PUTTING THEORIES TOGETHER TO MAKE SPECIFICATIONS

R.M. Burstall and J.A. Goguen*

Dept. of Artificial Intelligence
Edinburgh University
Edinburgh EH8 9NW

Dept. of Computer Science
University of California at L.A.
Los Angeles, California 90024

order not significant

We have been developing a language in which you can give structured descriptions of theories.

Why are we interested in theories? Because you need a theory to specify a problem before you can develop a program to solve it, whether you intend to develop the program intuitively or to synthesise it mechanically by rule. It only makes sense to say 'We want a program which can invert a matrix' in the context of some theory about matrices and the operations on them such as multiplication.

Why are we interested in structured descriptions of theories? Because people find it very hard to understand anything at all unless they have a well-structured description of it; as for machines, twenty years work in Artificial Intelligence has taught us to beware of letting them loose on an unstructured description.

What would an unstructured description of a theory be like? Imagine 217 axioms in Predicate Calculus telling you how to find your way around SRI, or 217 semantic equations describing the language Klugegol78. Minsky (1975) protests about 'attempts to represent knowledge as collections of separate simple fragments'. No-one could approve of such monsters as these.

Now consider the analogous situation with programs. They are structured by statements, iterations and procedures. For large programs these have proved inadequate (217 LISP functions!), and SIMULA classes, CLU clusters and ALPHARD forms have been devised to ward off the threatened chaos (Dahl et. al 1970, Liskov 1975, Wulf et al 1976). They all introduce abstract data structures by giving the collections of procedures which define the primitive operations on them. They separate the part of the program which implements a structure from other parts which use it but have no concern with its representation. Similarly in AI Minsky's frame notion (Minsky 1975) offers a way of bundling together LISP functions into some meaningful entities. Indeed one reason for the move away from a 'logical' representation of knowledge to a procedural one may be that we have some skill at structuring programs but hardly any at structuring theories.

Our work on theories derives from our attempts to clarify and generalise the above methods of building up programs in terms of abstract data structures. Tackling problem specifications rather than programs turned out to use the same mathematical tools but to be rather less difficult. It is also an area overdue for illumination. The present paper sets forth in an informal way our first, tentative, proposal for a language in which one may describe theories. This language, called 'Clear', is intended primarily as a tool for program specification, but it might also serve to represent knowledge in a machine manipulable form. We have largely worked out the mathematical semantics of Clear, but we have not attempted to implement it.

We will first explain our notion of theory in general terms, then discuss possible areas of application. After this we will describe our theory language and give some simple illustrations of its use.

What we mean by a theory

The notion of theory is a loose intuitive one in mathematics. There should be axioms, rules of inference and theorems, but the language in which they are expressed is open to choice. A popular choice of a formal language would be first order predicate calculus, or more boldly a higher order calculus. Some people, like the predicate calculus programmers (Kowalski 1974), would use a more restricted calculus, say Horn clauses with free variables but no explicit quantifiers. We have chosen an algebraic notion of theory, due to Lawvere (1963), making it many-sorted (Goguen, Thatcher and Wagner 1977) and with provision for errors (Goguen 1977).

A many-sorted algebraic theory is given by naming a set of sorts, a set of operators over those sorts and a set of laws which those operators must satisfy. The laws take the form of equations with free variables but no quantifiers. Since we may introduce truth values as a sort and two no-argument operators (constants) true and false, we can introduce predicates as operators producing a truth value as result (just like LISP).

Here are two examples:-

Vector spaces The sorts are scalar and vector.

The operators are scalar addition and multiplication, scalar zero and one, vector addition, vector negation, vector zero, and vector-by-scalar multiplication. The laws are associativity and commutativity for scalar addition, identity for scalar zero with addition and so on.

GPS (General Problem Solver) The sorts are states, actions, action-sequences, state descriptions, attributes, values and differences. (In GPS attributes of states have values, which give rise to differences between states.)
The operators are (i) apply, taking an action and a state to a state, (ii) descriptionof,

taking a state to a description, (iii) valueof, taking a description and an attribute to a value, (iv) undefined, a constant for a state, and so on.
There are some laws, for example:-
apply(a,undefined) = undefined,
concatenation of action sequences is associative.

Lawvere showed how such a theory description can be taken to denote a more abstract algebraic structure, namely a collection of operators susceptible to 'composition' (substitution) and 'tupling'. This is important because he was able to develop some theory about 'theories' (if you can have a theory about 'groups' you can have a theory about 'theories'), and his work enables us to give a mathematical basis to our language for denoting theories. It is not appropriate to go into thi3 mathematics here, but it is a comfort to us that we have managed to outline a proper semantics for our language; we hope to develop this and write it up soon for publication*

Not only is it relatively easy to reason about algebraic theories, but there is evidence that it is relatively easy to reason within an algebraic theory, indeed that is just the domain which was tackled very successfully by Boyer and Moore (1975) with their LISP theorem prover, subsequently enhanced to deal with many sorts by Aubin (1976).

We already have some encouraging experience of using algebraic theories (but not structured ones) as a specification tool (Goguen 1976, Goguen, Thatcher and Wagner 1977).

The use of algebraic techniques for specifying abstract data types has been studied extensively by Zilles (1974) and by Guttag, Horowitz and Musser (Guttag 1975, Guttag et al 1976) who give examples of program verification using such specifications.

We should remark that although we have chosen to use algebraic theories rather than predicate logic or lambda calculus theories, the methods we have used to combine them are rather general and may well apply to other kinds of theory.

We have not written out the above examples of theories' in full because they would be long and hard to understand; even eight operators and a dozen laws is a lot to swallow in one bite. A mathematics book would scarcely present the concept of vector space without some preparation on semigroups, groups and fields. Indeed most of the structure can be explained by saying that the scalars form a field and the vectors a group. We then have to impose some extra conditions (e.g. commutativity of vector addition) and enrich the structure with an extra operator, multiplication of vectors by scalars, which is distributive, etc.

Similarly GPS becomes much easier to understand if we first describe a state-action system, then say that action-sequences are just strings of actions whose effect is the composition of the effect of the component actions. We can independently enrich the idea of state with attributes and values, saying that descriptions are just-arrays (finite functions) from attributes to values. Only then can we put it all together and introduce the notion of the differences reduced by an operator.

This then is intuitively what we mean by building up a theory in a structured way: any good textbook does it all the time. Luckily Lawvere's notion of the category of theories supplies the mathematical correlate of this informal exposition and enables us to apply known mathematical methods ('colimits') to the task of constructing theories by using other theories as ingredients.

One may view a theory as a natural generalisation of the notion of abstract data type. Such a type is characterised by the operations which create its elements or apply to them. A theory may consist of several such types with the operations between them, thus avoiding the difficulty of arbitrarily assigning an operation from A's to B's to type A or to type B. Analogous to a group of procedures which realise a data-type, as in SIMULA, CLU or ALPHARD, would be a group of procedures which realise a theory. We learned recently that Nakajima, Honda and Nakahara (1977) had also been working on this idea and had designed a programming language to incorporate it. Their use of theories in a programming language is close to what we had in mind. We hope that the mathematical methods we have for structuring specifications can be adapted to give semantics for such a programming language (see our tentative remarks about programs as theory morphisms later on).

Specifications required for -program verification, transformation and synthesis

Program verification has been a continuing concern since McCarthy's classic paper (1963)• Recently there has been considerable interest in synthesising programs from their specifications (Manna and Waldinger 1971, 1975), Dijkstra (1975), Darlington (1975, 1976), one promising method being to take a very naive program as the specification and transform it into an acceptably efficient one (Darlington and Burstall 1976, Burstall and Darlington 1977, Arsac 1977). All of these techniques for obtaining correct programs must start from a specification. Verification, whether by hand or by machine, makes heavy weather even of non-trivial 'text-book' programs and still seems impractical for the much longer programs met with in practice. This comparative lack of success of verification techniques has obscured the fact that for large programs not only are we unable to carry through a correctness proof, but usually we cannot even specify the problem which the program is supposed to solve.* Similar remarks apply to program synthesis.

* For an overview of specification techniques, with many references, see Liskov and Berzins (1977).

There are exceptions. To specify a compiler, and hence verify it, you need to define the source language and the target machine. Scott and Strachey (1971 and subsequent papers) battled valiantly to give us precise semantic definitions of programming languages. Unfortunately, for large languages the specifications are very hard to understand (Robert Milne had one for Algol-68 which he declined to publish on the grounds that no-one would read it). We surmise that a good part of the trouble may be the lack of structure in such a formal definition, the structure that the writer of an informal manual for a language must be very careful to make clear.*

Thus we feel that a better grip on the way to structure the theory in terms of which specifications are made is a prerequisite for raising verification and synthesis techniques above the toy problem level.

Specifying AI problems

In AI research, as in other disciplines dealing with complex programs, there is a tendency to write the program but never get around to specifying the problem. Further any large program must be composed of subprograms, and these cannot be understood without a clear specification of the subproblems they are supposed to solve. Thus better tools for problem specification are a pressing need in AI.

Not only Slould the theories used in specifying these problems and subproblems be well structured, they should also be sufficiently abstract. They should be concerned with the abstract nature of the data and the operations to be performed rather than the particular problem domain or the specific machine representation of the data. For example Walts's (1975) work is about the abstract notion of networks of relations, rather than just about blocks and shadows or about LISP S-expressions; its full utility can only be exploited if this is kept in mind (see Mackworth 1977). Our theory language must be able to handle such abstraction and enable us to hide unnecessary detail.

Representing knowledge within an AI program

AI programs are commonly conceived to embody knowledge, whether as program or as data. Procedural embedding of knowledge may promote efficiency, and it may enable one to use existing program structuring techniques to impose some order on the embedded knowledge. However procedural embedding has disadvantages of inflexibility, and it makes it difficult to incorporate new knowledge, whether input or from inductive learning. Much of the disquiet with knowledge held as data seems to us to stem from its lack of structure, a large collection of axioms or facts unorganised, slowing processing down with irrelevant information.

Again we are exhorted to consider our commonsense knowledge of the world as composed of a large number of micro-theories about particular aspects. But we are left largely in the dark as to how to put these micro-theories together. This question of 'putting theories together' is close to the heart of our concern.

Thus although we cannot yet speak from experience, we very much hope that our theory-building techniques may eventually give some fresh insight into the appropriate organisation of knowledge in AI programs. A way of presenting theories so that people can understand them might help us to see how machines can make use of them.

Theories

We start our exposition of the language Clear by looking more closely at the notion of many-sorted algebraic theory. Our theories will also make provision for errors, like division by zero, but we will defer consideration of this until we have the basic ideas straight.

First we need the notion of a signature, that is a vocabulary of operators with given sorts.

A <u>signature</u> is a set of sort names and a set of operator symbols, each with a given sequence of 3orts for its arguments and a sequence of sorts for its results (possibly more than one result). We write $w : s_1,\ldots,s \to sj,\ldots\ldots,s_n'$ to show that w is an operator with input 30rts $s_1,\ldots,s_m$ and output sorts $s_1',\ldots,s'_n$.*

<u>Example 1 Natural numbers</u>

```
sorts       nat, bool
operations  zero   : -> nat
            succ   : nat -> nat
            iszero : nat -> bool
            true   : -> bool
            false  : -> bool
            not    : bool -> bool
            or     : bool, bool -> bool
```

<u>Example 2 Geometry</u> (a fragment)

```
sorts       line, point, bool
operations  join         : point, point -> line
            intersection : line, line -> point
            colinear     : point,point,point->bool
            true         : -> bool
            false        : -> bool
            not          : bool -> bool
```

A <u>theory presentation</u> is a signature together with a set of equations using the operators of the signature and respecting their input and output sorts. The equations have variables which are implicitly universally quantified.

<u>Example 1</u> (continued)

```
variables  m,n: nat
equations  iszero(zero) = true
           iszero(succ(n)) = false
           not(true) = false
           not(false) = true
```

* Mosses (1977) makes a start in this direction.

* for multi-result operators we could use a syntax like "...a...r... <u>where</u> <q,r> = quotient - and -remainder(m,n)", but we will not need them.

Example 2 (continued)

```
variables p,q,r: point; l,m: line
equations intersection(join(p.q).join(q,r)) = q
          join(p,q) = join(q,p)
          not(true) = false                  etc.
```

A theory is a signature together with a set of equations closed under inference by reflexivity, transitivity and symmetry of equality and by substitution. For example 'false = iszero(succ(succ (zero)))' is an equation in the theory defined by the presentation above.

Thus each theory presentation gives rise to a theory but the same theory can be presented in more than one way by choosing different sets of 'axiom' equations to generate it, (The notion of Theory is more basic than Theory Presentation in the sense that one would like to talk about the theory of groups,for example,irrespective of any particular axiomatisation of it,)*

The interpretations of a theory are algebras, where an algebra is a collection of sets, one for each sort, with a function over these sets assigned to each operator of the theory. These functions must obey the equations of the theory. In practice for theories containing bool we will only be interested in 'consistent' interpretations in which true ≠ false.

Theory-building operations

In the last section we were just writing down theories explicitly one at a time. As soon as they get to be interesting they become incomprehensible. We wind up with a large set of equations that no-one can understand and which are almost certainly wrong. So we must build our theories up from small intelligible pieces. For this we need

(i) the ability to write (small!) explicit theories, as above, thus

```
theory sorts ...
       opns  ...
       eqns  ... endth
```

(ii) four operations on theories, combine.enrich. induce and derive, which enable us to build up theory expressions denoting complex theories.

We will explain these operations informally, using examples.
First we define two explicit theories which will be useful

The theory NatO

```
theory sorts nat
       opns O    : -> nat
            succ: nat -> nat
       eqns                    endth
```

Technically we should call this a 'theory with signature' rather than just a 'theory' because the choice of a particular set of operators is irrelevant to the abstract notion, just as is the choice of particular axioms (see Lawvere 1963 or Manes 1976).

The theory BoolO

```
theory sorts bool
       opns true : -> bool
            false: -> bool
               ¬: bool -> bool
               ∧: bool, bool -> bool
       eqns  ¬ true    = false
             ¬ false   = true
             false ∧ p = false
             true ∧ p  = p       endth
```

(Strictly we ought to insert 'variable p: bool' before 'eqns'. but we will assume that undeclared single letter identifiers are variables; their type will be obvious. We will also allow ourselves to use traditional infixed symbols like A.)

Combine

This operation is dull but plays its part in the larger scheme of things. We simply take two theories and add them together. The sorts of the resulting theories are the union of the sorts of the given theories, the operators are the union of their operators, and the equations are the union of their equations. We use the + sign for the combine operation.

For example BoolO + NatO could be written explicitly as

```
theory
   sorts bool, nat
    opns true : -> bool
         false: -> bool
            ¬ : bool -> bool
            ∧ : bool, bool -> bool
             O: -> nat
          succ: nat -> nat
   eqns ¬ true    = false
        ¬ false   = true
        false ∧ p = false
        true ∧ p  = p       endth
```

We will see later that combine does not necessarily produce the disjoint union of two theories; it allows for sharing of common subtheories.

Enrich

Suppose that we want to build up a useful theory of the natural numbers,including operators for ordering and for equality. The operators < and eq belong neither to BoolO nor NatO, but they can be added to their combination to obtain a new theory

The theory Nat1

```
enrich BoolO + NatO by
  opns ≤: nat, nat -> bool
      eq: nat, nat -> bool
  eqns              O ≤ n = true
            succ(m) ≤ O = false
     succ(m) ≤ succ(n) = m ≤ n
               eq(m,n) = m ≤ n ∧ n ≤ m  enden
```

This whole expression denotes the new enriched theory.

In general one may add new sorts as well, thus

enrich . . . . by
aorta . . .
opns ...
eqns ... enden

The enriched theory inherits all the sorts. operators and equations of the original one. The new operators can have as input and output sorts the original sorts or the new sorts.

Given the enrichment operation we could just start with the empty theory, say Φ , and regard any explicitly written theory as an enrichment of Φ .

Induce

The theory Nat1 has an operator eq which satisfies not only the three equations explicitly given but also by substitution, the equations

eq(0,0) = true
eq(succ(0),succ(0)) = true
eq(succ(succ(0)),succ(succ(0))) = true

and so on. But the equation

eq(n,n) = true

is not inferable by substitution and is not part of the theory.

We would like to be able to extend the equations of a theory so that an equation holds for a variable, n, if it holds for every equation obtained by substituting a constant, that is a variable-free term, for n. So we have an operation induce on a theory which does just this.* Thus in the theory induce Nat1, which we will call Nat, the equations

eq(n,n) = true	reflexive
eq(m,n) = eq(n,m)	symmetric
eq(l,m) $\wedge$ eq(m,n) $\wedge\neg$ eq(l,n) = false	transitive

all hold.

To find equations holding in a theory created by induce we may prove by induction on the structure of terms, using the equations of the original theory, that a certain form of equation holds for every variable-free term; we may then assert that the form with a variable must hold in the induced theory. (Such methods of inference will not in general be complete however.)

In Bool0 every variable-free term is equivalent to true or false by using the equations for $\neg$ and $\wedge$, so it suffices to show that an equation holds for both true and false to see that the general equation holds in induce Bool0. Induction here just amounts to case analysis. For example $\neg\neg$ true = true and $\neg\neg$ false = false both hold in Bool0, so $\neg\neg$ p = p holds in induce Bool0. We call induce Bool0 simply Bool.

Derive

The third operation derive enables us to take a more complex theory than we need, perhaps built by combining and enriching some familiar theories, and then to select out from it just those sorts and operations which we require. For example, if we only need eq from Nat (and not < or zero or succ) we may write

The theory Natequal

derive
sorts element, bool
opns equal, true, false

from Nat by
element is nat
bool is bool
equal is eq
true is; true
false is, false endde

This denotes a new theory with two sorts and three operators. The equations governing the new operator, equal, are not specified. Indeed we have only given the signature of the new thonry. but the properties of equel are given implicitly by the correspondence equal is eq. Notiee- thet the equations for eq use the auxiliary operator $\leq$. In general an operator of the derived theory may correspond to a λ defined operator of the original theory, thus "plus2 is λ n.succ(succ(n))".

For brevity we will omit pairs of the form 'x iS x', such as 'true is true'. Also if we already have a theory T we may write'signature T' to denote its signature.

We use derive when we want to define a theory in terms of some other theories with which we are already familiar but which, taken together, are too rich for our purpose. We are making a construction from familiar mathematical objects, but the details of the construction are discarded in the more abstract result. An analogy would be the construction of the natural numbers in terms of the sets $\emptyset$, $[\emptyset]$,$\{\emptyset$, $[\emptyset]\}$,... . The operations we need on the natural numbers are, say, zero, successor and <, Other possible operations on these sets, such as cartesian product of two sets, are not meaningful for numbers. In programming work it is well-known that the operations on an abstract data type are defined in terms of those on a more concrete type which represents it; but at the abstract program level the more concrete operationsshould not be available.

Procedures for theory-building

We have some primitive operations on theories. The next step is to enable the user to define his own operations using these. For this we introduce procedures - no self-respecting language could be without them.

We shall introduce the simplest mechanisms which provide tolerably convenient facilities, namely

(i) Theory constants, enabling us to give a name to a theory

(ii) Theory procedures, taking theories as their parameters and producing a theory as a result. Their bodies use the primitive operations already defined and may call other theory procedures (but we eschew recursion])

* Technically, induce T, is the theory of the initial algebra of T7

(iii) Local theory definitions, permitted in the bodies of theory procedures, thus:-
<u>let</u> T = ... in ....

These facilities would be very similar whatever domain we were working in. Let us introduce them in the familiar domain of numbers and truth-values as a warming up exercise. We will assume the primitives * (multiply), /(divide) and if... <u>then</u>...<u>else</u>...

A constant declaration, .just assigning a fixed value to an identifier pi, would be

<u>constant</u> pi - 22/7

A procedure declaration for a procedure producing a number as result would be

<u>procedure</u> f(x: number, b: boolean) =
<u>if</u> b <u>then</u> pi * x <u>else</u> 0

A procedure declaration with an auxiliary local variable z would be

<u>procedure</u> *g(y:* number) =
<u>let</u> z = f(y * y, true) in s * z * z

Now if we evaluate g(2), *7,* takes the value f(2*?, true), i.e. (22/7)*2*2, and g(2) is the cube of this value.

Now the same definitional methods and syntax will apply to theories, using the theory-building operations instead of *,if-then-else etc. (we do not need conditional expressions for theories). We will need the type specification for parameters, as in x: number, since it turns out that there is notion of type for theory parameters.

There is one major difference however between numbers and theories as a domain. <u>Two theories may share some common subtheory</u>. For example the theory of natural numbers has <u>Bool</u> as a subtheory since it has predicates like <, so does the theory of character strings which has predicates like <u>isempty</u>. We may want to enrich the combination of these two theories to allow operators like

length: string -> nat

In this new theory we want the truthvalues produced by < to be the same as those produced by <u>isempty</u>. We want one copy of <u>Bool</u> not two. Thus <u>Bool</u> is a shared subtheory.

Where have we met this situation before? In LISP or any other language with pointers we have shared substructures. Let us see how the definitional mechanisms we have introduced would behave with shared lists. We will then be ready to tackle theories. We need atoms "A, "B etc., the list constructor cons, the selectors car and cdr and, crucially, the predicate eq which tests whether two lists are equal in the sense of starting with the same list cell, not just having the same shape. We will not use LISP syntax, but we intend LISP semantics, passing parameters by pointer than than copying the list structure. Let us look at examples (the intended values of the expression are given on the right)

(i) eq(cons("A,"B),cons("A,"B)) .... false

(ii) <u>constant</u> ab = cons("A,"B)
eq(ab,ab) ... true

(iii) <u>constant</u> ab = cons("A, "B)
eq(cons(ab,"c) , cons(ab," C)) .... false
eq(car(cons(ab,"c)) ,car(cons(ab, "c))) .... true

(iv) <u>procedure</u> P(l; list) = eq(l,l)
P(cons("A,"B)) .... true

(v) <u>procedure</u> P(l: list) -
let m - cons(l,"C) in e1(m,m)
P(cons(A,"B)) ..... true

Thus every use of a constant, parameter or local variable refers to the <u>same</u> list, to within eq, but writing down an explicit expression twice using cons refers to a different list (i). Two different lists can share a common sublist (iii).

Now our theory-making operations, explicitly writing a theory and enriching one, behave ju3t like cons. But by using theory constants or variables we can arrange for the theories we create to contain shared sub-theories- One of our main technical problems was to make this remark precise, since for theories we do not have the addre3s/value storage model as we do for data structures. In fact the category theory ideas of "diagram" and its "colimit" give a rather general notion of shared substructure. We hope that the reader's intuition using the LISP analogy will give him a reasonably good grip on the intended semantics. Those who wish to know the precise method for determining the denotation of our specifications must await the mathematical semantics in the paper which we are preparing.

<u>The specification language Clear</u>

We will call our proposed specification language "Clear". A specification in Clear consists of a sequence of constant and procedure declarations followed by an expression. The expression denotes a theory, not explicitly but using the theory-building operations and the declared constants and procedures.

Clear can be viewed as a language for communicating a precise specification of a problem to people, such as programmers- It could also be implemented on a machine so that 'evaluation' of a Clear specification yielded an explicit representation of the theory it denotes- A more useful implementation however would be to link Clear to an equatinnal theorem prover which would try to prove that a given equation held in this theory without producing the theory explicitly. Or it could be incorporated in a system which tried to show that a given program implemented some operations of this theory. This raises interesting, but still unanswered questions, about the relation between specification structure and program structure.

We will explain Clear by example. Let us start by building up the theory <u>Nat</u> of natural numbers using the <u>constant</u> facility and the <u>let</u> facility, thus repeating in succinct form the more fragmentary development of <u>Nat</u> above. We start with <u>Bool</u>, the theory of truth values.

(const is short for constant)

```
const Bool =
   induce theory
          sorts bool
          opns true, false: -> bool
                         ¬ : bool -> bool
                         ∧ : bool,bool -> bool
          vars p: bool
          eqns ¬ true  = false
               ¬ false = true
              true ∧ p = p
             false ∧ p = false  endth
```

The constant Bool now denotes the theory of truth values. Since we applied the induce operation it also has equations such as ¬¬ p = p obtained by case analysis.

```
const Nat =
   induce let Nat0 =
              theory sorts nat
                     opns   0: nat
                         succ: nat -> nat endth
          in enrich Nat0 + Bool by
             opns ≤, eq: nat, nat -> bool
             vars m,n: nat
             eqns 0≤n = true
                succ(m) ≤ 0 = false
                succ(m) ≤ succ(n) = m≤n
                eq(m,n) = m≤n ∧ n≤m   enden
```

The constant Nat now denotes the theory of natural numbers. It is built up by first making a local definition of the simple theory NatO with just zero and suec. We then combine this with Bool, enriching the combination with extra predicates < and eq. finally induce applied to the whole expression ensures that the theory contains general equations like eq(m,m) - true-

Procedures in Clear

We often build one theory on top of another. Suppose for example that we have some partially ordered set, then we can form strings from its elements and define the predicate 'ordered' for strings- This is just what we would have to do if we wanted to specify some sorting task. A theory of ordered strings can be developed for any partially ordered set (poset) of elements and the latter can be regarded as a theory parameter (compare Form parameters in ALPHARI)). We can write a procedure using the theory-building operations to construct the theory of ordered strings from this parameter. Now we can apply this procedure to any theory which has a 'less than or equal' operator satisfying the reflexivity, transitivity and antisymmetry laws, for example the theory Nat- Thus the procedure can only accept as parameter a certain sort of theory; we had better call it a 'meta-sort' to avoid confusion with the sorts within theories. This meta-sort is itself a theory, in this case the theory of partial orderings-

A degenerate example would be a theory procedure which can take any set as parameter and does not need any operators, for example the procedure which, given a set of elements, produces the theory of 3tring3 of those elements. The nieta-sort here is the trivial theory with one sort and no operators.

const Triv = theory sorts element endth

The theory procedure to make strings is then (proc being short for procedure)

```
proc Strings (X: Triv) =
     induce enrich X by
            sorts string
            opns unit: element -> string
                   Λ : -> string
                   . : string, string -> string
            eqns Λ.s = s
                 s.Λ = s
               (s.t).u = s.(t.u)  enden
```

As an example we can apply this procedure to Nat to get strings of natural numbers, but we need to associate the sorts and operators of the meta-sort (Triv)of the formal parameter with those of the actual parameter (Nat), that is we need a sort to sort function and an operator to operator function just as in derive. We write these in brackets after the actual parameter, thus

Strings (Nat [element is nat])

We may omit pairs of the form 'x *is.* x'-

The actual parameter theory must include all the equations of the meta-sort theory as rewritten under this operator to operator function. We must prove this for every procedure application- Unlike conventional type checking it is not in general decideable.

Now to do ordered strings we need the theory of partial order for use as a meta-sort.

```
const Poset =
      enrich Bool by
       sorts element
        opns ≤, eq: element, element -> bool
        eqns x ≤ x = true
            x ≤ y ∧ y ≤ z ∧ ¬ (x ≤ z) = false
                                    (transitivity)
              eq(x,y) = x ≤ y ∧ y ≤ x  enden
```

We can now write the procedure for ordered strings. We use the procedure Strings defined above.

```
proc Orderedstrings (P: Poset) =
   induce enrich Strings (P) by
          opns ordered: string -> bool
          eqns ordered(Λ) = true
          ordered(unit(x)) = true
          ordered(unit(x).unit(y)) = x ≤ y
          ordered(s.t.u) = ordered(s.t)
                            ∧ ordered(t.u) enden
```

Use of a theory as a meta-sort is rather distinct from its use in defining some data structure such as natural numbers- It enables us to state the presuppositions for some task which we wish to specify, and we are interested in any interpretation of the theory rather than some particular canonical one.

Shared subtheories

We observed already that just as two lists may share substructure so may two theories; this in accomplished by having the same variable appear in both the expressions denoting these theories. The details, which follow, are a little technical and may be skipped if desired.

Suppose that we have a theory variable T ei ther a formal parameter or bound by a let," Then the theory "enrich T by...enden" contains this theory T as a subthe~ory, and so do "T+..." and "Induce T". The theory ''derive aigrigture T1 from T? by, .Tendde" contains T as a subtheory if TInjontains it. Should T2. also contain T_ as a subtheory then the "..." had better map its operators identically (if they both contain Bool, "true is false" would not be welcome). Now it we combine two theories T1_ and T2 which both have T as a subtheory then TT+ T2 "only contains T once". The same rules hold IT T Ts not a variable but is introduced by "const T = ...". All this enables us to have Bool, say, as"a subtheory of several theories without proliferating many copies of it. Sometimes we do need a fresh copy of a theory T_, so we let ''copy T" denote one, to save writing it out again explicitly.

When we apply a procedure P to an argument _T the result always includes T just as if we had - written P(T) + T instead of P(T). For example String (Nat) is a theory whic has not only string operators but also the operators like succ defined in Nat, Of course when we are writing the definition of the procedure String these latter operators are not available; the importance of such 'insulation' mechanisms has been pointed out by Wulf and others.

Errors and condi tionals

Before going on to look at examples of specifications written in Clear we will incorporate two USEFU features: errors and conditionals.

Some applications of ar operator to its arguments will, not give a meaningful result, for example dividing by zero or popping an empty stack. Thus we need to consider errors, a topic which is often glo33ed over in algebraically oriented work, but whose proper treatment is essentialfor a realistic specification language. It is important too that the different levels of abstraction provided by our language should not become confused as soon as an error is encountered; we do' not want a stack underflow to produce an error message 'array subscript out of bounds'. Gtoguen (1977) studies this topic in depth, defining error algebras and error theories. We will confine ourselves to an informal, glance at error theories.

The idea is to extend each sort by a set of error elements of that sort, and to have error operators which produce these elements. Thus the theory of stacks might have an error operator

underflow: -> stack

and the theory of arrays might have an error operator

notdefinedfor: index -> value

meaning that there is no value for this index in the array. The term 'notdefinedfor(7)' would serve as an informative error result.

To say when an error occurs or to equate two different error expressions we need to use error equations, thus

```
pop(empty)      = underflow
pop(underflow) = underflow
```

We call the non-error elements of a 3ort "OK elements", the non-error operators "OK operators" and the non-error equations "OK equations". Now we can write a presentation of a theory with a set of erroropns in addition to the previous (OK)opns, and a set of erroreqns in addition to the previous (OK)eqns. An interpretation of such a theory is an 'error algebra', that is an algebra some of whose elements are designated error elements. This designation must obey the following rules:-

(0 Error operators always produce an error element.
(?) OK operators produce an error element if any of their arguments is an error element.

Now for an error algebra to satisfy the theory an OK equation or an error equation does not have to hold for all values of the variables. Only the following must be the case

(1) An OK equation must hold if both sides evaluate to an OK element.
(2) An error equation must hold if either side evaluates to an error element.

For example

```
theory sorts nat
       opns zero: -> nat
            succ: nat -> nat
            pred: nat -> nat
      erroropns neg: -> nat
      eqns pred(3succ(n)) = n
      erroreqns pred(zero) = neg
                succ(neg)  = neg
                pred(neg)  = neg  endth
```

Further examples, stack, array and symbol table, are given later.

It often happens that two expressions are equal only under a certain condition, thus

f(x) - *g(x)* if p(x)

Now we can permit such a conditional equation by regarding it as an abbreviation for

if (p(x),f(x),g(x)) = g(x)

where 'if' is the usual conditional operator defined for each type by the equations

if (true,y,z) = y   if (false,y,z) = z

Conditional axioms have been studied using a different approach by Thatcher, Wagner and Wright (1977).

Notice that the fact that our OK equations automatically do not apply to error values often saves us from adding a condition such as "... If s # underflow".

Notation

We should mention some small further points about notation. If we are naming sorts in a context where several theories are present, the same sort name, s, may appear in two different theories, T1 and T2, making a reference to s ambiguous. We then simply refer to "s of TV" or "s of T2.". A similar notation "f of TT"" will disambiguate operators.

Often a theory has a particular sort which is so to speak its raison d'etre, for example sort nat in theory Nat (even though Nat also has sort bool). To enable us to distinguish such a sort we define the principal sort of a theory to be the first sort mentioned in its definition, thus in 'theory sorts s,t ... endth' s is the principal sort and similarly in 'enrich T by sorts s,t ... enden'. Now a helpful convention is to allow the theory name, in lower case, to denote its principal sort. Also when we specify the correspondence between sorts in a derive operation we may omit a pair 's is t' if s and t are the principal 3orts of their respective theories; similarly for the [... notation used for actual parameters of procedures, thus 'Strings (Nat)' is acceptable for 'Strings (Nat [element is nat])'.

Examples of specification

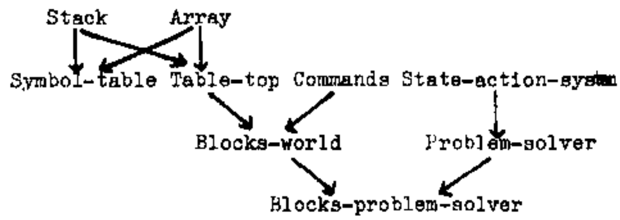
We will give two illustrations to show how Clear can be used to build up theories from pieces in a systematic way:-

(i) a theory to specify a symbol table such as one might need in an Algol compiler (an example given by Guttag et al_ 1976)

(ii) a theory to specify a problem solving system for a two dimensional blocks world.

These are, of course, rather small, simple examples, but we hope that they are .just complex enough to give the reader some idea of the modular structure that we wish to see in specifications. We hope the reader can grasp this structure without poring over *every* equation. The whole Clear description denotes a theory which does not itself have this structure, sc that the implementer would be at liberty to organise his program in some other way..,3 (Just as we might describe the number 19683 as *3,* but you are free to store it in the machine in any way you like, 3uch as binary.) Indeed by using derive we 'throw away' many of the operators introduced in our Clear description of the theory, so that they do not appear in the final theory and need have no corresponding procedures in the program which implements it. For example we describe a symbol table in terms of a stack of arrays, because stack and array are familiar concepts, but our specification does not demand that it be implemented in this way.

Here is our plan of campaign showing the main procedures or constants we will define and which other ones will use them.



We will use the theory Set of sets without defining it. The definition should be fairly obvious. We will also use expressions of the form $\{f(x): x \in X \wedge p(x)\}$ instead of defining such sets by explicit equations.

Stack

Since we can put any kind of element on a stack we take as a parameter theory a trivial theory, one with a single sort and no operators. This describes the 'values' which go on the stack. The operators, such as push and pop, are well-known. Notice that no 'side-effects' are allowed. We explicitly produce a new stack from push and pop.

```
proc Stack (Value: Triv) =
 induce enrich Value + Bool by
              sorts stack
                opns nilstack: -> stack
                     push    : value,stack->stack
                     empty   : stack -> bool
                     pop     : stack -> stack
                     top     : stack -> value
              erroropns underflow:  -> stack
                        undef    : -> value
              eqns empty(nilstack) = true
                   empty(push(v,s))= false
                   pop(push(v,s))  = s
                   top(push(v,s))  = v
              erroreqns pop(empty) = underflow
                        top(empty) = undef
                        pop(underflow) = underflow
                                          enden
```

Array

We define arrays with any kind of element as indices, not just integers. However the indices must have an equality relation defined over them in order for us to 'look up' indices in the array, so we have a parameter theory of meta-sort Id., a theory of identifiers with one sort besides bool and an equivalence operator == over that sort. We write the array access function as a[i] instead of, say, get(a,i).

```
const Id =
   enrich Bool by
     sorts identifier
      opns ==: identifier,identifier -> bool
      eqns i==i = true
           i==j = j==i
           (i==j) ∧ (j==k) ∧ ¬ (i==k) = false enden
```

```
proc Array (Index: Id, Value: Triv) =
 induce enrich Index + Value by
      sorts array
       opns nilarray: -> array
            put      : index,value,array->array
            ...[...]: array,index -> value
            in       : index,array -> bool
    erroropns undef: index -> value
     eqns put(i1,v,a)[i]    = v if i==i1
          put(i1,v,a)[i]    = a[i] if ¬ i==i1
          in(i,nilarray)    = false
          in(i,put(i1,v,a)) = i==i1 or in(i,a)
          put(i,v,put(i1,v1,a))=put(i1,v1,
                                  put(i,v,a))
                              if ¬ i==i1
    erroreqns nilarray[i]  = undef(i)  enden
```

Symbol table

A compiler needs to maintain a symbol table relating each identifier to a value such as a machine address or an address plus a type. In an Algol-like language with blocks each block introduces new identifiers which may or may not have occurred before. It associates new values with them, and these override any previous values until the end of the block is encountered and the table reverts to its prior state. Thus we need a theory with sorts: symbol, value, table; it has operators: nilst - an empty table, extend - used to mark entry to a new block, put - to add a symbol value pair, get(written-,. [...]) - to retrieve a value, contract - used when the end of the block is reached. Guttag e,t al_ (1976) have already given an equational specification of a symbol table as an abstract data structure. In contrast to their direct specification we will build up ours from the familiar concepts of stack and array, then use derive to extract just those operations which are required for a symbol table.

```
proc Symtab (Symbol: Id, Value: Triv) =
 let Table = Stack (Array (Symbol, Value)) in
 let Table1 = enrich Table by
     opns extend: table -> table
          putst : symbol,value,table -> table
          ...[...]: table,symbol -> value
          nilst : -> table
     erroropns undef: symbol -> value
     eqns extend(t) = push(nilarray,t)
          putst(s,v,t)=push(put(s,v,top(t)),pop(t))
          t[s] = top(t)[s] if in(i,top(t))
          t[s] = pop(t)[s] if ¬ in(i,top(t))
   erroreqns underflow[s] = undef(s)   in
 let T = enrich Symbol + Value by
     sorts table
      opns nilst : -> table
           extend: table -> table
           putst : symbol,value,table -> table
           ...[...]: table,symbol -> value
           contract: table -> table
   erroropns undef: symbol -> value  enden in
 derive signature T from Table1 by
           nilst is nilstack
           contract is pop  endde
```

Tabletop and Blocks World

Now let us specify a very crude model of a set of blocks on a tabletop together with some commands for moving them. We will stick to two dimensions and assume square blocks all of the same size. We can do this in terms of a one-dimensional array indexed by places on the table, each element of the array is a stack of blocks. We enrich this array of stacks theory with some extra operations: create an empty array of stacks, put a block on the stack at a given place, move a block from the stack at a place onto the stack at another place. We now use derive to get rid of the unwanted operations on stacks and arrays, just retaining these operations on an array of stacks, which we rename a tabletop. We do however need an equality for tabletops, because later we want to do problem solving and see whether we have the required goal tabletop. For this we use a theory procedure Stackeq (Value: Id) of stacks with equality (==: stack,stacic -> bool). Its definition from Stack (Value: Triv) by enrichment is left as an easy exercise. Similarly for Arrayeq (index: Id, Value: Id).

```
proc Tabletop (Block: Id, Place: Id.) =
 let Stackofblocka - Stackeq (Block) in
 let ArravofstacTcs" = Arraveq (Place,StackofblocksX"
 let T = enrich Array of stacks by
        opns empty: -> arrayofst
             put: place,block,arrayofst->arrayofst
             move: place,place,arrayofst->arrayofst
        erroropns error: -> arrayofst
        eqns empty[p] = nilstack
             put(p,b,a) = put(p,push(b,afp]),a)
             move(p,p',put(p,b,a)) - put(p'b.a)
        erroregns move (p,p,a)=serror if isemptv(a[p])
                                         enden in
 derive signature enrich Block + Place by
   sorts tabletop
    opns empty: -> tabletop
        put  : place,block,tabletop -> tabletop
        move : place,place,tabletop -> tabletop
        =-   : tabletop,tabletop -> bool
   erroropas euror: -> tabletop
  from T by. tabletop is, arrayofst  endde
```

The problem solver will seek a string of actions to transform one tabletop to another.. To provide these actions we define some commands, just expressions of the form "makemove(place1, place2)" using an operator "makemove" with no equations (like succ for numbers). Now we can define a dynamic Blocks World, in which you can execute commands to change the tabletop,

```
proc Commands (Place: Triv) =
 theory sorts command
        opns makemove: place,place -> command
                                        endth

proc Blocksworld (Block: Id, Place: Id) =
  enrich combine (Tabletop (Block), Place,
                  Commmands Place by
     opns execute: commands, tabletop -> Tabletop
     eqns execute(makemove(p,p1),t) = move(p,p1,t)
                                        enden
```

State-action system and Problem Solver

Quite separately from the Blocks World, but later to be combined with it, we define a Problem Solver theory for some arbitrary system with states and actions. First we define the state-action system alone with just these two sorts, then we have a procedure Iterate to enrich any state-action system to give the" effect of a whole string of actions. A problem solver is then defined for such a system, with an operation solve which must attain any reachable set of goal states- Note that we do not say how solve is to be programmed, just specify its desired result.

```
const State-action-system =
 let State = copy Id in let Action = copy Id in
   enrich State + Action + Set (Action) by
     opns do: action,state -> state
          acts: -> set(action)
     erroropns error: -> state  enden

proc Iterate (Sas: State-action-system) =
 let Action-string=String(Sas[element is action])in
 enrich Sas+Action-string by
   opns do: action-string,state -> state
   eqns do(nil,s) = s
        do(as.unit(a),s) = do(a,do(as,s))  enden


proc Problem-solver (Sas: State-action-system) =
 enrich Iterate (Sas) + Nat by
   opns reachable: nat,state -> set(state)
        solve    : nat,state,set(state) ->
                                      action-string
   erroropns error: -> action-string
   eqns reachable(0,s) = { }
        reachable(n+1,s) = {do(a,s1) | a ∈ acts
                                 ∧ s1 ∈ reachable(n,s)}
        do(solve(n,s,S),s) ∈ S = true
   erroreqns solve(n.s,S) = error
                           if reachable(n,s)∩S=={}
                                              enden
```

Blocks World Problem Solver

We now put this all together by deriving the required operations for a state-action system from the Blocks World, and applying the theory producing procedure Problem-Solver to it. The resulting theory specifies the notion of solving a problem for our Blocks World, that is finding a sequence of suitable moves to get from one state to a specified set of states. (In practice we would have to add extra operators to describe the start and goal states.) We choose to represent blocks and places by natural numbers, but we leave as a parameter the set of natural numbers determining just which places are involved.

```
const Setofnumbers = enrich Set (Nat) by
                          opns nset: -> set(nat)
                                            enden

proc Blocks-problem-solver (S: Setofnumbers) =
 let Sas = derive signature State-action-system
            from Blocksworld(Nat,Nat) + S by
                state is tabletop
                action is command
                acts is {move(p1,p2): p1 ∈ nset,
                                      p2 ∈ nset}
                  do is execute  endde in
 Problem-solver (Sas)
```

Some open questions

A number of questions arise from these examples.

(i) The language pays for the extra structure and localness by being rather cumbersome. Is this inevitable? We tried to moderate the longwindedness by 3ome conventions, but feared to sprinkle too much 3Ugar lest the reader lose sight of the basic mechanisms.

(ii) Should we distinguish two kinds of enrichment (a) adding new sorts and operators and equations about them, but without constraining existing operators further, (b) imposing further equations on the existing operators?

(iii) Could we improve on the rather clumsy way sharing is indicated in derive?

(iv) The induce operation is rather different from the others, a little mysterious. We stuck it in whenever we were talking about a particular data structure. Could it be inserted more systematically? Perhaps we should distinguish between theories used as metasorts, which generally do not need induce, and other theories, which generally do. Does induce allow us to make all the inductive inferences we need?

(v) Is our transfer of the LISP sharing paradigm to theories the best approach? Can we make good our claim to understand its semantics?

Programs and theory morphisms

In this section we discuss in a tentative way how programs, as opposed to specifications, might fit into our algebraic framework. For this we will need to define a 'morphism' between theories, which represents one theory in another. (The theories and their morphisms form a category, Lawvere 1963). The idea is that a program is essentially a means of representing one theory (the specification) in another theory (the machine), that is a morphism from one to the other.

We can often represent operators of one theory by operators of another, to be precise by derived operators of the other theory. By a derived operator of a theory we mean one which can be expressed in terms of the primitive operators. In a theory with primitives 'not' and 'and' the operator

$$\lambda xy.\, \neg(\neg x \wedge \neg y)$$

is a derived operator ('or'). In general we may build any term in the primitive operators using suitable variables, using the familiar λ notation to bind these variables. These operators include miliary ones, that is constant terms. An operator may be represented by more than one derived operator of the other theory. Since our theories may involve several sorts we must also represent each sort of the first theory by a sort of the second.

Now the operators of the first theory obey certain equations, so naturally the same equations must be true of the corresponding derived operators of the second theory.

We call such a connection between two

theories a theory morphism. Here is the definition.*

A <u>theory morphism</u> from a theory T to a theory T' is

(i) A function f from the sorts of T to the sorts of T. We write s is s' to mean $f(s) = s^1$.

(ii) A function g from the operators of T to non-empty sets of derived operators of T', such that any equation of T gives rise to an equation of T' when each operator ω of T is replaced by any operator in $g(\omega)$. The input and output sorts of an operator in $g(\omega)$ must be the f-images of those of ω. We write ω <u>is</u> ω' to mean $g(\omega)$ - ω'.

By the obvious extension, the* theory morphiam maps each derived operator of T to a set of derived operators of T'; this holds in particular for nullary operators i.e. constant terms.

Consider for example Id., "the theory of identifiers with an equivalence operator, and <u>Nat</u> the theory of natural numbers. We can define a morphism from <u>Id</u> to <u>Nat</u> by

```
sorts identifier is nat
      bool       is bool
 opns ==         is {eq}
      true       is {true}    false is {false}
        ¬        is {¬}         ∧   is {∧}
```

Suppose that we enrich <u>Nat</u> with a multiplication operator to get, say, <u>Natmult</u>. Then we cou have a morphism from <u>Bool</u> to <u>Natmult</u>

```
sorts bool is nat
 opns false is {0,2,4,...}
      true  is {1,3,5,...}
         ¬  is {succ}
         ∧  is {*}
```

Representing sets by strings and stacks by array-index pairs are other well-known examples.

As a matter of fact such theory morphisms play an essential role in our mathematical semantics for Clear. But here we are concerned with their connection with programs. It seems that if we restrict ourselves to an applicative language (without assignment) our theory morphisms are the mathematical correlate of a SIMULA class, CLU cluster or ALPHARD form, with the theory T playing the (generalised) role of the newly defined data type and the theory T' being the existing data type used to represent it. The derived operators in the morphism from T to T' are the procedures in the class, cluster or form declaration.

We do need one generalisation however since in the programming case the procedures may well be recursive. Fortunately Wright, Thatcher, Wagner and Goguen (1976) have defined a notion of rational theories and their morphisms** allowing recursively derived operations (not just λ but recursion too); this seems to model the real programming situation (always provided that we regard an imperative program as a notational variant of an applicative one!).

Now we see that a specification is just a theory, a machine (or more abstractly the primitive operators and sorts of a programming language) is another theory, and a program to realise the specification is just a (rational) morphism from the specification theory to the machine theory.

Of course we should not describe this morphism in an unstructured way, indeed there should be a programming language analogous to the specification language Clear, but describing morphisms not theories.** This would be the correlate of SIMULA etc. or more closely of the iota language of Naka.jima et al, and of Parnas' (197?) method of programming with modules. We have worked on such a language but decided to first get straight the rather easier case of a specification language.

How would the structure of such a program relate to the structure of the specification which it implements? The degree of closeness would be up to the implementer, but it would be natural to use the various theories defined for specification purposes to define the task of subparts of the program. In general one would expect the specification would be simpler than the program, and to specify parts of the program one would need to elaborate the theories used in the specification with new sorts and operators. For example one might decide to use the GPS method to solve the blocks world problem, and one would have to enrich the state-action theory with new sorts like 'difference' and operators like 'reduces'.

A speculative conclusion: <u>the main intellectual task of programming is elaborating the theories which describe all the concepts used in the actual program</u>. Writing the code (defining the morphisms) is a much more humdrum business.

Ah well! This is all delightfully vague and a great deal of work needs to be done. But it does promise to be interesting.

Conclusions

The main point of this paper is that it <u>is possible</u> to specify complex tasks provided that we do not try to write the specifications in an unstructured way. Our particular language proposal is only important in bringing into fbcus the problem of devising structured descriptions of specifications and suggesting the kind of operations which should be used to build them up. The basic ideas developed for data abstraction in programming languages should guide us in this task, and we firmly believe that the mathematical ideas about the category of theories can help us to grasp the rather deep concepts involved-

* Our theory morphisms are different from Lawvere's which represent an operator by a single derived operator.

** We would also need rational theories to make Clear deal properly with infinite data, such as infinite trees, defined inductively.

** We have a base for such development in the equational languages we have already implemented OBJ (Goguen and Tardo 1977) and NPL (Burstall 1977).

Acknowledgements

We owe a deep debt to the pioneers McCarthy, Landin, Eilenberg, MacLane and Lawvere, also to James Thatcher, Eric Wagner and Jesse Wright at IBM and to John Darlington at Edinburgh (now London) for long and educative collaboration. Many other colleagues have been very helpful especially Patrick Cousot (Grenoble), John Reynolds (Syracuse/Edinburgh), Gordon Plotkin, David MacQueen and Jerald Schwarz (Edinburgh) and Joseph Tardo (Los Angeles). J.A.G. wishes to thank Saunders MacLane *for* initiating him into category theory, and William Lawvere for help on the way. R.M.B. wishes to thank members of IFTP Working Group 2.3 for much education and encouragement, also Professor Veillon for a visit to USMG Grenoble where he started on this work, and to Professor Avizienis for help in visiting Los Angeles. Alan Bundy and Michael Woodger kindly read and made helpful comments on a draft.

We are grateful to the National Science Foundation and the Science Research Council for supporting this work, including an SRC Visiting Fellowship at Edinburgh for J.A.G.

Our very sincere thanks to Eleanor Kerse for her meticulous, speedy and cheerful typing.

Personal thanks go to Sei.ja Burstall, Charlotte Linde and Chogyam Trungpa Rinpocho.

References

Arsac, J. (1977) Program transformations as a programming tool. Research Report, Institut de Programmation, University de Paris VI.

Aubin, R. (1976) Mechanising structural induction. Ph.D. thesis. Depts. of Artificial Intelligence and Computer Science, University of Edinburgh.

Boyer, R.S. and Moore, J S. (1975) Proving theorems about LISP functions. JACM, 22, 1, 129-144.

Burstall, R.M. and Darlington, J. (1977) A transformation system for developing recursive programs. JACM, 14, *1*, 44-67.

Burstall, R.M. (1977) Program proof, program transformation, program synthesis for recursive programs. Lecture notes at Summer School, Erice, Sicily, 1976. To appear in Information the Journal of the Italian Association for Computer Science.

Dahl, 0-J., Myhrhaug, B. and Nygaard, K. (1970) The SIMULA 67 Common Base Language. Publication S22. Norwegian Computing Centre, Oslo.

Darlington, J. (1975) Application of program transformation to program synthesis. Proc. of International Symposium on Proving and Improving Programs, Arc-et-Senans, France, pp. 133-144.

Darlington, J. (1976) The use and implementation of very high level specifications. Invited paper at IFIP WG 2.3 Conference on Software Specifications. St. Pierre-de-Chartreuse, France.

Darlington, J. and Burstall, R.M. (1976) A system which automatically improves programs. Acta Informatica, 6, 41-60.

Dijkstra, E.W. (1975) Guarded commands, non-determinacy and formal derivation of programs. CACM, 18, 8, 453-457.

Goguen, J.A. (1976) Correctness and equivalence of data types. Proc. of 1975 Conference on Algebraic Systems, Udine, Italy, pp. 352-358. Springer-Verlag.

Goguen, J.A. (1917) Abstract errors for abstract data types. To appear in Proc. of IFIP Working Conference on the Formal Description of Programming Concepts, New Brunswick, N.J.

Goguen, J.A. and Tardo, J. (1977) OBJ-0 Preliminary Users Manual, Semantics and Theory of Computation Report, UCLA, Los Angeles.

Goguen, J.A., Thatcher, J.W. and Wagner, E.G. (1977) An initial approach to the specification, correctness and implementation of abstract data types. To appear in Current trends in programming methodology, Vol. 3. Data Structuring (ed. R.T. Yeh) Prentice Hall.

Guttag, J.V. (1975) The specification and application to programming of abstract data types. Computer Systems Research Technical Report CSRG-59, University of Toronto.

Guttag, J.V., Horowitz, E. and Musser, D.R. (1976) Abstract data types and software validation. Report ISI/RR-76-48, Information Sciences Institute, Marina del Rey, California.

Kowalski, R. (1974) Predicate logic as a programming language. Proc. of IFIP Congress '74, pp 569-574, North Holland.

Lawvere, F.W. (1963) Functional semantics of algebraic theories. Proc. of National Academy of Science. 50, pp. 869-872.

Liskov, B.H. (1975) A note on CLU. MAC-TR, MIT, Cambri dge, Mass•

Liskov, B.H. and Berzins, V. (1977) An appraisal of program specifications. Computation Structures Group Memo 141-1, MIT, Cambridge, Mass.

Mackworth, A.K. (1977) Consistency in networks of relations. Artificial Intelligence, 8, 1_, 99-118.

Manes, E.G. (1976) Algebraic theories. Springer Verlag.

Manna, Z. and Waldinger, R. (1971) Toward automatic program synthesis. CACM, 14, 3. 151-165.

Manna, Z. and Waldinger, R. (1975) Knowledge and reasoning in program synthesis. Artificial Intelligence, 6, 2, 175-208.

McCarthy, J. (1963) A basis for a mathematical theory of computation. Computer Programming and Formal Systems (eds. P. Braffort and D. Hirschberg) North Holland.

Minsky, M. (1975) A framework for representing knowledge. The Psychology of Computer Vision (ed. P. Winston) McGraw-Hill: New York.

Mosses, P. (1975) Making denotational semantics less concrete. To appear in Proc. of the Bad Honnef Workshop on Semantics of Programming Languages.

Nakajima, R., Honda, M. and Nakahara, H. (1977) Programming and verification schemes in the iota system. To appear in Proc. of IFIP Working Conference on the Formal Description of Programming Concepts, New Brunswick, N.J.

Parnas, D.L. (1972) A technique for module specification with examples. CACM, 15, 5, 330-336.

Scott, D. and Strachey, C. (1971) Towards a mathematical semantics for computer languages. Technical Monograph PRG 6, Computing Laboratory, Oxford University.

Thatcher, J.W., Wagner, E.G. and Wright, J.B. (1977) Specification of abstract data types using conditional axioms. Report IBM Laboratories, Yorktown Heights, N.Y.

Waltz, D. (1975) Understanding line drawings of scenes with shadows. The Psychology of Computer Vision (ed. P. Winston) MoCraw Hill: New York.

Wright, J.B., Thatcher, J.W., Wagner, E.G. and Goguen, J.A. (1976) Rational algebraic theories and fixed point solutions. Proc. of IEEE 17th Symposium on Foundations of Computer Science, Houston, pp. 147-158.

Wulf, W.A., London, R.L. and Shaw, M. (1976) Abstraction and verification in ALPIIARD. ISI/RR-76-46, Information Sciences Institute, Marina del Rey, California. Also as a Carnegie-Mellon Computer Science Report.

Zilles, S. (1974) Algebraic specification of data types. Computation Structures Group Memo 119, MIT, Cambridge, Mass.