of_theories I),
                         cat_of_theories I,
                         unitdiag(cat_of_theories I) )
                       ( theory_mor(th,g,th1), t )
```

That is, the result of enrich($\underline{T},\sigma:\langle\underline{\Sigma},\bar{\emptyset}\rangle\rightarrow\langle\underline{\Sigma}',E'\rangle$) is the theory $\langle\underline{\Sigma}',\overline{E'\cup\sigma(eqns(\underline{T}))}\rangle$ with the base of $\underline{T}$ attached (this is the action of right_compose).

3.3. Derive

The derive operation is used to change the signature of a theory. Under the usual institution this means forgetting some sorts and operators and possibly renaming the ones remaining; under an arbitrary institution signatures may not consist of names at all so we cannot speak about forgetting or renaming. Yet, the semantics of

derive under an arbitrary institution is the same as that given in the set-theoretic semantics for the special case of ordinary Clear. Given a $\underline{\Sigma}$-theory and a $\underline{\Sigma}'$-theory and a signature morphism $\sigma:\underline{\Sigma}\rightarrow\underline{\Sigma}'$, derive produces a theory with the signature and base of the $\underline{\Sigma}$-theory, which has for axioms the inverse image under σ of the axioms of the $\underline{\Sigma}'$-theory. The model-theoretic condition (the Satisfaction Lemma) which made this the appropriate set of axioms in the case of ordinary Clear reappears as a condition on the satisfaction relation ($\models$) of an institution, with the same result.

The semantics of derive is split into two parts. The quotient function produces the resulting theory, which must then be attached to the appropriate base.

Def: If $\underline{T}=\langle\underline{\Sigma},E\rangle$ and $\underline{T}'=\langle\underline{\Sigma}',E'\rangle$ are theories and $\sigma:\underline{T}\rightarrow\underline{T}'$ is a theory morphism, then the quotient of $\underline{T}$ by σ (written $\underline{T}/\sigma$) is the theory $\langle\underline{\Sigma},\sigma^{-1}(E')\rangle$, where $\sigma^{-1}(E') = \{e \mid \sigma(e)\in E'\}$. The identity signature morphism $1_{\underline{\Sigma}}$ gives a theory morphism $1_{\underline{\Sigma}}:\underline{T}\rightarrow\underline{T}/\sigma$ denoted by quotient($\underline{T}$,σ) (because σ is a theory morphism implies that $E\subseteq\sigma^{-1}(E')$).

$\underline{T}/\sigma$ will always be a theory because of the following fact, a generalised version of a fact from chapter III:

Fact: If E is closed then $\sigma^{-1}(E)$ is closed, under any institution.

Proof: Identical to the proof outlined in section III.2.4, except that we appeal to the condition on the $\models$ relation of an institution rather than to the Satisfaction Lemma.

An intermediate step in the proof of this fact shows (by a model-theoretic argument) why $\underline{T}/\sigma$ (with a suitable base) is the appropriate result of the derive operation -- see section III.2.4 for details.

Once the quotient function is defined as above, with type:

```
dec quotient : Institution(o,m,alpha,beta) ->
               (Theory(o,m,beta) # Theory_Mor(o,m,beta) ->
                      Theory_Mor(o,m,beta))
```

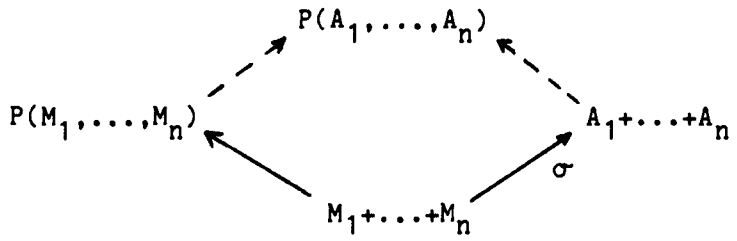
the derive operation can be easily defined:

```
dec derive :  Institution(o,m,alpha,beta) ->
          (Based_Theory(o,m,beta) # m # Based_Theory(o,m,beta) ->
                    Based_Theory(o,m,beta))

--- derive I <=
     lambda t1, sigma, t2 =>
         let tsigma ==
             extend_signature_morphism(apex t1,sigma,apex t2) in
         right_compose ( cat_of_diagrams(cat_of_theories I),
                         cat_of_theories I,
                         unitdiag(cat_of_theories I) )
                       ( quotient(apex t1,tsigma), t1 )
```

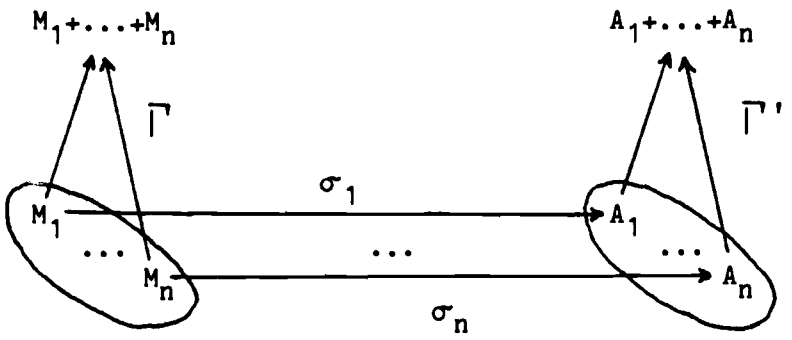
That is, it is the quotient with the base of the first theory attached.

3.4. Apply

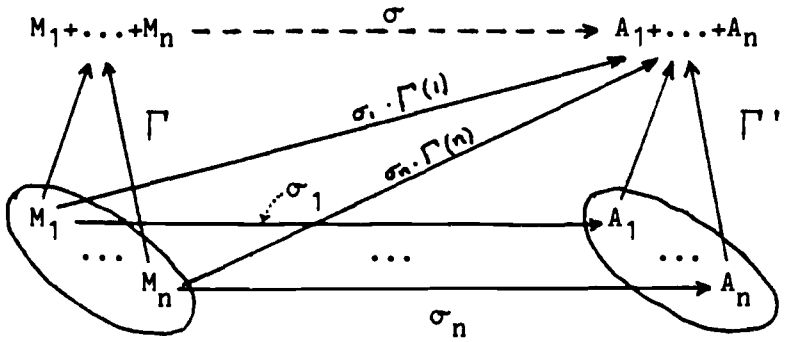
Apply defines the application of a theory procedure to its arguments. A theory procedure here is represented as a based theory morphism (from the coproduct of the metasort theories to the theory described by the procedure body); under the usual institution this morphism is an inclusion. The result of a procedure application is the pushout of this morphism and the combined fitting morphism from the coproduct of the metasorts to the coproduct of the actual parameter theories:



This is straightforward except for the construction of the combined fitting morphism σ. We are given based theory morphisms $\sigma_1 : M_1 \rightarrow A_1$, ..., $\sigma_n : M_n \rightarrow A_n$ and wish to construct $\sigma : M_1+\ldots+M_n \rightarrow A_1+\ldots+A_n$. Taking the two coproducts gives the following situation:



Now the 'universal part' of the metasort coproduct may be used to construct a morphism to the apex of the actual parameter colimit, using the 'pretend coproduct' (i.e. another cone on the same base) of the metasorts formed by composing $\sigma_1,\ldots,\sigma_n$ with $\Gamma'(1),\ldots,\Gamma'(n)$:

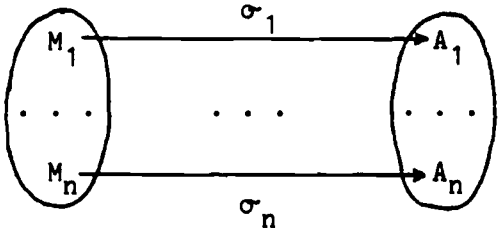


This σ must be the correct morphism because it is the unique morphism from $M_1+\ldots+M_n$ to $A_1+\ldots+A_n$ for which everything commutes.

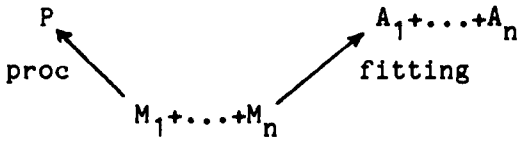
```
dec apply :  Institution(o,m,alpha,beta) ->
       (Based_Theory_Mor(o,m,beta)
        # list(Based_Theory_Mor(o,m,beta)) ->
                    Based_Theory(o,m,beta))
```

```
--- apply I <=
     lambda proc, fittings_list =>
         let colimit_cat(catbth,bthcolim) ==
                 colim_cat_of_based_theories I in
         let Dm & diagram_mor(Ds,_,_,Dt) ==
                 cpmdiagram catbth fittings_list in
         let actual_parameter,_ == colimit Dt in
         let _,univ == colimit Ds in
         let pretendcoprodcone ==
                 left_compose ( cat_of_diagrams catbth,
                                catbth,
                                unitdiag catbth )
                              ( Dm, actual_parameter ) in
         let fitting == apex_morphism(univ pretendcoprodcone) in
         let pocone,_ ==
                 colimit(podiagram catbth (proc,fitting)) in
         apex pocone
```

Given the morphisms $\sigma_1,\ldots,\sigma_n$ the constant cpmdiagram produces the diagram morphism:



and podiagram produces a pushout diagram:



The definitions of these auxiliary functions are omitted.

3.5. Copy

The copy operation makes a fresh copy of a theory, preserving a given set of subtheories. This is just a matter of attaching a restricted base to the theory, the base of the combined subtheories to be preserved. The semantics of copy was not included in [Burstall and Goguen 1980].

```
dec copy :  Institution(o,m,alpha,beta) ->
              (Based_Theory(o,m,beta) # Based_Theory(o,m,beta) ->
                    Based_Theory(o,m,beta))

--- copy I <=
     lambda t1, t2 =>
         left_compose ( cat_of_diagrams (cat_of_theories I),
                        cat_of_theories I,
                        unitdiag (cat_of_theories I) )
                      ( diagram_inclusion (cat_of_theories I)
                                          (base t2, base t1), t1 )
```

The constants base (base of a cone) and diagram_inclusion (producing a morphism which is the inclusion of one diagram in another) are auxiliary functions whose definitions are omitted. The second argument (of the lambda) of copy is the sum of the subtheories to be preserved. The base of this subtheory is attached to the theory to be copied using left_compose.

3.6. Data

The data operation cannot be used under an arbitrary institution. As mentioned in chapter I, we need a data institution; this is an institution in which the models of a theory will always form a category and a theory morphism gives rise to forgetful and free functors (see [Burstall and Goguen 1980] for details). This is an aspect we do not attempt to treat in our implementation. But the other special characteristic of a data institution is that the class of axioms must include data constraints. A data constraint is a theory morphism together with a signature morphism, and so we can define data axioms (axioms which include data constraints, but are otherwise unspecified) as follows:

```
data Data_Axiom(o,m,beta) ==
       axiom(beta) ++ data_constraint(Theory_Mor(o,m,beta),m)
```

A data institution is then an ordinary institution with the type of axioms instantiated to data axioms:

```
type Data_Institution(o,m,alpha,beta) ==
       Institution(o,m,alpha,Data_Axiom(o,m,beta))
```

Data theories and their morphisms are easily defined (based data

theories and their morphisms similarly):

```
type Data_Theory(o,m,beta) == Theory(o,m,Data_Axiom(o,m,beta))

type Data_Theory_Mor(o,m,beta) ==
                    Theory_Mor(o,m,Data_Axiom(o,m,beta))
```

Now we can define the semantics of the data operation. It takes a simple theory morphism representing an enrichment, and converts it to a data theory morphism with a target which includes a data constraint describing the enrichment. The modified enrichment can then be used by the enrich operation (defined above) to 'data-enrich' a theory. Since the data operation manipulates data constraints, it works only under a data institution; this fact is reflected in its type.

```
dec data :  Data_Institution(o,m,alpha,beta) ->
             (Theory_Mor(o,m,beta) -> Data_Theory_Mor(o,m,beta))

--- data(institution(colimit_cat(cat(_,_,id,_),_),_,_,_)) <=
     lambda F & theory_mor(t,sigma,theory(sig1,E1)) =>
         let constraint == data_constraint(F,id sig1) in
         let axioms == union(datafy E1,close {constraint}) in
         let t1 == theory(sig1,axioms) in
         theory_mor(t,sigma,t1)
```

If the enrichment is given by the (simple) theory morphism $\sigma:\underline{T}\longrightarrow\langle\underline{\Sigma}',E'\rangle$, then the data constraint added by application of the data operation will be $C = \langle\sigma,1_{\underline{\Sigma}'}\rangle$, and the resulting (data) enrichment is $\sigma:\underline{T}\longrightarrow\langle\underline{\Sigma}',\overline{E'\cup\{C\}}\rangle$. But E' is a simple agglomerate; it must first be converted to a 'data agglomerate' using the auxiliary constant 'datafy' whose definition is omitted.

3.7. Enrichment

This operation constructs a theory morphism representing a theory enrichment for use by the enrich operation defined above. As mentioned before, the enrichment operation must be dependent on a particular institution for it deals with the internal structure of signatures. The enrichment operation will be defined here for ordinary Clear (Clear under the usual institution). Although the definition is dependent on a particular notion of signature (the one discussed in section 2) it is independent of the other elements of

an institution and so we can leave these unrestricted for now. The enrichment operation is parameterised by a signed institution (that is, an institution with the usual kind of signatures, where character strings are used for sort and operator names):

```
type Signed_Institution(alpha,beta) ==
        Institution(Signature Name,Signature_Mor Name,alpha,beta)
```

The type Name was defined in section 1 as an abbreviation for 'list char'. Signed institutions could easily be parameterised by the type of sort and operator names -- in fact, this is done in the semantics program -- but for simplicity we will use the fixed type Name. Signed theories and their morphisms are then defined as follows (based signed theories and morphisms similarly):

```
type Signed_Theory(beta) ==
        Theory(Signature Name,Signature_Mor Name,beta)

type Signed_Theory_Mor(beta) ==
        Theory_Mor(Signature Name,Signature_Mor Name,beta)
```

Note that this specialisation to signed institutions and signed theories is orthogonal to the previous specialisation to data institutions and data theories. Signed data institutions, signed data theories and their morphisms, and based signed data theories and their morphisms are easily defined.

Enrichment takes the signature to be enriched (Σ) and some new sorts, operators and axioms (S', Σ', E'), and produces the theory inclusion from $\langle\Sigma,\emptyset\rangle$ to $\langle\Sigma \cup \langle S',\Sigma'\rangle,\overline{E'}\rangle$. This operation is defined using an auxiliary function whose definition is omitted which produces the signature of the enriched theory; the inclusion function on signatures (producing an inclusion of one signature in another) is also not defined here.

```
type S_Name == Tag Name                 ! sort name

type O_Name == Tag Name                 ! operator name

dec enriched_signature :
      Signature Name # set S_Name # set(O_Name # list S_Name) ->
                Signature Name          ! O_Name # list S_Name is an
                                        ! operator with its arity
```

```
dec enrichment : Signed_Institution(alpha,beta) ->
        (Signature Name # set S_Name # set(O_Name # list S_Name)
                        # set beta ->    Signed_Theory_Mor(beta))

--- enrichment SI <=
     lambda sig & (O,_,S), S1, O1, E1 =>
         let sig2 == enriched_signature(sig,S1,O1) in
         let t == theory(sig,close nil_set) in
         let t2 == theory(sig2,close E1) in
         theory_mor(t,inclusion(sig,sig2),t2)
```

Why must the sort and operator names be 'tagged' (types S_Name and O_Name rather than simply Name)? The reason is that the arities of the new operators may refer to sorts in the 'old' signature. Since this signature may have been formed by putting together several signatures (using combine, for example), it may contain several sorts or operators with the same name (but tagged in different ways as a result of the colimit inherent in the combine operation). The enrichment operation must be supplied with tagged arities to disambiguate in such cases, and the sort and operator names are required to be tagged as well for uniformity. This is of course invisible to the user of the specification language; it is a detail which must be handled by the semantics (specifically, by the notion of a dictionary discussed in section 4.1). Note that these tags bear some resemblance to the tags of the set-theoretic semantics, although here they are part of the colimit mechanism rather than an explicit ingredient of the semantics.

3.8. Add equality

A side effect of the data operation is the introduction of an equality predicate ==:s,s->bool for each 'data' sort s. The operators are easily added to the signature, and the add_equality agglomerate constructor defined model-theoretically in section III.2.3 is used here as well to add the axioms which specify the meaning of the new operators.

If S is the set of new sorts and E is the set of axioms already in a theory, then E^S is E together with all the axioms needed to define the new equality relations on sorts of S. This is denoted in

the program by the agglomerate add_equality(S,E). Note that the theory being enriched must include Bool.

To define the add equality operation (on theories), we use an auxiliary function which produces an equality operator with arity s,s->bool when given the sort s; its definition is omitted. The enriched_signature operation mentioned above is used to form a signature which includes the new equality operators.

```
dec equality_operator : S_Name -> O_Name # list S_Name

. . .

dec add_equality : Signed_Institution(alpha,beta) ->
            (Signed_Theory_Mor(beta) -> Signed_Theory_Mor(beta))

--- add_equality SI <=
     lambda theory_mor(t & theory(sig,_),sigma,theory(sig1,E1)) =>
         let data_sorts == S1 - S where (_,_,S) == sig
                                  where (_,_,S1) == sig1 in
         let new_operators == equality_operator * data_sorts in
         let sig2 == enriched_signature(sig1,nil_set,
                                        new_operators) in
         let t2 == theory(sig2,add_equality(S,E1)) in
         theory_mor(t,inclusion(sig,sig2),t2)
```

The definition is similar to that of the data operation above. A theory morphism describing an enrichment is modified to further enrich by the new equality operators and the axioms which define them.

4. Semantic equations

In this section the semantic equations for Clear are given, building on the semantic operations defined in the previous section. This parallels section 4 of the set-theoretic semantics. Since many of the equations are identical (i.e. all those in levels I and IIb) only those which are different are given, along with new definitions of dictionary and environment. The equations will be given in the notation of denotational semantics, rather than in HOPE. This should make them slightly easier to read, and the translation to HOPE is straightforward (see section 4.2 for an example).

4.1. Dictionaries

The notion of a dictionary in this semantics is identical to the one presented in section III.4.1 of the set-theoretic semantics. The only difference is the way that the dict operation (which produces a dictionary) is defined. Recall that a dictionary gives the correspondence between a sort expression or operator expression (such as 's of T') and the sort or operator to which it refers.

Def: A dictionary is a pair of functions <sd,od> where

sd : sort-name x theory-name → sort
od : operator-name x theory-name → operator

In the implementation, the two components of a dictionary are functions which return tagged names; this is because there may be more than one sort or operator with the same name, as discussed earlier.

```
data Dictionary == dictionary((Name # Name -> S_Name),
                              (Name # Name -> O_Name))
```

The operation dict constructs a dictionary from a based theory, yielding a dictionary which interprets expressions referring to sorts and operators in that theory.

```
dec dict : Signed_Based_Theory(beta) -> Dictionary

--- dict(_, diagram_mor(_,_,nm,_), _) <=
     let d ==
          (lambda tn =>
             let theory_mor(_,fcomma_mor(_,(mor(_,fo,_),
                                            mor(_,fs,_)),_),_) ==
                  nm of (const tn) in (fo,fs))  in
```

```
let sd == (lambda sn,tn =>
              let (_,fs) == d(tn) in find(fs,sn)) in
let od == (lambda on,tn =>
              let (fo,_) == d(tn) in find(fo,on)) in
dictionary(sd,od)
```

In the above definition, nm is the map taking nodes in the base of a based theory to theory morphisms from base theories to the apex theory (the flanks). The nodes in the base of a based theory are labelled by (tagged) theory names, since the base is always a subdiagram of the environment (see section 4.5 for the reason for the tag 'const'). The value of d applied to a theory name will thus be a pair (fo,fs), where fo is a map taking operators in the base theory to the corresponding operators in the apex theory, and fs does the same for sorts. Given an expression 'sn of tn' (similarly 'on of tn'), the sort sn should appear in the domain of fs (where (fo,fs) = d(tn)) and can thus be mapped to its name in the apex theory. But sn itself will not be in the domain of fs; some tagged version of sn will be (and it might not be simply 'just sn', since the theory at node tn may be the result of a combine or apply operation). There should be only one such sort, or else the expression is ambiguous. So the auxiliary function find is used to search for the result corresponding to a tagged version of the sort name; it gives an error if there is more than one choice. This is a subtle point which was not revealed in [Burstall and Goguen 1980]. In general there is a problem in determining which sort or operator in a theory produced using a series of theory-building operations corresponds to a sort or operator name. The problem could be solved by keeping track of the original name associated with each tagged name. In our implementation this correspondence is fortunately easy to establish.

4.2. Level I: Sorts, operators, terms

The semantic equations for level I are exactly the same as those for level I of the set-theoretic semantics (section III.4.2). In order to justify writing the semantic equations using the notation of denotational semantics rather than HOPE, an example of how the translation may be accomplished will now be given.

The syntax of sort, operator and term expressions is defined in section III.4.2 by the following BNF syntax:

```
sex ::= s | s of T
oex ::= o | o of T
tex ::= x | oex(tex1,...,texn)
```

where s is a sort name (lower case identifier), o is an operator name (identifier or operator symbol), T is a theory name (capitalised identifier) and x is a variable name (identifier). This may easily be converted to a sequence of HOPE data declarations:

```
infix of : 5
distfix _ << _ >>

data Sex == just Name ++ Name of Name
data Oex == just Name ++ Name of Name
data Tex == just Name ++ Oex << list Tex >>
```

Distributed-fix operators can be used to give an approximation to Clear syntax. Mutually recursive data definitions are also possible in HOPE using the with construct:

```
data . . . == . . .
with . . . == . . .
with . . . == . . .
```

The three semantic functions of this level may be declared as follows:

```
dec Sex : Sex -> (Signature Name -> (Dictionary -> S_Name))

dec Oex : Oex -> (Signature Name -> (Dictionary -> O_Name))

dec Tex : Tex -> (Signature Name -> (Dictionary ->
                          ((Name --> S_Name) -> Term)))
          ! Name --> S_Name associates variables with their sorts
```

The denotation of a term expression is a term:

```
data Term == just Name ++ O_Name << list Term >>
```

Now the semantic equations of section III.4.2 can be translated into HOPE. For example, the second equation defining the function Tex is:

```
Tex[[oex(tex_1,...,tex_n)]]Σ dX =
    let ω = Oex[[oex]]Σ d in
    let tm_1,...,tm_n = Tex[[tex_1]]Σ dX,...,Tex[[tex_n]]Σ dX in
      ω(tm_1,...,tm_n) (a Σ-term on X)
```

This becomes:

```
--- Tex(oex << list_tex >>) <=
    (lambda sigma => (lambda d => (lambda X =>
       let omega == Oex oex sigma d in
       let list_tm ==
              (lambda tex => Tex tex sigma d X) * list_tex in
       omega << list_tm>> )))
```

The notation of denotational semantic will be used henceforth for clarity, as mentioned already.

4.3. Level IIa: Enrichments

The level IIa semantic equations are very similar to those in the set-theoretic semantics (section III.4.3). The equations for Sd and Od (giving the semantics of sort and operator declarations) are the same except that the unique tags required by the set-theoretic semantics need not be attached here. The equations for Enrb and Enr (the semantics of enrichments) are different because the definitions of the enrich and data operations in section 3 operate on theory morphisms rather than directly on theories.

The semantic operations from section 3 will be needed in the equations below, so it is finally necessary to select a particular institution. We want axioms to be equations:

```
distfix all _ . _ = _

data Eqn == all (Name-->S_Name) . Term = Term
```

Note that an Eqn is a semantic object, as distinct from the equations which appear in specifications, defined as follows:

```
data Eq == all Varl . Tex = Tex
```

(Varl and Tex are other syntactic types.) The Eq semantic operation defined below translates an Eq to an Eqn.

We have already decided in section 3 on the kind of signatures we will use, so the institution we want is defined as follows:

```
dec Clear_Institution : Signed_Data_Institution(alpha,Eqn)

--- Clear_Institution <= institution(colimit_cat_of_signatures,
                                     functor( ..., ... ),
                                     functor( ..., ... ),
                                     ... )
```

Our implementation does not deal at all with the model-theoretic aspects of the semantics and does not manipulate equations in non-trivial ways, so the first component of the institution (and all types except those of models) are all that is needed. In the program, the ...'s are replaced by the function 'error', but anything (well-typed) will do since it will never be accessed.

The above definition is sufficient for the purposes of the program. But to make sure that such an institution really exists we must be more specific. All we have specified so far is the category of signatures $\underline{\underline{Sig}}$ and the form of axioms.

- The functor $Sen:\underline{\underline{Sig}} \longrightarrow \underline{\underline{Set}}$ takes a signature to the set of axioms (equations and data constraints) on that signature, and takes a morphism $\sigma:\underline{\Sigma} \longrightarrow \underline{\Sigma}'$ to the set morphism $\sigma:Sen(\underline{\Sigma}) \longrightarrow Sen(\underline{\Sigma}')$ defined in sections II.3 and II.5 which translates a $\underline{\Sigma}$-axiom to a $\underline{\Sigma}'$-axiom.

- We take as models the algebras defined in section II.2; the functor $Mod:\underline{\underline{Sig}} \longrightarrow \underline{\underline{Set}}^{op}$ takes a signature $\underline{\Sigma}$ to the set of all $\underline{\Sigma}$-algebras, and takes a morphism $\sigma:\underline{\Sigma} \longrightarrow \underline{\Sigma}'$ to the set morphism $(\_)|^{\sigma}_{\underline{\Sigma}}:Mod(\underline{\Sigma}') \longrightarrow Mod(\underline{\Sigma})$ (which takes a $\underline{\Sigma}'$-algebra to its $\underline{\Sigma}$-restriction).

- The relation $\models_{\underline{\Sigma}} \subseteq Mod(\underline{\Sigma}) \times Sen(\underline{\Sigma})$ is the satisfaction relation defined in sections II.3 and II.5 for equations and data constraints respectively.

This is clearly an institution (recall the Satisfaction Lemma of section II.3) so we can proceed.

We need to define theories and their morphisms under this institution:

type Clear_Theory == Signed_Data_Theory(Eqn)

type Clear_Theory_Mor == Signed_Data_Theory_Mor(Eqn)

Based Clear theories and morphisms are defined similarly.

Before we can use the enrichment operation below, we need to define another institution for dealing with simple theories and morphisms (i.e. without constraints). The definition is identical to the definition of Clear_Institution above except for the type declaration:

dec Simple_Clear_Institution : Signed_Institution(alpha,Eqn)

Simple Clear theories and morphisms are just the same as signed theories and their morphisms, defined above.

Semantic functions

Sd, Od, Varl, Eq : same as in section III.4.3
Enrb : enrichment-body $\longrightarrow$ signature $\longrightarrow$ dictionary
$\longrightarrow$ simple-Clear-theory-morphism
Enr : enrichment $\longrightarrow$ signature $\longrightarrow$ dictionary
$\longrightarrow$ Clear-theory-morphism

Semantic equations

$Sd[\![s]\!] = just\ s$

$Od[\![o: sex_1,\ldots,sex_n \to sex]\!]\Sigma d =$
$\quad$ let $s_1,\ldots,s_n,s = Sex[\![sex_1]\!]\Sigma d,\ldots,Sex[\![sex_n]\!]\Sigma d,Sex[\![sex]\!]\Sigma d$ in
$\quad\quad \langle just\ o, string(s_1,\ldots,s_n,s)\rangle$

$Varl[\![x_{11},\ldots,x_{1n_1}:sex_1,\ldots,x_{m1},\ldots,x_{mn_m}:sex_m]\!]\Sigma d =$
$\quad$ let $s_1,\ldots,s_m = Sex[\![sex_1]\!]\Sigma d,\ldots,Sex[\![sex_m]\!]\Sigma d$ in
$\quad\quad \{ \langle x_{11},s_1\rangle,\ldots,\langle x_{1n_1},s_1\rangle,$
$\quad\quad\quad \ldots$
$\quad\quad\quad \langle x_{m1},s_m\rangle,\ldots,\langle x_{mn_m},s_m\rangle \}$ (a map Name --> S_Name)

$Eq[\![$all varl. $tex_1 = tex_2]\!]\Sigma d =$
$\quad$ let $X = Varl[\![varl]\!]\Sigma d$ in
$\quad$ let $tm_1,tm_2 = Tex[\![tex_1]\!]\Sigma dX,Tex[\![tex_2]\!]\Sigma dX$ in
$\quad\quad$ all X. $tm_1 = tm_2$ (an Eqn)

```
Enrb[[sorts sd_1,...,sd_m opns od_1...od_n eqns eq_1...eq_p]]Σd =
     let S' = {Sd[[sd_1]],...,Sd[[sd_m]]} in
     let Σ' = enriched_signature(Σ,S',∅) in
     let Σ' = {Od[[od_1]]Σ'd,...,Od[[od_n]]Σ'd} in
     let Σ" = enriched_signature(Σ',∅,Σ') in
     let E' = {Eq[[eq_1]]Σ"d,...,Eq[[eq_p]]Σ"d} in
       enrichment Simple_Clear_Institution (Σ,S',Σ',E')

Enr[[enrb]]Σd = datafy Enrb[[enrb]]Σd

Enr[[data enrb]]Σd =
     add_equality Clear_Institution
                  (data Clear_Institution Enrb[[enrb]]Σd)
```

Datafy (used in the first Enr equation above) is an auxiliary function which converts a simple Clear theory morphism to a (data) Clear theory morphism; its definition is omitted.

4.4. Level IIb: Signature changes

This level is absolutely identical with section III.4.4 of the set-theoretic semantics.

4.5. Environments

It has already been mentioned (when based objects were discussed in section 2) that the environment must record the relationships between values (theories) as well as the values themselves. This leads to the natural representation of the environment as a diagram on the category of theories, where the edges describe how theories have been put together to make other theories. This is a generalisation of the usual notion of environment in denotational semantics, which simply maps names to values. Metatheories and constant theories must both be stored in the same environment, since the relationship between a metatheory and all its constant subtheories must be recorded as well as the relationships between constant theories. So two of the three environments used in the

set-theoretic semantics are combined here into a single environment, where the two kinds of theories are bound in different ways; the third environment (the procedure environment) remains separate.

Here several operations for creating and manipulating environments are defined. Environments can be defined without reference to the properties of the values which they contain, so these operations are parameterised by an arbitrary category (with a colimit function, which is needed for the node_morphism operation). The type of an environment is just the same as the type of a diagram:

```
type Env(o,m) == Diagram(o,m)
```

None of the programs are given here; they are all straightforward albeit somewhat long and complicated.

The first operation is easy; nil_diagram (the diagram with no nodes) is the empty environment.

Next, we need to bind new values into the environment. Ordinary (constant) theories are bound in a different way from metatheories, since the two cases must be handled differently when the time comes to retrieve values from the environment. Each name is tagged to indicate whether the associated value is constant or meta (recall that the names of nodes in a diagram are tagged already):

```
data Tag alpha == . . . ++ const alpha ++ meta alpha
```

The operation bind is used to bind a constant (theory) into the environment:

```
dec bind : Colimit_Cat(o,m) ->
                (Name # BasedObj(o,m) # Env(o,m) -> Env(o,m))
```

Bind is defined as follows:

Def: Given an environment diagram D, a name i not in D, and a based object O (with base included in D), the value of bind(i,O,D) is the diagram D' where:

- The nodes of D' are those of D together with a node with the name const(i) and the value apex(O), and
- The edges of D' are those of D together with an edge for each morphism in the flanks of O (going from the base node in D' to the apex of O at const(i) in D').

The operation bind_meta for binding a meta(theory) into an environment is defined identically (with the same type), except that the name meta(i) is used instead of const(i).

The operation bind is also defined for n-tuples of names and based objects:

```
dec bind : Colimit_Cat(o,m) ->
                (list Name # list(BasedObj(o,m)) # Env(o,m)
                                -> Env(o,m))
```

This operation binds each name in the list to the corresponding value (as a constant). There is no need for a bind_meta operation on n-tuples.

The retrieve operation finds the value in an environment which is associated with a given name, and constructs the corresponding based object. As mentioned, it works differently depending on whether the value is a constant or a meta (theory); the only difference is that the base of the result for a metatheory will not include the metatheory itself.

```
dec retrieve : Colimit_Cat(o,m) ->
                (Name # Env(o,m) -> BasedObj(o,m))
```

Def: Given a name i and an environment diagram D including either const(i) or meta(i), the value of retrieve(i,D) is the based object O (with base included in D), where:

- The apex of O is the value attached to the node const(i) or meta(i) in D.
- For the base of O there are two cases. By support(j,D) we mean the set of nodes in D which have a path to j (but not including j itself).
  - . D includes const(i): The base of O is D restricted to the nodes support(const(i),D)$\cup${const(i)}.
  - . D includes meta(i): The base of O is D restricted to the nodes support(meta(i),D).
- For each node k in the base of O, the flank morphism from k to the apex of O is the composition of morphisms along the path from k to const(i) or meta(i).

The same result would be obtained if metatheories were treated as parameterless procedures, as in section III.4.5.

We will need an operation to restrict the base of a based object to make it compatible with a restricted environment. This is necessary for the semantics of local declarations, since locally declared theories have limited scope. At the end of their scope they must be removed from the bases of objects they have been used to build.

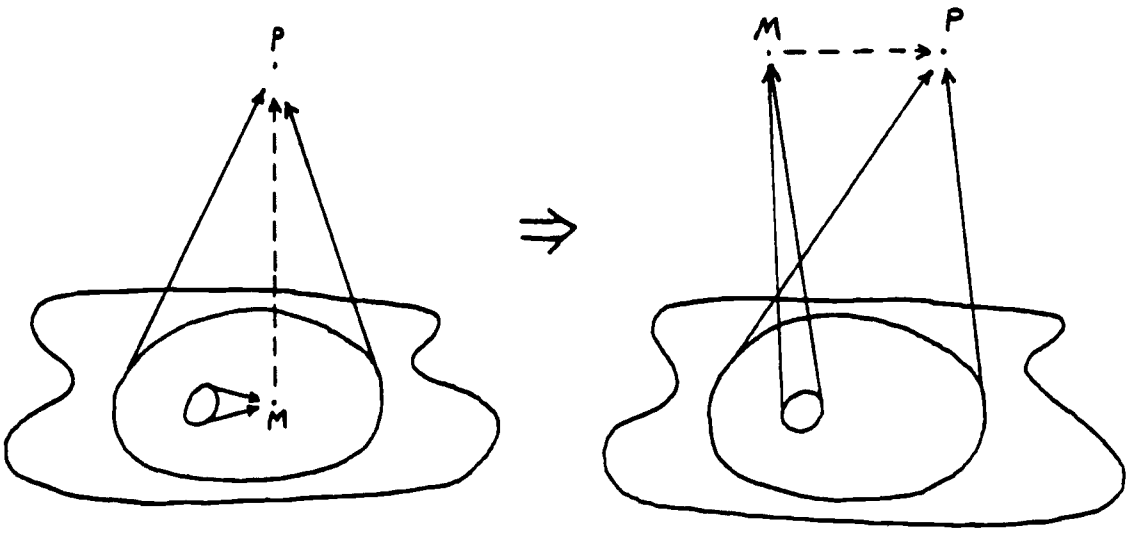
```
dec restrict : Colimit_Cat(o,m) ->
                  (BasedObj(o,m) # Env(o,m) -> BasedObj(o,m))
```

Def: If O is a based object and D is an environment diagram, then restrict(O,D) is the based object O' where the base of O' is the intersection of the base of O and D, apex(O')=apex(O), and the flank morphisms of O' are those of O which come from nodes appearing in the base of O'.

The operation restrict is also defined on based object morphisms:

```
dec restrict : Colimit_Cat(o,m) ->
                (BasedObj_Mor(o,m) # Env(o,m) -> BasedObj_Mor(o,m))
```

Finally, we need a special operation (called node_morphism) for constructing the denotation of a procedure which (as has already been mentioned) is a based theory morphism from the coproduct of the metasort theories to the theory described by the procedure body. The metasort theories are (normally) included in the base of the theory given by the body, so except for the complication of taking a coproduct in the case of multiple metasorts the result is essentially the flank morphism from the metasort to the apex of the procedure body. For the case of a single metasort:




```
dec node_morphism : Colimit_Cat(o,m) ->
                    (list Name # BasedObj(o,m) # Env(o,m)
                                  -> BasedObj_Mor(o,m))
```

Def: If D is an environment diagram, P is a based object (with base included in D) and $I=[i_1,\ldots,i_n]$ is a list of names of nodes in D, then the value of node_morphism(I,P,D) is:

- The unique morphism from the coproduct retrieve(i_1,D) + ... + retrieve(i_n,D) to P, if the nodes $i_1,\ldots,i_n$ are in the base of P
- Error, if some node i_j is not in the base of P

The result of node_morphism is constructed using the 'universal part' obtained from the coproduct of the metasort theories. If n=1, then the result is as shown above.

All of these operations are parameterised by the (colimit)

category of values stored in the environment. In the case of ordinary Clear the values are based Clear theories, defined previously. The name 'Clear_cat' will be used for this category rather than the more descriptive but long-winded 'colim_cat_of_based_Clear_theories'.

```
dec Clear_cat : Colimit_Cat(Based_Clear_Theory,
                            Based_Clear_Theory_Mor)

--- Clear_cat <= colim_cat_of_based_theories(Clear_Institution)
```

The environment which keeps track of constant theories and metatheories has just been defined. We also need an environment for theory procedures. This is just a map from procedure names to their values, as in the set-theoretic semantics. The denotation of a theory procedure is a based theory morphism (from the coproduct of the metasorts to the procedure body). However, in order to apply the procedure we also need to know the metasort theories so that we can determine the fitting morphisms between the metasorts and the actual parameter theories. The procedure environment therefore must map procedure names to pairs consisting of a based theory morphism and a list of based theories (the metasorts):

```
type Proc_Env(o,m) == Name --> Based_Theory_Mor(o,m,beta)
                               # list(Based_Theory(o,m,beta))
```

No special operations will be needed for manipulating procedure environments; bind and retrieve are as usual for maps (we write $\overline{w}$(pn) to retrieve the value associated with the name pn from $\overline{w}$, and $\overline{w}$[v/pn] to bind the value v to pn in $\overline{w}$).

4.6. Level III: Theory-building operations

The final level is similar to section III.4.6 of the set-theoretic semantics. The only differences are those stemming from the use of a different set of semantic operations and the more complex notion of environment.

Values

$\underline{T}$: based Clear theory
ρ : environment (constant theories and metatheories)
π : procedure environment

Semantic functions

E : expression → environment → procedure-environment
→ based-Clear-theory
Spec : specification → environment → procedure-environment
→ based-Clear-theory

Semantic equations

$E[\![T]\!]\rho\pi$ = retrieve Clear_cat (T,ρ)

$E[\![$theory enr endth$]\!]\rho\pi$ =
enrich Clear_Institution ($\underline{\Phi}$,Enr$[\![$enr$]\!]\underline{\Phi}$dict($\underline{\Phi}$))
($\underline{\Phi}$ is the empty based theory;
Φ is the empty signature)

$E[\![e_1 + e_2]\!]\rho\pi$ = combine Clear_Institution ($E[\![e_1]\!]\rho\pi$,$E[\![e_2]\!]\rho\pi$)

$E[\![$enrich e by enr enden$]\!]\rho\pi$ =
let $\underline{T}$ = $E[\![e]\!]\rho\pi$ in
enrich Clear_Institution ($\underline{T}$,Enr$[\![$enr$]\!]$signature($\underline{T}$)dict($\underline{T}$))

$E[\![$derive enr using $e_1,\ldots,e_n$ from e by sic endde$]\!]\rho\pi$ =
let $\underline{T}$ = combine Clear_Institution ($E[\![e_1]\!]\rho\pi$,
combine Clear_Institution ($E[\![e_2]\!]\rho\pi$,...)) in
let $\underline{T}'$ = enrich Clear_Institution
($\underline{T}$,Enr$[\![$enr$]\!]$signature($\underline{T}$)dict($\underline{T}$)) in
let $\underline{T}''$ = $E[\![e]\!]\rho\pi$ in
let σ = Sic$[\![$sic$]\!]$signature($\underline{T}'$)signature($\underline{T}''$)dict($\underline{T}''$) in
derive Clear_Institution ($\underline{T}'$,σ,$\underline{T}''$)

```
E[[P(e1[sic1],...,en[sicn])]]ρπ =
     let T1',...,Tn' = E[[e1]]ρπ,...,E[[en]]ρπ in
     let <F,<T1,...,Tn>> = π(P) in
     let σ1,...,σn =
              Sic[[sic1]]signature(T1)signature(T1')dict(T1'),
               . . .
              Sic[[sicn]]signature(Tn)signature(Tn')dict(Tn') in
     let F1,...,Fn =
              extend_signature_morphism Clear_Institution
                                        (T1,σ1,T1'),
               . . .
              extend_signature_morphism Clear_Institution
                                        (Tn,σn,Tn') in
       apply Clear_Institution (F,<F1,...,Fn>)

            (where extend_signature_morphism is the corresponding
            function on based theories rather than theories)


E[[let T = e1 in e2]]ρπ =
     let T = E[[e1]]ρπ in
     let ρ' = bind Clear_cat (T,T,ρ) in
       restrict Clear_cat (E[[e2]]ρ'π,ρ)


E[[copy e using e1,...,en]]ρπ =
     let T = E[[e]]ρπ in
     let T' = combine Clear_Institution (E[[e1]]ρπ,
                    combine Clear_Institution (E[[e2]]ρπ,...)) in
       copy Clear_Institution (T,T')

Spec[[e]]ρπ = E[[e]]ρπ

Spec[[const T = e spec]]ρπ =
     let ρ' = bind Clear_cat (T,E[[e]]ρπ,ρ) in
       Spec[[spec]]ρ'π


Spec[[meta T = e spec]]ρπ =
     let ρ' = bind meta Clear_cat (T,E[[e]]ρπ,ρ) in
       Spec[[spec]]ρ'π
```

```
Spec⟦proc P(T1:e1,...,Tn:en) = e spec⟧ρπ =
    let T1,...,Tn = E⟦e1⟧ρπ,...,E⟦en⟧ρπ in
    let ρ' = bind Clear_cat (<T1,...,Tn>,<T1,...,Tn>,ρ) in
    let T' = E⟦e⟧ρ'π in
    let F = node_morphism Clear_cat (<T1,...,Tn>,T',ρ') in
    let F' = restrict Clear_cat (F,ρ) in
    let T1',...,Tn' = restrict Clear_cat (T1,ρ),
                        . . .
                      restrict Clear_cat (Tn,ρ) in
    let π' = π[<F',<T1',...,Tn'>>/P] in
      Spec⟦spec⟧ρπ'
```

5. Implementation

In the preceding sections an implementation of the category-theoretic semantics of Clear has been presented in parallel with the semantics itself. The finished program written entirely in HOPE is about 1700 lines long and occupies 110K words on a DEC KL-10 computer (where the HOPE system itself occupies 66K words of this total). The only theory in the initial environment of the system is a simple version of Bool.

The system has been tested on several small examples, but as the timing figures below demonstrate it is rather too slow to be used on realistic large specifications such as those in section IV.2.

Example 1

The denotation of the specification

```
const Nat =
  enrich Bool by
    data sorts nat
         opns 0 : nat
              pred, succ : nat -> nat
         eqns all n:nat. pred(succ(n)) = n
              all n:nat. succ(pred(n)) = n    enden
```

```
Nat + Nat
```

is the based theory

```
sorts nat, bool
opns 0, pred, succ, true, false, ...
eqns . . .
```

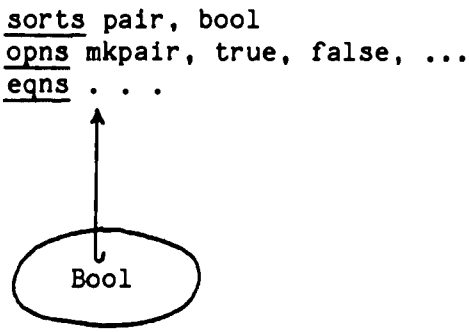
The computation of this result required 4.325 minutes of CPU time (excluding garbage collections).

Example 2

The denotation of the specification

```
meta Triv =
  theory sorts element    endth

proc Pair(X:Triv) =
  enrich Bool + X by
    data sorts pair
         opns mkpair : element of X,element of X -> pair
         eqns all n,m:element of X. mkpair(n,m) = mkpair(m,n)
                                                        enden
Pair(Bool[element is bool])
```

is the based theory



The computation of this result required 1.85 minutes (excluding garbage collections).

Nearly all of the time these rather trivial examples required was consumed in the computation of colimits in the category of based Clear theories. Recall that the application of a theory procedure requires 3 simple colimits in the category of based theories. Each of these colimits requires a larger colimit in the category of theories, which in turn requires a number of colimits in the category of signatures. Each colimit in the category of signatures requires 2 colimits in the category of sets (the category of signatures is a comma category) and each of these requires a number of set coproducts and coequalisers. The second example above required 88 set coproducts and 32 set coequalisers. An intricate and complex manipulation of the results of these operations is then required to convert them into the result of the theory procedure

application. Of course, the speed of the HOPE system itself is an important factor; recoding an earlier version of the colimit program in POP-2 resulted in a very substantial increase in speed.

Although the idea of giving a very general semantics of Clear using colimits and the way that colimits in categories of complex objects are built from colimits in categories of their components are both (in some ways) extremely elegant, they contribute to a computationally discouraging result. But some possibilities for speeding up the program remain. Since it was written without regard for efficiency, there is a chance that some of the algorithms used can be substantially improved. Also, rewriting the program in POP-2 or LISP would certainly improve its performance, probably by at least one order of magnitude. There is at least one special case (i.e. a certain class of institutions) which can be treated separately and made very much more efficient; when signatures are essentially collections of sets and the morphisms within based theories are all inclusions, the necessary colimits in the category of based theories can be computed quickly using the representation and algorithms described in the set-theoretic semantics of chapter III. This class of institutions includes ordinary Clear and all other institutions which have been proposed so far (see section III.6). The necessary manipulations of theories in the special case are actually very simple as compared with those performed when a powerful general technique is applied as in the present program; the same result can be computed for example 1 in 2.3 seconds, giving a factor of more than 100 speedup.

CHAPTER SIX

PROVING THEOREMS IN CLEAR THEORIES

We have discussed in earlier chapters two versions of Clear's semantics, and we have seen how an implementation of either semantics can be useful both for checking the semantic definitions for mistakes and for checking specifications for syntactic and semantic errors. This is surely commendable in its own right, but what is to be done with the theory produced by this program as the denotation of a specification? It is nice to know that a specification contains no errors (at least at the level of theories -- whether or not it has the intended class of models is another matter) but it would be even nicer if the result of laboriously computing its denotation could be used to shed further light on the specification and its models.

There are several things which could conceivably be done with the denotation of a specification. The most obvious thing is to simply print the signature and (some representation of) the set of equations for the user to examine. The signature at least is often slightly different from that expected; it is especially easy to forget about the == operators contributed automatically by the data operation. This could also be useful in determining the effect of unusual uses of Clear's theory-building operations. Both Clear implementations print their results, although the result printed by the category-theoretic version is rather difficult to read.

A system like OBJ [Goguen and Tardo 1979] could be used to 'run' the theory in some cases. OBJ evaluates expressions by treating the equations as left-to-right rewrite rules, with special provisions for permutative equations like a+b=b+a. With this the user could check examples to see if the specified behaviour is consistent with his intentions. Such a system could not cope with all theories; loose and implicitly specified theories would both cause (probably) insurmountable problems.

The DAISTS system [Gannon, McMullin and Hamlet 1981] tests if a model (program) is consistent with an equational specification. The

idea is to run the program on a set of examples and see if the results satisfy the equations. Such a system would have a use similar to that of the OBJ-like system just mentioned; it would be more laborious to use (the user has to write a program as well as a specification) but it would be able to handle all specifications with equal ease. Of course it could also be used to test if a program satisfies its specification, provided that we are sure the specification is correct. The system checks only for consistency and not for completeness -- the program might satisfy some extra (wrong) equations as well as those in the specification -- so it will not always find the flaw in an incorrect program.

In a later chapter we shall see how the denotations of specifications would be needed in a system for stepwise refinement of specifications. The goal of such a system would be to check the validity of (and perhaps assist with) the development of a program from a specification by rewriting the specification at successively lower and lower levels. The resulting program is guaranteed to satisfy the specification, provided that the correctness of each refinement step has been verified by the system.

But in this chapter we will discuss the problem of proving theorems in the theory described by a specification. If a theorem prover of some kind were available it could be used by the Clear system itself to check that specifications are semantically well-formed; the conditions attached to the apply and derive semantic operations require that the signature morphism provided be a theory morphism, which entails checking that the equations and constraints in the source theory (translated via the signature morphism) hold in the target theory. Even better, the user could pose questions about his specification in the form of equations, which the theorem prover would try to answer. Guttag and Horning [1980] demonstrate how this can be of use in analysing specifications. Also, a program development system would need a theorem prover to check the validity of refinement steps.

Most useful would be a fully automatic theorem prover. But theorem proving technology is not yet sufficiently advanced to provide this, although some remarkably good automatic theorem

provers do exist (see for example [Boyer and Moore 1980]). Here we will discuss how a semi-automatic theorem proving system based on Edinburgh LCF [Gordon, Milner and Wadsworth 1979] was attached to the set-theoretic implementation of Clear. This system proves many theorems automatically, but in difficult cases it leaves the user to design a proof strategy from high-level primitives. He also can build his own primitives (tactics, in LCF jargon) using the inference rules provided. The structure of Clear theories seems to be very useful in directing the search for a proof in an interactive system, although so far little experimentation has been done to confirm this suspicion.

In section 5 it is shown that no complete proof system exists for Clear. Although this result has important consequences, in practice the difficulty of mechanical theorem proving is the limiting factor. Usually the theorems we wish to prove will have routine proofs; our task is to automate the easy proofs and provide the user with tools for attacking the harder ones.

1. Edinburgh LCF

Since the system we are about to discuss both is built upon and draws inspiration from Edinburgh LCF, we now briefly describe the most important features of that system.

Edinburgh LCF (usually called simply 'LCF') is a large system, and as such it is probably easiest to understand when it is decomposed into several more or less independent subsystems. First is ML, a general-purpose applicative language with polymorphic types. ML is very much like HOPE; one useful feature which is found in ML but not in HOPE is a failure generating (and failure trapping) mechanism.

Built on top of ML is the second component, PPLAMBDA -- a family of deductive calculi or theories with terms from typed lambda-calculus and (for each member of the family) a set of types, constants and axioms. There are facilities akin to enrich and combine in Clear for putting together several theories and extending the result to make a new theory. A theorem in PPLAMBDA is an ML data structure like a term or formula, but with a crucial difference: the only way to construct a theorem is by application of built-in inference rules. This ensures that any object of type thm must be true in the theory in which it was formed. Thus the type security provided by the ML type checker is used to maintain logical security.

The final component of LCF is not a program but a methodology for goal-directed proof in PPLAMBDA using ML. Given a theorem to be proved (we use the notation $a_1 \ldots a_n \stackrel{?}{\vdash} c$), we apply a tactic; that is, a proof rule in the form of a little ML program. This may fail if the goal is not of the appropriate form. If it succeeds then it delivers a list of subgoals together with a proof; this is a function built from inference rules which will produce a theorem (written $a_1 \ldots a_n \vdash c$) corresponding to the original goal if it is given a theorem corresponding to each of the subgoals. Proving a theorem is then a matter of applying one tactic after another until the empty list of goals is obtained. Tacticals like

THEN : tactic x tactic $\longrightarrow$ tactic

are provided for composing tactics into larger tactics called

strategies.

LCF is sometimes described as an interactive theorem-proving system, but as it stands it is not well-adapted to this end (although Luca Cardelli, Jacek Leszczylowski and Brian Monahan have each written a collection of 'interactive' tactics). The bookkeeping problem of remembering how to compose proof functions (obtained by the application of tactics to goals at various stages of the proof) is handled well by the tacticals but is nontrivial for humans. LCF is most useful for interactively designing and testing strategies for proof; the idea is to produce a strategy which will solve the entire problem by reducing the top-level goal to the empty goal list, rather than to attack subgoals individually by hand (although this can be useful for designing a strategy).

2. The theorem prover

The denotation of a Clear specification is a theory -- that is, a signature Σ together with a closed set of Σ-equations (and Σ-constraints). Of course, the set is often infinite, so it cannot be represented explicitly. Both Clear implementations represent a closed set of equations by an agglomerate; this is a value of the term algebra generated by the following constructors:

close : equation-set x constraint-set → agglomerate
union : agglomerate x agglomerate → agglomerate
translate : signature-morphism x agglomerate → agglomerate
inv-translate : signature-morphism x agglomerate → agglomerate
add-equality : signature-morphism x agglomerate → agglomerate

For the formal meanings of these operators, consult the next section (they have already been defined informally in section IV.1). This is a sufficient set of operators to describe the manipulations on agglomerates required by the semantics of Clear. Roughly speaking, each operator corresponds to a theory-building operation of Clear. The operator close is used for enrich, union for combine (and enrich), inv-translate for derive, and add-equality for data-enrich. Translate is needed for enrich, combine, and apply. For example, the Clear expression A + B generates the following agglomerate:

union(translate(σ_A,A-agglomerate),
translate(σ_B,B-agglomerate))

where σ_A and σ_B are the inclusions of the signatures of A and B respectively into the signature of the combined theories.

The theorem prover's job is to implement the membership operation, determining if an equation occurs in the set of equations described by an agglomerate:

is-in : equation x agglomerate → bool

Given an equation e and an agglomerate A, we try to show that e is contained in the denotation of A; if this can be established then we write $A \vdash e$. This is called a fact. Facts in our system coexist with PPLAMBDA theorems (which we write with a subscripted turnstile, $\vdash_{LCF}$ from now on) and play a parallel role. Like theorems in PPLAMBDA, facts can only be constructed by application of certain rules of inference which we will shortly discuss. The system provides a set of tactics for attacking goals (which are written

$A \vdash^{?} e$ -- LCF goals are henceforth written $a_1 \ldots a_n \vdash^{?}_{LCF} c$); these are analogous to LCF tactics and can be combined into strategies using the standard tacticals.

Thus we adopt wholesale the LCF proof methodology, and use exactly the same trick for ensuring the validity of facts as LCF uses for theorems. We use PPLAMBDA forms for representing equations and constraints, and perform all of the necessary straightforward equational deduction using the standard PPLAMBDA rules of inference. The system itself is written in ML. The only important feature of LCF we do not use is the facility for building new PPLAMBDA theories by extending old theories. The role of theories in PPLAMBDA is played by agglomerates in our system. As we shall see shortly, much of the work of the theorem prover consists of rapidly switching contexts from one agglomerate to another (usually embedded) one -- LCF does not permit switching between PPLAMBDA theories in the course of a proof (although such a facility could be added). Moreover, agglomerates may be related in ways different from the simple parent-daughter relationship between theories supported by LCF. The theorem prover operates in a PPLAMBDA theory containing all the types (sorts) and constants (operators) it will need to use because these need to be declared before appearing in a form, but no axioms are included except for those built into PPLAMBDA. The axioms of a Clear specification are contained in the agglomerate which is its denotation; these are brought into play in the course of the proof but never become part of the underlying PPLAMBDA theory itself.

We actually use an impoverished version of LCF in which many of the usual built-in types, operations and inference rules of PPLAMBDA are not available. This is necessary because of a mismatch between the models of PPLAMBDA and Clear theories. Clear deals entirely with total functions, while PPLAMBDA is designed for reasoning about recursively-defined functions which may be partial. A model of a PPLAMBDA theory is given by a family of domains, each with a distinguished minimum element and an order relation (see [Milner, Morris and Newey 1975]). A PPLAMBDA type always includes an implicit minimum element and an order relation, and inference rules

are provided for reasoning about them. This means that the PPLAMBDA rules of inference are not sound for reasoning about Clear theories; an example will be given in section 3. Soundness is restored by removing the implicit order and minimum element and all inference rules concerning them. The subset of PPLAMBDA which remains is described in appendix 3.

The goal of this system is to provide a set of tools sufficient to enable a user to conduct proofs of 'facts' in LCF. As mentioned earlier, our intent is not to give a general-purpose automatic proof system, for this would be an impossible task. To this end the system contains definitions of agglomerates and facts (with their inference rules); a set of basic tactics are supplied as well, although the user may design his own tactics from the inference rules given. A strategy which is capable of automatically proving a restricted class of facts is provided. If this strategy fails, it will at least have reduced the problem at hand to one of ordinary equational deduction using standard PPLAMBDA inference rules. At this point the user must assume control of the proof attempt, with all the usual facilities of LCF at his disposal.

3. Inference rules

Suppose we are somehow able to construct the fact $A \vdash e$ in our system. We understand this to mean that e is a member of the set described by the agglomerate A. We had better explore the semantics of agglomerates before attempting to give inference rules for reasoning about them; without a semantics, we cannot even prove the soundness of our system.

The abstract syntax of agglomerates was given at the beginning of the last section. They have a straightforward semantics, given by the semantic function $\mathbb{E}$ (recall that $\overline{E}$ refers to the model-theoretic closure of E, and $\sigma^{-1}(E) = \{e \mid \sigma(e) \in E\}$; see section III.2.3 for the meaning of the notation E^S, the augmentation of E by equations defining the 'data' equality predicate == on the sorts of S).

$\mathbb{E}$: agglomerate $\longrightarrow$ (equation and constraint) -set

$\mathbb{E}[\![\text{close}(E,C)]\!] = \overline{E \cup C}$
$\mathbb{E}[\![\text{union}(A,A')]\!] = \overline{\mathbb{E}[\![A]\!] \cup \mathbb{E}[\![A']\!]}$
$\mathbb{E}[\![\text{translate}(\sigma,A)]\!] = \overline{\sigma(\mathbb{E}[\![A]\!])}$
$\mathbb{E}[\![\text{inv-translate}(\sigma,A)]\!] = \sigma^{-1}(\mathbb{E}[\![A]\!])$
$\mathbb{E}[\![\text{add-equality}(\sigma,A)]\!] = \mathbb{E}[\![A]\!]^S$ (S=sorts($\underline{\Sigma}'-\underline{\Sigma}$), where $\sigma:\underline{\Sigma} \hookrightarrow \underline{\Sigma}'$)

where E denotes a set of equations,
C denotes a set of constraints,
and A denotes an agglomerate.

Observe that for any agglomerate A, $\mathbb{E}[\![A]\!]$ is closed (the denotation of inv-translate is always closed due to a result in section III.2.4). Also note that not all terms are semantically well-formed -- for example, if A denotes a set of $\underline{\Sigma}$-equations and constraints and $\sigma:\underline{\Sigma}' \rightarrow \underline{\Sigma}''$ is a signature morphism where $\underline{\Sigma} \neq \underline{\Sigma}'$, then translate($\sigma$,A) is meaningless. It is assumed throughout this chapter that any restrictions necessary to maintain well-formedness are tacitly stated whenever a term appears.

A number of identities follow from the semantics, including the following:

translate(σ,union(A,A')) = union(translate(σ,A),translate(σ,A'))
translate(σ,inv-translate(σ,A)) = A

But the following identity does not hold in general:

$$\text{inv-translate}(\sigma,\text{translate}(\sigma,A)) = A$$

Using this semantics, we can give a set of inference rules which allow us to reason about facts of the form $A \vdash e$, where e is an equation or a constraint. (Note that $A \vdash e$ means $e \in \mathbb{E}[\![A]\!]$.) The problem with this is that we do not have any means available for reasoning about constraints; we know what it means for an algebra to satisfy a constraint and how to translate constraints by signature morphisms, but this does not provide a rich deductive calculus similar to what we have for equations. Moreover, constraints cannot be converted into equations; the language of equations is not rich enough to capture the meaning of a constraint. But certainly we do not want to throw away the information encapsulated in constraints if at all possible, since this would dramatically restrict the class of facts we would be able to prove.

We need a notion of fact in which something more than an equation is allowed on the right of the turnstile. This 'something' should be powerful enough to express constraints, and should have a readily-available proof theory. A very convenient choice is PPLAMBDA forms (formulae); these include equations, and also allow higher-order quantification and combination of forms with the conjunction and implication connectives. We will see shortly that an induction rule for a sort s (derived from a data constraint) can be expressed as a second-order form $\forall P.\forall Q. \ldots$ where P and Q have the polymorphic type $s \longrightarrow *$. Moreover, we know how to reason about forms; that is precisely what LCF was built to do.

It is easy to extend facts to be of the form $A \vdash f$, where f is any PPLAMBDA form. We can define $A \vdash f$ to mean $f \in \mathbb{E}[\![A]\!]^{*+}$, where

$*$: (Σ-equation and Σ-constraint) -set $\longrightarrow$ Σ-algebra set

is the function defined in section II.4 (recall that $\overline{E} = E^{**}$) and

$+$: Σ-algebra set $\longrightarrow$ Σ-form set

is defined by:

$M^{+} = \{f \mid m \text{ satisfies } f \text{ for each } m \in M\}$

It turns out that even though the language of forms is

sufficiently powerful to express the information contained in a constraint, it is impossible to extract all of it because of incompleteness. But that portion of the information which is most necessary for our purposes may be translated into a form.

To see how this arises we must examine the definition of constraint satisfaction given in section II.5. If we consider for the moment only cases where the second part of the constraint (the signature morphism) is the identity, then this amounts less formally to the following definition:

Def (Constraint satisfaction, informally): An algebra A satisfies a constraint $\langle i:\underline{T}\hookrightarrow\underline{T}', 1_{sig(\underline{T}')}\rangle$ if the following conditions hold:

1. A is a model of T'.
2. No terms are identified in A unless the equations of T' force them to be.
3. Every A element is the value of a term having variables only in sorts of T for some assignment of values to variables.

These three conditions are statements which will be true of any algebra in $\mathbb{E}[\![A]\!]^*$, for any constraint in $\mathbb{E}[\![A]\!]$. This means that any statement which follows from them which can be encoded as a form will be in $\mathbb{E}[\![A]\!]^{*+}$, and hence a fact in A.

Condition 1 is redundant. The equations of T' will appear elsewhere in the agglomerate which contains the constraint, so we can safely ignore them now. Condition 2 entails only inequations -- these can be given as PPLAMBDA forms (a≠b is written as "a=b IMP TT=FF"), but it is impossible in general to determine which inequations will hold because of the incompleteness result mentioned earlier. This is not a problem if T' is anarchic. But because we are mainly interested in proving equations, and because PPLAMBDA does not include facilities for reasoning about inequations, we choose to ignore this special case.

Condition 3 gives rise to an induction rule for each sort in sorts(T')-sorts(T), since all values of these sorts are generated by the 'constructors' in T'. This rule can be expressed as a polymorphic second-order form -- in the case of natural numbers with

operations 0 and successor the rule becomes:

```
!P:nat->*. !Q:nat->*.
   [  P(0)=Q(0) & !x.[P(x)=Q(x) IMP P(succ x)=Q(succ x)]
      IMP
      !x.[P(x)=Q(x)]  ]
```

In LCF the universal quantifier becomes '!', type variables are written '*' (or **, ***, etc.), and IMP means logically implies. We use the operator '=' instead of the LCF '==' to write PPLAMBDA equations in this chapter; the '==' operator is reserved for Clear's 'data' equality predicate.

This rule could be instantiated to prove the equation $n+m\geq n = true$ (that is, to prove the fact $A\vdash$"!n.[$n+m\geq n$=true]" for the agglomerate A which arises from enriching the natural numbers with an order relation). The type variable * is instantiated to nat->bool, P becomes $\lambda n.\lambda m.n+m\geq n$ and Q becomes $\lambda n.\lambda m.true$ to give:

```
λm.0+m≥0=λm.true
& !x.[λm.x+m≥x=λm.true IMP λm.succ(x)+m≥succ(x)=λm.true]
IMP
!x.[λm.x+m≥x=λm.true]
```

In general, given a constraint $\langle i:\underline{T}\hookrightarrow\underline{T}', 1_{sig(\underline{T}')}\rangle$ and a sort $s\in sorts(\underline{T}')-sorts(\underline{T})$ with

$$constructors(s) \equiv \{o\in opns(\underline{T}') \mid arity(o) \text{ is } v\text{->}s, \text{ for some } v\}$$
$$= \{\ldots, \omega:u\text{->}s, \ldots\},$$

we can extract the following induction rule:

```
!P:s->*. !Q:s->*.
   [  ...
      & !x1:u1. ... !xn:un.[ ... & P(xj)=Q(xj) & ...
                             IMP
                             P(ω(x1,...,xn))=Q(ω(x1,...,xn)) ]
      & ...
      IMP
      !x:s.[P(x)=Q(x)]  ]
```

where $u=u_1\ldots u_n$ and $u_j=s$.

Recall that the preceeding discussion related only to constraints with the identity morphism (on the signature of $\underline{T}'$) as a second part. Given a constraint $\langle i:\underline{T}\hookrightarrow\underline{T}', \sigma:signature(\underline{T}')\rightarrow\underline{\Sigma}\rangle$ (where σ

need not be the identity), we can produce an induction rule by first generating a rule for the constraint $\langle i, 1_{sig(\underline{T}')} \rangle$ using the method just described, and then applying the signature morphism σ to translate the rule to the signature $\sum$.

We have just described a way of extracting a set of induction rules from a constraint; this gives a function

induction-rules : constraint $\longrightarrow$ form-set

It is easy to define another function

eqn-to-form : equation $\longrightarrow$ form

for converting equations to forms. Now we can have a try at an inference rule:

$f \in (\text{eqn-to-form}^* E \cup \text{induction-rules}^* C) \Rightarrow \text{close}(E,C) \vdash f$

This is a satisfactory rule, but since the original equations and constraints are no longer of any use (but only the forms derived from them) we could just as well forget them and deal only with forms. Accordingly we modify the abstract syntax of agglomerates so that close accepts a set of forms:

close : form-set $\longrightarrow$ agglomerate

The rest of the abstract syntax remains the same. The agglomerates used by the Clear implementation (call them E-agglomerates) are translated into agglomerates with the new close (F-agglomerates), with the only nontrivial part of the translation being the conversion of the constraints to forms. This translation occurs at the interface between Clear and LCF, as described in a later section.

An incidental benefit of the switch from equations to forms is that the theorem prover is now equally capable of handling specifications using conditional equations, predicate calculus formulae, or any other kind of axioms which can be translated into PPLAMBDA forms. The only difference is at the interface between the specification language and the theorem prover, where the axioms must be translated into forms.

A semantics for F-agglomerates is given by the semantic function

$\mathbb{F}$, defined as follows:

$\mathbb{F}$: agglomerate $\rightarrow$ form-set

$\mathbb{F}[\![\text{close}(F)]\!] = \overline{F}$
$\mathbb{F}[\![\text{union}(A,A')]\!] = \overline{\mathbb{F}[\![A]\!] \cup \mathbb{F}[\![A']\!]}$
. . .

All the semantic equations except for the close operation are identical to those at the beginning of the section. Note that $\overline{F} = F^{++}$, where the first + is the operation

$+ : \underline{\Sigma}$-form set $\rightarrow$ $\underline{\Sigma}$-algebra set

defined by

$F^{+} = \{m \mid m \text{ satisfies } F\}$

and the second + is

$+ : \underline{\Sigma}$-algebra set $\rightarrow$ $\underline{\Sigma}$-form set

as described earlier.

Theorem: For any E-agglomerate A, $\mathbb{F}[\![\tau(A)]\!] \subseteq \mathbb{E}[\![A]\!]^{*+}$, where τ:E-agglomerate$\rightarrow$F-agglomerate is the translation mentioned above.

Proof: See Appendix 4; the proof relies on a proof of the Satisfaction Lemma (section II.3) for PPLAMBDA forms, also given.

This theorem tells us that the new semantics for agglomerates is consistent with the old semantics -- so any fact we can prove using inference rules which are sound with respect to the new semantics will hold in the corresponding theory (but not vice versa).

The inference rules can now be stated. It is easy to prove from the semantics that each of the rules is sound (note that $A \vdash f$ now means $f \in \mathbb{F}[\![A]\!]$). Each rule is given an upper-case name, following LCF convention.

CLOSE: $f \in F \Rightarrow \text{close}(F) \vdash f$
UNIONLEFT: $A \vdash f \Rightarrow \text{union}(A,A') \vdash f$
UNIONRIGHT: $A' \vdash f \Rightarrow \text{union}(A,A') \vdash f$
TRANSLATE: $A \vdash f \Rightarrow \text{translate}(\sigma,A) \vdash \sigma(f)$
INVTRANSLATE: $A \vdash \sigma(f) \Rightarrow \text{inv-translate}(\sigma,A) \vdash f$
ADDEQUALITY: $A \vdash f \Rightarrow \text{add-equality}(\sigma,A) \vdash \sigma(f)$
EQUALITYOPN: ω:s,s->bool$_{\text{Bool}} \in \text{opns}(\underline{\Sigma}'-\underline{\Sigma})$ and $\sigma:\underline{\Sigma} \hookrightarrow \underline{\Sigma}'$
$\Rightarrow$ add-equality$(\sigma,A) \vdash$!x:s.!y:s.[x=y IMP $\omega(x,y)$=true$_{\text{Bool}}$]
LCFINFER: $A \vdash f_1$ & ... & $A \vdash f_n$ & $f_1,\ldots,f_n \vdash_{LCF} f \Rightarrow A \vdash f$

EQUALITYOPN provides us with a way of proving equality (the operator ω will always be the == data equality) but no way of proving inequality. Proving inequality is impossible in general because of incompleteness, but in the special case of an anarchic theory it is trivial. Burstall [1980a] has devised a way of proving inequality in a nonanarchic theory, but the method requires help from the user, analogous to but different from supplying induction hypotheses to a theorem prover. We do not attempt to deal with this problem; no inference rules are provided for reasoning about inequality. Note that inequalities are subtly different from inequations, discussed earlier.

LCFINFER provides a 'gateway' between standard PPLAMBDA and the superstructure of inference rules about facts which is needed to adapt LCF to reason within Clear theories. Viewing the theorem prover as a goal-manipulation system, the previous seven rules provide a means for reducing a goal (prove a fact A⊢f) to a problem in ordinary equational logic. LCFINFER permits this to be translated into an LCF goal, whereupon the proof can proceed using the facilities of standard LCF.

We must be careful in our use of LCF for two reasons. The first problem stems from the mismatch between the models of PPLAMBDA and Clear theories mentioned in section 2. Recall that in standard PPLAMBDA a type always includes an implicit minimum element (written "UU"). If full PPLAMBDA is used then LCFINFER is not sound. Consider the theory Bool; it contains a data constraint which gives rise to the following induction rule:

```
!P:bool->*. !Q:bool->*.
   [  P(true)=Q(true) & P(false)=Q(false)
      IMP
      !x.[P(x)=Q(x)]  ]
```

Taking the example

```
        P(UU) = UU,  P(true) = P(false) = true
   and  Q(UU) = Q(true) = Q(false) = true
```

this rule leads to the conclusion UU=true. A similar example can be used to prove that UU=false, and by symmetry and transitivity this means that true=false.

In order to retain soundness, we restrict PPLAMBDA so that examples like the one above do not occur by excluding UU and all inference rules which refer to UU or the order relation. In fact, we really want to replace the turnstile $\vdash_{LCF}$ in LCFINFER by $\vdash_{EQ}$, where EQ is a system for purely equational deduction with the ability to apply the induction rules described earlier. We use LCF only for convenience and because it contains a powerful simplifier which is capable of assuming much of the work of equational deduction.

A second problem with the PPLAMBDA inference rules is demonstrated by the following example from Goguen and Meseguer [1981]:

```
const T = theory
            sorts a, bool
            opns true, false : bool
                 not : bool -> bool
                 and, or : bool,bool -> bool
                 f : a -> bool
            eqns not(true) = false      not(false) = true
                 p or not(p) = true     p and not(p) = false
                 p or p = p             p and p = p
                 f(a) = not(f(a))       endth
```

We can now make the following deduction using the inference rules of PPLAMBDA (symmetry, transitivity, substitutivity and specialisation of quantifiers are sufficient):

```
true  = f(a) or not(f(a))
      = f(a) or f(a)
      = f(a)
      = f(a) and f(a)
      = f(a) and not(f(a))
      = false
```

But true=false is not satisfied by the model of T with bool={true,false} and a=∅ (with the usual interpretation of the boolean operators).

This is again due to a mismatch between the models of PPLAMBDA and Clear theories. The inference rule for specialising quantified variables is not sound for many-sorted theories (e.g. Clear theories) unless the variable is of a non-void sort:

Def: A sort s is void in a signature Σ if $s \in sorts(\Sigma)$ and:

- There are no constants of sort s in Σ, and
- There is no operator $\omega:s_1,\ldots,s_n\text{->}s$ in Σ with all of $s_1,\ldots,s_n$ non-void.

It is difficult to change the inference rule because it is built into the LCF simplifier, which plays a vital role in our equational deduction tactic (this tactic is described at the end of the next section). But void sorts are very unusual in practice. The quantifier specialisation inference rule remains valid as long as all sorts are nonvoid, so for reasonable examples there will be no problem. It is best to eliminate unsound inference rules, so a future reimplementation should incorporate a version of quantifier specialisation modified to fail for variables of void sorts.

4. Tactics and strategies

The inference rules given in section 3 could be used to prove theorems in a 'forward' direction, but the LCF style is to instead proceed backwards in a goal-directed fashion. A step consists of transforming the goal into a list of goals which, if they can be achieved (converted to theorems), entail the desired theorem. The transformation steps are carried out by backwards inference rules called tactics, which can be composed using tacticals to give strategies, as discussed in section 1.

The theorem prover provides tactics corresponding to each of the inference rules given in section 3. These are all simple ML programs, operating on goals of the form $A \stackrel{?}{\vdash} f$ and returning a list of goals (together with a proof, not shown).

CLOSETAC: $close(F) \stackrel{?}{\vdash} f \longmapsto [\,]$ if $f \in F$, else failure

UNIONLEFTTAC: $union(A,A') \stackrel{?}{\vdash} f \longmapsto [\ A \stackrel{?}{\vdash} f\ ]$

UNIONRIGHTTAC: $union(A,A') \stackrel{?}{\vdash} f \longmapsto [\ A' \stackrel{?}{\vdash} f\ ]$

TRANSLATEWITHTAC: $f \longmapsto translate(\sigma,A) \stackrel{?}{\vdash} f' \longmapsto [\ A \stackrel{?}{\vdash} f\ ]$
if $\sigma(f)=f'$, else failure

INVTRANSLATETAC: $inv\text{-}translate(\sigma,A) \stackrel{?}{\vdash} f \longmapsto [\ A \stackrel{?}{\vdash} \sigma(f)\ ]$

ADDEQUALITYTAC: $add\text{-}equality(\sigma,A) \stackrel{?}{\vdash} f \longmapsto [\ A \stackrel{?}{\vdash} f\ ]$
if $\sigma^{-1}(f) \neq \emptyset$, else failure

EQUALITYOPNTAC: $add\text{-}equality(\sigma,A) \stackrel{?}{\vdash} \omega(x,y)=true_{Bool} \longmapsto [\ add\text{-}equality(\sigma,A) \stackrel{?}{\vdash} x=y\ ]$
if $\omega:s,s\text{->}bool_{Bool} \in opns(\underline{\Sigma}'-\underline{\Sigma})$ where $\sigma:\underline{\Sigma} \hookrightarrow \underline{\Sigma}'$, else failure

LCFINFERTAC: $[\ A \vdash f_1, \ldots, A \vdash f_n\ ] \longmapsto A \stackrel{?}{\vdash} f \longmapsto [\ f_1,\ldots,f_n \stackrel{?}{\vdash}_{LCF} f\ ]$

Each of these tactics gives a way of diving into an agglomerate with a form, yielding a goal concerning a subagglomerate and the (possibly transformed) form. UNIONRIGHTTAC and UNIONLEFTTAC take different choices when given a union; similarly, TRANSLATEWITHTAC yields a different result for the goal $translate(\sigma,A) \stackrel{?}{\vdash} f$ depending on which element of the set $\sigma^{-1}(f)$ it is given. The system provides

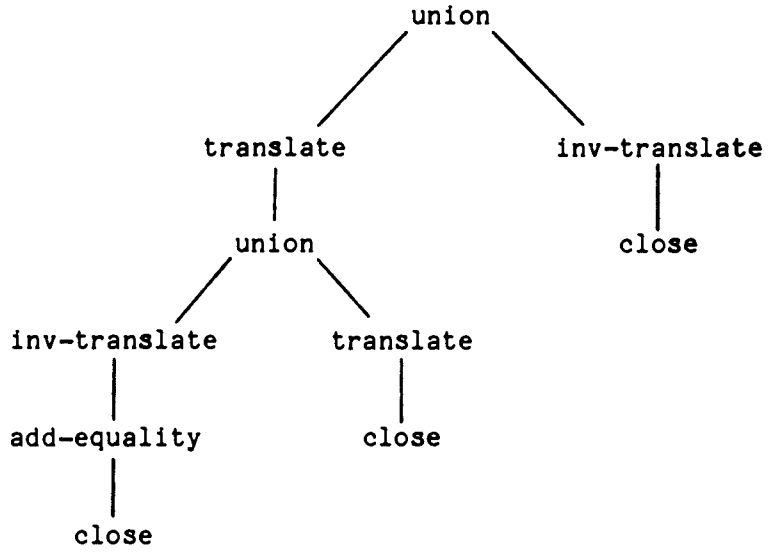
tacticals which automate these choices:

UNIONTACTHEN: tac $\longmapsto$
(UNIONLEFTTAC THEN tac) ORELSE (UNIONRIGHTTAC THEN tac)

TRANSLATETACTHEN: tac $\longmapsto$ translate$(\sigma,A)\stackrel{?}{\vdash}f \longmapsto$
((TRANSLATEWITHTAC f_1 THEN tac)
ORELSE ... ORELSE
(TRANSLATEWITHTAC f_n THEN tac)) translate$(\sigma,A)\stackrel{?}{\vdash}f$
where $\{f_1, \ldots, f_n\} = \sigma^{-1}(f)$

The standard LCF tactical ORELSE, given the two tactics tac_1 and tac_2, applies tac_1 to the goal unless it fails, in which case tac_2 is applied. The action of UNIONTACTHEN tac is therefore to first try choosing the left-hand branch of the union; if this causes tac to fail, then it tries the right-hand branch. TRANSLATETACTHEN tac tries each possible choice of argument for TRANSLATEWITHTAC, rejecting those which cause tac to fail.

It is helpful to think of an agglomerate as a tree. For example:



Each of the tactics given so far dive from an agglomerate to the subagglomerate(s) immediately underneath (with the exception of EQUALITYOPNTAC, which remains at the same node). A composite tactical called DIVETAC is provided which, given an LCF tactic (i.e., a tactic for attacking LCF goals), explores the entire

agglomerate by diving repeatedly until it reaches a tip (a close agglomerate). At this point LCFINFERTAC is applied, followed by the tactic provided as argument. If this results in the empty goal list, then the goal is achieved; otherwise a failure is generated which is trapped at the most recent choice point (an application of UNIONTACTHEN or TRANSLATETACTHEN). The same process is then used to explore another branch of the tree (or the same branch, with a different form), until the entire tree has been traversed.

DIVETAC: tac $\longmapsto$ g $\longmapsto$
  if g = close(F) $\vdash^{?}$ f:
    (TRY (LCFINFERTAC [close(F) $\vdash$ f_1
                        . . .
                        close(F) $\vdash$ f_n] THEN tac)) g
    where F = $\{f_1, \ldots, f_n\}$
  if g = union(A,A') $\vdash^{?}$ f: (UNIONTACTHEN DIVETAC tac) g
  if g = translate(σ,A) $\vdash^{?}$ f: (TRANSLATETACTHEN DIVETAC tac) g
  if g = inv-translate(σ,A) $\vdash^{?}$ f: (INVTRANSLATETAC THEN DIVETAC tac) g
  if g = add-equality(σ,A) $\vdash^{?}$ f:
          ((DO EQUALITYOPNTAC) THEN ADDEQUALITYTAC
                               THEN DIVETAC tac) g

This uses two auxiliary tacticals. The first is called TRY; it fails unless the tactic supplied is able to achieve the goal.

TRY: tac $\longmapsto$ g $\longmapsto$ if tac g = [] then [], else failure

The second is called DO; it applies the given tactic, returning the original goal if the result is failure.

DO: tac $\longmapsto$ g $\longmapsto$ if tac g = failure then [g], else tac g

DIVETAC EQTAC (where EQTAC is an LCF tactic for performing equational deduction; one such is described at the end of this section) can automatically provide proofs for a wide range of facts, provided that EQTAC performs adequately. It dives down to the tip which contains the information needed to prove the fact at hand (of course, finding the proper tip may involve a backtracking search), and uses EQTAC to do the 'dirty work' of the proof.

This is quite a good way to go about proving facts concerning

large agglomerates. For example, if the goal is $A \stackrel{?}{\vdash} p+q=q+p$ where A is obtained from the specification of a compiler, then almost all of the information buried in A is completely irrelevant and should be ignored lest the proof get bogged down by silly proof attempts. DIVETAC will fail quickly when attempting to follow most silly paths (going on to find the correct path) because of a mismatch between the form at hand and the signature of the irrelevant subagglomerate. For instance, the Clear expression Nat + Useless gives rise to the agglomerate

$$\begin{aligned}&\text{union}(\text{translate}(\sigma_{Nat}:\Sigma_{Nat} \hookrightarrow \Sigma_{Nat+Useless}, A_{Nat}),\\ &\quad \text{translate}(\sigma_{Useless}:\Sigma_{Useless} \hookrightarrow \Sigma_{Nat+Useless}, A_{Useless}))\end{aligned}$$

An attempt to prove that p+q=q+p in the combined theory using DIVETAC will ignore the subagglomerate $A_{Useless}$ because TRANSLATETACTHEN anytac will fail immediately when applied to the goal

$$\text{translate}(\sigma_{Useless}, A_{Useless}) \stackrel{?}{\vdash} p+q=q+p$$

for $\sigma^{-1}_{Useless}(p+q=q+p)$ is empty. That is, provided that $\Sigma_{Useless}$ does not include the + operator.

Unfortunately, a large class of facts remains which cannot be proved using DIVETAC. These are the cases in which there is not enough information in any <u>single</u> tip to prove the fact. For example, proving that the equation

length(append(l,k)) = length(l) + length(k)

holds in the theory of lists and natural numbers requires the use of equations and induction rules from both subtheories. DIVETAC will fail for this reason.

The theorem prover provides a tactic for handling this eventuality. Instead of diving into an agglomerate with a form, we want to '<u>dredge up</u>' facts from the depths of the agglomerate, forming the union of all the information available in the tips. Then LCFINFER and EQTAC can be used to prove the form.

This is more difficult than it sounds. Consider the following contrived but illustrative specification:

```
ABCD = theory sorts abcd
           opns a,b,c,d : abcd  endth

ACD = derive sorts acd
          opns a,c,d : acd
      from enrich ABCD by
          eqns a = b
               b = c  enden
      by acd is abcd  endde
```

This gives rise to the agglomerate
inv-translate(σ,close(a=b,b=c))

where σ: acd $\longmapsto$ abcd
a $\longmapsto$ a
c $\longmapsto$ c
d $\longmapsto$ d

The equation a=c holds in ACD. How are we to discover this? It is easy to prove the fact

inv-translate(σ,close(a=b,b=c))$\vdash$a=c

using DIVETAC, but extracting all of the facts which are true in a situation like this (without knowing beforehand which facts are needed) is difficult. It is impossible in general because of the existence of theories which have finite presentations when derive is allowed, but only infinite presentations otherwise (see [Thatcher, Wagner and Wright 1978]).

DREDGETAC therefore does not try to dredge up all of the information available, but only that which is conveniently accessible. The following auxiliary function produces the set of conveniently accessible forms from an agglomerate:

dredge: close(F) $\longmapsto$ F
union(A,A') $\longmapsto$ dredge(A)$\cup$dredge(A')
translate(σ,A) $\longmapsto$ σ(dredge(A))
inv-translate(σ,A) $\longmapsto$ σ^{-1}(dredge(A))
add-equality(σ,A) $\longmapsto$ σ(dredge(A))$\cup$
$\{!x{:}s.!y{:}s.[x{=}y \text{ IMP } \omega(x,y){=}true_{Bool}] \mid \omega{:}s,s\text{->}bool_{Bool} \in opns(\underline{\Sigma}'-\underline{\Sigma}) \text{ where } \sigma{:}\underline{\Sigma}\hookrightarrow\underline{\Sigma}'\}$

Note the similarity between the function dredge and the semantic function $\mathbb{F}$ defined in section 3. The only difference is that dredge (being only a program running on a finite computer) must abstain from use of the closure ('bar') and the add-equality-axioms operations.

It is easy to prove the following derived inference rule, using the fact that $F \subseteq \overline{F}$ and $F \subseteq F' \Rightarrow \overline{F} \subseteq \overline{F'}$:

DREDGE: $f \in dredge(A) \Rightarrow A \vdash f$

DREDGETAC uses dredge to extract forms form the agglomerate at hand. Then LCFINFERTAC is applied to give an LCF goal, which has as assumptions the set of facts thus accumulated.

DREDGETAC: $A \vdash^{?} f \longmapsto$ LCFINFERTAC [$A \vdash f_1$, ..., $A \vdash f_n$] $a \vdash^{?} f$
where $\{f_1, \ldots, f_n\} = dredge(A)$

We have seen that DIVETAC is capable of proving a certain class of facts, yet DREDGETAC seems to be needed to collect the information necessary for the proofs of other facts. DREDGETAC alone (followed by EQTAC) is not capable of proving many of the facts which are handled with ease by DIVETAC. Some combination of diving and dredging seems to be necessary in a general strategy for proof in Clear.

Our strategy rests on the observation (mentioned above) that often the agglomerate at hand contains a great deal of information which is utterly irrelevant to the proof of the desired fact. This seems to be a pitfall to which most theorem-proving systems are susceptible; it is easy to get irretrievably bogged down in exploring the large number of blind alleys made available by a wealth of information (see the introduction of [Boyer and Moore 1979], for example). It is therefore important to restrict the available information as much as possible before attempting the proof using standard techniques.

But how is the theorem prover to automatically determine exactly which subset of the available information is necessary for the proof of a fact? In the case of a conventional theorem prover, where the axioms, previously proved theorems, etc. are stored in a list, the only approach seems to be some kind of heuristic filter which passes only 'relevant' facts. The construction of such a filter is a formidable task, for it is not always immediately obvious what is relevant.

This problem is not so perplexing when we are given the information in a highly structured form, such as an agglomerate. As we observed above, it is easy when diving to exclude certain irrelevant subagglomerates entirely because a 'translate' node acts as a barrier to inappropriate goals. Moreover, the agglomerate will reflect the structure of the human-constructed specification from which it arises, and so it is likely that all of the information necessary to prove the fact will be located in a relatively small subagglomerate. DREDGETAC applied to this subagglomerate will normally collect all of the information neccessary to prove the fact, without much that is irrelevant.

The strategy we use is based on DIVETAC and DREDGETAC, as expected. Recalling the explanation of DIVETAC, the approach now is to visit each node in the agglomerate in precisely the same order as in DIVETAC, performing the same action at the tips. But after trying both paths of a 'union' node and failing, DREDGETAC is used to attempt the proof in the combined theory. This means that dredging takes place on a subagglomerate only after all other methods have failed.

This strategy is implemented by the tactical SUPERTAC (again, this takes as parameter an LCF tactic for doing equational deduction).

```
SUPERTAC:  tac ⟼ g ⟼
   if g = close(F) ⊢? f:
          (TRY (LCFINFERTAC [ close(F) ⊢ f1
                                  . . .
                              close(F) ⊢ fn ] THEN tac)) g
          where F = {f1, ..., fn}
   if g = union(A,A') ⊢? f:  ((UNIONTACTHEN SUPERTAC tac) ORELSE
                              (TRY (DREDGETAC THEN tac))) g
   if g = translate(σ,A) ⊢? f:  (TRANSLATETACTHEN SUPERTAC tac) g
   if g = inv-translate(σ,A) ⊢? f:
                              (INVTRANSLATETAC THEN SUPERTAC tac) g
   if g = add-equality(σ,A) ⊢? f:
                  ((DO EQUALITYOPNTAC) THEN ADDEQUALITYTAC
                                       THEN SUPERTAC tac) g
```

Note that DREDGETAC could be applied at nodes other than union, but

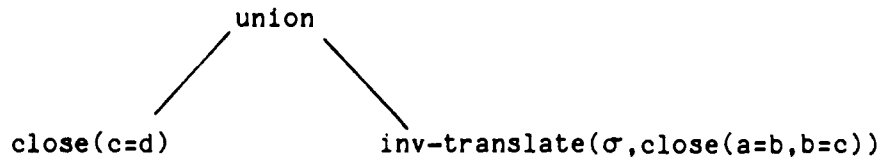
any fact which can be proved using DREDGETAC THEN tac on a non-union node can also be proved using the appropriate diving tactic followed by SUPERTAC tac, so this would be a waste of effort.

It is interesting to observe that the structure of the specification from which the agglomerate is taken is an important factor in the performance of SUPERTAC. It is certainly possible to write a specification which defeats the heuristics upon which SUPERTAC is based. But this specification would probably have a rather strange structure. The locality of reference which SUPERTAC exploits seems to be one criterion for a well-structured specification.

There remains an important class of facts which cannot be automatically proved using SUPERTAC. Recall the theory ACD given as an example earlier in this section; this was used to demonstrate the difficulty of dredging from an inv-translate. But in some cases dredging is necessary; for example, consider the theory

Tricky = enrich ACD by
eqns c = d enden

This gives rise to the following agglomerate:



where σ is as before. Now suppose we want to prove the fact $A_{Tricky} \vdash a=d$. This requires a dredge, since the necessary information is spread over both branches of the union. But the important equation a=c cannot be dredged from the inv-translate, so SUPERTAC will fail.

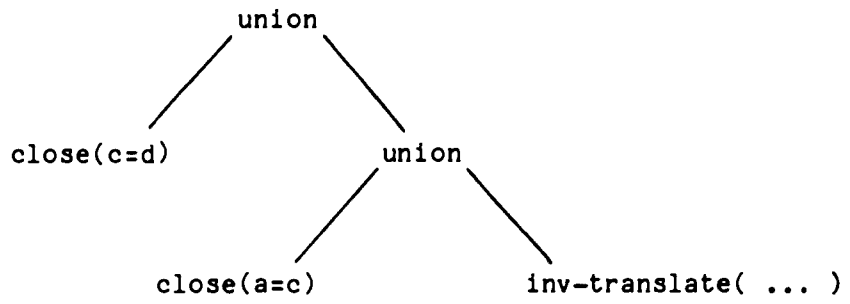
The lemma a=c is a necessary step in the proof. This can easily be proved by diving down the right-hand branch of the union. It is then easy to prove a=d using the equation c=d.

But where does the idea for this lemma come from? We avoid the difficult problem of automatic lemma generation by requiring the

user to supply such lemmas. A new construct has been added to Clear for this purpose to enable the user to propose lemmas along with the specification; we feel that this is preferable to requiring the lemmas to be inserted at theorem-proving time. To propose the lemma a=c in the above specification, the user would write:

```
ACD' = enrich ACD by
         thms a = c  enden

Easy = enrich ACD' by
         eqns c = d  enden
```

A 'thm' is treated exactly as an equation, except that it must be provable from the existing equations and constraints or else an error occurs. The theory Easy gives rise to the agglomerate



and now the equation a=d may be proved easily using DREDGETAC. We got this idea from the Z specification language [Abrial, Schuman and Meyer 1979] which also permits theorems to be included in specifications. This is a useful facility, apart from its use in assisting the theorem prover. The user can insert theorems which he believes to be correct as a check on the correctness of his specification, or he can use a theorem to prominently display an important consequence of the axioms.

It should be noted that Nelson and Oppen [1979] have described an elegant method for combining decision procedures for several independent theories into a decision procedure for the combined theory; this can be seen as an alternative to our DREDGETAC. Unfortunately, their method does not work when the theories share operators, so in general it cannot be applied to the combination of Clear theories. But in the special case where the theories do not

share operators (and perhaps also for cases with certain restricted kinds of sharing) their algorithm could be applied in place of DREDGETAC.

The theorem prover of the ι system [Nakajima, Honda and Nakahara 1980] also exploits the structure of specifications to facilitate proofs. It uses theory-focusing techniques [Honda and Nakajima 1979] which are related to the strategy embodied in SUPERTAC.

Equational deduction

The strategies given above assume the existence of an LCF tactic for performing equational deduction. We give here a brief description of the one provided by the system; this is able to prove a reasonable number of examples completely automatically, but it is far from the best possible. Several equational theorem provers (see [Musser 1980], [Goguen 1980] and [Huet and Hullot 1980]) have recently been built using the Knuth-Bendix [1970] completion algorithm; this method seems to give far better results than the naive approach used here.

EQTAC is built from five component tactics, to be described below. It tries each tactic in turn, repeating the sequence until a tactic fails or the goal is achieved.

```
EQTAC = REPEAT (SIMPTAC THEN INDTAC THEN CONJTAC
                        THEN EXTTAC THEN IMPLTAC)
```

Actually, INDTAC (induction tactic) is the only one of these which can fail, so EQTAC fails only if all possible induction variables have disappeared.

SIMPTAC is the standard LCF simplification tactic. It uses the basic simplification rules provided by LCF (beta-conversion, etc.) together with all of the assumptions of the theorem EQTAC is trying to prove (contributed by LCFINFERTAC) with the exception of induction rules. If a permutative rule such as p+q = q+p is included in a specification, then SIMPTAC will loop.

INDTAC does induction on the leftmost outermost universally quantified variable in the goal for which an induction rule is

available. An example of its result when applied to the goal $\ldots \stackrel{?}{\vdash}_{LCF} n+m \geq n = true$ was given in section 3, except that the result shown there has already been simplified using beta-conversion.

CONJTAC converts a goal of the form

$$a_1,\ldots,a_n \stackrel{?}{\vdash}_{LCF} f_1 \;\&\; \ldots \;\&\; f_m$$

to a list of goals

$$a_1,\ldots,a_n \stackrel{?}{\vdash}_{LCF} f_1 \quad \ldots \quad a_1,\ldots,a_n \stackrel{?}{\vdash}_{LCF} f_m$$

This splits the goal generated by INDTAC into cases which can be treated separately.

EXTTAC converts any occurrence of $\lambda x.t1 = \lambda x.t2$ in a goal to $!x.(t1=t2)$. Equations like these are generated by INDTAC when it is applied to an equation containing universally quantified variables other than the induction variable.

IMPLTAC converts a goal of the form

$$a_1,\ldots,a_n \stackrel{?}{\vdash}_{LCF} !x\ldots[f_1 \text{ IMP } f_2]$$

to the goal

$$a_1,\ldots,a_n,f_1 \stackrel{?}{\vdash}_{LCF} f_2$$

adding f_1 to the set of simplification rules. This assumes the inductive assumptions generated by INDTAC. The next time around the EQTAC loop, SIMPTAC will (we hope) simplify most of the goals to tautologies and a further induction will be attempted on the remaining variables.

EQTAC is able to prove routine theorems involving multiple inductions without difficulty. Typical examples are the transitivity of $\leq$ and the associativity of addition and append. Commutativity of addition is much more difficult because induction causes rules like x=x+0 and x+y=y+x to be entered as assumptions for use by SIMPTAC, causing it to loop. More care with the use of such permutative equations as simplification rules is needed to avoid this behaviour.

An example of a theorem which EQTAC cannot prove is reverse(reverse(l))=l. The proof of this theorem requires the application of a few clever heuristics rather than brute force. Induction on l followed by simplification reduces the problem to one

of proving

reverse(append(reverse(l),cons(a,nil))) = cons(a,l)

with reverse(reverse(l))=l as the inductive assumption. At this point EQTAC fails. Boyer and Moore's [1979] theorem prover continues the proof by applying the inductive assumption in reverse to the right-hand side of the goal (they call this cross fertilisation) and then replacing reverse(l) by the new variable z everywhere (generalisation). This gives the goal

reverse(append(z,cons(a,nil))) = cons(a,reverse(z)) (*)

and induction on z completes the proof. Alternatively, the user could supply a lemma such as (*) above; the theorem prover is able to prove this lemma and then use it to complete the proof of the theorem.

5. Incompleteness

Formally, we shall define a proof system as any relation between theories and sentences such that the set of sentences provable in a theory is recursively enumerable. In practice a proof system is a set of inference rules together with a notion of proof leading to such a relation. The recursive enumerability requirement captures the idea that a proof system is an effective procedure for generating the theorems of a theory.

Def: A proof system is a relation $\vdash \subseteq \text{Theories} \times \text{Sentences}$ such that if a theory T is effectively given (e.g. T is a theory with a finite presentation) then the set of provable sentences $\{s \mid T \vdash s\}$ is recursively enumerable.

Def: A proof system $\vdash$ is called complete for a theory T if any sentence s of signature(T) which is satisfied in every model of T (i.e. $T \models s$) is provable from T using $\vdash$ (i.e. $T \vdash s$).

It is well-known that equational logic (i.e. reflexivity, symmetry, transitivity and substitutivity) is complete for one-sorted equational theories (this is due to Birkhoff [1935]). Goguen and Meseguer [1981] show that this result extends to the many-sorted case only if equational logic is modified slightly by the introduction of explicit quantifiers and rules to add and delete them. For initial models of equational theories, this modified logic is complete with respect to ground equations but Nourani [1981] shows that no proof system is sound and complete with respect to non-ground equations (he actually shows that equational logic with induction is not complete, but his proof generalises easily). But the modified equational logic is not complete for Clear theories (i.e. theories with equations and data constraints) with respect to ground equations, even when induction is permitted. This fact is demonstrated by the following simple example:

```
const T = enrich Nat by
            opns f : nat -> nat
            eqns f(n) = 2*f(n+1)    enden
```

where Nat is the usual theory of the natural numbers with addition and multiplication. For all models A of T we have $A \models f(0)=0$

(remember that f must be total and the sort nat does not include an 'infinite' element). But this equation is not provable by equational logic with induction; this may be shown by induction and case analysis on the terms which may be derived from f(0).

It is easy to prove the equation f(0)=0 in T if proof by contradiction is allowed. But for some theories there is no proof system which is strong enough to prove even all true ground equations:

Theorem: There exists no proof system for Clear which is sound and complete with respect to ground equations.

Proof [MacQueen and Sannella 1982]: Proposition 4 of [Bergstra, Broy, Tucker and Wirsing 1981] states that for any total recursive function $f:\mathbb{N}x\mathbb{N}\longrightarrow\mathbb{N}$ there is a finite Clear theory T_f having as its only model (to within isomorphism) an algebra A_f consisting of the natural numbers $\mathbb{N}$ enriched by the function

$$ex_f(y) = \begin{cases} 1 \text{ if } \exists x\in\mathbb{N} \text{ such that } f(x,y)>0 \\ 0 \text{ otherwise} \end{cases}$$

Suppose f is the total recursive function

$$f(x,y) = \begin{cases} 1 \text{ if } x \text{ codes a convergent computation of } \varphi_y(y) \\ 0 \text{ otherwise} \end{cases}$$

(where φ_y is the partial recursive function with Gödel number y). Then ex_f is the characteristic function of the complete recursively enumerable set K (see [Rogers 1967]) so ex_f is not recursive and therefore its graph is not recursively enumerable. Hence the set of equations $ex_f(n)=m$ true in A_f is not r.e., where n,m are ground terms ($succ^j(0)$ for some j). Since for any proof system $\vdash$ the set of theorems which can be derived from a theory is r.e., there must be ground terms n,m such that $A_f\models ex_f(n)=m$ (so $T_f\models ex_f(n)=m$ since A_f is the only model of T_f) but $T_f\not\vdash ex_f(n)=m$.

6. Implementation and an example

The theorem prover described here has been implemented on the Edinburgh KL-10 computer, on top of the Edinburgh LCF system. The system is called SOGGIE, which stands for Semi-Otomatic (sic) Goal-directed Generation of Irrefutable Equations. The Clear implementation described in chapter IV (only the prolific version, so far) communicates with SOGGIE by constructing files containing ML declarations which describe agglomerates corresponding to the theories in which facts are to be proved. At present a file is produced whenever the semantics demands that a signature morphism be a theory morphism (i.e. one for each apply or derive in a specification). To prove that $\sigma:\langle\underline{\Sigma},EC\rangle\longrightarrow\langle\underline{\Sigma}',EC'\rangle$ is a theory morphism, we prove that $\sigma(EC)\subseteq EC'$; the file contains the two agglomerates $\sigma(EC)$ and EC'. The user ensures that the specification is semantically well-formed by using SOGGIE to prove in each case that the denotation of one agglomerate (the second) is included in the denotation of the other. The following rules allow this task to be decomposed into a list of goals of the form $A\stackrel{?}{\vdash}f$ (SOGGIE does this automatically):

$$A\vdash f_1 \;\&\; \ldots \;\&\; A\vdash f_n \Leftrightarrow \mathbb{F}[\![\text{close}(\{f_1,\ldots,f_n\})]\!]\subseteq\mathbb{F}[\![A]\!]$$
$$\mathbb{F}[\![A]\!]\subseteq\mathbb{F}[\![A'']\!] \;\&\; \mathbb{F}[\![A']\!]\subseteq\mathbb{F}[\![A'']\!] \Leftrightarrow \mathbb{F}[\![\text{union}(A,A')]\!]\subseteq\mathbb{F}[\![A'']\!]$$
$$\mathbb{F}[\![A]\!]\subseteq\mathbb{F}[\![\text{inv-translate}(\sigma,A')]\!] \Leftrightarrow \mathbb{F}[\![\text{translate}(\sigma,A)]\!]\subseteq\mathbb{F}[\![A']\!]$$
$$\mathbb{F}[\![A]\!]\subseteq\mathbb{F}[\![\text{translate}(\sigma,A')]\!] \Rightarrow \mathbb{F}[\![\text{inv-translate}(\sigma,A)]\!]\subseteq\mathbb{F}[\![A']\!]$$
(but not vice versa)

There is no analogous rule concerning add-equality. But this does not cause a problem; it is very unusual in practice for the source of an alleged theory morphism to include add-equality (which can only arise from application of the <u>data</u> operation) except as part of a subtheory shared with the target (such as Bool). In such a case both agglomerates will include identical subagglomerates containing add-equality nodes, and so the target agglomerate obviously includes that fragment of the source agglomerate.

The tags attached to sorts and operators are retained; the tagged name $name_{tag}$ becomes the LCF identifier name'tag (quotation marks are permitted in LCF identifiers). Equations and constraints are translated to PPLAMBDA forms as described in section 3. Error

equations are ignored at present, and quantifiers in equations are not permitted. Although the specification is not strictly semantically well-formed unless the facts given in the files are proved, the responsibility for this is left to the user.

SOGGIE together with LCF fits into 128K words, with sufficient workspace left for simple proofs (LCF itself accounts for 96K of this total). The system can be expanded to provide extra workspace for more ambitious proofs. Timing statistics may be misleading in comparison with statistics obtained for other theorem provers; ML is run interpretively, and SOGGIE was written without much concern for efficiency.

As implemented, the theorem prover is slightly different than described in the preceding sections. One difference is in the inference rule LCFINFER. The version used in SOGGIE is as follows:

LCFINFER: $A \vdash f_1 \;\&\; \ldots \;\&\; A \vdash f_n \;\&\; f'_1, \ldots, f'_m \vdash_{LCF} f \;\Rightarrow\; A \vdash f$
where each f'_j is a type instance of some f_k

This modification is necessary because of a restriction on the PPLAMBDA inference rule for type variable instantiation, which requires us to instantiate type variables in induction rules on the left-hand side of the $\vdash_{LCF}$ before using them. The change is transparent so long as the built-in induction tactic INDTAC is used.

A second difference is that DREDGE is implemented as a primitive inference rule, rather than constructed from other inference rules as a derived rule. Also, DREDGE accepts a list of forms and produces a list of facts, rather than transforming a single form to a single fact. These changes are necessary for reasons of efficiency; much of the time consumed by SOGGIE is devoted to dredging (typically about forty percent) and so optimisation of this step is important.

During the course of a proof attempt SOGGIE draws the shape of the agglomerate as it explores. Each 'dive' exposes a new node of the tree, labelling it according to its contructor. A 'dredge' draws the outline of an entire subtree without labelling the nodes. This enables the user to follow the progress of the proof as it

proceeds. Except for this, the user interface of SOGGIE is rather primitive. To use the theorem prover, the user loads a file containing the type and operator declarations for his theory and then a file containing the agglomerates he wants to work with (both these files are produced by the Clear system). This binds a list of goals (agglomerate x form pairs) to be proved to the variable goallist. The user selects a goal from this list and applies PROVE to it. PROVE prepares the display for drawing the agglomerate and then applies SUPERTAC EQTAC to the goal. This produces either the empty goal list and a proof (a function which when applied to the empty theorem list yields a fact corresponding to the goal), or else failure.

A typical example for SOGGIE is to prove the equation

length(k) plus length(l) = length(append(k,l))

in the theory given by the following Clear specification:

```
const Nat =
  let Nat0 =
    enrich Bool by
      data sorts nat
           opns zero : nat
                succ : nat -> nat   enden in
   enrich Nat0 by
     opns (_ plus _) : nat,nat -> nat
     eqns zero plus n = n
          succ(n) plus m = succ(n plus m)   enden


meta Triv = theory sorts element endth


proc List(X:Triv) =
  let List0 =
    enrich X + Bool by
      data sorts list
           opns nil : list
                cons : element,list -> list   enden in
  enrich List0 + Nat by
    opns length : list -> nat
         append : list,list -> list
    eqns length(nil) = zero
         length(cons(a,l)) = succ(length(l))
         append(nil,l) = l
         append(cons(a,l),m) = cons(a,append(l,m))   enden
```

```
proc Sequence(X:Triv) =
  enrich X + Bool by
    data sorts sequence
         opns empty : sequence
              unit : element -> sequence
              (_ conc _) : sequence,sequence -> sequence
         eqns empty conc s = s
              s conc empty = s
              (s conc t) conc v = s conc (t conc v)    enden

 List( Sequence(Nat[element is nat]) [element is sequence] )
```

Nonalphabetic operators such as . (sequence concatenation), + and 0 are not allowed in LCF (actually, 0 is allowed but tagged operators like 0'E24 are not allowed) so conc, plus and zero have been used instead.

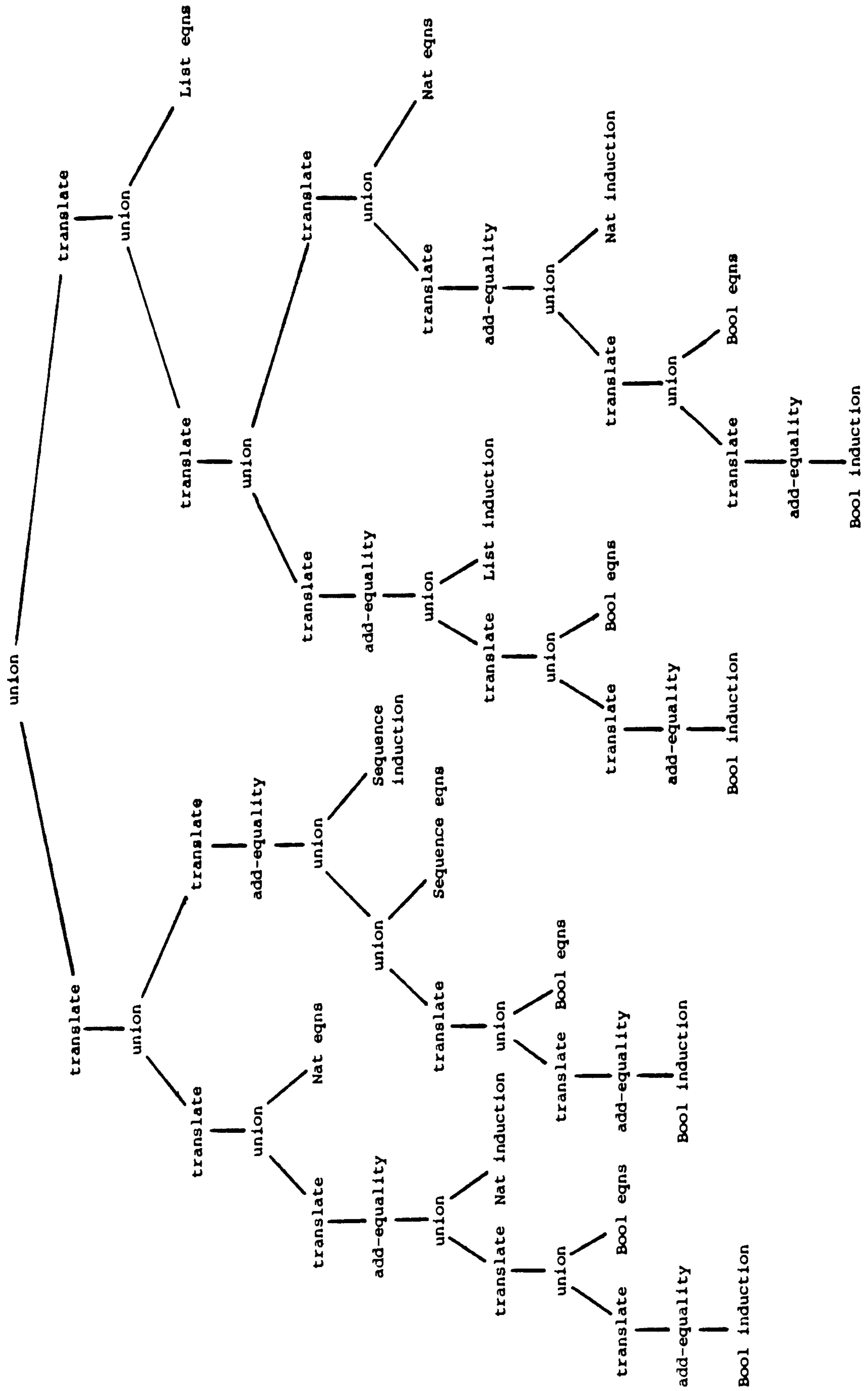
The agglomerate produced by the Clear system as the denotation of this specification is shown on the next page in the form of a tree. Note that the theory Nat appears twice in the tree, and Bool appears four times.

The initial goal is a pair consisting of this agglomerate and the PPLAMBDA form:

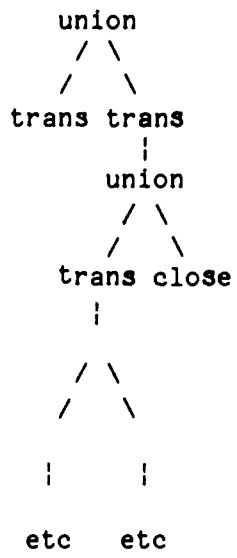
```
"!k:list'P22. !l:list'P22.
 plus'E5(length'P22(k),length'P22(l)) = length'P22(append'P22(k,l))"
```

Note that sorts and operators are tagged, and that infix (and other distributed-fix) operators have become prefix.

SUPERTAC EQTAC applied to the goal (via PROVE as described above) begins by diving down the left branch of the topmost union node of the agglomerate. But the first translate node forms a barrier to further diving because the source of the signature morphism it contains has no sort corresponding to list'P22 and no operator corresponding to length'P22 or append'P22 (all information pertaining to lists is contained in the right-hand subagglomerate of the topmost union node). This failure causes the system to backtrack and dive down the right branch of the union. It succeeds in diving through the translate node and down the left branch of the next union node. At this point it meets another barrier; no


union
translate
union
List eqns
translate
union
translate
union
Nat eqns
translate
add-equality
union
Nat induction
translate
union
Bool eqns
translate
add-equality
Bool induction
translate
add-equality
union
List induction
translate
union
Bool eqns
translate
add-equality
Bool induction
translate
union
translate
add-equality
union
Sequence induction
union
Sequence eqns
translate
union
Bool eqns
translate
add-equality
Bool induction
translate
union
Nat eqns
translate
add-equality
union
Nat induction
translate
union
Bool eqns
translate
add-equality
Bool induction

operator corresponding with append'P22 or length'P22 is available below this translate node. The system backtracks to the union node immediately above and dives down the right branch to the close node containing the equations defining append'P22 and length'P22. LCFINFERTAC (applied to the set of equations available at that node) THEN EQTAC is applied but this fails to achieve the goal. This failure causes the system to backtrack again to the immediately preceeding union node at which point it applies DREDGETAC THEN EQTAC. DREDGETAC produces an LCF goal in which equations defining plus'E5, length'P22 and append'P22 along with an induction rule for list'P22 (as well as other equations and induction rules) are available as assumptions. This goal is achieved by EQTAC; the proof does an induction on k followed by two separate inductions on l (one each for the base case and induction step). The result is an empty goal list and a proof function which yields the desired fact when applied to the empty thm list. The following is (an abbreviated version of) the display drawn by SOGGIE while searching for the proof:



The CPU time to achieve the original goal is 22.6 seconds (excluding garbage collection); dredging accounts for 7 seconds of this total. The CPU time to perform the proof (transforming the empty thm list to the desired fact) is 11.5 seconds where dredging again accounts for 7 seconds of this total.

It is important to note how easily the theorem prover was able to

avoid all the irrelevant information contained in the left half of the agglomerate. It would use exactly the same sequence of reasoning to prove the fact in the theory List(T) for any theory T. This is because the fact is true of any sort of list, whether the elements are sequences of natural numbers or something else. This seems to be a common situation for proofs about parameterised theories. If T is very large then it is important that the system ignores T if it is irrelevant to the proof.

The example above is typical of the facts which SOGGIE is able to handle. Experimentation with SOGGIE has so far been limited, but it has been used to prove simple boolean identities, reflexivity and transitivity of $\leq$, and associativity of + and append. In each case the axioms relevant to the proof were buried within a larger agglomerate. Comparison with a theorem proving system such as the one described by Boyer and Moore [1979] would certainly not be favourable, but this is entirely due to the mediocre facility for equational deduction we use. Our goal is not to automatically prove all theorems, but to provide a set of tools sufficient to enable a user to construct his own proofs. It is nice that SOGGIE is able to prove a certain class of theorems automatically, but more important is that it is able to reduce any proof problem to one of ordinary equational deduction. Also important is the way that SUPERTAC takes advantage of the structure of Clear specifications to simplify the theorem-proving task; this appears to be a novel approach to theorem proving.

7. Possible improvements

It is easy to think of ways in which SOGGIE could be improved. A better EQTAC which utilises state-of-the-art methods for equational deduction would improve the performance of the system substantially. Failing this, SOGGIE could at least be a little bit more careful about adding equations to the simplification set. It is easy to filter out at least the more obvious permutative rules, protecting the system from looping in the course of simplification.

It would be great help if SOGGIE could check the consistency of enrichments (i.e. that equations added in an enrichment do not violate any previous data constraints). Again, this amounts to proving inequality. As mentioned before this is easy in an anarchic theory but impossible in general, so SOGGIE does not attempt to deal with the problem.

The theorem prover needs most of all a good user interface. It is important that when a proof attempt fails, the user should know what happened and be able to return to the point of failure so that he can fill in missing steps manually. A good first attempt at a more friendly user interface would be a version of SUPERTAC which upon failure prints a table containing the goals at which it failed together with the sequence of choices which led to each of those goals. The user could examine this list to find the goal which he thinks would be easiest to achieve manually. He would then use another tactic to repeat the particular line of reasoning which led to the selected goal; this tactic would take as a parameter the sequence of choices provided by SUPERTAC. Once the proper environment has been re-established, the user would have all the facilities of LCF at his disposal to achieve the goal. If he is successful, then the proof of the goal can be composed with the partial proof which SOGGIE was able to perform by itself to give the proof of the original goal. Note that the goals at which the system fails are always LCF goals; SOGGIE is able to automatically reduce any problem to the level of equational deduction.

Once a fact has been established, it would be helpful to add it to the agglomerate for use as a lemma in future proofs. If the agglomerate were represented as a DAG (directed acyclic graph) with

identification of identical subagglomerates rather than as a tree then the lemma would automatically be incorporated in the appropriate places throughout the agglomerate. Common theories such as Bool typically appear many times in even a small agglomerate. In a similar vein, if the dredge function were altered to deposit intermediate results at each node it visits (the dredge of each subagglomerate would be deposited at its root), then subsequent calls of dredge could be made to run much faster. These enhancements require an ability to destructively update data structures. This is awkward in DEC-10 ML but easy in Luca Cardelli's version of ML for VAX.

Present users of SOGGIE are required to view a specification as a huge and complex tree with an elaborate relation to the original specification. This undesirable state of affairs results from the separation of theorem proving into a separate activity which must be performed in isolation. Ideally, SOGGIE would be combined with the Clear semantics program into a single integrated system. This could be done in such a way that the user would never have to know that his theories denote complicated agglomerates, or that sorts and operators carry tags, although agglomerates and tags would still exist at some lower level. Interaction between the system and the user would be in Clear, using the sorts and operators defined in the user's specification. But the user needs some way of directing the system when an automatic proof fails. LCF provides a powerful tool, but the ordinary user would not be interested in writing his own tactics in ML. A simple facility for interactive proof using a set of tactics provided by the system would be sufficient for all but the most sophisticated users. Such users could use ML in the usual way to define higher-level strategies from the tactics provided.

CHAPTER SEVEN

IMPLEMENTATION OF SPECIFICATIONS AND PROGRAM DEVELOPMENT

Clear specifications can be viewed as abstract programs. Some specifications are so completely abstract that they give no hint of a method for finding an answer. For example, a function for inverting an nxn matrix can be specified as follows:

```
const Inverse =
  enrich Matrices by
    opns inv : matrix -> matrix
    eqns inv(A) x A = I
         A x inv(A) = I    enden
```

(provided that the theory Matrices includes specifications of matrix multiplication and the identity nxn matrix). Other specifications are just HOPE programs written in a slightly different notation. For example:

```
proc Reverse(X:Triv) =
  enrich List(X) by
    opns reverse : list -> list
    eqns reverse(nil) = nil
         reverse(a::l) = append(reverse(l),a::nil)    enden
```

A Clear specification amounts to a HOPE program if all data is anarchic and all axioms are equations with simple left-hand sides, enabling their use as rewrite rules.

It is usually easiest to specify a problem at a relatively abstract level. We can then work gradually and systematically toward a low-level 'program' which satisfies the specification. This will normally involve the introduction of auxiliary functions, particular data representations and so on. This approach to program development is related to the well-known programming discipline of stepwise refinement advocated by Wirth [1971] and Dijkstra [1972].

A formalisation of this programming methodology depends on some precise notion of the implementation of a specification by a lower-level specification. This turns out to be a rather difficult and subtle problem. Previous notions have been given for the implementation of both non-parameterised specifications ([Goguen,

Thatcher and Wagner 1978], [Nourani 1979], [Hupbach 1980], [Ehrig, Kreowski and Padawitz 1980], [Ehrich 1982]) and parameterised specifications ([Ganzinger 1980], [Hupbach 1981], [Ehrig and Kreowski 1982]), but none of these approaches deals adequately with Clear-style specifications which may be constructed in a hierarchical fashion using data and which may be loose. A definition of implementation is presented in this chapter which agrees with our intuitive notions built upon programming experience and which handles Clear-style specifications, based on a new (and seemingly fundamental) concept of the simulation of a theory by an algebra. This definition extends to give a definition of the implementation of parameterised specifications. An example of an implementation is given and several other examples are sketched.

For most of the chapter a variant of Clear is employed in which the notion of a data constraint is replaced by the weaker notion of a hierarchy constraint. The result is still a viable specification language, although specifications tend to be somewhat longer than in ordinary Clear. We later show that all results hold for Clear with data constraints, but only under more restrictive conditions.

The 'putting-together' theme of Clear and the proposals of Goguen and Burstall [1980] for CAT (a proposed system for systematic program development using Clear) lead us to wonder if implementations can be put together as well. We prove that if P is implemented by P' (where P and P' are 'well-behaved' parameterised theories) and A is implemented by A', then P(A) is implemented by P'(A').

We prove that implementations compose in another dimension as well. If a high-level theory A is implemented by a lower-level theory B which is in turn implemented by a still lower-level theory C (and an extra compatibility condition is satisfied), then A is implemented by C. These two results allow large specifications to be refined in a gradual and modular fashion, a little bit at a time.

All of the definitions and results in this chapter are the product of work done in collaboration with Martin Wirsing, Technische Universität München, reported in [Sannella and Wirsing 1982].

1. Clear with hierarchy constraints

In section I.1.1 Clear's data operation was introduced as a way of restricting the class of models of a theory to exclude trivial and other undesirable models. In section II.5 the notion of a data constraint was defined; an application of the data operation contributes a data constraint to the resulting theory, and satisfaction was defined so that only an algebra without 'junk' (elements which are not the value of any term) and without 'confusion' (identification of terms not required by the equations) satisfies a data constraint, where the precise nature of junk and confusion depend on the data constraint in question.

A notion for the implementation of one theory by another will be given in the next section. In section 4 it is shown that the implementation relation is transitive; in practical terms this means that the result of refining a specification several times in succession is an implementation of the original specification. Another very desirable property would be that the theory-building operations of Clear preserve implementations, so combining the implementations of two theories gives an implementation of the combined theory. Unfortunately, in the presence of data constraints this property only holds in general under a seriously restrictive condition. As a result, our notion of implementation is apparently of limited usefulness in practice.

This situation can be improved if the notion of a data constraint is replaced by the weaker notion of a hierarchy constraint (see [Broy et al 1979] and [Wirsing and Broy 1981]). Hierarchy constraints are identical to data constraints except that models need only satisfy the inequation true≠false rather than the stronger "no confusion" condition. The same definition of implementation works if theories include hierarchy constraints in place of data constraints, and in this case more reasonable conditions guarantee the preservation of implementations under Clear's theory-building operations. Accordingly, for the bulk of this chapter we use hierarchical Clear, where hierarchy constraints are contributed to a theory by an operation called 'data'. Since hierarchy constraints

are weaker than data constraints, specifications in hierarchical Clear tend to be somewhat longer than in ordinary Clear -- as in the terminal algebra approach of Wand [1979], it is sometimes necessary to add extra operators to avoid trivial models. At the end of the chapter it is shown that all results hold for Clear with data constraints but only under more restrictive conditions.

We now give formal definitions concerning hierarchy constraints; note that in most respects hierarchy and data constraints are identical.

Def: A Σ-hierarchy constraint c is a pair $\langle i,\sigma\rangle$ where $i:\underline{T}\hookrightarrow\underline{T}'$ is a simple theory inclusion and $\sigma:\text{signature}(\underline{T}')\rightarrow\Sigma$ is a signature morphism.

Def: If $\sigma':\Sigma\rightarrow\Sigma'$ is a signature morphism and $\langle i,\sigma\rangle$ is a Σ-hierarchy constraint, then σ' applied to $\langle i,\sigma\rangle$ gives the Σ'-hierarchy constraint $\langle i,\sigma.\sigma'\rangle$.

Without loss of generality we assume that every theory contains the theory Bool (with sort bool and constants true and false) as a primitive subtheory.

Def: A Σ-algebra $\underline{A}$ satisfies a Σ-hierarchy constraint $\langle i:\underline{T}\hookrightarrow\underline{T}',\sigma:\text{sig}(\underline{T}')\rightarrow\Sigma\rangle$ if

$$[\ \text{letting } \underline{A}_{\text{target}} = \underline{A}|^{\sigma}_{\text{sig}(\underline{T}')} \quad \text{and } \underline{A}_{\text{source}} = \underline{A}|^{i.\sigma}_{\text{sig}(\underline{T})}\ ]$$

$\underline{A}_{\text{target}}$ is a model of $\underline{T}'$ and

- "No crime": $\underline{A} \models \text{true}\neq\text{false}$ (i.e. $\underline{A} \not\models \text{true}=\text{false}$).
- "No junk": Every element in $\underline{A}_{\text{target}}$ is the value of a $\underline{T}'$-term which has variables only in sorts of $\underline{T}$, for some assignment of $\underline{A}_{\text{source}}$ values.

Note that the only difference between a data constraint and a hierarchy constraint is in the definition of satisfaction; compare the "no crime" condition above with the "no confusion" condition in section II.5.

Def: A hierarchical Σ-theory presentation is a pair $\langle\Sigma,EC\rangle$ where Σ is a signature and EC is a set of Σ-equations and Σ-hierarchy constraints.

The notions of hierarchical theory, satisfaction (of a hierarchical theory), closure and hierarchical theory morphism follow as before. The denotation of a hierarchical Clear specification is a hierarchical theory. For the remainder of the chapter (except where noted at the end of section 4) all discussion will concern only hierarchical Clear. We will use terms like 'theory' in place of longer terms like 'hierarchical theory'. For the purposes of this chapter it is convenient to dispense with the equality predicates == normally added by the data operation; these extra operators cause no problems but only serve to make the examples longer. We will assume in this chapter that all theories have been constructed using Clear (so e.g. no theory may contain both $\langle TA \hookrightarrow TA',\sigma\rangle$ and $\langle TB \hookrightarrow TB',\sigma'\rangle$ as constraints if $TA \subset TB \subset TA'$ and σ,σ' are inclusions). This assumption is implicit in some of the proofs of section 4.

A short example will illustrate the difference between data and hierarchy constraints. Consider the following specification in ordinary Clear (with data constraints):

```
const Nat =
  enrich Bool by
    data sorts nat
         opns 0 : nat
              succ : nat -> nat    enden

const T =
  enrich Nat by
    data sorts newnat
         opns f : nat -> newnat    enden
```

T includes two data constraints, $C1=\langle\Phi \hookrightarrow Nat, sig(Nat) \hookrightarrow sig(T)\rangle$ and $C2=\langle Nat \hookrightarrow T, 1_{sig(T)}\rangle$. Given a sig(T)-algebra, we can check if it satisfies these constraints. For example:

$A_{nat} = \{0,1,2,\ldots\}$
$A_{newnat} = \{0,I,II,\ldots\}$
$f(0)=0 \quad f(1)=I \quad f(2)=0 \quad f(3)=III \quad f(4)=IV \quad \ldots$

(with the usual interpretation of Bool). This satisfies constraint C1, but fails to satisfy the "no confusion" condition for constraint C2 (consider the equation f(x)=f(y) under the injective assignment [x↦0, y↦2]). It also violates the "no junk" condition (the element $II \notin A_{newnat}$ is not the value of any term). But if the function f is altered so that f(2)=II then the constraint is satisfied. In general, any algebra satisfying these data constraints will have both carriers isomorphic to $\mathbb{N}$ with f 1-1 and onto.

Changing data above to 'data' changes both data constraints to hierarchy constraints. The following algebra is then a model of T, although it does not satisfy the "no confusion" condition for constraint C2:

$A_{nat} = \{0,1,2,...\}$
$A_{newnat} = \{0\}$
$f(0) = f(1) = f(2) = ... = 0$

(again with the usual interpretation of Bool). It is necessary to add some new operators and equations to retain the original class of models, for example:

```
const Nat' =
  enrich Bool by
    'data' sorts nat
           opns 0 : nat
                succ : nat -> nat
                eq : nat,nat -> bool
           eqns eq(n,n) = true
                eq(n,m) = eq(m,n)
                eq(0,succ(n)) = false
                eq(succ(n),succ(m)) = eq(n,m)    enden

const T' =
  enrich Nat by
    'data' sorts newnat
           opns f : nat -> newnat
                eq : newnat,newnat -> bool
           eqns eq(f(n),f(m)) = eq(n,m)    enden
```

Further examples appear throughout the rest of this chapter.

For later results we need a generalisation of Guttag's notion of sufficient completeness [Guttag and Horning 1978] and of the

classical notion of conservativeness from logic:

Def: A theory $\underline{T}$ is sufficiently complete with respect to a set of operators Σ, sorts S, a subset Σ' of Σ, and variables of sorts X (where $S,X \subseteq sorts(\underline{T})$, $\Sigma \subseteq opns(\underline{T})$) if for every term t of an S sort containing operators of Σ and variables of X sorts, there exists a term t' with variables of X sorts and operators of Σ' such that $\underline{T} \vdash t=t'$.

Def: A theory $\underline{T}$ is conservative with respect to a theory $\underline{T}' \subseteq \underline{T}$ if for all equations e containing operators only of $\underline{T}'$, $\underline{T} \vdash e \Rightarrow \underline{T}' \vdash e$.

Sufficient completeness means that $\underline{T}$ does not contain any new term of an old sort which is not provably equal to an old term (where 'new' and 'old' depend on Σ, S, Σ' and X). Conservativeness means that old terms (from $\underline{T}'$) are not newly identified in $\underline{T}$. Instances of these general notions guarantee that all models of a theory possess a convenient hierarchical structure.

To apply the above definitions it will be convenient to refer to the following notions of constrained sort and constructor.

Def: Let $\underline{T}''$ be a theory and let $c=\langle \underline{T} \hookrightarrow \underline{T}', \sigma: sig(\underline{T}') \rightarrow sig(\underline{T}'') \rangle$ be a constraint of $\underline{T}''$.

- A sort s of $\underline{T}''$ is called constrained (with respect to c) if $s \in \sigma(sorts(\underline{T}')-sorts(\underline{T}))$.

- An operator f: ...->s of $\underline{T}''$ is called a constructor (with respect to c) if $f \in \sigma(opns(\underline{T}'))$ and $s \in$ constrained-sorts(c), or if $s \notin$ constrained-sorts(c).

2. A notion of implementation

A formal approach to stepwise refinement of specifications must begin with some notion of the implementation of a specification by another (lower level) specification. Armed with a precise definition of this notion, we can prove the correctness of refinement steps, providing a basis for a methodology for the systematic development of programs which are guaranteed to satisfy their specifications. But first we must be certain that the definition itself is sound and agrees with our intuitive notions built upon programming experience. It turns out that a formal definition of implementation adequate to deal with all cases which arise in practice is rather elaborate, and so it is better to carefully examine the situation first from a less formal point of view.

Suppose we are given two theories $T=\langle\Sigma,EC\rangle$ and $T'=\langle\Sigma',EC'\rangle$. We want to implement the theory T (the abstract specification) using the sorts and operators provided by T' (the concrete specification). Previous formal approaches (see [Goguen, Thatcher and Wagner 1978], [Nourani 1979], [Hupbach 1980], [Ehrig, Kreowski and Padawitz 1980], [Ganzinger 1980], [Ehrich 1982]) agree that T' implements T if there is some way of deriving sorts and operators like those of T from the sorts and operators of T'. Each approach considers a different way of making the 'bridge' from T' to T. We will require that there be a more or less direct correspondence between the sorts and operators of T and those of T'. Each sort or operator in Σ must be implemented by a sort or operator in Σ' -- this correspondence will be embodied by a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$. Note that two different sorts or operators in Σ may map to the same Σ' sort or operator, and also that there may be some (auxiliary) sorts and operators in Σ' which remain unused. This is a simplification over previous approaches, which generally allow some kind of restricted enrichment of T' to T'' before matching T with T''. But the power is the same; we would say that T'' implements T and leave the enrichment from T' to T'' to the user. As a consequence of a later theorem (see section 4) our results extend to more complex notions.

Given a signature morphism $\sigma:\underline{\Sigma}\rightarrow\underline{\Sigma}'$, what relationship must hold between $\underline{T}$ and $\underline{T}'$ before we can say that $\underline{T}'$ implements $\underline{T}$? One might suspect that $\sigma:\underline{T}\rightarrow\underline{T}'$ is required to be a theory morphism -- i.e. that if $\underline{A}'$ is a model of $\underline{T}'$ then its restriction $\underline{A}'|_{\underline{\Sigma}}^{\sigma}$ must be a model of $\underline{T}$ -- but this condition is too strong. We shall say that $\underline{T}'$ implements $\underline{T}$ if the $\underline{\Sigma}$-restriction of each model of $\underline{T}'$ *simulates* $\underline{T}$. A $\underline{\Sigma}$-algebra simulates $\underline{T}$ if it satisfies the axioms of $\underline{T}$ after allowing for the representation of data.

We have decomposed the notion of implementation into three separate issues:

1. Enriching the concrete theory $\underline{T}'$ (adding derived operators and possibly some new sorts) to give an intermediate $\underline{\Sigma}''$-theory $\underline{T}''$.

2. Renaming some of the sorts and operators of $\underline{\Sigma}''$ and forgetting others, so as to match $\underline{\Sigma}$.

3. Simulation of $\underline{T}$ by a $\underline{\Sigma}$-algebra (obtained by $\underline{\Sigma}$-restricting a model of $\underline{T}''$).

As already mentioned we can safely ignore (1) and assume that $\underline{T}''=\underline{T}'$ because a later theorem allows all of our results to be extended to the case where $\underline{T}''\neq\underline{T}'$. Issue (2) presents no problems since the restriction of an algebra to a subsignature (with renaming) was defined in chapter II. The fundamental issue is (3); we need a satisfactory definition of simulation which captures our intuition concerning data representation.

We said above that a $\underline{\Sigma}$-algebra $\underline{A}$ simulates a $\underline{\Sigma}$-theory $\underline{T}$ if it satisfies the axioms of $\underline{T}$ modulo data representation. In particular, we must allow for two kinds of flexibility:

- A subset of the values of an $\underline{A}$ sort may be used to represent all the values of a $\underline{T}$ sort. Example: the natural numbers are simulated by the integers, where the negative integers are not needed.

- More than one $\underline{A}$ value may be used to represent the same $\underline{T}$ value. Example: simulating sets by strings -- the order does not matter, so "1.2.3" = "3.2.1" (as sets).

Now $\underline{A}$ simulates $\underline{T}$ if (and only if) $\underline{A}$ is a model of $\underline{T}$ after these two considerations have been taken into account. This ensures that operators will yield the specified result (modulo data

representation) which seems to be the central issue.

For the definition of simulation we need an auxiliary notion. As mentioned above, a subset of the values of A may be used to represent all values required by T. Restricting the carriers of A to the values which are actually used yields an intermediate algebra which plays an important role in the definition of simulation. We do not want to restrict the carrier for every sort, but only for those sorts of Σ which are constrained in T (for unconstrained sorts we do not know which values are unused). This is where we depart from the usual practice of restricting to 'reachable' values (see for example [Ehrig, Kreowski and Padawitz 1980]). We want the subalgebra which has been reduced just enough to satisfy the "no junk" condition for each constraint in T.

Def: If Σ is a signature, A is a Σ-algebra and T is a Σ-theory, then $\text{restrict}_T(A)$ is the largest subalgebra A' of A satisfying the "no junk" condition (section 1) for every constraint $\langle i:T' \hookrightarrow T'', \sigma:\text{sig}(T'') \rightarrow \Sigma \rangle$ in T, that is:

[letting $A'_{target} = A'|^{\sigma}_{sig(T'')}$
and $A'_{source} = A'|^{i.\sigma}_{sig(T')}$]

- Every element in A'_{target} is the value of a T"-term which has variables only in sorts of T', for some assignment of A'_{source} values.

Note that the subalgebra A' does not always exist. Consider the following example:

```
const T = let Nat = enrich Bool by
                      'data' sorts nat
                             opns 0 : nat
                                  succ : nat -> nat    enden in
          enrich Nat by
            opns neg : nat    enden
```

Let Σ be the signature of T. Suppose A is the Σ-algebra with carrier {-1,0,1,...}, the usual interpretation for the operators 0 and succ, and neg=-1. Now $\text{restrict}_T(A)$ does not exist because every subalgebra of A must contain -1 (the value of neg) and hence fails to satisfy the "no junk" condition for the constraint of T.

A Σ-algebra $\underline{A}$ simulates a Σ-theory $\underline{T}$ if it satisfies the equations and constraints of $\underline{T}$ after allowing for unused carrier elements and multiple representations.

Def: If Σ is a signature, $\underline{A}$,$\underline{A}'$ are Σ-algebras and $\underline{T}$ is a Σ-theory, then $\underline{A}$ simulates $\underline{A}'$ if there is a surjective Σ-homomorphism rep:restrict$_{\underline{T}}(A)\rightarrow\underline{A}'$. $\underline{A}$ simulates $\underline{T}$ if there is a model of $\underline{T}$ which is simulated by $\underline{A}$.

Note that simulation of an algebra by an algebra is with respect to a theory because it is defined in terms of the restrict operation. It is not possible to allow for unused elements of the 'concrete' algebra otherwise; without the constraints of $\underline{T}$ we cannot distinguish between an element (of a constrained sort) which is truly unused and an element (of an unconstrained sort) which is not the value of any term.

The following definition of simulation is equivalent to the definition above (this is easy to show) but more constructive.

Def: If Σ is a signature, $\underline{A}$ is a Σ-algebra and $\underline{T}=\langle\Sigma,EC\rangle$ is a Σ-theory, then $\underline{A}$ simulates $\underline{T}$ if restrict$_{\underline{T}}(\underline{A})/\equiv_{EC}$ (call this $RI_{\underline{T}}(\underline{A})$) exists and is a model of $\underline{T}$.

[$\equiv_{EC}$ is the Σ-congruence generated by EC -- i.e. the least Σ-congruence on restrict$_{\underline{T}}(\underline{A})$ containing the relation determined by the equations in EC]

RI stands for restrict-identify, the composite operation which forms the heart of this definition. To determine if a Σ-algebra $\underline{A}$ simulates a hierarchical Σ-theory $\underline{T}$, we restrict $\underline{A}$, removing those elements from the carrier which are not used to represent the value of any Σ-term, for constrained sorts; the result of this satisfies the "no junk" condition for each constraint in $\underline{T}$. We then identify multiple concrete representations of the same abstract value by quotienting the result by the Σ-congruence generated by the equations of $\underline{T}$, obtaining an algebra which (of course) satisfies those equations and also continues to satisfy the "no junk" condition of the constraints. If this is a model of $\underline{T}$ (i.e. it satisfies the "no crime" condition for each constraint in $\underline{T}$) then $\underline{A}$

simulates $\underline{T}$. Note that any model of $\underline{T}$ simulates $\underline{T}$. It has been shown by Ehrig, Kreowski and Padawitz [1980] that the order restrict-identify gives greater generality than identify-restrict.

Clear (both the ordinary version and our variant) differs from most specification approaches/languages in that it allows the construction of loose theories having an assortment of non-isomorphic models. Such a theory need not be implemented by a theory with the same broad range of models. A loose theory leaves certain details unspecified and an implementation may choose among the possibilities or not as is convenient. That is:

- A loose theory may be implemented by a 'tighter' theory. Example: implementing the operator choose:set->integer (choose an element from a set of integers) by an operator which chooses the smallest.

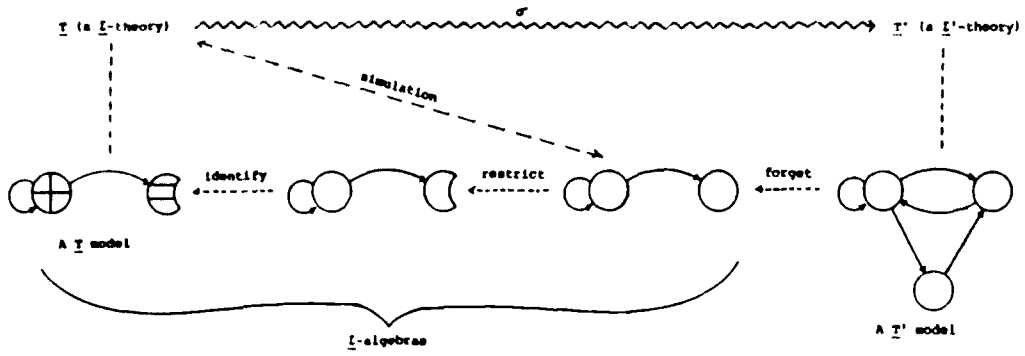
This is intuitively necessary because it would be silly to require that a program (the final result of the refinement process) embody all the vagueness of its original specification. This kind of flexibility is already taken into account by the discussion above, and is an important feature of our notion of implementation. Previous notions do not allow for it because they generally consider only a single model for any specification.

Now we are finally prepared to define our notion of the implementation of one theory by another. This definition is inspired by the notion of [Ehrig, Kreowski and Padawitz 1980] but it is not the same; they allow a more elaborate 'bridge' but otherwise their notion is more restrictive than ours. Our notion is even closer to the one of Broy *et al* [1980] but there the 'bridge' is less elaborate than ours. It also bears some resemblance to a more programming-oriented notion due to Schoett [1981].

Def: If $\underline{T}=\langle\underline{\Sigma},EC\rangle$ and $\underline{T}'=\langle\underline{\Sigma}',EC'\rangle$ are satisfiable theories and $\sigma:\underline{\Sigma}\longrightarrow\underline{\Sigma}'$ is a signature morphism, then $\underline{T}'$ *implements* $\underline{T}$ (via σ), written $\underline{T}\overset{\sigma}{\rightsquigarrow}\underline{T}'$, if for any model $\underline{A}'$ of $\underline{T}'$, $\underline{A}'|^{\sigma}_{\underline{\Sigma}}$ simulates $\underline{T}$.

Note that any theory morphism $\sigma:\underline{T}\longrightarrow\underline{T}'$ where $\underline{T}'$ is satisfiable is an implementation $\underline{T}\overset{\sigma}{\rightsquigarrow}\underline{T}'$. In particular, if $\underline{T}'$ is an enrichment of $\underline{T}$ (e.g. by equations which 'tighten' a loose theory) then $\underline{T}\rightsquigarrow\underline{T}'$.

The following diagram shows how the definitions of restriction, simulation and implementation fit together:

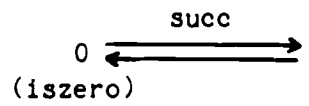


An implementation $\underline{T} \xrightarrow{\sigma} \underline{T}'$

A simple example will show how this definition works (other implementation examples are given in the next section). Consider the theory of the natural numbers modulo 2, specified as follows:

```
const Natmod2 =
  enrich Bool by
    'data' sorts natmod2
           opns 0, 1 : natmod2
                succ : natmod2 -> natmod2
                iszero : natmod2 -> bool
           eqns succ(0) = 1          succ(1) = 0
                iszero(0) = true     iszero(1) = false    enden
```

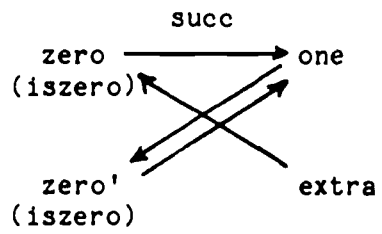
Here is a picture which shows the situation described by the equations:



Can Natmod2 be implemented by the following theory?

```
const Fourvalues =
  enrich Bool by
    'data' sorts fourvals
           opns zero, one, zero', extra : fourvals
                succ : fourvals -> fourvals
                iszero : fourvals -> bool
                eq : fourvals, fourvals -> bool
           eqns succ(zero) = one        succ(one) = zero'
                succ(zero') = one       succ(extra) = zero
                iszero(zero) = true     iszero(one) = false
                iszero(zero') = true    iszero(extra) = false
                eq(zero,one) = false    eq(zero,zero') = false
                     . . .                      . . .
                eq(p,q) = eq(q,p)       eq(p,p) = true    enden
```

Here is the picture (omitting the eq operator):



The iszero operator of Natmod2 and the eq operator of Fourvalues are needed to avoid trivial models.

All models of Fourvalues have a carrier containing 4 elements, and all models of Natmod2 have a 2-element carrier. Now consider the signature morphism σ:sig(Natmod2)$\longrightarrow$sig(Fourvalues) given by [natmod2$\mapsto$fourvals, 0$\mapsto$zero, 1$\mapsto$one, succ$\mapsto$succ, iszero$\mapsto$iszero] (and everything in Bool maps to itself). Intuitively, Natmod2$\xrightarrow{\sigma}$Fourvalues (zero and zero' both represent 0, one represents 1 and extra is unused) but is this an implementation according to the definition? Consider any model of Fourvalues (e.g. the term model -- all models are isomorphic). 'Forgetting' to the signature sig(Natmod2) eliminates the operators zero', extra and eq. Now we check if this algebra (call it A) simulates Natmod2.

- 'Restrict' removes the value of extra from the carrier.
- 'Identify' identifies the values of the terms "succ(1)" (=zero') and "0" (=zero).

The "no crime" condition of Natmod2's constraint requires that the values of true and false remain separate; this condition is

satisfied, so $\underline{A}$ simulates Natmod2 and Natmod2$\xrightarrow{\sigma}$Fourvalues is an implementation.

Suppose that the equation succ(zero')=one in Fourvalues were replaced by:

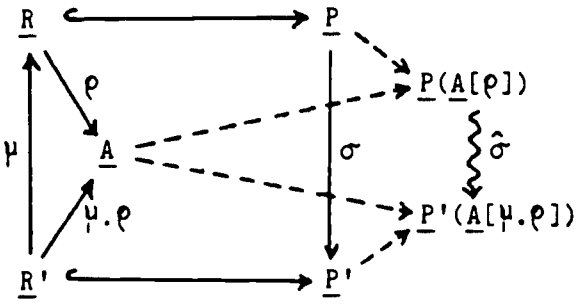
succ(zero')=zero.

Forget (producing an algebra $\underline{B}$) followed by restrict has the same effect on any model of Fourvalues, but now identify collapses the carrier for sort natmod2 to a single element (the closure of the equations in Natmod2 includes the equation succ(succ(p))=p, so "succ(succ(0))" (=zero') is identified with "0" (=zero), and "succ(succ(1))" (=zero) is identified with "1" (=one)). Furthermore, the carrier for sort bool collapses; "iszero(succ(succ(1)))" (=true) is identified with "iszero(1)" (=false). The result fails to satisfy the "no crime" condition of the constraint, so $\underline{B}$ does not simulate Natmod2 and Natmod2$\xrightarrow{\sigma}$Fourvalues is no longer an implementation.

It is not difficult to extend our notion of implementation to deal with parameterised theories. We will consider here only the single-parameter case, but the extension to multiple parameters should pose no problems.

Since a parameterised theory $\underline{R}\hookleftarrow\underline{P}$ (that is, a procedure with requirement theory $\underline{R}$ and body $\underline{P}$ -- $\underline{R}$ will always be included in $\underline{P}$) is a function taking a theory $\underline{A}$ as an parameter and producing a theory $\underline{P}(\underline{A})$ as a result, an implementation $\underline{R}'\hookleftarrow\underline{P}'$ of $\underline{R}\hookleftarrow\underline{P}$ is a function as well which takes any parameter theory $\underline{A}$ of $\underline{P}$ as argument and produces a theory $\underline{P}'(\underline{A})$ which implements $\underline{P}(\underline{A})$ as result. But this does not specify what relation (if any) must hold between the requirement theories $\underline{R}$ and $\underline{R}'$. Since every actual parameter $\underline{A}$ of $\underline{R}\hookleftarrow\underline{P}$ (which must match $\underline{R}$) should be an actual parameter of $\underline{R}'\hookleftarrow\underline{P}'$, it must match $\underline{R}'$ as well. This requires a theory morphism $\mu:\underline{R}'\rightarrow\underline{R}$ (then a fitting morphism $\rho:\underline{R}\rightarrow\underline{A}$ gives a fitting morphism $\mu.\rho:\underline{R}'\rightarrow\underline{A}$).

Def: If $\underline{R} \hookrightarrow \underline{P}$ and $\underline{R}' \hookrightarrow \underline{P}'$ are parameterised theories, $\mu:\underline{R}' \to \underline{R}$ is a theory morphism and $\sigma:sig(\underline{P}) \to sig(\underline{P}')$ is a signature morphism, then $\underline{R}' \hookrightarrow \underline{P}'$ implements $\underline{R} \hookrightarrow \underline{P}$ (via σ and μ), written $\underline{R} \hookrightarrow \underline{P} \xrightarrow[\mu]{\sigma} \underline{R}' \hookrightarrow \underline{P}'$, if for all theories $\underline{A}$ with fitting morphism $\rho:\underline{R} \to \underline{A}$, $\underline{P}(\underline{A}[\rho]) \xrightarrow{\hat{\sigma}} \underline{P}'(\underline{A}[\mu.\rho])$ where $\hat{\sigma}$ is the extension of σ from $\underline{P}$ to $\underline{P}(\underline{A}[\rho])$ defined using the universal property of the pushout $\underline{P}(\underline{A}[\rho])$ in the obvious way (so $\hat{\sigma}|_{sig(\underline{P})-sig(\underline{R})} = \sigma$ and $\hat{\sigma}|_{sig(\underline{A})} = id$).



Ordinarily $\underline{R}$ and $\underline{R}'$ will be the same theory, or at least the same modulo a change of signature. Otherwise $\underline{R}'$ must be weaker than $\underline{R}$.

Note that the definition of implementation for parameterised theories requires a certain property to hold for every possible actual parameter theory and fitting morphism. Better would be a definition which refers only to the parameterised theories themselves. Unfortunately, such a definition does not seem to work under the existing framework. Perhaps it would be possible to give some conditions on $\underline{R} \hookrightarrow \underline{P}$ and $\underline{R}' \hookrightarrow \underline{P}'$ under which the above definition reduces to the simpler form, but we have so far been unable to discover satisfactory ones.

Sometimes it is natural to split the implementation of a parameterised theory into two or more cases, implementing it for reasons of efficiency in different ways depending on some additional conditions on the parameters. For example:

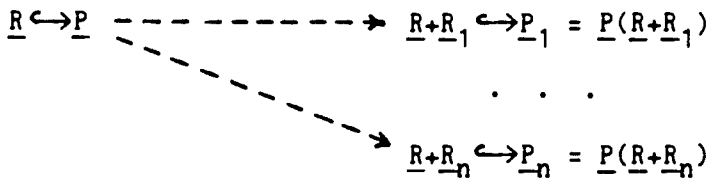
- Sets: A set can be represented as a binary sequence if the range of possible values is small; otherwise it must be represented as a sequence (or tree, etc) of values.

- Parsing: Different algorithms can be applied depending on the nature of the grammar (operator precedence, LR, context sensitive, etc -- see [Aho and Ullman 1977]).

- Sorting: Distribution sort can be used if the range of values is small; otherwise quicksort (see [Knuth 1973]).

In each instance the cases must exhaust the domain of possibilities, but they need not be mutually exclusive.

Our present notion of implementation does not treat such cases. We could extend it to give a definition of the implementation of a parameterised theory $\underline{R} \hookrightarrow \underline{P}$ by a collection of parameterised theories $\underline{R}' + \underline{R}'_1 \hookrightarrow \underline{P}'_1$, ..., $\underline{R}' + \underline{R}'_n \hookrightarrow \underline{P}'_n$ (where for every theory $\underline{A}$ with a theory morphism $\sigma : \underline{R} \rightarrow \underline{A}$ there must exist some $i \geq 1$ such that $\sigma' : \underline{R}' + \underline{R}'_i \rightarrow \underline{A}$ exists). But we force the case split to the <u>abstract</u> level, rather than entangle it with the already complex transition from abstract to concrete:



This collection of n parameterised theories is equivalent to the original $\underline{R} \hookrightarrow \underline{P}$, in the sense that every theory $\underline{P}(\underline{A}[\sigma])$ with $\sigma : \underline{R} \rightarrow \underline{A}$ is the same as the theory $\underline{P}_i(\underline{A}[\sigma'])$ with $\sigma' : \underline{R} + \underline{R}_i \rightarrow \underline{A}$ for some $i \geq 1$. (A theory of the transformation of Clear specifications is needed to discuss this matter in a more precise fashion; no such theory exists at present.) Now each case may be handled separately, using the normal definition of parameterised implementation:

$$\underline{R} + \underline{R}_1 \hookrightarrow \underline{P}_1 \longrightarrow \underline{R}' + \underline{R}'_1 \hookrightarrow \underline{P}'_1$$

$$\underline{R} + \underline{R}_n \hookrightarrow \underline{P}_n \longrightarrow \underline{R}' + \underline{R}'_n \hookrightarrow \underline{P}'_n$$

3. Examples

Sets can be implemented using sequences by representing a set S as a sequence containing the elements of S in any order without repetitions. Sets may be specified in hierarchical Clear as follows:

```
proc Set(X:Ident) =
  let Set0 =
    enrich X by
      'data' sorts set
             opns ∅ : set
                  singleton : element -> set
                  (_ U _) : set,set -> set
                  (_ is_in _) : element,set -> bool
             eqns ∅ U S = S
                  S U S = S
                  S U T = T U S
                  S U (T U V) = (S U T) U V
                  a is_in ∅ = false
                  a is_in singleton(b) = a==b
                  a is_in S U T = a is_in S or a is_in T enden in
  enrich Set0 by
         opns choose : set -> element
         eqns choose(singleton(a) U S) is_in (singleton(a) U S)
                                                = true    enden
```

This specification includes an operator choose which is defined (loosely) as selecting an arbitrary element from a non-empty set. The value of choose(∅) is left undefined — although the same notion of implementation should work for error theories and algebras, we prefer to avoid the issue of errors for now. Note that the membership operator is_in is included within the 'data' in contrast to the specification of sets in ordinary Clear in section I.1.2. This subtle change is necessary to avoid trivial models.

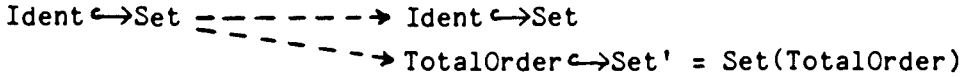
The concrete specification must include a definition of sequences as well as operators on sequences corresponding to all the operators in Set. We begin by defining everything except the choose operator:

```
proc Sequence(X:Triv) =
  enrich X + Bool by
    'data' sorts sequence
           opns empty : sequence
                unit : element -> sequence
                (_ . _) : sequence,sequence -> sequence
                head : sequence -> element
                tail : sequence -> sequence
           eqns empty.s = s
                s.empty = s
                s.(t.v) = (s.t).v
                head(unit(a).s) = a
                tail(unit(a).s) = s    enden

proc SequenceOpns(X:Ident) =
  enrich Sequence(X) by
    opns (_ is_in _) : element,sequence -> bool
         add : element,sequence -> sequence
         (_ U _) : sequence,sequence -> sequence
    eqns a is_in empty = false
         a is_in unit(b) = a==b
         a is_in s.t = a is_in s or a is_in t
         add(a,s) = s  if a is_in s
         add(a,s) = unit(a).s  if not(a is_in s)
         empty U s = s
         unit(a).t U s = add(a,t U s)    enden
```

The head and tail operators of Sequence and their defining equations are needed to avoid trivial models; they serve no other function in the specification.

Before dealing with the choose operator, we split Set into two cases:

```
meta TotalOrder =
  enrich Ident by
    opns (_ ≤ _) : element,element -> bool
    eqns a≤a = true
         a≤b and b≤a --> a==b = true
         a≤b and b≤c --> a≤c = true
         a≤b or b≤a = true    enden
```



These two cases may be handled separately. The choose operator can select the minimum element when the element type is totally ordered; otherwise we can leave the precise choice unspecified as

before.

```
proc SequenceAsSet(X:Ident) =
  enrich SequenceOpns(X) by
    opns choose : sequence -> element
    eqns choose(unit(a).t) is_in (unit(a).t) = true    enden


proc SequenceAsSet'(X:TotalOrder) =
  enrich SequenceOpns(X) by
    opns choose : sequence -> element
    eqns choose(unit(a)) = a
         choose(unit(a).unit(b).s) = choose(unit(a).s) if a≤b
                                   else choose(unit(b).s)    enden
```

Now Ident$\hookrightarrow$Set $\xrightarrow{\sigma}_{\mu}$ Ident$\hookrightarrow$SequenceAsSet and TotalOrder$\hookrightarrow$Set' $\xrightarrow{\sigma}_{\mu'}$ TotalOrder$\hookrightarrow$SequenceAsSet', where σ = [element$\mapsto$element, ==$\mapsto$==, set$\mapsto$sequence, $\emptyset\mapsto$empty, singleton$\mapsto$unit, U$\mapsto$U, is_in$\mapsto$is_in, choose$\mapsto$choose] (and everything in the signature of Bool maps to itself), and μ and μ' are the identity morphisms on Ident and TotalOrder respectively. Note that an incorrect implementation results if choose in SequenceAsSet is changed to select the first element; Set contains an equation

```
  choose(singleton(x) U singleton(y))
= choose(singleton(y) U singleton(x))
```

so the identify step would collapse the parameter sort (and consequently bool).

This example illustrates all of the features of our notion of implementation. Not all sequences are needed to represent sets -- sequences with repeated elements are not used. Each set is represented by many sequences, since the sequence representation of a set keeps track of the order in which elements were inserted. Set is split into two theories before implementation, and finally SequenceAsSet' is 'tighter' than Set' because the choose operator (select an element) is implemented by an operator which chooses the minimum element.

A nonparameterised example is obtained by applying Set or Set' and SequenceAsSet or SequenceAsSet' to an argument, for example:

Set(Nat[element is nat])$\xrightarrow{\sigma}$SequenceAsSet(Nat[element is nat])

where σ' is the same as σ above except that element$\mapsto$element is replaced by nat$\mapsto$nat.

Two additional examples:

- Lists can be implemented using arrays of (value,index) pairs, where the index points to the next value in the list (and where some distinguished index value denotes nil). There are many representations for the same list (the relative positions of cells in the array are irrelevant, for example) and circular structures are not needed to represent the value of any list.

- The specification of matrix inversion in the Introduction can be implemented by a specification of matrix inversion using the Gauss-Seidel method. Conversely, this specification can be implemented by the specification in the Introduction (enriched by some auxiliary functions).

The matrix inversion example shows that the expectation that A$\longrightarrow$B should imply that B is 'lower level' than A is not always justified. This is because the definition of implementation is concerned with classes of models rather than with the equations used to describe those classes. In this case both theories will have the same class of models except that the Gauss-Seidel method will probably require auxiliary operators.

4. Horizontal and vertical composition

Clear is a language for writing structured specifications, providing facilities for combining small theories in various ways to make large theories. These facilities allow a large specification to be built in a modular fashion from smaller bits. Following Goguen and Burstall [1980] the structure of such a specification shall be called horizontal structure.

Likewise, the implementation of a large specification is not done all at once; it is good programming practice to implement and test pieces of the specification separately and then construct a final system from the finished components. If the theories which make up a Clear specification are implemented separately, it should be possible to put together (horizontally compose) the implementations in the same way that the theories themselves are put together, yielding an implementation of the entire specification.

Although the problem of developing a program from a specification is simplified by dividing it into smaller units, the step from specification of a component to its implementation as a program is still often uncomfortably large. A way to conquer this is to break the development of a program into a series of consecutive refinement steps. That is, the specification is refined to a lower level specification, which is in turn refined to a still lower level specification, and so on until a program is obtained. Again following Goguen and Burstall [1980], this is called the vertical structure (of the development process). If a specification A is implemented by another specification B, and B is implemented by C, then these implementations should vertically compose to give an implementation of A by C. That is, the implementation relation should be transitive. Goguen and Burstall [1980] propose a system called CAT for the structured development of programs from specifications by composing implementations in both the horizontal and vertical dimensions. (Note: Horizontal and vertical compositions were originally defined on natural transformations. The general structure admitting two such compositions is called a 2-category [Kelly and Street 1974].)

The vertical composition of two implementations is not always an implementation. For example, consider the following theories:

```
const T = enrich Bool by
            opns extra : bool   enden

const T' = enrich Bool by
             opns extra : bool
             eqns extra = true   enden

const T" = theory 'data' sorts threevals
                         opns tt, ff, extra : threevals   endth
```

Now $T \longrightarrow T'$ and $T' \longrightarrow T''$ but $T \not\longrightarrow T''$ (consider the model of T" where tt≠ff≠extra). The theories must satisfy an extra condition.

Def: A theory $\underline{T}$ is reachably complete with respect to a theory $\underline{T}' \subseteq \underline{T}$ if for all constraints c of $\underline{T}'$, $\underline{T}$ is sufficiently complete with respect to opns($\underline{T}'$), constrained-sorts(c), constructors(c), and variables of unconstrained-sorts($\underline{T}'$).

In the example above T" is not reachably complete with respect to T because extra is not provably equal to either tt or ff.

Reachable completeness with respect to a theory T is sufficient to guarantee that the result of the operation restrict_T will always exist:

Restriction lemma: If a theory $\underline{T}$ is reachably complete with respect to $\sigma(\underline{T}') \subseteq \underline{T}$ then for every model M of $\underline{T}$ $\text{restrict}_{\underline{T}'}(M|^{\sigma}_{\text{sig}(\underline{T}')})$ exists.

Proof: We may assume for simplicity that $\underline{T}' \subseteq \underline{T}$ and σ is the inclusion; the following proof generalises to arbitrary $\underline{T}'$ and σ.

Let $\tilde{M}$ be the sig($\underline{T}'$)-subalgebra of $M|_{\text{sig}(\underline{T}')}$ which is finitely generated by opns($\underline{T}'$) and elements of unconstrained-sorts($\underline{T}'$) (i.e. every element of $\tilde{M}$ is the value of a term built from operators of $\underline{T}'$ and variables of unconstrained sorts of $\underline{T}'$, for some assignment of M values). We will show that $\tilde{M}$ satisfies the "no junk" condition for every constraint $c=\langle \underline{Tc} \hookrightarrow \underline{Tc}', \sigma' \rangle$ of $\underline{T}'$; $\tilde{M}$ is then clearly the largest such subalgebra.

Let a be an element of $\tilde{M}_{\text{target}} = \tilde{M}|^{\sigma'}_{\text{sig}(\underline{Tc}')}$. Then a is the value

of some term t built from opns($\underline{T}'$) and variables of unconstrained-sorts($\underline{T}'$) for some assignment of these variables. If a is not of a constrained sort of c then it trivially satisfies the "no junk" condition. Otherwise, the reachable completeness of $\underline{T}$ with respect to $\underline{T}'$ implies the existence of a term t' built from constructors(c) and variables of unconstrained-sorts($\underline{T}'$) such that $\underline{T}\vdash t=t'$.

Now, let $t_1,\ldots,t_k$ be the largest subterms of t' of σ'(sorts($\underline{Tc}$)) and consider the term t" containing variables $x_1,\ldots,x_k$ of $sort(t_1),\ldots,sort(t_k)$ such that $t' = t"[t_1/x_1,\ldots,t_k/x_k]$. Then t" does not contain any operator f:...->s with s$\notin$constrained-sorts(c). Thus (since opns(t")$\subseteq$opns(t')) all operators of t" are in σ'(opns($\underline{Tc}'$)).

Since $a=\varphi(t)=\psi(t')=\varphi(t"[\psi(x_1),\ldots,\psi(x_k)])$ for some assignments φ and ψ such that $\varphi(t_i)=\psi(x_i)$, a is the value of some sig($\underline{Tc}'$)-term with variables in sorts($\underline{Tc}$). Thus $\tilde{M}$ satisfies the "no junk" condition for c. □

We can use this lemma to prove that implementations can be vertically composed if the target of the composition is reachably complete with respect to the source.

Vertical composition theorem

1. [Reflexivity] $\underline{T}\xrightarrow{id}\underline{T}$ (the proof is obvious).

2. [Transitivity] If $\underline{T}\xrightarrow{\sigma}\underline{T}'$ and $\underline{T}'\xrightarrow{\sigma'}\underline{T}"$ and $\underline{T}"$ is reachably complete with respect to $\sigma.\sigma'(\underline{T})$, then $\underline{T}\xrightarrow{\sigma.\sigma'}\underline{T}"$.

Proof of transitivity: Let M" be a model of $\underline{T}"$ and consider $FRI_{\underline{T}}(M") =_{def} restrict_{\underline{T}}(M"|^{\sigma.\sigma'}_{sig(\underline{T})})/\equiv_{eqns(\underline{T})}$. The existence of $FRI_{\underline{T}}(M")$ follows from the restriction lemma. Because $\underline{T}\xrightarrow{\sigma}\underline{T}'$ and $\underline{T}'\xrightarrow{\sigma'}\underline{T}"$, $FRI_{\underline{T}}(FRI_{\underline{T}'}(M"))\vdash true\neq false$. Since there is a homomorphism from $FRI_{\underline{T}}(M")$ onto $FRI_{\underline{T}}(FRI_{\underline{T}'}(M"))$, $FRI_{\underline{T}}(M")\vdash true\neq false$ as well. Therefore $M"|^{\sigma.\sigma'}_{sig(\underline{T})}$ simulates $\underline{T}$. □

Corollary

1. [Reflexivity of parameterised implementations] $\underline{R} \hookrightarrow \underline{P} \xrightarrow[id]{id} \underline{R} \hookrightarrow \underline{P}$ (the proof is obvious).

2. [Transitivity of parameterised implementations] If $\underline{R} \hookrightarrow \underline{P} \xrightarrow[\mu]{\sigma} \underline{R}' \hookrightarrow \underline{P}'$ and $\underline{R}' \hookrightarrow \underline{P}' \xrightarrow[\mu']{\sigma'} \underline{R}'' \hookrightarrow \underline{P}''$ and $\underline{P}''$ is reachably complete with respect to $\sigma.\sigma'(\underline{P})$, then $\underline{R} \hookrightarrow \underline{P} \xrightarrow[\mu'.\mu]{\sigma.\sigma'} \underline{R}'' \hookrightarrow \underline{P}''$.

Proof of transitivity: Suppose $\rho:\underline{R} \rightarrow \underline{A}$ is a fitting morphism; then so is $\rho'':\underline{R}'' \rightarrow \underline{A} = \mu'.\mu.\rho$. Let M'' be a model of $\underline{P}''(\underline{A}[\rho''])$. Since $M''|_{sig(\underline{P}'')}$ is a model of $\underline{P}''$ and $\underline{P}''$ is reachably complete with respect to $\sigma.\sigma'(\underline{P})$, by the restriction lemma $FR_{\underline{P}}(M'') =_{def} restrict_{\underline{P}}(M''|^{\sigma.\sigma'}_{sig(\underline{P})})$ exists. Since $FR_{\underline{A}}(M'') = M''|_{sig(\underline{A})}$ and all theories are built using Clear, it follows that $FR_{\underline{P}(\underline{A}[\rho])}(M'')$ exists. By definition $\underline{P}(\underline{A}[\rho]) \xrightarrow{\hat{\sigma}} \underline{P}'(\underline{A}[\mu.\rho])$ and $\underline{P}'(\underline{A}[\mu.\rho]) \xrightarrow{\hat{\sigma}'} \underline{P}''(\underline{A}[\rho''])$ and so $FRI_{\underline{P}(\underline{A}[\rho])}(M'') \vdash true \neq false$ by the same argument as in the nonparameterised case. □

In the absence of constraints (as in the initial algebra [Goguen, Thatcher and Wagner 1978] and final algebra [Wand 1979] approaches), reachable completeness is guaranteed so this extra condition is unnecessary.

To prove that implementations of large theories can be built by arbitrary horizontal composition of small theories, it is necessary to prove that each of Clear's theory-building operations preserves implementations. We will concentrate here on the application of parameterised theories and the enrich operation. Extension of these results to the remaining operations should not be difficult.

For the apply operation our object is to prove the following property of implementations:

Horizontal composition property: If $\underline{R} \hookrightarrow \underline{P} \xrightarrow[\mu]{\sigma} \underline{R}' \hookrightarrow \underline{P}'$, $\underline{A} \xrightarrow{\sigma'} \underline{A}'$, and $\rho:\underline{R} \rightarrow \underline{A}$ is a theory morphism, then $\underline{P}(\underline{A}[\rho]) \xrightarrow{\sigma''} \underline{P}'(\underline{A}'[\mu.\rho.\sigma'])$, where σ'' is constructed from σ, σ', μ and ρ (see the horizontal composition theorem below for details).

But this is not true in general; in fact, $\underline{P}'(\underline{A}'[\mu.\rho.\sigma'])$ is not even always defined. Again, some extra conditions must be satisfied for the desired property to hold.

Def: Let $\underline{R} \hookrightarrow \underline{P}$ be a parameterised theory.

- $\underline{R} \hookrightarrow \underline{P}$ is called structurally complete if $\underline{P}$ is sufficiently complete with respect to the parameter $\underline{R}$ (i.e. with respect to opns($\underline{P}$), sorts($\underline{R}$), opns($\underline{R}$) and variables of unconstrained-sorts($\underline{R}$)), and if for all constraints c of $\underline{P}$, $\underline{P}$ is sufficiently complete with respect to c (i.e. with respect to opns($\underline{P}$), constrained-sorts(c), constructors(c), and variables of unconstrained-sorts($\underline{P}$)). A nonparameterised theory $\underline{A}$ is called structurally complete if $\emptyset \hookrightarrow \underline{A}$ is structurally complete.

- $\underline{R} \hookrightarrow \underline{P}$ is called parameter consistent if $\underline{P}$ is conservative with respect to $\underline{R}$.

- $\underline{R} \hookrightarrow \underline{P}$ is called persistent if it is both structurally complete and parameter consistent.

If $\underline{R}' \hookrightarrow \underline{P}'$ is persistent and reachably complete, and $\underline{A}'$ is a valid actual parameter of $\underline{R}' \hookrightarrow \underline{P}'$, then the horizontal composition property holds. The proof of this result relies on the following lemma:

Horizontal composition lemma: If $\underline{R} \hookrightarrow \underline{P}$ is persistent, $\rho:\underline{R} \rightarrow \underline{A}$ and $\rho.\sigma:\underline{R} \rightarrow \underline{A}'$ are theory morphisms and $\underline{A} \xrightarrow{\sigma} \underline{A}'$ then $\underline{P}(\underline{A}[\rho]) \xrightarrow{\tilde{\sigma}} \underline{P}(\underline{A}'[\rho.\sigma])$, where $\tilde{\sigma}|_{sig(\underline{P}(\underline{A}[\rho]))-sig(\underline{A})} = id$ and $\tilde{\sigma}|_{sig(\underline{A})} = \sigma$.

The proof of this lemma relies in turn on the following result:

Theorem [Wirsing and Broy 1981]: If $\underline{R} \hookrightarrow \underline{P}$ is persistent then any model of $\underline{R}$ can be extended to both an initial model and a terminal model of $\underline{P}$. Thus for every structurally complete and satisfiable theory $\underline{A}$ with $\rho:\underline{R} \rightarrow \underline{A}$, $\underline{P}(\underline{A}[\rho])$ has both initial and terminal models.

Proof of the lemma: Let $\underline{PA} =_{def} \underline{P}(\underline{A}[\rho])$ and $\underline{PA}' =_{def} \underline{P}(\underline{A}'[\rho.\sigma])$, and suppose M is a model of $\underline{PA}'$. We will show first that $FR_{\underline{PA}}(M) =_{def} restrict_{\underline{PA}}(M|_{sig(\underline{PA})})$ exists, and then that $FRI_{\underline{PA}}(M) =_{def} FR_{\underline{PA}}(M)/\equiv_{eqns(\underline{PA})} = FR_{\underline{PA}}(M)/\equiv_{eqns(\underline{A})}$ satisfies true≠false. Since

$FRI_{\underline{PA}}(M)$ must satisfy the equations and the "no junk" condition of the constraints of $\underline{PA}$, this implies that $FRI_{\underline{PA}}(M)$ is a model of $\underline{PA}$ and therefore that $\underline{PA}'$ implements $\underline{PA}$.

Let $\tilde{M}$ be the $sig(\underline{PA})$-subalgebra of $M|_{sig(\underline{PA})}$ which is finitely generated by $opns(\underline{PA})$, elements of M of $unconstrained\text{-}sorts(\underline{PA})$, and elements of $FR_{\underline{A}}(M)$. Since $\underline{R} \hookrightarrow \underline{P}$ is sufficiently complete (with respect to the parameter $\underline{R}$) $\tilde{M}|_{sig(\underline{A})} = FR_{\underline{A}}(M) = FR_{\underline{A}}(M|_{sig(\underline{A}')})$, which satisfies the "no junk" condition for every constraint of $\underline{A}$ since $\underline{A} \xrightarrow{\sigma} \underline{A}'$. The only remaining constraints of $\underline{PA}$ are on sorts of $\underline{P}$-$\underline{R}$, since $\underline{P}$ is built from $\underline{R}$ using Clear and ρ is a theory morphism. Suppose c is such a constraint. An argument analogous to the proof of the restriction lemma shows that (since $\underline{R} \hookrightarrow \underline{P}$ is sufficiently complete with respect to c) $\tilde{M}$ satisfies the "no junk" condition of c. Therefore $\tilde{M}$ satisfies the "no junk" condition for all constraints of $\underline{PA}$; it is clearly the largest such subalgebra of $M|_{sig(\underline{PA})}$ so $FR_{\underline{PA}}(M) = \tilde{M}$.

To show that $FRI_{\underline{PA}}(M) \not\models true \neq false$ we begin by introducing a constant c_a for every element a in $FR_{\underline{PA}}(M)^- =_{def} FR_{\underline{PA}}(M)|_{sig(\underline{A})}$. Call this new algebra $FR_{\underline{PA}}(M)^{-\dagger}$. Let $\underline{T}$ be the theory with the signature of $FR_{\underline{PA}}(M)^{-\dagger}$ (i.e. $sig(\underline{A})$ together with all the new nullary operators c_a) and the axioms $(FR_{\underline{PA}}(M)^{-\dagger})^*$ -- recall the operation * defined in section II.4. Since $FR_{\underline{PA}}(M)$ satisfies all the equations of $\underline{P}$ and all the constraints of $\underline{PA}$, $FR_{\underline{PA}}(M)^-$ satisfies all the equations and constraints of $\underline{R}$ (translated via ρ). Thus $\rho: \underline{R} \rightarrow \underline{T}$ is a theory morphism and $FR_{\underline{PA}}(M)$ (when appropriately extended) is a model of $\underline{P}(\underline{T}[\rho])$.

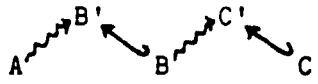
Now, $FRI_{\underline{A}}(M^-)$ (which is $FR_{\underline{PA}}(M)^-/_{\equiv_{eqns(\underline{A})}}$ by structural completeness of $\underline{R} \hookrightarrow \underline{P}$) is a model of $\underline{A}$ (since $\underline{A} \dashrightarrow \underline{A}'$) and $FRI_{\underline{A}}(M^-)$ (when appropriately extended) is also a model of $\underline{T}$. Moreover, since $\underline{R} \hookrightarrow \underline{P}$ is persistent, $FRI_{\underline{A}}(M^-)$ can be extended to some model S of $\underline{P}(\underline{T}[\rho])$. And since $\underline{T}$ is structurally complete and satisfiable, $\underline{P}(\underline{T}[\rho])$ possesses a terminal model Z satisfying $true \neq false$. There exist homomorphisms from S onto Z and from $FR_{\underline{PA}}(M)$ to Z because Z is terminal. Hence Z satisfies all equations of $\underline{A}$ (because of S) and all equations satisfied by $FR_{\underline{PA}}(M)$. Therefore there exists a homomorphism from $FRI_{\underline{PA}}(M)$ onto Z and $Z \not\models true \neq false$ implies $FRI_{\underline{PA}}(M) \not\models true \neq false$. □

Corollary (Horizontal composition for enrich): If $\underline{A} \overset{\sigma}{\rightsquigarrow} \underline{A}'$ and $sig(\underline{A}) \hookrightarrow$ enrich $sig(\underline{A})$ by <stuff> is persistent then enrich $\underline{A}$ by <stuff> $\overset{\tilde{\sigma}}{\rightsquigarrow}$ enrich $\underline{A}'$ by $\tilde{\sigma}$<stuff>, where $\tilde{\sigma}|_{sig(<stuff>)} = id$ and $\tilde{\sigma}|_{sig(\underline{A})} = \sigma$.

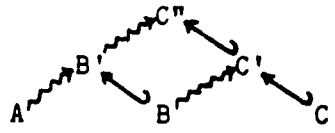
Proof: Consider the (persistent) parameterised theory $\underline{R} \hookrightarrow \underline{P}$ where $\underline{R}$ = <sig($\underline{A}$),∅> and $\underline{P}$ = enrich $\underline{R}$ by <stuff>. Since $id:\underline{R} \rightarrow \underline{A}$ and $id.\sigma:\underline{R} \rightarrow \underline{A}'$ are (trivially) theory morphisms, the horizontal composition lemma applies to give the desired result. □

A consequence of this corollary is that our vertical and horizontal composition theorems extend to more elaborate notions of implementation such as the one discussed by Ehrig, Kreowski and Padawitz [1980]. They would say that T is implemented by T' (which we will write $T \rightsquigarrow T'$) if there is a theory T" which is an enrichment of T' (written $T' \hookrightarrow T''$) such that $T \overset{id}{\rightsquigarrow} T'$ (in our sense). In pictorial form:

Then $A \rightsquigarrow B \rightsquigarrow C$ implies (under appropriate conditions) $A \rightsquigarrow C$, since if:



then by the corollary:



and then $A \rightsquigarrow C''$ by the vertical composition theorem.

We can now use the above lemma to prove the horizontal composition theorem.

Horizontal composition theorem: If $\underline{R}' \hookrightarrow \underline{P}'$ is persistent, $\underline{P}'$ is reachably complete with respect to $\sigma(\underline{P})$, $\underline{R} \hookrightarrow \underline{P}$ $\overset{\sigma}{\underset{\mu}{\rightarrow}}$ $\underline{R}' \hookrightarrow \underline{P}'$ and $\underline{A} \xrightarrow{\sigma'} \underline{A}'$, and $\rho:\underline{R} \rightarrow \underline{A}$ and $\rho':\underline{R}' \rightarrow \underline{A}'$ are theory morphisms where $\rho' = \mu.\rho.\sigma'$, then $\underline{P}(\underline{A}[\rho]) \xrightarrow{\hat{\sigma}.\tilde{\sigma}'} \underline{P}'(\underline{A}'[\rho'])$.

Proof: Let $\underline{PA} =_{def} \underline{P}(\underline{A}[\rho])$ and $\underline{PA}' =_{def} \underline{P}'(\underline{A}[\mu.\rho])$. From the reachable completeness of $\underline{P}'$ it follows that $\underline{P}'(\underline{A}'[\rho'])$ is reachably complete with respect to $\hat{\sigma}.\tilde{\sigma}'(\underline{PA})$ for all constraints of $\underline{P}$. Let c be a constraint of $\underline{A}$. Suppose f:...->s where $\rho(s) \in$ constrained-sorts(c) is an operator of $\underline{P}$-$\underline{R}$; then $s \in$ sorts($\underline{R}$). Because $\underline{R} \hookrightarrow \underline{P}$ is structurally complete, any sig($\underline{P}$)-term f(...) is provably equal to a sig($\underline{R}$)-term t'. Thus $\tilde{\rho}$(f(...)) is provably equal to a 'constrained' sig($\underline{A}$)-term ρ(t') (where $\tilde{\rho}:\underline{P} \rightarrow \underline{PA}$ is the extension of ρ). Therefore $\underline{P}'(\underline{A}'[\rho'])$ is reachably complete with respect to $\hat{\sigma}.\tilde{\sigma}'(\underline{PA})$ for c.

Suppose M is a model of $\underline{P}'(\underline{A}'[\rho'])$. By the restriction lemma, $FR_{\underline{PA}}(M)$ exists. According to the horizontal composition lemma, $\underline{PA}' \xrightarrow{\tilde{\sigma}'} \underline{P}'(\underline{A}'[\rho'])$. By definition, $\underline{PA} \xrightarrow{\hat{\sigma}} \underline{PA}'$. Therefore, $FRI_{\underline{PA}}(FRI_{\underline{PA}'}(M)) \sqsubseteq \text{true} \neq \text{false}$. Since there is a homomorphism from $FRI_{\underline{PA}}(M)$ onto $FRI_{\underline{PA}}(FRI_{\underline{PA}'}(M))$, $FRI_{\underline{PA}}(M) \sqsubseteq \text{true} \neq \text{false}$ as well. □

In [Sannella and Wirsing 1982] examples are given which demonstrate the necessity of all the conditions on this theorem. It is also shown there that if $\underline{R}=\underline{R}'$ (this is normally the case, as in all of our examples) then reachable completeness of $\underline{P}'$ with respect to $\sigma(\underline{P})$ is not needed.

The vertical and horizontal composition theorems give us freedom to build the implementation of a large specification from many small implementation steps. The correctness of all the small steps guarantees the correctness of the entire implementation, which in turn guarantees the correctness of the low-level 'program' with respect to the high-level specification. This provides a formal foundation for a methodology of programming by stepwise refinement. An analogue of CAT's 'double law' [Goguen and Burstall 1980] is a consequence of the vertical and horizontal composition theorems.

That is, given:

$$\underline{R} \hookrightarrow \underline{P} \longrightarrow \underline{R}' \hookrightarrow \underline{P}' \qquad \underline{A} \longrightarrow \underline{A}'$$
$$\underline{R}' \hookrightarrow \underline{P}' \longrightarrow \underline{R}'' \hookrightarrow \underline{P}'' \qquad \underline{A}' \longrightarrow \underline{A}''$$

(and appropriate fitting morphisms) we can apply the horizontal composition theorem to give:

1. $\underline{P}(\underline{A}) \longrightarrow \underline{P}'(\underline{A}')$ 2. $\underline{P}'(\underline{A}') \longrightarrow \underline{P}''(\underline{A}'')$

or else apply the vertical composition theorem (and its corollary) to give:

3. $\underline{R} \hookrightarrow \underline{P} \longrightarrow \underline{R}'' \hookrightarrow \underline{P}''$ 4. $\underline{A} \longrightarrow \underline{A}''$

Now we can either apply the vertical composition theorem to (1) and (2), or else apply the horizontal composition theorem to (3) and (4); either way we get the same implementation of $\underline{P}(\underline{A})$ by $\underline{P}''(\underline{A}'')$. This means that the order in which parts of an implementation are composed makes no difference, and that our notion of implementation is appropriate for use in CAT.

Our notions of simulation and implementation extend without modification to ordinary Clear (with data constraints rather than hierarchy constraints). The vertical and horizontal composition results then hold only under additional conditions.

Vertical composition theorem (with data): In Clear with data,

1. [Reflexivity] $\underline{T} \xrightarrow{id} \underline{T}$ (the proof is obvious as before).

2. [Transitivity] If $\underline{T} \xrightarrow{\sigma} \underline{T}'$, $\underline{T}' \xrightarrow{\sigma'} \underline{T}''$, all sorts in $\underline{T}$ are constrained and $\underline{T}''$ is reachably complete with respect to $\sigma.\sigma'(\underline{T})$, then $\underline{T} \xrightarrow{\sigma;\sigma'} \underline{T}''$.

Proof of transitivity: Let M be a model of $\underline{T}''$. As in the hierarchical case, $FR_{\underline{T}}(M)$ exists because of the reachable completeness of $\underline{T}''$. Let E be the set of all ground equations which hold in $\text{restrict}_{\underline{T}}(M)$, and define:

$\tilde{T} =_{def}$ enrich T by eqns E

$\tilde{T}' =_{def}$ enrich T' by eqns $\sigma(E)$

$\tilde{T}'' =_{def}$ enrich T'' by eqns $\sigma.\sigma'(E)$

$T \xrightarrow{\sigma} T' \xrightarrow{\sigma'} T''$ implies that $\tilde{T} \xrightarrow{\sigma} \tilde{T}' \xrightarrow{\sigma'} \tilde{T}''$. The reachable completeness of T'' ensures that for every ground sig(T)-term t there exists a 'constrained' term t' such that $E \vdash t=t'$. Thus $\tilde{T}$ is structurally complete, and since every sort of $\tilde{T}$ is constrained it has (up to isomorphism) only one model which is initial in the class of 'hierarchical' models of $\tilde{T}$ (i.e. in the class of algebras which are models of $\tilde{T}$ when the data constraints of $\tilde{T}$ are viewed as hierarchy constraints).

By the vertical composition theorem for hierarchical theories, $FRI_T(M) = FRI_{\tilde{T}}(M)$ is a hierarchical model of $\tilde{T}$. There is a homomorphism from $FRI_{\tilde{T}}(M)$ onto $FRI_{\tilde{T}}(FRI_{\tilde{T}'}(M))$. The initiality of $FRI_{\tilde{T}}(FRI_{\tilde{T}'}(M))$ implies the existence of a homomorphism in the opposite direction. Thus $FRI_{\tilde{T}}(M)$ is initial in the class of hierarchical models of $\tilde{T}$ so (equivalently) it is a model of $\tilde{T}$. Therefore it is a model of T. □

An example showing that constraints on all sorts of T are required for this theorem is given in [Sannella and Wirsing 1982].

Corollary: In Clear with data,

1. [Reflexivity of parameterised implementations] $R \hookrightarrow P \xrightarrow{id} R \hookrightarrow P$ (the proof is obvious as before).

2. [Transitivity of parameterised implementations] If $R \hookrightarrow P \xrightarrow{\sigma}_{\mu} R' \hookrightarrow P'$ and $R' \hookrightarrow P' \xrightarrow{\sigma'}_{\mu'} R'' \hookrightarrow P''$, all non-parameter sorts of $R \hookrightarrow P$ are constrained and P'' is reachably complete with respect to $\sigma.\sigma'(P)$ then $R \hookrightarrow P \xrightarrow{\sigma.\sigma'}_{\mu'.\mu} R'' \hookrightarrow P''$.

The proof of transitivity relies on a lemma.

Lemma: In Clear with data, if $R \hookrightarrow P \xrightarrow[\mu]{\sigma} R' \hookrightarrow P'$ and $R' \hookrightarrow P' \xrightarrow[\mu']{\sigma'} R'' \hookrightarrow P''$, all non-parameter sorts of $R \hookrightarrow P$ are constrained, P'' is reachably complete with respect to $\sigma.\sigma'(P)$ and $\rho:R \to A$ is a theory morphism where all sorts in A are constrained, then $P(A[\rho]) \xrightarrow{\hat{\sigma}.\hat{\sigma}'} P''(A[\mu'.\mu.\rho])$.

Proof of lemma: All sorts of $P(A[\rho])$ are constrained. Let M be a model of $P''(A[\mu'.\mu.\rho])$ and let ground(M^*) be the set of (constraints and) ground equations which hold in M. Then the theory $T =_{def}$ enrich $P''(A[\mu'.\mu.\rho])$ by eqns ground(M^*) is reachably complete with respect to $\hat{\sigma}.\hat{\sigma}(P(A[\rho]))$. M is a model of T and transitivity in the nonparameterised case implies that $FRI_{P(A[\rho])}(M)$ is a model of $P(A[\rho])$. □

Proof of transitivity: Suppose $\rho:R \to A$ is a fitting morphism, and let M be a model of A. Let $M^\dagger$ be the algebra obtained by introducing a constant c_a into M for every element a of M. Let T be the theory with the signature of $M^\dagger$ and the axioms ground($M^{\dagger *}$); T will include a data constraint for every sort of A. Then $\rho:R \to T$ is a theory morphism. Since every sort of T is constrained, the lemma implies that for every model $\tilde{M}$ of $P''(T[\mu'.\mu.\rho])$, $\tilde{M}|_{sig(P(T[\rho]))}$ simulates $P(T[\rho])$. Therefore $\tilde{M}|_{sig(P(A[\rho]))}$ simulates $P(A[\rho])$. Every model of $P''(A[\mu'.\mu.\rho])$ (suitably extended) is a model of $P''(T[\mu'.\mu.\rho])$ for some such T, so this implies the desired result. □

Def: A data theory T is hierarchical submodel consistent if for every model M of T and every hierarchical submodel M^- of M (i.e. every submodel of M satisfying the constraints of T when viewed as hierarchy constraints), M^- satisfies the data constraints of T.

Horizontal composition lemma (with data): In Clear with data, if $R \hookrightarrow P$ is persistent and P is hierarchical submodel consistent, $\rho:R \to A$ and $\rho.\sigma:R \to A'$ are theory morphisms and $A \xrightarrow{\sigma} A'$ then $P(A[\rho]) \xrightarrow{\tilde{\sigma}} P(A'[\rho.\sigma])$, where $\tilde{\sigma}|_{sig(P(A[\rho]))-sig(A)} = id$ and $\tilde{\sigma}|_{sig(A)} = \sigma$.

Proof: Let M be a model of $P(A'[\rho.\sigma])$, and let $PA =_{def} P(A[\rho])$. The horizontal composition lemma for hierarchical Clear says that

$FRI_{\underline{PA}}(M)$ exists and is a model of $\underline{PA}$ when the data constraints are viewed as hierarchy constraints. It remains to show that $FRI_{\underline{PA}}(M)$ satisfies the "no confusion" condition for every data constraint c of $\underline{PA}$.

Because $\underline{R} \hookrightarrow \underline{P}$ is persistent, $M_{sig(\underline{A}')} = M^{-}$ where M^{-} is a model of $\underline{A}'$ in which all elements are finitely generated from operators and elements of M of sorts unconstrained in $\underline{A}'$. Thus $FR_{\underline{PA}}(M)|_{sig(\underline{A})} = FR_{\underline{A}}(M^{-})$. Once more, persistency ensures that $FRI_{\underline{PA}}(M)|_{sig(\underline{A})} = FRI_{\underline{A}}(M^{-})$. Since $\underline{A} \xrightarrow{\sigma} \underline{A}'$, $FRI_{\underline{A}}(M^{-})$ is a model of $\underline{A}$ and hence $FRI_{\underline{PA}}(M)$ satisfies the data constraints of $\underline{A}$. Since ρ is a theory morphism it also satisfies the data constraints of $\underline{R}$.

$FR_{\underline{PA}}(M)$ satisfies all the equations and constraints (when viewed as hierarchy constraints) of $\underline{P}$. Thus $FR_{\underline{PA}}(M)|_{sig(\underline{P})}$ is a hierarchical model of $\underline{P}$ and moreover is a submodel of M. Hierarchical submodel consistency of $\underline{P}$ guarantees that $FR_{\underline{PA}}(M)|_{sig(\underline{P})}$ and thus $FR_{\underline{PA}}(M)$ satisfies the "no confusion" condition for every constraint of $\underline{P}$. Then $FRI_{\underline{PA}}(M)$ (which is $FR_{\underline{PA}}(M)/\equiv_{eqns(\underline{A})}$) satisfies the "no confusion" condition for the constraints of $\underline{P}$ as well. □

Corollary (Horizontal composition for enrich with data): If $\underline{A} \xrightarrow{\sigma} \underline{A}'$ and $sig(\underline{A}) \hookrightarrow \underline{P}$=enrich $sig(\underline{A})$ by <stuff> is persistent and $\underline{P}$ is hierarchical submodel consistent then enrich $\underline{A}$ by <stuff> $\xrightarrow{\tilde{\sigma}}$ enrich $\underline{A}'$ by $\tilde{\sigma}$<stuff>, where $\tilde{\sigma}|_{sig(<stuff>)} = id$ and $\tilde{\sigma}|_{sig(\underline{A})} = \sigma$.

Proof: As before, applying the horizontal composition lemma to the parameterised theory $<sig(\underline{A}),\emptyset> \hookrightarrow \underline{P}$. □

Horizontal composition theorem (with data): In Clear with data, if $\underline{R}' \hookrightarrow \underline{P}'$ is persistent and $\underline{P}'$ is hierarchical submodel consistent, $\underline{P}'$ is reachably complete with respect to $\sigma(\underline{P})$, all nonparameter sorts of $\underline{R} \hookrightarrow \underline{P}$ are constrained, $\underline{R} \hookrightarrow \underline{P} \xrightarrow[\mu]{\sigma} \underline{R}' \hookrightarrow \underline{P}'$ and $\underline{A} \xrightarrow{\sigma'} \underline{A}'$ where all sorts of $\underline{A}$ are constrained, and $\rho: \underline{R} \to \underline{A}$ and $\rho': \underline{R}' \to \underline{A}'$ are theory morphisms where $\rho' = \mu.\rho.\sigma'$, then $\underline{P}(\underline{A}[\rho]) \xrightarrow{\hat{\sigma}.\tilde{\sigma}'} \underline{P}'(\underline{A}'[\rho'])$.

Proof: Let M be a model of $\underline{P}'(\underline{A}'[\rho'])$, and let $\underline{PA} =_{def} \underline{P}(\underline{A}[\rho])$ and $\underline{PA}' =_{def} \underline{P}'(\underline{A}[\mu.\rho])$. The horizontal composition theorem for hierarchical Clear says that $FRI_{\underline{PA}}(M)$ exists and is a model of $\underline{PA}$

when the data constraints are viewed as hierarchy constraints. It remains to show that $FRI_{\underline{PA}}(M)$ satisfies the "no confusion" condition for every data constraint c of $\underline{PA}$.

By the horizontal composition lemma, $\underline{PA}' \xrightarrow{\tilde{\sigma}'} \underline{P}'(\underline{A}'[\rho'])$ and by definition, $\underline{PA} \xrightarrow{\hat{\sigma}} \underline{PA}'$. Thus $\tilde{M} =_{def} FRI_{\underline{PA}}(FRI_{\underline{PA}'}(M))$ is a model of $\underline{PA}$ (satisfying the data constraints of $\underline{PA}$). Since $FR_{\underline{PA}}(FR_{\underline{PA}'}(M)) = FR_{\underline{PA}}(M)$, there is a homomorphism from $FRI_{\underline{PA}}(M)$ onto $\tilde{M}$. Let constr($\underline{PA}$) denote the theory $\underline{PA}$ with non-constructors omitted. Since $\tilde{M}$ satisfies the data constraints of $\underline{PA}$, $\tilde{M}^- =_{def} \tilde{M}|_{sig(constr(\underline{PA}))}$ is an initial model of constr($\underline{PA}$). $FRI_{\underline{PA}}(M)^- =_{def} FRI_{\underline{PA}}(M)|_{sig(constr(\underline{PA}))}$ is also a model of constr($\underline{PA}$) and there is a homomorphism from $FRI_{\underline{PA}}(M)^-$ onto $\tilde{M}^-$. On the other hand, the initiality of $\tilde{M}^-$ implies the existence of a homomorphism in the opposite direction. Hence $FRI_{\underline{PA}}(M)^-$ and $\tilde{M}^-$ are isomorphic, and $FRI_{\underline{PA}}(M)^-$ satisfies the data constraints of $\underline{PA}$, which implies that $FRI_{\underline{PA}}(M)$ satisfies the "no confusion" condition of the data constraints. □

An example is given in [Sannella and Wirsing 1982] which shows the necessity of the condition that all nonparameter sorts of $\underline{R} \hookleftarrow \underline{P}$ be constrained. It is also shown there that if $\underline{R}=\underline{R}'$ then this condition can be dropped along with reachable completeness of $\underline{P}'$ with respect to $\sigma(\underline{P})$ and the condition that all sorts of $\underline{A}$ be constrained.

The vertical and horizontal composition results for theories with data constraints are encouraging because ordinary Clear is easier to use than our 'hierarchical' variant. However, the hierarchical submodel consistency condition on the horizontal composition theorem is rather strong and it may be that it is too restrictive to be of practical use. Here is an example which shows that the proposition (and therefore the theorem) does not hold without the hierarchical submodel consistency condition:

```
meta Natlike =
  enrich Bool by
    sorts nat
    opns 0 : nat
         succ : nat -> nat    enden

proc P(X:Natlike) =
  enrich X by
    data sorts s
         opns a, b : s
              f : nat -> s
         eqns f(0) = a
              f(succ(x)) = b    enden

const A = Nat as usual but with only the operators 0 and succ

const A' =
  enrich Bool by
    data sorts nat'
         opns -1, 0 : nat'
              succ : nat' -> nat'
         eqns succ(-1) = 0    enden
```

Now A and A' are both valid actual parameters of Natlike$\hookrightarrow$P, and A$\longrightarrow$A' (where -1 is an unused value). But P(A)$\not\longrightarrow$P(A') (since P(A)$\models$a$\neq$b and P(A')$\models$a=b). The problem is that P is not hierarchical submodel consistent. Consider the following model M of P:

M_{nat} = {-1,0,1,2,...}
M_s = {a}
succ defined on M_{nat} in the usual way

(with the usual interpretation of Bool). Now suppose we remove -1 from M_{nat} to give an algebra M^-:

M^-_{nat} = {0,1,2,...}
M^-_s = {a}
succ as before

M^- is a hierarchical submodel of M but it does not satisfy the "no confusion" condition of the data constraint on the sort s, and therefore P is not hierarchical submodel consistent. There may be some weaker condition than hierarchical submodel consistency which

is sufficient to guarantee that implementations of data theories can be horizontally composed, but we have so far been unable to discover any such condition.

CONCLUSION

In the Introduction we described the wide variety of roles which specifications play in the development of every program. A specification of one sort or another is necessary to describe the task which the program is to perform, for communication between designers and programmers, for checking or proving the correctness of the resulting program, and for documentation. Of course, this is a very loose use of the word "specification" which includes everything from the vague ideas in a programmer's head to a precise description written in a formal language.

We argued that formal specifications are highly desirable because all informal specifications are to some degree imprecise, and the cost of ambiguity can be immense. It is not enough to write specifications in a language with a formally-defined syntax; this gives only a dangerous illusion of precision. It is essential that the specification language have a complete formal semantics. Only then can we be confident that our specifications have a precise and unambiguous meaning — the exact meaning of any specification can be determined mechanically by consulting the semantics.

Burstall and Goguen [1980] were the first to give a complete formal semantics of a specification language. They define the meaning of Clear's theory-building operations using the language of category theory, and then supply a denotational semantics of the language as a whole by building upon these definitions. Chapter V describes the semantics and a HOPE program which implements it. Besides being an experiment in 'categorical programming' as practiced by Burstall [1980] and Rydeheard [1981], the program exposed several minor errors and one rather serious error in [Burstall and Goguen 1980]. The semantics given in chapter V is a corrected version of the original semantics. The serious error was a failure to distinguish between theories and <u>metatheories</u> (necessary for supplying metasorts in parameterised theories); the rather subtle difference is discussed in section III.3. Unfortunately the program is too slow to be of much practical use.

A different but equivalent semantics for Clear is given in chapter III. This uses straightforward set-theoretic constructions to define the semantics of the theory-building operations; the denotational semantics built upon these definitions is virtually the same as in chapter V. The simplicity of the constructions depends on the use of tags to distinguish different sorts and operators which have the same name but originate in different theories. Both versions of the semantics are prolific -- two applications of the same parameterised theory to the same actual parameter (using the same fitting morphism) give two different copies of the same theory. Section III.5 describes how the set-theoretic semantics can be altered to remove this undesirable characteristic.

Why do we need two versions of the semantics? Is this not too much of a good thing? The category-theoretic semantics was developed at the same time as Clear was being designed. This had an altogether positive effect on the resulting language, as predicted by Ashcroft and Wadge [1982]; a desire to give Clear an elegant category-theoretic semantics led Burstall and Goguen to reject certain features and embrace others. The idea of 'parameterising' by an institution came from the realisation that the semantics of the theory-building operations relied only on the existence of colimits in the category of signatures. The language of category theory is perfect for expressing this kind of flexibility. The set-theoretic semantics has the advantage of being down-to-earth and constructive and therefore more useful for practical applications. But without the motivation provided by the category-theoretic semantics, the constructions of chapter III may seem mysterious and complicated. The set-theoretic semantics does not seem to readily generalise to an arbitrary institution, but in section III.6 we show that it can be easily adapted to deal with all institutions which have so far been proposed.

Winograd [1979] has argued convincingly for the need to force specifications into the foreground of the program development process and code into the background, in contrast to present-day programming practice. He makes the point that programming nowadays is concerned more with the integration of existing modules into

larger systems and the modification of existing programs than with the creation of new programs from scratch. In such cases a high-level specification of a module is far more important than the sequence of instructions which actually does the job. He suggests that the organisation and manipulation of these specifications should be regarded as a programmer's primary task. We agree wholeheartedly with his proposals. But these ideas are not yet practical because formal specifications are unfortunately rather difficult (or at least tedious) to construct. Although formal specifications have the advantage of precision, they are harder to understand than informal specifications and it is difficult to be sure that a formal specification is a correct description of the intended idea or behaviour.

There are two ways to attack this problem. The first is to develop an expressive and flexible specification language with a solid mathematical basis, but which does not require a great deal of mathematical sophistication to understand and use. Although addition of ad hoc features is never desirable, it is important that the language should not force specifications into an unnatural form for reasons of theoretical elegance. With a carefully-designed language users can worry about describing their problems without struggling with the language. The specification language may even aid users in expressing and thinking about their problems by encouraging them to construct specifications in a certain systematic way. Clear is a first attempt toward such a language -- the facilities it provides for structuring specifications in particular seem to be a great asset. But in many ways Clear is clumsy. ORDINARY [Goguen and Burstall 1980a] seems to be continuing in the right direction by retaining Clear's structuring facilities and institutional approach but emphasising useability.

The other approach to the problem is to develop automated aids to help us write, understand and manipulate specifications. Chapter IV discusses an implementation in HOPE of the set-theoretic semantics of chapter III, along with some examples of specifications which have been processed. As well as helping expose bugs in early versions of the semantics, this has shown itself to be invaluable in

checking specifications for syntax and type errors. It is surprisingly difficult to write even a small specification without making some kind of silly mistake. Since the semantics does not assign any meaning to a syntactically or semantically ill-formed specification it is imperative to detect such errors. The implementation could also serve as a front end to any system which requires specifications as input (such as a program verification or development system). A helpful addition would be to add a check for the persistency of enrichments, but this is a difficult problem which is undecidable in general. On the other hand, it would be easy to add a check for void sorts. The program described in chapter IV is presently rather slow and lacks a really good user interface, but these faults could easily be cured by a careful reimplementation in some lower-level language with more attention to error reporting and recovery.

A theorem prover is a useful tool for exploring the meaning of a specification, and is a necessary basis for building almost any system making serious use of specifications. In fact, the Clear implementation needs a theorem prover to check that specifications are semantically well-formed. In chapter VI a semi-automatic theorem prover for Clear built on top of the Edinburgh LCF system [Gordon, Milner and Wadsworth 1979] is described. It is able to prove many theorems automatically, exploiting the structure of Clear specifications to restrict the information available to that which is relevant to the theorem at hand. If the built-in strategy fails the user is free to attempt to prove the theorem using the high-level primitives (LCF tactics) and inference rules provided; our use of the LCF proof methodology guarantees that only valid theorems can be proved. Our goal was not to produce a powerful theorem prover full of clever heuristics, but to provide a set of tools sufficient for users to conduct proofs interactively and to explore some of the possibilities for automatic proof, with particular emphasis on finding evidence for our suspicion that the structure of Clear specifications can aid both interactive and automatic proof. A more powerful equational deduction component which uses state-of-the-art methods would improve the performance of the system substantially. Another area for improvement is the user interface, which is at

present rather primitive.

Chapter VII lays a foundation for the use of Clear in program development. A formal notion of the implementation of a theory by a lower-level theory is given which seems to agree with our intuitive ideas built on programming experience. This notion extends to give a definition of the implementation of parameterised theories. We prove that the implementation relation is transitive under certain conditions, and that separate implementations of a parameterised theory P and an actual parameter theory A can be combined to give an implementation of the application P(A), again provided that the theories are 'well-behaved'. These two results (together with an analogous result for each of the remaining theory-building operations of Clear — we only considered the apply operation) mean that large high-level specifications can be refined in a gradual and modular fashion to low-level HOPE-style 'programs', where the correctness of all the small individual refinements guarantees the correctness of the final program. A question not addressed was how to prove that a refinement is indeed a correct implementation according to our model-theoretic definition. This seems to be a difficult problem; Martin Wirsing and I have tried to produce a set of conditions sufficient to guarantee correctness of implementations, but so far we have had only limited success.

An ambitious project would be to integrate all of this work (together with efforts like OBJ [Goguen and Tardo 1979] and DAISTS [Gannon, McMullin and Hamlet 1981]) into a system for the specification, verification and systematic development of programs. The main barriers to such a system at present seem to be the lack of a means of proving the correctness of refinement steps, and the limitations of automatic theorem-proving technology. An important problem to which we have not yet devoted much attention is the construction of a comprehensive library of basic specifications which can be used to build large specifications without starting from scratch; the library in appendix 2 is just a feeble beginning. A great deal of work must also be done to develop a specification language which permits greater ease of expression than Clear, and on other problems of user engineering.

It is almost certain that a systematic approach to program development such as we have described will never be easier than the 'quick and dirty' approach. But in the long run the initial high cost of carefully developing a program should be balanced by the guaranteed correctness of the result and the relative ease of maintenance and later modification.

REFERENCES

Abrial, J.R., Schuman, S.A. and Meyer, B. (1979) Specification language Z. Massachusetts Computer Associates Inc., Boston, Massachusetts.

Aho, A.V. and Ullman, J.D. (1977) Principles of Compiler Design. Addison-Wesley.

Arbib, M.A. and Manes, E.G. (1975) Arrows, Structures and Functors. Academic Press.

Ashcroft, E.A. and Wadge, W.W. (1982) $\mathcal{R}$ for semantics. TOPLAS 4, 2.

Aubin, R. (1977) Strategies for mechanizing structural induction. Proc. 5th Intl. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts, pp. 363-369.

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM 21, 8 pp. 613-641.

Bauer, F.L. et al (the CIP Language Group) (1981) Report on a wide spectrum language for program specification and development (tentative version). Report TUM-I8104, Technische Univ. München.

Bekič, H., Bjørner, D., Henhapl, W., Jones, C.B. and Lucas, P. (1974) A formal definition of a PL/I subset. IBM Vienna Technical Report TR25.139.

Bergstra, J.A., Broy, M., Tucker, J.V. and Wirsing, M. (1981) On the power of algebraic specifications. Proc. 10th Intl. Symp. on Mathematical Foundations of Computer Science, Strbske Pleso, Czechoslovakia. Springer Lecture Notes in Computer Science, Vol. 118, pp. 193-204.

Birkhoff, G. (1935) On the structure of abstract algebras. Proc. of the Cambridge Philosophical Society 31, pp. 433-454.

Birkhoff, G. (1948) Lattice Theory. American Mathematical Soc. Colloq. Publications, Vol. 25, New York.

Boyer, R.S. and Moore, J.S. (1978) A formal semantics for the SRI hierarchical program design methodology. Technical report, SRI International.

Boyer, R.S. and Moore, J.S. (1979) A Computational Logic. Academic Press.

Broy, M., Dosch, W., Partsch, H., Pepper, P. and Wirsing, M. (1979) Existential quantifiers in abstract data types. Proc. 6th Intl. Colloq. on Automata, Languages and Programming. Springer Lecture Notes in Computer Science, Vol. 71, pp. 73-87.

Broy, M., Möller, B., Pepper, P. and Wirsing, M. (1980) A model-independent approach to implementations of abstract data types. Proc. of the Symp. on Algorithmic Logic and the Programming Language LOGLAN, Poznan, Poland. Springer Lecture Notes in Computer Science (to appear).

Burge, W.H. (1975) Recursive Programming Techniques. Addison-Wesley.

Burstall, R.M. (1977) Design considerations for a functional programming language. Infotech State of the Art Conference: The Software Revolution, Copenhagen.

Burstall, R.M. (1980) Electronic category theory. Proc. 9th Intl. Symp. on Mathematical Foundations of Computer Science, Rydzyna, Poland. Springer Lecture Notes in Computer Science, Vol. 88, pp. 22-39.

Burstall, R.M. (1980a) Proving inequalities. Unpublished notes.

Burstall, R.M. and Darlington, J. (1977) A transformation system for developing recursive programs. JACM 24, 1 pp. 44-67.

Burstall, R.M. and Goguen, J.A. (1977) Putting theories together to make specifications. Proc. 5th Intl. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts, pp. 1045-1058.

Burstall, R.M. and Goguen, J.A. (1980) The semantics of Clear, a specification language. Proc. of Advanced Course on Abstract Software Specifications, Copenhagen. Springer Lecture Notes in Computer Science, Vol. 86, pp. 292-332.

Burstall, R.M. and Goguen, J.A. (1981) An informal introduction to specifications using Clear. The Correctness Problem in Computer Science (R.S. Boyer and J.S. Moore, eds.), Academic Press, pp. 185-213.

Burstall, R.M., MacQueen, D.B. and Sannella, D.T. (1980) HOPE: an experimental applicative language. Proc. 1980 LISP Conference, Stanford, California, pp. 136-143; also Report CSR-62-80 (Revised version, Feb. 1981), Dept. of Computer Science, Univ. of Edinburgh.

Damas, L. and Milner, R. (1982) Principal type-schemes for functional programs. Proc. 9th ACM Symp. on Principles of Programming Languages, Albuquerque, New Mexico.

Darlington, J. and Reeve, M. (1981) ALICE: a multi-processor reduction machine for the parallel evaluation of applicative languages. Proc. ACM/MIT Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire.

Dijkstra, E.W. (1972) Notes on structured programming. Notes on Structured Programming (Dahl O.-J., Dijkstra, E.W. and Hoare, C.A.R.), Academic Press, pp. 1-82.

Dijkstra, E.W. (1980) Some beautiful arguments using mathematical induction. Acta Informatica 13 pp. 1-8.

Dybjer, P. (1981) Higher order continuous theories and their algebras. Unpublished draft, Dept. of Computer Science, Univ. of Edinburgh.

Ehrich, H.-D. (1981) On realization and implementation. Proc. 10th Intl. Symp. on Mathematical Foundations of Computer Science, Strbske Pleso, Czechoslovakia. Springer Lecture Notes in Computer Science, Vol. 118.

Ehrich, H.-D. (1982) On the theory of specification, implementation, and parametrization of abstract data types. JACM 29, 1 pp. 206-227.

Ehrich, H.-D. and Lohberger, V.G. (1978) Parametric specification of abstract data types, parameter substitution and graph replacements. Proc. of Workshop on Graphentheoretische Konzepte in der Informatik, Applied Computer Science, Carl Hanser Verlag.

Ehrig, H. (1981) Algebraic theory of parameterized specifications with requirements. Proc. 6th CAAP, Genova, Italy.

Ehrig, H. and Fey, W. (1981) Methodology for the specification of software systems: from formal requirements to algebraic design specifications. Proc. GI 81.

Ehrig, H. and Kreowski, H.-J. (1982) Parameter passing commutes with implementation of parameterized data types. Proc. 9th Intl. Colloq. on Automata, Languages and Programming, Aarhus, Denmark. Springer Lecture Notes in Computer Science (to appear).

Ehrig, H., Kreowski, H.-J. and Padawitz, P. (1980) Algebraic implementation of abstract data types: concept, syntax, semantics and correctness. Proc. 7th Intl. Colloq. on Automata, Languages and Programming, Noordwijkerhout, Netherlands. Springer Lecture Notes in Computer Science, Vol. 85, pp. 142-156.

Ehrig, H., Kreowski, H.-J., Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1980) Parameterized data types in algebraic specification languages (short version). Proc. 7th Intl. Colloq. on Automata, Languages and Programming, Noordwijkerhout, Netherlands. Springer Lecture Notes in Computer Science, Vol. 85.

Feather, M.S. (1982) A system for assisting program transformation. TOPLAS 4,1 1-20.

Gannon, J., McMullin, P. and Hamlet, R. (1981) Data-abstraction implementation, specification, and testing. TOPLAS 3,3 pp. 211-223.

Ganzinger, H. (1980) Parameterized specifications: parameter passing and implementation. TOPLAS (to appear).

Goguen, J.A. (1978) Abstract errors for abstract data types. Proc. IFIP Working Conf. on the Formal Description of Programming Concepts, New Brunswick, New Jersey.

Goguen, J.A. (1978a) Order sorted algebras: exceptions and error sorts, coercions and overloaded operators. Semantics and Theory of Computation Report No. 14, Computer Science Dept., UCLA.

Goguen, J.A. (1980) How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type implementation. Proc. 5th Conf. on Automated Deduction, Les Arcs, France. Springer Lecture Notes in Computer Science, Vol. 87.

Goguen, J.A. (1981) Two ORDINARY specifications. Technical report CSL-128, Computer Science Laboratory, SRI International.

Goguen, J.A. and Burstall, R.M. (1978) Some fundamental properties of algebraic theories: a tool for semantics of computation. Report 53, Dept. of Artificial Intelligence; to appear in Theoretical Computer Science.

Goguen, J.A. and Burstall, R.M. (1980) CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, Computer Science Laboratory, SRI International.

Goguen, J.A. and Burstall, R.M. (1980a) An ORDINARY design. Unpublished draft, Computer Science Laboratory, SRI International.

Goguen, J.A. and Meseguer, J. (1981) Completeness of many-sorted equational logic. SIGPLAN Notices 16, 7 pp. 24-32.

Goguen, J.A. and Tardo, J.J (1979) An introduction to OBJ: a language for writing and testing formal algebraic program specifications. Proc. of Conf. on Specification of Reliable Software, Cambridge, Massachusetts.

Goguen, J.A., Thatcher, J.W. and Wagner, E.G. (1978) An initial algebra approach to the specification, correctness, and implementation of abstract data types. Current Trends in Programming Methodology, Vol. 4: Data Structuring (R.T. Yeh, ed.), Prentice-Hall, pp. 80-149.

Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1973) A junction between computer science and category theory I: basic definitions and examples, part 1. IBM Research Report RC4526.

Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1977) Initial algebra semantics and continuous algebras. JACM 24, 1 pp. 68-95.

Gordon, M.J., Milner, A.J.R. and Wadsworth, C.P. (1979) Edinburgh LCF. Springer Lecture Notes in Computer Science, Vol. 78.

Grätzer, G. (1979) Universal Algebra (2nd edition), Springer.

Guttag, J.V. and Horning, J.J. (1978) The algebraic specification of abstract data types. Acta Informatica 10 pp. 27-52.

Guttag, J.V. and Horning, J.J. (1980) Formal specification as a design tool. Proc. 7th ACM Symp. on Principles of Programming Languages, Las Vegas.

Guttag, J.V., Horowitz, E. and Musser, D.R. (1978) Abstract data types and software validation. CACM 21, 12 pp. 1048-1064.

Henderson, P. and Snowdon, R. (1972) An experiment in structured programming. BIT 12 pp. 38-53.

Honda, M. and Nakajima, R. (1979) Interactive theorem proving on hierarchically and modularly structured sets of very many axioms. Proc. 6th Intl. Joint Conf. on Artificial Intelligence, Tokyo, pp. 400-402.

Hopcroft, J.E. and Ullman, J.D. (1979) Introduction to Automata Theory, Languages, and Computation. Addison-Wesley.

Huet, G. and Hullot, J.-M. (1980) Proofs by induction in equational theories with constructors. Rapport de Recherche 28, INRIA.

Hupbach, U.L. (1980) Abstract implementation of abstract data types. Proc. 9th Intl. Symp. on Mathematical Foundations of Computer Science, Rydzyna, Poland. Springer Lecture Notes in Computer Science, Vol. 88, pp. 291-304.

Hupbach, U.L. (1981) Abstract implementation and parameter substitution. Proc. 3rd Hungarian Computer Science Conference, Budapest.

Hupbach, U.L., Kaphengst, H. and Reichel, H. (1980) Initial algebraic specification of data types, parameterized data types, and algorithms. VEB Robotron, Zentrum für Forschung und Technik, Dresden.

IFIP WG 2.1 (1979) [Specification examples]. Document WG 2.1 334 (Bru-2), distributed prior to December 1979 IFIP WG 2.1 meeting in Brussels.

Iverson, K. (1962) A Programming Language. John Wiley and Sons.

Jenks, R.D. (1974) The SCRATCHPAD language. Proc. Symp. on Very High Level Languages.

Jones, C.B. (1978) The meta-language: a reference manual. The Vienna Development Method: The Meta-language (D. Bjørner and C.B. Jones, eds.). Springer Lecture Notes in Computer Science, Vol. 61, pp. 218-277.

Kaphengst, H. and Reichel, H. (1971) Algebraische Algorithmentheorie. VEB Robotron, Zentrum für Forschung und Technik, Dresden.

Kelly, G.M. and Street, R. (1974) Review of the elements of 2-categories. Category Seminar (G.M. Kelly, ed.), Springer Lecture Notes in Mathematics, Vol. 420, pp. 75-103.

Knuth, D.E. (1973) The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley.

Knuth, D.E. and Bendix, P.B. (1970) Simple word problems in universal algebras. Computational Problems in Abstract Algebra (J. Leech, ed.), Pergammon Press, pp. 263-297.

Landin, P.J. (1966) The next 700 programming languages. CACM 9, 3 pp. 157-166.

Lawvere, F.W. (1963) Functorial semantics of algebraic theories. Proc. Nat. Acad. Sci. USA 50, pp. 869-872.

Lehmann, D.J. and Smyth, M.B. (1981) Algebraic specification of data types: a synthetic approach. Mathematical Systems Theory 14, pp. 97-139.

Levitt, K.N., Robinson, L. and Silverberg, B. (1979) HDM handbook Vols. I, II, III. SRI International.

Levy, M.R. (1980) Specifying data types with variables and referencing. Report DCS-5-IR, Dept. of Computer Science, University of Victoria.

Liskov, B.H. and Berzins, V. (1977) An appraisal of program specifications. MIT Computation Structures Group Memo 141-1.

Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. (1977) Abstraction mechanisms in CLU. CACM 20, 8 pp. 564-576.

MacLane, S. (1971) Categories for the Working Mathematician. Springer.

MacQueen, D.B. (1981) Structure and parameterization in a typed functional language. Symp. on Functional Languages and Computer Architecture, Gothenburg, Sweden.

MacQueen, D.B. and Sannella, D.T. (1982) Completeness of proof systems for equational specifications. In preparation.

Manna, Z. and Waldinger, R. (1980) A deductive approach to program synthesis. TOPLAS 2, 1 pp. 90-121.

Manna, Z. and Waldinger, R. (1981) Deductive synthesis of the unification algorithm. Automatic Program Construction (G. Guiho, ed.), NATO Scientific Series, D. Reidel Pub. Co., Dordrecht, Holland.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I. (1962) LISP 1.5 Programmer's Manual. MIT Press.

Milner, R.G. (1978) A theory of type polymorphism in programming. JCSS 17, 3 pp. 348-375.

Milner, R., Morris, L. and Newey, M. (1975) A logic for computable functions with reflexive and polymorphic types. Proc. of Conf. on Proving and Improving Programs, Arc-et-Senans, France; also LCF Report 1, Dept. of Computer Science, Univ. of Edinburgh.

Mosses, P.D. (1976) Compiler generation using denotational semantics. Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science, Gdansk, Poland. Springer Lecture Notes in Computer Science, Vol. 45, pp. 436-441.

Musser, D.L. (1980) On proving inductive properties of abstract data types. Proc. 7th ACM Symp. on Principles of Programming Languages, Las Vegas, Nevada.

Mycroft, A. (1981) Abstract Interpretation and Optimising Transformations for Applicative Programs. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh.

Nakajima, R., Honda, M. and Nakahara, H. (1980) Hierarchical program specification and verification — a many-sorted logical approach. Acta Informatica 14 pp. 135-155.

Nelson, G. and Oppen, D.C. (1979) Simplification by cooperating decision procedures. TOPLAS 1, 2 pp. 245-257.

Nourani, F. (1979) Constructive extension and implementation of abstract data types and algorithms. Ph.D. thesis, Dept. of Computer Science, UCLA.

Nourani, F. (1981) On induction for programming logic: syntax, semantics, and inductive closure. Bulletin EATCS 13, pp. 51-64.

Parnas, D.L. (1972) A technique for software module specification with examples. CACM 15, 5 pp. 330-336.

Parnas, D.L. (1972a) On the criteria to be used in decomposing systems into modules. CACM 15, 12 pp. 1053-1058.

Reichel, H. (1980) Initially-restricting algebraic theories. Proc. 9th Intl. Symp. on Mathematical Foundations of Computer Science, Rydzyna, Poland. Springer Lecture Notes in Computer Science, Vol. 88, pp. 504-514.

Rogers, H. (1967) Theory of Recursive Functions and Effective Computability. McGraw-Hill.

Roubine, O. and Robinson, L. (1977) SPECIAL reference manual (3rd edition). SRI Technical Report CSG-45.

Rydeheard, D.E. (1981) Applications of category theory to programming and program specification. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh.

Sannella, D.T. (1981) A new semantics for Clear. Report CSR-79-81, Dept. of Computer Science, Univ. of Edinburgh.

Sannella, D.T. and Wirsing, M. (1982) Implementation of parameterised specifications. Report CSR-103-82, Dept. of Computer Science, Univ. of Edinburgh; extended abstract in: Proc. 9th Intl. Colloq. on Automata, Languages and Programming, Aarhus, Denmark. Springer Lecture Notes in Computer Science (to appear).

Schoett, O. (1981) Ein Modulkonzept in der Theorie Abstrakter Datentypen. Report IFI-HH-B-81/81, Fachbereich Informatik, Universität Hamburg.

Scott, D.S. (1976) Data types as lattices. SIAM Journal on Computing 5, 3 pp. 522-587.

Spitzen, J.M., Levitt, K.N. and Robinson, L. (1978) An example of hierarchical design and proof. CACM 21, 12 pp. 1064-1075.

Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1976) Specification of abstract data types using conditional axioms. IBM Research Report RC6214.

Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1978) Data type specification: parameterization and the power of specification techniques. SIGACT 10th Annual Symp. on the Theory of Computing, San Diego, California.

Turner, D.A. (1979) SASL language manual. Dept. of Computer Science, Univ. of St. Andrews.

Wand, M. (1979) Final algebra semantics and data type extensions. JCSS 19 pp. 27-44.

Warren, D.H.D., Pereira, L.M. and Pereira, F.C.N. (1977) PROLOG — the language and its implementation compared with LISP. Proc. ACM Symp. on Artificial Intelligence and Programming Languages, Rochester, New York.

Winograd, T. (1979) Beyond programming languages. CACM 22, 7 pp. 391-401.

Wirsing, M. and Broy, M. (1980) Abstract data types as lattices of finitely generated models. Proc. 9th Intl. Symp. on Mathematical Foundations of Computer Science, Rydzyna, Poland. Springer Lecture Notes in Computer Science, Vol. 88, pp. 673-685.

Wirsing, M. and Broy, M. (1981) An analysis of semantic models for algebraic specifications. Proc. 1981 Marktoberdorf Intl. Summer School on Theoretical Foundations of Programming Methodology.

Wirth, N. (1971) Program development by stepwise refinement. CACM 14, 4 pp. 221-227.

APPENDIX ONE

HOPE

The following description of HOPE is a condensation of [Burstall, MacQueen and Sannella 1980], brought up to date. Lazy evaluation is not mentioned, since none of the programs in this thesis use that facility of HOPE. After a brief presentation of the notation and features of HOPE, a simple example of a HOPE program is given. This is followed by a discussion of some of the advantages and disadvantages of HOPE, and notes concerning its implementation.

A precursor of HOPE called NPL is described by Burstall [1977]. Major influences in the design of HOPE were LISP and ISWIM [Landin 1966]. It bears some resemblance to a number of other languages, including PROLOG [Warren, Pereira and Pereira 1977], ML [Gordon, Milner and Wadsworth 1979], SASL [Turner 1979], OBJ [Goguen and Tardo 1979], SCRATCHPAD [Jenks 1974], and languages by Burge [1975] and Backus [1978].

1. Data declarations

Conceptually, all data in HOPE is represented as terms consisting of a data constructor applied to a number of subterms, each of which in turn represents another data item. The tips of this tree are nullary data constructors or functional objects. An example is succ(succ 0) in which succ is a unary constructor and 0 is a nullary one (i.e. a constant). Constructor functions are uninterpreted; they just construct.

A data declaration is used to introduce a new data type along with the data constructors which create elements of that type. For example, the data declaration for natural numbers would be:

```
data num == 0 ++ succ num
```

defining a data type called num with data constructors 0 and succ. So the elements of num are 0, succ(0), succ(succ 0), ... ; that is, 0, 1, 2, ...

To define a type 'tree-of-numbers' we could say

```
data numtree == empty ++ tip num ++ node(numtree,numtree)
```

One of the elements of numtree is:

```
node(tip(succ 0),node(tip(succ(succ 0)),tip 0))
```

But we would like to have trees of lists and trees of trees as well, without having to define them all separately. So we declare a type variable

```
typevar alpha
```

which when used in a type expression denotes any type (including second- and higher-order types). A general definition of tree as a parameterised type is now possible:

```
data tree(alpha) == empty ++ tip alpha
                          ++ node(tree alpha,tree alpha)
```

Now tree is not a type but a unary type constructor -- the type numtree can be dispensed with in favour of tree(num).

Another example of a data declaration is

```
data graph == mkg(set vertex,(vertex#vertex->truval))
```

(the sign # gives the cartesian product of types). This says that a graph is (the data constructor mkg applied to) a set of vertices together with a binary relation which tells if there is an edge between any two vertices.

Another way to define graphs is using a type declaration:

```
type graph == set vertex # (vertex#vertex->truval)
```

Now graph is just an abbreviation for a type tuple, rather than a new data type. With this definition no data constructor is used to construct a graph. Type definitions may be parameterised in the same way as data declarations, but they may not be recursive.

HOPE currently comes equipped with the data types num, truval, char, list, set, and map (finite functions).

2. Expressions

The simplest expressions of HOPE are constants (i.e. data constructors and functions -- the 'usual' concept of a constant is just the class of nullary functions and data constructors) and variables.

An application may be formed by simply juxtaposing two expressions:

```
factorial 6
```

For functions of several arguments we use tuples, formed with commas; thus 3,4 is a 2-tuple. Parentheses are used for grouping, for example:

```
g (3,4)
```

In the expression

```
(f x) y
```

the subexpression f x would have to produce a function; thus the types would be

$$f : T_1 \rightarrow T_2 \rightarrow T_3$$

with $x:T_1$ and $y:T_2$.

It is possible to use function symbols as infix or postfix operators if they are declared and given a precedence; for example:

```
infix +, - : 8
```

A similar form is used to assign a precedence to a prefix symbol.

Distributed-fix operators (see [Goguen and Tardo 1979]) are also available; for example:

```
distfix while _ do _
distfix _ unless _ in which case _
```

Some convenient notations have been implemented for built-in types; thus $e_1::(e_2:: \ldots ::nil)$ is abbreviated $[e_1,e_2, \ldots]$, ['a','b', ...] is "ab..." and sets are written $\{e_1,e_2, \ldots\}$. Note that we write cons as infix :: .

There are two equivalent forms of conditional expression:

e_1 if c else e_2

and

c then e_1 else e_2

(in many languages written if c then e_1 else e_2).

Lambda-expressions (denoting functions) are formed as described in section 3.

Local variables may be introduced and associated with values using either of the equivalent forms

e_1 where p == e_2

or

let p == e_2 in e_1

where p is an expression formed by application of data constructors to a number of distinct variables (this is called a pattern). For example:

a+b where a::(b::l) == f(t)

Upon evaluation, f(t) is expected to yield a value which 'matches' the pattern a::(b::l). The corresponding subterms in the value of f(t) are then bound to a, b, and l while evaluating a+b.

3. Defining functions

Before a function is defined, its type must be declared. For example:

dec reverse : list alpha -> list alpha

HOPE is a very strongly-typed language, and the HOPE system includes a polymorphic typechecker (a modification of the algorithm in [Milner 1978]) which is able to detect all type errors at compile time. Function symbols may be overloaded. When this is done, the typechecker is able to determine which function definition belongs to each instance of the function symbol.

Functions are defined by a sequence of one or more equations, where each equation specifies the function over some subset of the possible argument values. This subset is described by a pattern (see section 2) on the left-hand side of the equation. For example:

```
--- reverse nil <= nil                                    (1)
--- reverse(a::l) <= reverse l <> [a]                     (2)
```

(the symbol <> is infix append). This defines the (top-level) reverse of a list; for example:

```
reverse(1::(2::nil)) = reverse(2::nil) <> [1]
                     = (reverse nil <> [2]) <> [1]
                     = (nil <> [2]) <> [1]
```

So reverse [1,2] = [2,1] (by two applications of equation 2 followed by a single application of equation 1). The left-hand-side patterns

will normally be disjoint and be related to the structure of the type definition:

```
data list alpha == nil ++ alpha :: list alpha
```

The set of equations defining a function should exhaust the possibilities given in the data-statement introducing the argument types. For example, a definition of the Fibonacci numbers:

```
dec fib : num -> num
--- fib 0 <= 1
--- fib(succ 0) <= 1
--- fib(succ(succ n)) <= fib(succ n) + fib n
```

In this case the three patterns 0, succ 0, and succ(succ n) exhaust the set of values belonging to num. The pattern 1 may be used as shorthand for succ(0).

Nullary 'functions' may also be defined; for example:

```
dec pi : rational
--- pi <= mkrational(22,7)
```

which assumes that the type rational has been defined.

Lambda-expressions are defined similarly. For example, a function to compute the conjunction of two truth values (already available as the function 'and'):

```
lambda true,p => p
     ¦ false,p => false
```

Another example of a lambda-expression occurs in the definition of function composition:

```
typevar alpha,beta,tau
dec compose : (alpha->beta) # (beta->tau) -> (alpha->tau)
--- compose(f,g) <= lambda x => f(g x)
```

Patterns may be somewhat more complex than those used above; for example:

```
--- f(ll & (_ :: (c::_))) <= c::ll
```

This pattern uses "don't care" variables (underscores) to give the shape of the pattern without specifying variable bindings, and the multilevel pattern operation (ampersand) to bind variables to the same value at different levels. The expression f[1,2,3] will have the value [2,1,2,3].

4. Modules

Any sequence of statements may be made into a module by surrounding it with the statements

module mname

and

end

Data types defined in a module may be referred to outside only if a statement

pubtype tname

is included in the module. Similarly, constants (including data constructors) may be referenced only if a statement

pubconst cname

is included.

Nothing defined outside a module may be referenced within it, unless the module includes the statement

uses mname

In this case, all of the types and constants declared as public to the indicated module are available. In addition, certain global types and constants (num, truval, char, list, set and map, together with some primitive operations) may be referenced within any module.

This is an effective tool for the encapsulation of data abstractions; if the primitive constructors and low-level operations on the data representation are not declared public, then the implementation of the abstraction is hidden from the rest of the program.

5. An example

An example of a complete HOPE program is given below. This illustrates how we can use HOPE to implement a data type (ordered trees), and then how that type can be used in a program for treesort.

```
module ordered_trees
  pubtype otree
  pubconst empty, insert, flatten

  data otree == empty ++ tip num ++ node(otree,num,otree)

  dec insert : num # otree -> otree
  dec flatten : otree -> list num

  --- insert(n,empty)  <= tip n
  --- insert(n,tip m) <= n<m then node(tip n,m,empty)
                          else node(empty,m,tip n)
  --- insert(n,node(t1,m,t2)) <= n<m then node(insert(n,t1),m,t2)
                                 else node(t1,m,insert(n,t2))

  --- flatten empty  <= nil
  --- flatten(tip n) <= [n]
  --- flatten(node(t1,n,t2)) <= flatten t1 <> (n::flatten t2)

end


module list_iterators
  pubconst *, **

  typevar alpha, beta

  dec * : (alpha->beta) # list alpha -> list beta
  dec ** : (alpha#beta->beta) # (list alpha # beta) -> beta

  infix *, ** : 6

  --- f * nil     <= nil
  --- f * (a::al) <= (f a)::(f * al)

  --- g ** (nil,b)   <= b
  --- g ** (a::al,b) <= g ** (al,g(a,b))

end


module tree_sort
  pubconst sort
  uses ordered_trees, list_iterators

  dec sort : list num -> list num

  --- sort l <= flatten(insert ** (l,empty))

end
```

Ordered trees

The first module contains an implementation of the abstract type ordered-tree-of-numbers (data type otree in the program). An otree is defined to be either empty, a tip (containing a number), or a node containing two otrees and a number. The special property of otree is that for any term node(t1,n,t2), all numbers contained in t1 are less than n, which is in turn less than or equal to all numbers contained in t2. We define three public constants:

empty	the empty otree
insert	adds a number to an otree, preserving the 'orderedness' of the otree
flatten	inorder traversal of an otree

Ordinarily an abstract data type would have a few more operations; only those which are used in the remainder of the program have been included here.

Note that the data constructor 'node' is not public. Consequently, the only functions available to the 'outside world' for constructing and modifying otrees are 'empty' and 'insert'. Both of these preserve the properties of otrees, so the integrity of the implementation is assured. However, insert is not a data constructor, and hence may not be used in patterns.

List iterators

This module defines two second-order functions which apply a given function to every element of a list and collect the results. These two functions are representatives of a group of functions which are widely used in HOPE programs in an attempt to eliminate explicit recursion as far as possible. Both of these are in fact provided as primitive operations in HOPE, but their definitions are repeated here nonetheless.

The function * is identical to mapcar in LISP. It produces a list containing the results of applying the function supplied to each element of the given list. This operation is not actually used in the example.

The function ** is slightly more complicated. When supplied with a function g of type alpha#beta -> beta, a list of alpha-objects, and an 'initial' beta-object, it applies g to each element of the list, beginning with the given beta-object as a second argument and subsequently recycling the result of the previous application. This operation is analogous to the 'reduction' operator of APL [Iverson 1962]; an example of its use would be to compute the union of a list of sets:

```
union ** (setlist,nil_set)
```

In this case, the module facility is used as a means of packaging a number of related functions rather than as a device for protecting a delicate abstraction. However, if one of the operations requires an auxiliary function which has no utility of its own, then it might be desirable to keep this function local to the module.

Tree sort

A function for sorting a list of numbers is now defined using the primitives developed in the preceding modules. The ** operation from list_iterators is used to successively insert the list elements into an initially empty otree. The result is then flattened to produce the final answer.

6. Advantages and disadvantages

The greatest triumph of HOPE is that we have found it to be significantly easier to construct programs in HOPE than in any other programming language we know. In particular, it is rather easy to write programs which are absolutely correct the first time they are run. It seems quite difficult to commit an error which remains undiscovered for long -- the simple errors are caught during compilation by the typechecker, while the more fundamental errors (stemming usually from an insufficient understanding of the problem) display themselves glaringly during even a casual test.

An important aim of language design is to make it easier to verify that a program meets a given specification. In this respect

applicative languages such as HOPE seem to offer considerable advantages; the absence of assignment statements and the consequent replacement of iteration by recursion gives programs a simple and easy to analyse form. Powerful verification systems for applicative languages have been written by Boyer and Moore [1980] and by Aubin [1977].

HOPE has faults, too; one is illustrated in the example in the last section. The sorting program will only sort a list of numbers, because otree is 'ordered-tree-of-numbers'. We want a more general sorting program, and this depends on a more general definition of ordered trees; we would like to define 'ordered-tree-of-alphas'. The data declaration is easy to generalise. But to generalise insert to type

alpha # otree alpha -> otree alpha

we must have a more general order relation than <, which is defined only for numbers. But a general order (of type alpha#alpha->truval) cannot be defined; for each data type the order must be defined separately.

The solution is to associate a collection of operations with each data type (so types become algebras instead of simply sets). Rather than generalising to otree(alpha) we could generalise to otree(alpha[<]), requiring an order relation to exist on the parameter type. This is the approach taken in CLU [Liskov, Snyder, Atkinson and Schaffert 1977] and in Clear. We really want HOPE modules to have parameters, a collection of types and operators, just as CLU clusters have parameters.

As a further example, refer again to the sorting program and note that the module tree_sort does not depend on the fact that otrees are trees, but just on certain properties of insert and flatten. We may substitute a module ordered_lists for ordered_trees, where empty becomes nil, insert becomes the obvious order-preserving insertion in an ordered list, and flatten is the identity function. Essentially, tree_sort is a parameterised module which may be 'applied' to any module satisfying certain (nontrivial) properties.

Parameterised modules do not exist in present-day HOPE, but

MacQueen [1981] has proposed an extension to the type system of HOPE based on ideas from Clear which accommodates them nicely. In MacQueen's language, an abstraction is made up of an interface (the 'meta-type' of the abstraction, declaring the types and operators which it makes available) and a structure (an implementation of the types and operators promised by the interface). Interfaces and structures are defined and manipulated separately, and may be parameterised by other interfaces and structures.

7. Implementation

The HOPE system consists of a compiler (from HOPE programs to code for an abstract stack machine) and an implementation of the target machine. The system is written in POP-2, and currently runs in approximately 51K words (plus a 15K shareable segment) on a DEC KL-10.

Timing tests indicate that a program written in HOPE runs approximately 3 times slower than the same algorithm coded in LISP running under the Rutgers/UCI interpreter (and 50 times slower than compiled LISP). Large programs run more slowly because of page thrashing. A machine code implementation of the interpreter should run a lot faster.

A very high-level language such as HOPE pays penalties of inefficiency because it is remote from the machine level. It could be thought of as a specification language in which the specifications are 'walkable' (if not 'runnable'), or as a language for making a first try at a programming project. But recent work on efficiency issues in applicative languages gives us hope that we can produce tolerably efficient programs with less effort than in a conventional language.

An advantage of an applicative language is the fact that programs lend themselves very well to the technique of program transformation [Burstall and Darlington 1977], whereby a simple but inefficient program is transformed into an acceptably efficient one by steps which maintain its correctness. A very simple example of program transformation would be the production of the following linear-time

program for generating Fibonacci numbers from the equivalent program in section 3 which requires exponential time.

```
dec g : num -> num#num
--- g 0 <= 1,1
--- g(succ n) <= (a + b),a   where a,b == g n

dec fib' : num -> num
--- fib' 0 <= 1
--- fib' 1 <= 1
--- fib'(succ(succ n)) <= a + b   where a,b == g n
```

Feather [1982] has produced a system for transforming large programs, which is connected to an earlier version of the HOPE system. Mycroft [1981] describes a method for detecting automatically when 'applicative' operators can be replaced by destructive operators in a program written in an applicative language without changing its semantics. The transformed program will consume storage less rapidly with the result that garbage collection will occur less frequently.

In addition, there is another advantage of applicative languages which may come to our rescue: applicative languages are not so tightly bound to the notion of a sequential machine as are imperative languages. The value of the function application

$$e_0(e_1, \ldots, e_n)$$

is independent of the order of evaluation of the expressions $e_0, \ldots, e_n$ (if parameters are passed 'by value'); this is guaranteed by the absence of an assignment statement. If a parallel machine is available, $e_0, \ldots, e_n$ may be evaluated simultaneously. Not only that, but if $e_0, \ldots, e_n$ are themselves function applications, then their arguments may all be evaluated simultaneously. Darlington and Reeve [1981] describe the architecture of a machine which is capable of running HOPE programs in such a parallel fashion.

HOPE is still somewhat incomplete, lacking such conveniences as sensible input/output facilities. A way of neatly adding interactive input/output to HOPE using streams was proposed by Burstall, MacQueen and Sannella [1980], but this was never implemented. At the present time there is no provision for interactive input, and only the most rudimentary printing facility

is available (a function which has the side effect of printing its argument at the terminal).

In order to make up for deficiencies such as these for the time being, a facility has been added to HOPE which allows a HOPE function to be defined by a POP-2 program. The function is declared as usual, and its meaning is attached later using a set of POP-2 macros. This provides the means for supplying all the power of POP-2 in HOPE (of particular interest is the possibility of using POP-2 input/output facilities), and it also could be used to make important HOPE programs more efficient. Naturally, there is no way to typecheck the POP-2 code at compile time, and since there is no runtime typechecking in HOPE it is easy to violate the HOPE type system in this fashion. But when used with care and discretion this facility makes it possible to construct large and useful systems in HOPE. The Clear implementation described in chapter IV is an example; it uses the HOPE parser and typechecker as well as input and file handling routines written in POP-2.

APPENDIX TWO

LIBRARY OF BASIC SPECIFICATIONS

Listed below are all the theories included in the initial environment of the Clear system described in chapter IV. All with the exception of Bool are shown exactly as they are given to the system (except that all keywords have been underlined). Bool must be treated specially because the data-enrich operation expects the tagged sort $bool_{Bool}$ to be present, and if Bool is added in the normal fashion the sort bool will be given an arbitrary tag.

```
const Bool =
  let Bool0 =
    theory
      data sorts bool
           opns true, false : bool    endth in
  enrich Bool0 by
    opns not : bool -> bool
         (_ or _), (_ and _), (_ --> _) : bool,bool -> bool
    eqns not(true) = false              not(false) = true
         p or true = true               p or false = p
         p and true = p                 p and false = false
         p-->q = not(p and not(q))           enden
```

```
const Nat =
  let Nat0 =
    enrich Bool by
      data sorts nat
           opns 0 : nat
                succ : nat -> nat    enden in
  enrich Nat0 by
    opns 1, 2, 3, 4, 5, 6, 7, 8, 9 : nat
         (_ =< _), (_ >= _), (_ < _), (_ > _) : nat,nat -> bool
         (_ plus _), (_ - _), (_ * _),
                     (_ div _), (_ mod _) : nat,nat -> nat
    erroropns neg : nat
    eqns 1 = succ(0)   2 = succ(1)   3 = succ(2)
         4 = succ(3)   5 = succ(4)   6 = succ(5)
         7 = succ(6)   8 = succ(7)   9 = succ(8)
         0=<n = true                 succ(m)=<0 = false
         succ(m)=<succ(n) = m=<n     m>=n = n=<m
         m<n = m=<n and not(m==n)    m>n = n<m
         0 plus n = n                succ(m) plus n = succ(m plus n)
         m plus n - m = n            0*n = 0
         succ(m)*n = m*n plus n      m*n plus p  div m = n  if p<m
         m*n plus p  mod m = p  if p<m
    erroreqns m-n = neg  if m<n                          enden
```

```
const Int =
  let Int0 =
    enrich Bool by
      data sorts int
           opns 0 : int
                pred, succ : int -> int
           eqns pred(succ(n)) = n
                succ(pred(n)) = n    enden in
  enrich Int0 by
    opns 1, 2, 3, 4, 5, 6, 7, 8, 9,
         (_ =< _), (_ >= _), (_ < _), (_ > _) : int,int -> bool
         (- _), magnitude : int -> int
         (_ plus _), (_ - _), (_ * _),
                     (_ div _), (_ mod _) : int,int -> int
    eqns 1 = succ(0)     2 = succ(1)     3 = succ(2)
         4 = succ(3)     5 = succ(4)     6 = succ(5)
         7 = succ(6)     8 = succ(7)     9 = succ(8)
         0 - n = - n
         n=<n = true                     n=<pred(n) = false
         pred(n)=<m = true if n=<m       n=<pred(m) = false if not(n=<m)
         n=<succ(m) = true if n=<m       succ(n)=<m = false if not(n=<m)
         m>=n = n=<m                     m<n = m=<n and not(m==n)
         m>n = n<m                       0 plus n = n
         succ(m) plus n = succ(m plus n)
         pred(m) plus n = pred(m plus n)
         m plus n - m = n                0*n = 0
         succ(m)*n = m*n plus n          pred(m)*n = m*n - n
         magnitude(m) = m  if m>=0       magnitude(m) = - m  if m<0
         m*n plus p  div m = n  if p<magnitude(m) and p>=0
         m*n plus p  mod m = p  if p<magnitude(m) and p>=0
                                                    enden


const Character =
  derive sorts character
         opns blank, A, B, C, D, E, F, G, H, I : character
              (_ == _), (_ =< _), (_ >= _), (_ < _), (_ > _) :
                                    character,character -> bool
    using Bool
    from Nat
    by character is nat,          blank is 0,
       A is 1,        B is 2,        C is 3,
       D is 4,        E is 5,        F is 6,
       G is 7,        H is 8,        I is 9    endde


meta Triv =
  theory sorts element    endth
```

```
meta Ident =
  enrich Bool + Triv by
    opns (_ == _) : element,element -> bool
    eqns all i:element. i==i = true
         all i,j:element. i==j = j==i
         all i,j:element. i==j and j==k --> (i==k) = true    enden


meta POSet =
  enrich Ident by
    opns (_ =< _) : element,element -> bool
    eqns i=<i = true
         i=<j and j=<i --> (i==j) = true
         i=<j and j=<k --> (i=<k) = true    enden


proc Sequence(X:Triv) =
  let Seq0 =
    enrich X + Bool by
      data sorts sequence
           opns empty : sequence
                unit : element -> sequence
                (_ . _) : sequence,sequence -> sequence
           eqns empty.s = s
                s.empty = s
                s.t.v = s.(t.v)    enden in
  enrich Seq0 + Nat by
    opns length : sequence -> nat
    eqns length(empty) = 0        length(unit(a)) = 1
         length(s.t) = length(s) plus length(t)    enden


proc Pair(X:Triv,Y:Triv) =
  enrich X + Y + Bool by
    data sorts pair
         opns (_ # _) : element of X,element of Y -> pair    enden


proc Sum(X:Triv,Y:Triv) =
  enrich X + Y + Bool by
    data sorts sum
         opns inl : element of X -> sum
              inr : element of Y -> sum    enden
```

```
proc Set(X:Triv) =
  let Set0 =
    enrich X + Bool by
      data sorts set
           opns empty : set
                singleton : element -> set
                (_ U _) : set,set -> set
           eqns S U empty = S
                S U S = S
                S U T = T U S
                S U T U V = S U (T U V)    enden in
  enrich Set0 + Nat by
    opns (_ is_in _) : element,set -> bool
         (_ - _), (_ intersect _) : set,set -> set
         card : set -> nat
    eqns a is_in empty = false
         a is_in singleton(b) = singleton(a)==singleton(b)
         a is_in (S U T) = a is_in S or a is_in T
         empty-S = empty
         singleton(a)-S = empty  if a is_in S
         singleton(a)-S = singleton(a)  if not(a is_in S)
         T U V - S = (T-S) U (V-S)
         S intersect T = S-(S-T)
         card(empty) = 0
         card(singleton(a)) = 1
         card(S U T) = card(S) plus card(T)-card(S intersect T)
                                                        enden
```

```
proc Bag(X:Triv) =
  let Bag0 =
    enrich X + Bool by
      data sorts bag
           opns empty : bag
                singleton : element -> bag
                (_ U _) : bag,bag -> bag
           eqns S U empty = S
                S U T = T U S
                S U T U V = S U (T U V)    enden in
  enrich Bag0 + Nat by
    opns (_ is_in _) : element,bag -> bool
         occurrences : element,bag -> nat
    eqns a is_in empty = false
         a is_in singleton(b) = singleton(a)==singleton(b)
         a is_in (S U T) = a is_in S or a is_in T
         occurrences(a,empty) = 0
         occurrences(a,singleton(b)) = 0
                              if not(a is_in singleton(b))
         occurrences(a,singleton(b)) = 1  if a is_in singleton(b)
         occurrences(a,S U T) = occurrences(a,S)
                                plus occurrences(a,T)     enden
```

```
proc Stack(X:Triv) =
  let Stack0 =
    enrich X + Bool by
      data sorts stack
           opns empty : stack
                push : element, stack -> stack    enden in
  enrich Stack0 by
    opns top : stack -> element
         pop : stack -> stack
         isempty : stack -> bool
    erroropns undef : element
              underflow : stack
    eqns top(push(a,s)) = a          pop(push(a,s)) = s
         isempty(empty) = true       isempty(push(a,s)) = false
    erroreqns top(empty) = undef     pop(empty) = underflow   enden


proc Map(X:Ident,Y:Triv) =
  let Map0 =
    enrich X + Y by
      data sorts map
           opns empty : map
                insert : map,element of X,element of Y -> map
           eqns insert(insert(f,a,b),a,d) = insert(f,a,d)
                insert(insert(f,a,b),c,d) = insert(insert(f,c,d),a,b)
                                            if not(a==c)    enden in
  enrich Map0 + Set(X) by
    opns (_ << _ >>) : map,element of X -> element of Y
         domain : map -> set
         (restrict _ to _) : map,set -> map
         (_ is_in _) : element of X,map -> bool
    erroropns undef : element of Y
    eqns insert(f,a,b)<<a>> = b
         insert(f,a,b)<<c>> = f<<c>>  if not(a==c)
         domain(empty) = empty
         domain(insert(f,a,b)) = singleton(a) U domain(f)
         restrict empty to S = empty
         restrict insert(f,a,b) to S = restrict f to S
                                            if not(a is_in S)
         restrict insert(f,a,b) to S = insert(restrict f to S,a,b)
                                            if a is_in S
         a is_in f = a is_in domain(f)
    erroreqns empty<<a>> = undef    enden
```

```
proc Relation(X:Ident,Y:Ident) =
  let Rel0 =
    enrich X + Y by
      data sorts relation
           opns empty : relation
                insert : relation,element of X,element of Y
                                                  -> relation
           eqns insert(insert(R,a,b),a,b) = insert(R,a,b)
                insert(insert(R,a,b),c,d) = insert(insert(R,c,d),a,b)
                                            if not(a==c) or not(b==d)
                                                                enden in
  enrich Rel0 + Set(X) by
    opns isrelated: relation,element of X,element of Y -> bool
         domain : relation -> set
    eqns isrelated(empty,a,b) = false
         isrelated(insert(R,a,b),a,b) = true
         isrelated(insert(R,a,b),c,d) = isrelated(R,c,d)
                                            if not(a==c) or not(b==d)
         domain(empty) = empty
         domain(insert(R,a,b)) = singleton(a) U domain(R)    enden
```

APPENDIX THREE

SUBSET OF PPLAMBDA USED BY THE THEOREM PROVER

The impoverished version of PPLAMBDA used by the theorem prover discussed in chapter VI is described here. It is necessary to remove the implicit order relation and 'bottom' element because models of Clear theories do not possess these; other irrelevant elements of PPLAMBDA have been removed as well. Refer to [Gordon, Milner and Wadsworth 1979] for details concerning the items mentioned briefly below.

Types

The built-in type constructors 'prod' (cartesian product) and 'fun' (function space) are still available. The following type constructors have been deleted: ., tr, u, sum

Constants

Only the built-in constant 'PAIR' is still available. The following constants have been deleted: TT, FF, UU, COND, FST, SND, INL, INR, OUTL, OUTR, ISL, FIX, UP, DOWN, DEF, ()

Formulae

All the usual PPLAMBDA formulae are allowed except for inequations (e.g. f<<f').

Inference rules

The following inference rules are available:

AXTRUTH = $\vdash_{LCF}$TRUTH

ASSUME f = {w}$\vdash_{LCF}$w

CONJ($A\vdash_{LCF}f$, $A'\vdash_{LCF}f'$) = $A \cup A' \vdash_{LCF} f\&f'$

GEN x $A\vdash_{LCF}f$ = $A\vdash_{LCF}$!x.f (fails if x occurs free in A)

DISCH f' $A\vdash_{LCF}f$ = $A'\vdash_{LCF}f'$ IMP f (where A' is the set of assumptions in A not alpha-convertible to f')

SEL1 $A\vdash_{LCF}f\&f'$ = $A\vdash_{LCF}f$

SEL2 $A\vdash_{LCF}f\&f'$ = $A\vdash_{LCF}f'$

SPEC t $A\vdash_{LCF}$!x.f = $A\vdash_{LCF}f[t/x]$

MP $A\vdash_{LCF}(f$ IMP $f')$ $A'\vdash_{LCF}f$ = $A \cup A' \vdash_{LCF} f'$

INST [t1,x1;...] $A \vdash_{LCF} f$ = $A \vdash_{LCF} f[t1/x1...]$
(fails if any xi occurs free in A)

INSTTYPE [ty1,vty1;...] $A \vdash_{LCF} f$ = $A \vdash_{LCF} f\{ty1/vty1...\}$
(fails if any vty1 is not a vartype, or is a vartype in A)

REFL t = $\vdash_{LCF} t=t$

SYM $A \vdash_{LCF} t=t'$ = $A \vdash_{LCF} t'=t$

TRANS($A \vdash_{LCF} t=t'$, $A' \vdash_{LCF} t'=t''$) = $A \cup A' \vdash_{LCF} t=t''$

SUBST [$A1 \vdash_{LCF} t1=t1'$,x1;...] f $A' \vdash_{LCF} f'[t1/x1...]$
= Union(Ai) $\cup A' \vdash_{LCF} f'[t1'/x1...]$

SUBS [$A1 \vdash_{LCF} t1=t1'$;...] $A' \vdash_{LCF} f'$ = Union(Ai) $\cup A' \vdash_{LCF} f'[t1'/t1...]$

SUBSOCCS [intl1,$A1 \vdash_{LCF} t1=t1'$;...] $A' \vdash_{LCF} f'$ = As for SUBS,
but substitutes according to occurrence numbers in intli

APTERM t $A \vdash_{LCF} t'=t''$ = $A \vdash_{LCF} t\ t'=t\ t''$

APTHM $A \vdash_{LCF} t'=t''$ t = $A \vdash_{LCF} t'\ t=t''\ t$

LAMGEN x $A \vdash_{LCF} t=t'$ = $A \vdash_{LCF} \lambda x.t=\lambda x.t'$
(fails if x occurs free in A)

BETACONV $(\lambda x.t)t'$ = $\vdash_{LCF} (\lambda x.t)t'=t[t'/x]$

ETACONV $\lambda x.(t\ x)$ = $\vdash_{LCF} \lambda x.(t\ x)=t$ (fails if x occurs free in t)

EXT $A \vdash_{LCF} !x.(t\ x=t'\ x)$ = $A \vdash_{LCF} t=t'$
(fails if x occurs free in t or t')

ABS x $A \vdash_{LCF} t\ x=t'$ = $A \vdash_{LCF} t=\lambda x.t$ (fails if x occurs free in t or A)

SIMP ss $A \vdash_{LCF} f$ = $A \cup A' \vdash_{LCF} f'$ where f' is the result of simplifying f using ss and A' is a subset of the hypotheses of theorems in ss

The following inference rules have been deleted: SYNTH, ANAL, HALF1, HALF2, MIN, MINAP, MINFN, FIXPT, FIX, INDUCT, AXDEF, DEFUU, DEFCONV, CONDCONV, CONDTRCONV, CASES, CONTR, DOT, DOWNCONV, UPCONV, SELCONV, PAIRCONV, ISCONV, OUTCONV, INCONV

Simplification

The only simplification rules in BASICSS are those corresponding with the inference rules BETACONV and ETACONV.

The following simplification rules have been deleted: MINAP, MINFN, DEFCONV, CONDCONV, CONDTRCONV, UPCONV, DOWNCONV, SELCONV,

PAIRCONV, ISCONV, OUTCONV, INCONV

Tactics

The standard LCF tactics CASESTAC, CONDCASESTAC, INDUCTAC and INDUCOCCSTAC have been deleted, leaving GENTAC, SUBSTAC, SUBSOCCSTAC, SIMPTAC and all tactics provided by the Clear theorem prover.

APPENDIX FOUR

PROOF OF SOUNDNESS OF THE THEOREM PROVER

The following results imply the soundness of the theorem prover described in chapter VI; see section VI.3 for definitions and motivation.

Notation: If A is a Σ-algebra, X is a set and $f:X\rightarrow|A|$, then $f^{\#}:W_{\Sigma}(X)\rightarrow A$ is the unique homomorphism extending f.

Satisfaction Lemma: If $\sigma:\Sigma\rightarrow\Sigma'$ is a signature morphism, f is a Σ-formula and $\underline{A}'$ is a Σ'-algebra, then $\underline{A}'\models\sigma(f)$ iff $\underline{A}'|_{\sigma}\models f$.

Proof (Satisfaction Lemma): By structural induction.

Case 1: f is TRUTH

trivial since $\sigma(\text{TRUTH})=\text{TRUTH}$ for any σ and $A\models\text{TRUTH}$ for any A

Case 2: f is t=t'

$A'\models\sigma(t=t')$
$\Leftrightarrow A'\models\sigma(t)=\sigma(t')$
$\Leftrightarrow \forall k:FV(\sigma(t)\cup\sigma(t'))\rightarrow|A'|.\ A'\models k^{\#}(\sigma(t))=k^{\#}(\sigma(t'))$
$\Leftrightarrow \forall h:FV(t)\cup FV(t')\rightarrow|A'|_{\Sigma}|.\ A'|_{\Sigma}\models h^{\#}(t)=h^{\#}(t')$
(by the proof of the Satisfaction Lemma for equations; see [Burstall and Goguen 1980])
$\Leftrightarrow A'|_{\Sigma}\models t=t'$

Case 3: f is f' & f"; we know $A'\models\sigma(f')\Leftrightarrow A'|_{\Sigma}\models f'$ and similarly for f"

$A'\models\sigma(f'\ \&\ f")$
$\Leftrightarrow A'\models\sigma(f')\ \&\ \sigma(f")$
$\Leftrightarrow A'\models\sigma(f')$ and $A'\models\sigma(f")$
$\Leftrightarrow A'|_{\Sigma}\models f'$ and $A'|_{\Sigma}\models f"$ (by the inductive assumptions)
$\Leftrightarrow A'|_{\Sigma}\models f'\ \&\ f"$

Case 4: f is f' IMP f"; we know $A'\models\sigma(f')\Leftrightarrow A'|_{\Sigma}\models f'$ and similarly for f"

$A'\models\sigma(f'\ \text{IMP}\ f")$
$\Leftrightarrow A'\models\sigma(f')\ \text{IMP}\ \sigma(f")$
$\Leftrightarrow \forall k:FV(\sigma(f'))\cup FV(\sigma(f"))\rightarrow|A'|.\ A'\models k^{\#}(\sigma(f'))\Rightarrow k^{\#}(\sigma(f"))$
$\Leftrightarrow \forall k:FV(\sigma(f'))\cup FV(\sigma(f"))\rightarrow|A'|.\ A'\models k^{\#}(\sigma(f'))\Rightarrow A'\models k^{\#}(\sigma(f"))$
$\Leftrightarrow \forall h:FV(f')\cup FV(f")\rightarrow|A'|_{\Sigma}|.\ A'|_{\Sigma}\models h^{\#}(f')\Rightarrow A'|_{\Sigma}\models h^{\#}(f")$
(by the inductive assumptions)
$\Leftrightarrow \forall h:FV(f')\cup FV(f")\rightarrow|A'|_{\Sigma}|.\ A'|_{\Sigma}\models h^{\#}(f')\Rightarrow h^{\#}(f")$
$\Leftrightarrow A'|_{\Sigma}\models f'\ \text{IMP}\ f"$

Case 5: f is !x.f'; we know $A' \models \sigma(f') \Leftrightarrow A'|_{\Sigma} \models f'$

$A' \models \sigma(!x.f')$

$\Leftrightarrow A' \models \sigma(f')$ (assuming A'_s is nonempty, where x is of sort s; otherwise $A' \models \sigma(!x.f') \Leftrightarrow A'|_{\Sigma} \models !x.f'$ vacuously)

$\Leftrightarrow A'|_{\Sigma} \models f'$ (by the inductive assumption)

$\Leftrightarrow A'|_{\Sigma} \models !x.f'$

Theorem: For any E-agglomerate A, $\mathbb{F}[\![\tau(A)]\!] \subseteq \mathbb{E}[\![A]\!]^{*+}$, where τ:E-agglomerate$\rightarrow$F-agglomerate is the following translation function:

$$\tau \begin{cases} \text{union}(A,A') \longmapsto \text{union}(\tau(A),\tau(A')) \\ \quad \cdots \\ \text{close}(E,C) \longmapsto \text{close}(\text{eqn-to-form}^*E \cup \text{induction-rules}^*C) \end{cases}$$

Proof: By structural induction.

Case 1: A is close(E,C)

For any model M,

$M \models E \Leftrightarrow M \models \text{eqn-to-form}^*E$

and $M \not\models C \Rightarrow M \models \text{induction-rules}^*C$

so $(E \cup C)^{***} = (E \cup C)^{*} \subseteq (\text{eqn-to-form}^*E \cup \text{induction-rules}^*C)^{+}$

hence $(\text{eqn-to-form}^*E \cup \text{induction-rules}^*C)^{++} \subseteq \overline{E \cup C}^{*+}$

so $\mathbb{F}[\![\tau(A)]\!] \subseteq \mathbb{E}[\![A]\!]^{*+}$

Case 2: A is translate(σ,A'); we know $\mathbb{F}[\![\tau(A')]\!] \subseteq \mathbb{E}[\![A']\!]^{*+}$

$\mathbb{F}[\![\tau(\text{translate}(\sigma,A'))]\!]$

$= \mathbb{F}[\![\text{translate}(\sigma,\tau(A'))]\!]$

$= \sigma(\mathbb{F}[\![\tau(A')]\!])^{++}$

$\subseteq \sigma(\mathbb{E}[\![A']\!]^{*+})^{++}$ (by the inductive assumption)

$= \sigma(\mathbb{E}[\![A']\!])^{*+}$ (by the Satisfaction Lemma; for any E, $\sigma(E^{*+})^{+} = \sigma^{-1}(E^{*++}) = \sigma^{-1}(E^{*}) = \sigma(E)^{*}$)

$= \mathbb{E}[\![\text{translate}(\sigma,A')]\!]^{*+}$

Case 3: A is inv-translate(σ,A'); we know $\mathbb{F}[\![\tau(A')]\!] \subseteq \mathbb{E}[\![A']\!]^{*+}$

$\mathbb{F}[\![\tau(\text{inv-translate}(\sigma,A'))]\!]$

$= \mathbb{F}[\![\text{inv-translate}(\sigma,\tau(A'))]\!]$

$= \sigma^{-1}(\mathbb{F}[\![\tau(A')]\!])$

$\subseteq \sigma^{-1}(\mathbb{E}[\![A']\!]^{*+})$ (by the inductive assumption)

$= \sigma^{-1}(\mathbb{E}[\![A']\!])^{*+}$ (by the Satisfaction Lemma; for any closed E, $\sigma^{-1}(E^{*+}) = \sigma(E^{*})^{+} = \sigma(E^{*})^{**+} = \sigma^{-1}(E^{**})^{*+} = \sigma^{-1}(E)^{*+}$)

$= \mathbb{E}[\![\text{inv-translate}(\sigma,A')]\!]^{*+}$

Case 4: A is union(A',A"); we know $\mathbb{F}[\![\tau(A')]\!] \subseteq \mathbb{E}[\![A']\!]^{*+}$ and $\mathbb{F}[\![\tau(A")]\!] \subseteq \mathbb{E}[\![A"]\!]^{*+}$

$\mathbb{F}[\![\tau(\text{union}(A',A"))]\!]$
$= \mathbb{F}[\![\text{union}(\tau(A'),\tau(A"))]\!]$
$= (\mathbb{F}[\![\tau(A')]\!] \cup \mathbb{F}[\![\tau(A")]\!])^{++}$
$\subseteq (\mathbb{E}[\![A']\!]^{*+} \cup \mathbb{E}[\![A"]\!]^{*+})^{++}$ (by the inductive assumption)
$\subseteq (\mathbb{E}[\![A']\!] \cup \mathbb{E}[\![A"]\!])^{*+}$
(because $(\mathbb{E}[\![A']\!] \cup \mathbb{E}[\![A"]\!])^{*}$ satisfies $\mathbb{E}[\![A']\!]^{*+}$ and $\mathbb{E}[\![A"]\!]^{*+}$)
$= \mathbb{E}[\![\text{union}(A',A")]\!]^{*+}$

Case 5: A is add-equality(σ,A'); we know $\mathbb{F}[\![\tau(A')]\!] \subseteq \mathbb{E}[\![A']\!]^{*+}$

$\mathbb{F}[\![\tau(\text{add-equality}(\sigma,A'))]\!]$
$= \mathbb{F}[\![\text{add-equality}(\sigma,\tau(A'))]\!]$
$= \mathbb{F}[\![\tau(A')]\!]^{S}$ (for appropriate S)
$\subseteq \mathbb{E}[\![A']\!]^{*+S}$ (by the inductive assumption)
$= \mathbb{E}[\![A']\!]^{S*+}$
$= \mathbb{E}[\![\text{add-equality}(\sigma,A')]\!]^{*+}$

This theorem says that the $\mathbb{F}$ semantics is consistent with the $\mathbb{E}$ semantics, so any fact provable using inference rules which are sound with respect to the $\mathbb{F}$ semantics will be valid.