A THEORY OF ABSTRACT DATA TYPES FOR PROGRAM DEVELOPMENT: BRIDGING THE GAP?*

T.S.E. Maibaum
Department of Computing
Imperial College of
Science and Technology
London
UK.

Paulo A.S. Veloso
Depto. de Informatica
PUC/RJ, Rio de Janeiro
Brasil.

M.R. Sadler
Department of Computing
Imperial College of
Science and Technology
London
UK.

Abstract: This paper outlines a logical approach to abstract data types, which is motivated by, and more adequate for (than algebraic approaches), the practice of programming. Abstract data types are specified as axiomatic theories and notions concerning the former are captured by syntactical concepts concerning the latter. The basic concepts of namability, conservative extensions and interpretations of theories explain implementation, refinement and parameterisation. Being simple, natural and flexible, this approach is quite appropriate for program development.

Key Words: abstract data types, axiomatic theories, incomplete specifications, program development, stepwise refinement, implementation, parameterisation, interpretation, conservative extension, namability.

1. Introduction

This paper outlines and illustrates a logical approach to the specification and implementation of abstract data types (ADT's) and software, which is directly motivated by, and more adequate for the practice of software engineering. There is still, we feel, a large gap between existing formal methods work on theories of ADT's and the actual practice of programming. The most important aspects of this gap concern the lack of a clear relationship between formal specification and actual programs and the technical inadequacies of formal theories of specification and implementation.

Let us look more closely at this gap with reference to examples from the current literature. Theories of specification and implementation usually do not give an adequate account of how programs are actually produced to realise the presented specification and implementation. For example, in [GHM] an implementation is defined equationally with claims that their equational definitions can be translated easily into programs in some programming language. This may be reasonably seen to be the case for functional (applicative) languages [He'80] and easy examples, but is certainly not the case for other kinds of languages and more complicated examples. In any case, what criteria can we use to decide whether this allegedly easy step is actually correct? This is not discussed. In the algebraic theories [GTW'78, Eh'81, EKP'79, WPPDB'80, WB'82, SW'82, Ga'83, BG'81], development is kept totally within the same formalism with no interface defined other than inadequate statements such as: At the end of the development process, the primitive data types of a programming language can be used. How they can be used and how programs are actually produced is never adequately discussed or is left as an open problem [BG'81].

In [L'79] we can see an informal approach to program specification and development in the ADT framework which interfaces well with a given programming language. In fact, descriptions of behaviour are given in a form which is very close to the language. However, the specifications and development are highly informal and so it is difficult to see how to verify correctness of any of the steps.

Another aspect of the gap is the difference between what software

*Research partly sponsored by CNPq, FINEP and SERC (UK).

engineers feel is required at the specification stage and what formal theories require one to state. The algebraic theories require details to be specified which might more often be left as details of implementation (or as we will point out, may not need to be dealt with at all!). One example is an operation like choosing an arbitrary element of a set. We would like to say just this (i.e. have an underdetermined operation) but, for example, the algebraic theories based on initiality force us to define at the specification level, which element of the set we will choose. Errors present a similar situation. Often (when fault tolerance is not an important consideration, for example), we wish to specify the behaviour under normal conditions of the system we are developing and do not require anything about error situations. This is what is expressed by conventional input/output specifications. Algebraic theories generally require us to give complete specifications and require us to deal with these abnormal situations at the specification level, either by using error specifications whose theory is quite complex ([Go'77,Go'83,P'84]) or partial algebras and definedness predicates which again tend to be complex and difficult ([BW'82, WPPDB'83]) or else by being somewhat informal [G'77].

Finally, let us turn to the process of implementation/development. This process is generally seen in terms of the following diagram:

$$\begin{tikzcd}
S1 \arrow[r, dashed] & S2' & & & & \\
& S2 \arrow[u, dashed] \arrow[r, dashed] & S3' & & & \\
& & S3 \arrow[u, dashed] & \cdots & & \\
& & & & SN\text{-}1 \arrow[r, dashed] & SN' \\
& & & & & SN \arrow[u, dashed]
\end{tikzcd}$$

We wish to implement specification S0 and we decide that we wish to use S1 as a basis for a first step in this process. We then enrich (extend) S1 by adding new symbols and properties so that we can realise the operations of S0 in terms of those of S1. This realisation is effected by a translation from S0 to S1' (the extension of S1); for example, theory morphisms in algebraic theories. The software engineer would mimic this process by writing a cluster of procedures C01 to implement the operations of S0 assuming those of S1 as primitive. It is not clear how he would prove the correctness of his procedures with respect to the original specification, even if he were interested, as he has no obvious interface between this implementation step (S0 in S1 via S1') and the programming language and its associated logic.

Supposing that this first implementation step is done, the software engineer might then wish to proceed by implementing S1 in S2 (via the extension/enrichment S2'). Having then written the corresponding cluster of procedures C12, he would then have a collection of procedures (C01 and C12) which he would say together implemented S0 in S2. Moreover, he would expect that the modularised reasoning he had done (in justifying the steps from S0 to S1 via S', the writing of the cluster C01, the step from S1 to S2 via S2' and the development of the cluster C12) to justify the correctness of the composed implementation of S0 in S2 and use the set of procedures in C01 and C12 as the programming language realisation of this composition. Unfortunately, most theories would not allow him to make this assumption! For example, in [Eh'82], in order to compose S0 in S1 via S1' with S1 in S2 via S2', one is obliged to "reprogram" S0 in S2 via a further extension. So one would have two extensions of S2:S2' to implement S1 and S2" to implement S0. Thus no cluster C01 could have been written _before_ the compostion as its proof of correctness can only be supplied at the end of the process of implementation when we have finally reprogrammed S0 in SN. So it would seem that we have to keep redoing work we have already done - proving the correctness of the implementation of S0. Moreover, in [Eh'82] translations (implementations) do not preserve all properties of the specification SO - only so-called ground properties (formulae in which no variables appear). Thus, any reasoning about a program P using SO would bot be valid once an implementation step is taken as invariably this involves non-ground properties and these may not be preserved. (Actually algebraic methods are also deficient in another sense with respect to verifying program properties since reasoning about programs generally requires something equational logic does not provide - the use of quantifiers. This is particularly true, for example, in the case of loop invariants in imperative language programs).

Again, in [SW'82], implementations do not in general compose. Again, property preservation is the problem (see counter example on page 485 of [SW'82]). A technical way of putting this is that the categories of specifications as objects and translations as arrows/morphisms have too many such arrows.

To bridge the gap, it is our intention to outline a theory which makes a better approximation to the objects (specifications) and arrows (translations) which should be present in order to support the practices of software engineering, in particular stepwise development and certification.

Among the methodological advantages of this method are:

(i) Specifications need say only as much as is thought desirable. There is an allowance for underdetermined operations and partially defined operations (if popping the empty stack is never used in a program, then there is no need to specify what happens in this situation), thereby simplifying the treatment of errors.

(i) The language of first order logic is a powerful and succinct formalism compared to the more restrictive formalisms used elsewhere.

(iii) The theory is closer to the usual logics of programs (usually extensions of first order logic).

(iv) The restrictive notion of sufficient completeness ([Gu'77, Ga'83, WPPDB'83, WB'82]) is replaced by a more permissive concept, thus contributing to the ease of use and expressiveness of the formalism.

Among the technical advantages of this method are:

(i) There is a powerful proof theory which, for example, allows the use of general formulae including quantifiers, justifies the use of structural induction and the development of canonical forms and provides a natural basis for proofs of termination for programs using ADT's.

(ii) There is a natural interface between implementation steps and input/output specifications for programs to realise these steps.

(iii) Implementations always compose and the composition is constructed directly and automatically from the components. Moreover, the correctness of the composition is guaranteed by the correctness of the components. Compostion of implementations is associative and implementations "commute" with instantiation of parameters in a parameterised type.

(iv) Implementations preserve <u>all</u> provable properties. Thus, proofs of correctness of programs remain valid after implementation.

(v) Equality is dealt with as any other predicate (i.e. it is not interpreted as <u>identity</u>). Thus there is no need as in the algebraic theories, to introduce a "semantic" equality different from the "=" used in equations. This reflects the reality that abstract objects are often represented by more than one concrete object. These concrete objects are equivalent but not identical.

The technical justification of these claims is provided elsewhere ([MV'81, MSV'83, MSV'83a, SM'84]). Here we hope to present our case for the methodological benefits of the theory.

2. Namability and Incomplete Specifications

For a very simple example consider the ADT with one sort (Nat), one constant symbol (zero) and one unary operation symbol (succ) specified by the following two sentences (with leading universal quantifiers implicit, as usual):

(1) $\sim$ zero = succ(n)

(2) succ(m) = succ(n) -> m = n

Notice the occurrence of the binary predicate symbol =. We shall consider = to be present in every specification together with the usual axioms stating that the realisation of = is a congruence [E'72]. Also assumed present in every specification is the following namability axiom

(N) $(\forall$ n:Nat$)$ [n = zero v n = succ (zero) v ...

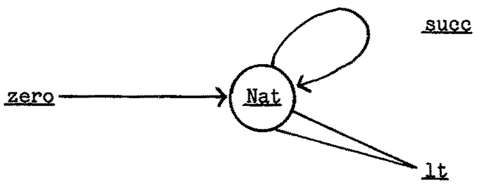
... v n = succ(...succ(zero ...) v ...]

This is an infinitary sentence (in $L_{\omega_1\omega}$), stating that every element of the domain of Nat must be the value of a ground (variable-free) term.

It is well-known, and easy to see, that every model of (1), (2) and (N) in which = is realised as identity is isomorphic to the standard model N of the natural numbers. In fact, any model A of the above axioms is such that the quotient A/=A (where =A is the realisation of = in A) is isomorphic to N. (We shall not require that = be realised always as identity for reasons to be clarified in the sequel). One very important consequence of the namability axiom will be an induction axiom

$(\forall$ n:Nat$)$[n = zero v$(\exists$ m:Nat$)$[n = suc(m)]]

Now consider the result of enriching the above ADT with a binary predicate intended to mean "less than". Call it NAT; its language is



We can specify Nat by adding to the above specification, for instance, the following sentences (which amount to a recursive definition of lt)

(3) lt(succ(m),succ(n)) <-> lt(m,n)

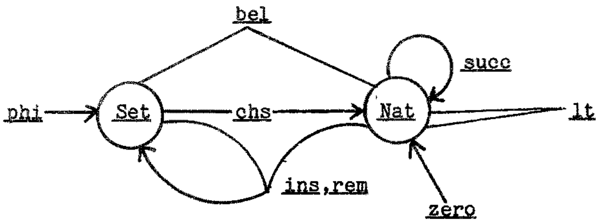
(4) lt(zero,succ(n))

(5) ~lt(zero,zero)

(6) ~lt(succ(m),zero)

We remark that this theory is a conservative extension [Sh'67] of the preceding one, for the addition of the new axioms (3)-(6) does not enable the derivation of any new theorem in the old language, i.e. without lt.

A more interesting example is the ADT SET of NAT, intended to mean finite sets of naturals. Its language is



Consider the following axioms (where the sorts of the variables are m,n:Nat; s,t:Set)

(7) [(∀ n:Nat) bel(n,s) <-> bel(n,t)] -> s = t

(8) ~ bel(n,phi)

(9) bel(m,ins(s,n)) <-> m = n v bel(m,s)

(10) bel(m,rem(s,n)) <->~ m = n & bel(m,s)

(11) ~ s = phi -> bel(chs(s),s)

Axiom (7) can be regarded (as its converse is a consequence of the underlying axioms for =) as defining = (short for $=_{Set}$) in terms of bel (onging). That is part of the reason why we do not require = to be realised as identity. Namely, in a complex data type equality among objects of a (structured) sort will in general depend upon its component objects, having to be programmed (cf. equality among arrays), rather than being simple logical identity.

Axioms (8), (9), (10) define, in the same spirit, phi, ins and rem in terms of bel. But, in contrast, we give no similar complete definition for chs. For, we want chs to be an underdetermined operation to choose an element from a non-empty set. And axiom (11) states just that! Notice in particular, that it says nothing about chs(phi) because at this point we have decided not to be interested in this particular error situation.

In order to clarify this let us consider a specific ground term

t = ins(ins(phi,succ(zero)),zero)

(which denotes the set {0,1}). From the preceding axioms we are able to deduce (as expected) sentences like

bel(succ(zero),t)
~ t = phi
t = ins(ins(phi,zero),succ(zero))
rem(t,succ(zero)) = ins(phi,zero)

properties of the specification S0 - only so-called ground properties (formulae in which no variables appear). Thus, any reasoning about a program P using S0 would bot be valid once an implementation step is taken as invariably this involves non-ground properties and these may not be preserved. (Actually algebraic methods are also deficient in another sense with respect to verifying program properties since reasoning about programs generally requires something equational logic does not provide - the use of quantifiers. This is particularly true, for example, in the case of loop invariants in imperative language programs).

Again, in [SW'82], implementations do not in general compose. Again, property preservation is the problem (see counter example on page 485 of [SW'82]). A technical way of putting this is that the categories of specifications as objects and translations as arrows/morphisms have too many such arrows.

To bridge the gap, it is our intention to outline a theory which makes a better approximation to the objects (specifications) and arrows (translations) which should be present in order to support the practices of software engineering, in particular stepwise development and certification.

Among the methodological advantages of this method are:

(i) Specifications need say only as much as is thought desirable. There is an allowance for underdetermined operations and partially defined operations (if popping the empty stack is never used in a program, then there is no need to specify what happens in this situation), thereby simplifying the treatment of errors.

(i) The language of first order logic is a powerful and succinct formalism compared to the more restrictive formalisms used elsewhere.

(iii) The theory is closer to the usual logics of programs (usually extensions of first order logic).

(iv) The restrictive notion of sufficient completeness ([Gu'77, Ga'83, WPPDB'83, WB'82]) is(replaced by a more permissive concept, thus contributing to the ease of use and expressiveness of the formalism.

Among the technical advantages of this method are:

(i) There is a powerful proof theory which, for example, allows the use of general formulae including quantifiers, justifies the use of structural induction and the development of canonical forms and provides a natural basis for proofs of termination for programs using ADT's.

(ii) There is a natural interface between implementation steps and input/output specifications for programs to realise these steps.

(iii) Implementations always compose and the composition is constructed directly and automatically from the components. Moreover, the correctness of the composition is guaranteed by the correctness of the components. Compostion of implementations is associative and implementations "commute" with instantiation of parameters in a parameterised type.

(iv) Implementations preserve all provable properties. Thus, proofs of correctness of programs remain valid after implementation.

(v) Equality is dealt with as any other predicate (i.e. it is not interpreted as identity). Thus there is no need as in the algebraic

We can also deduce, of course, bel(chs(t),t) and even

chs(t) = zero v chs(t) = succ(zero)

But we cannot deduce either equation of the above disjunction! In other words, the above axioms do not enable us to compute a specific natural number as the value of chs(t). And they should not! If they did we would have overspecified chs, which is still regarded as arbitrary choice. It would be premature at this level of specification to describe exactly how such an element is to be picked. This should be left to a future refinement, or perhaps to the implementation phase, when the consequences of such a decision can be better evaluated.

But how do we guarantee that chs(t) is indeed a natural number? Notice that, by syntax, chs(t) is of sort Nat. And our namability axiom (N) guarantees that any object of a domain of sort Nat (in particular the object denoted by chs(t)) is a standard natural number, thus preventing chs(t) from becoming a non-standard natural number. Compare this with the more conceptually complicated notions of data and hierarchy constraints in [BG'79,BG'81, WPPDB'83] and the semantic constraints of [MV'81].

As we have a new sort, we also have the corresponding namability axiom

$$(\forall s:\text{Set}) \bigvee_{t \text{ in } T} (s = t)$$

where T is an enumeration of all ground terms of sort Set and V represents infinite disjunction. As a consequence we again have a schema of induction. More important is the fact that every object of sort Set has a "normal form" involving only phi and ins, which are then the constructor operations [GH'78]. This is again a consequence of the above namability axiom.

In general, a specification for an ADT consists of a many-sorted first-order theory presented by a language L and a set of axioms G. For each sort s in S we assume a binary predicate symbol $=_s$ in L. The machinery of the logic of namability has for each sort s in S

- the usual equality axioms
- a namability axiom $(\forall x:s) \bigvee_{t \text{ in } T} (x =_s t)$

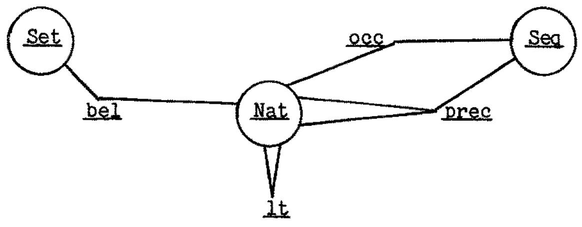
where T is an enumeration of the ground terms of sort s (in addition to the usual logic axioms). The logic of namability also has an infinitary rule of inference (a ω-rule) which allows us to manipulate the namability axiom to derive induction schema, normal forms, etc. See [B'77] for details of ω-rules.

3. Program Development and Verification

The proposed approach aims at being partucularly appropriate for program construction by means of ADT's. As a simple example to illustrate this we shall consider sorting. We can formulate it as the construction of a program P that receives as input a set t of natural numbers and outputs a sequence q of natural numbers such that is-sort(q,t), where is-sort is defined by

(12) is-sort(q,t) <-> ordered(q) & same(q,t)

Here the language consists of the following sorts and predicates



(the intended realisation of occ is occurrence of a number in a sequence and that of prec(m,n,q) is that m occurs in q before n) so that ordered and same are defined by

(13) ordered(q) <-> (∀ m,n:Nat) [lt(m,n)

& occ(m,q) & occ(n,q) -> prec (m,n,q)]

(14) same(q,t) <-> (∀ n:Nat) [occ(n,q) <-> bel(n,t)]

We now have a natural interface to our programming language as the input output specification of our sort porgram P is: {true} P {is_sort(q,S_o)} where S_o is the input set.

For operations we have all those of SET of NAT plus a constant symbol lmbd of sort Seq such that

(15) ~ occ(n,lmbd)

and an operation symbol ordins: (Seq,Nat) -> Seq partly specified by

(16) ordered(q) -> ordered(ordins(q,n))

(17) occ(m,ordins(q,n)) <-> m = n & occ(m,q)

Given this ADT SORT of NAT it is quite natural to conceive our program P first as an abstract program that repeatedly removes elements from the input set and inserts them (respecting their relative order) into an initially empty sequence. In order to formalise this intuition it is useful to extend the specification of our ADT by the following definition

(18) is-transf(q,t,s_o) <-> (∀n:Nat)[bel(n,s_o)
<-> bel(n,t) v occ(n,q)]

We are thus led to the following abstract program

```
t:=$s_o$; q:=lmbd;
  {ordered(q) & is-transf(q,t,$s_o$)}
while ~t = phi do
     n:=chs(t)      {bel(n,t)};
     q:=ordins(q,n);
     t:=rem(t,n)    {~ bel(n,t)}
end
```

We have already annotated the program with the loop invariant

ordered(q) & is-transf(q,t,s_o)

and some assertions following immediately from axioms (10) and (11).

In view of (12), the verification conditions [Ma'74] (for partial correctness) include for instance,

(19) $\text{is-transf}(q,t,s_o) \dashrightarrow [\text{bel}(n,t) \dashrightarrow \text{is-transf}(\text{ordins}(q,n), \text{rem}(t,n),s_o)]$

This (and the other verification conditions) follow easily from the preceding axioms.

A usual method to prove termination is that of the well founded set [Ma'74]. Here we can employ the set of ground terms with the well founded relation of "being a subterm". Indeed we can show, using the normal form of section 2,

$\text{bel}(n,t) \rightarrow \text{rem}(t,n) < t$

which suffices to guarantee termination.

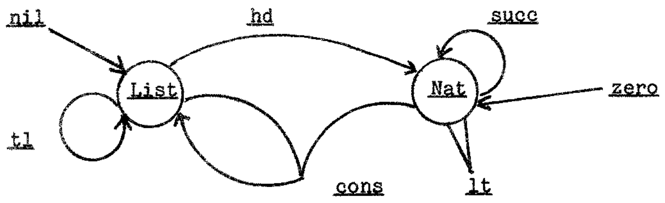
An advantage of the proposed approach is exactly this: we generally can employ the syntactical well-founded relation of being a subterm in order to prove termination, rather than having to create a special well founded set for almost every program.

Notice that we have proved the total correctness of our program based only on the ADT specification, thus without needing complete definitions for chs or ordins.

In fact, in line with the methodology of program construction by means of ADT's, we employed an ADT close to the problem. That is why we use ordins and prec, not usually thought of as available to manipulate sequences. We shall take care of this in the next section by implementing the ADT SORT of NAT in terms of more "concrete" ADT's.

4. Implementations (and Refinements) As Interpretations

The ADT used in the program for sorting is SORT of NAT. We are now going to implement it in terms of the list of naturals. We will use the ADT LIST of NAT with the following language



Its specification consists of that for NAT plus the following axioms

(20) $\text{cons}(m,x) = \text{cons}(n,y) \rightarrow x = y \;\&\; m = n$

(21) $\text{hd}(\text{cons}(m,x)) = m$

(22) $\text{tl}(\text{cons}(m,x)) = x$

(23) $\text{tl}(\text{nil}) = \text{nil}$

(Recall that we also have namability and $=_{\text{List}}$ as a congruence). Notice in particular that we cannot deduce a value for hd(nil). All we know from (N) is that hd(nil) is some natural.

We can implement *SORT of NAT* into *LIST of NAT*, sort by sort. Consider first the sort *Set*.

The first thing to do is decide which lists will represent sets. Our intuition tells us we need only those lists with nonrepeated occurrences of elements. So, we extend *LIST of NAT* with the

(24) $\underline{\text{set-rep}}(x) <\!\!-\!\!> (\forall\, n{:}\underline{\text{Nat}})\ [\underline{\text{is-in}}(n,x) \to \underline{\text{once}}(n,x)]$

where

(25) $\underline{\text{is-in}}(n,x) <\!\!-\!\!> \sim x = \underline{\text{nil}}\ \&\ [\underline{\text{hd}}(x) = n \vee \underline{\text{is-in}}(n,\underline{\text{tl}}\ (x))]$

(26) $\underline{\text{once}}(n,x) <\!\!-\!\!> \sim x = \underline{\text{nil}}\ \&\ [\underline{\text{hd}}(x) = n\ \&\ \sim \underline{\text{is-in}}(n,\underline{\text{tl}}(x))]$
$\vee\ [\sim \underline{\text{hd}}(x) = n\ \&\ \underline{\text{once}}(n,\underline{\text{tl}}(x))]])$

set-rep is called a *relativisation predicate* and it is used to delineate those lists which actually represent a set from those that do not. (It sorts the wheat from the chaff!).

We now have to extend *LIST of NAT* by concepts corresponding to those of *SET of NAT*: for each symbol *phi*, *bel*, etc., we introduce in *LIST of NAT* the corresponding primed one *phi*', *bel*', etc. For instance

(27) $\underline{\text{phi}}' = \underline{\text{nil}}$

(28) $\underline{\text{ins}}'(x,m) = y <\!\!-\!\!> [\underline{\text{is-in}}(m,x)\ \&\ y = x] \vee [\sim \underline{\text{is-in}}(m,x)\ \&\ y = \underline{\text{cons}}(m,x)]$

(29) $\underline{\text{rem}}'(x,m) = y <\!\!-\!\!> [\sim \underline{\text{is-in}}(m,x)\ \&\ y = x]$
$\vee\ [\underline{\text{is-in}}(m,x)\ \&\ y = \underline{\text{tl}}(x)]$

(30) $\underline{\text{chs}}'(x) = m \to \underline{\text{is-in}}(m,x)$

(31) $\underline{\text{bel}}'(m,x) <\!\!-\!\!> \underline{\text{is-in}}(m,x)$

(32) $x =\ '_{\underline{\text{Set}}}\ y <\!\!-\!\!>: (\forall\, n\ \underline{\text{Nat}})\ [\underline{\text{is-in}}(n,x) <\!\!-\!\!> \underline{\text{is-in}}(n,y)]$

Notice that = , not being considered a logical symbol realised as identity, undergoes the same treatment as the other symbols (we employ here = $'_{\underline{\text{Set}}}$ for clarity). Also notice that some of the above axioms define a primed symbol in terms of list symbols (e.g. *phi*' as *nil*) but others only give partial definitions. In particular, notice that we have not yet defined completely how *chs* is to operate, nor have we imposed that each set be represented by a unique list.

Having translated one specification into another, we can write a cluster of procedures to realise this translation/refinement/implementation. The procedures correspond in a one to one fashion to the operations and predicates of the abstract specification being implemented. Thus, in the example above, we have procedures defining *phi*', *ins*', *rem*', *chs*', *bel*' and = $'_{\underline{\text{Set}}}$ (as well as the operations of NAT). We can define the input/output specifications for these procedures by using (24), ..., (32). For example, the function (procedure) INS (corresponding to *ins* and *ins*') takes arguments m_o of sort *Nat* and x_o of sort *List* and has the specification

$\{\underline{\text{set-rep}}(x_o)\ \&\ \underline{\text{nat-rep}}(m_o)\}$

$\text{INS}(x_o,m_o)$

$\{(\underline{\text{is-in}}(m_o,x_o)\ \&\ \text{INS} = x_o)$
$\vee\ (\sim \underline{\text{is-in}}(m_o,x_o)\ \&\ \text{INS} = \underline{\text{cons}}(m_o,x_o))\}$

So the input specification indicates that the function is guaranteed or expected to work only for concrete representatives of abstract objects while the output specification is the definiens of the definition of <u>ins</u>' in terms of list operations. Developing and proving such a program correct can of course use the logic of the programming language and assume the properties of lists.

Now for the sort <u>Seq</u> it is natural to use a list as representing a sequence. So the corresponding representation predicate is trivial, and similarly for equality between sequences.

For the constant, operation and predicate symbols of sort <u>Seq</u>, we introduce, for instance

(33) <u>ordins</u>'(x,m) = y -> <u>ordered</u>' (y)
 & (∀ n:<u>Nat</u>) (<u>is-in</u>(n,y) <-> n = m v <u>is-in</u>(n,x))

(34) <u>prec</u>'(m,n,x) <-> ~ x = <u>nil</u> & [(<u>hd</u>(x) = m & <u>is-in</u>(n,<u>tl</u>(x)))
 v <u>prec</u>'(m,n,<u>tl</u>(x))]

Notice that we do not have to worry about symbols like <u>same</u>, etc., that were introduced by definition. As we will see later, such definitions can be "carried forward" in implementations in an automatic fashion.

Finally, we can naturally represent the sort <u>Nat</u> of <u>SORT of NAT</u> identically by the sort <u>Nat</u> of <u>LIST of NAT</u>. So this part is trivial.

By adding axioms (24) through (34) to <u>LIST of NAT</u> we have built a conservative extension of the latter. Call this extension <u>LIST of NAT by SORT of NAT</u>. Now, each sentence of the language of <u>SORT of NAT</u> can be translated into a corresponding one, its primed and relativised version. For instance consider axiom (10), which after being written with explicit leading universal quantifiers is translated to

(35) (∀ x:<u>List</u>) (∀ i,j:<u>Nat</u>) {<u>set-rep</u>(x) & <u>nat-rep</u>(i) & <u>nat-rep</u>(j)
 ->[<u>bel</u>' (i,<u>rem</u>'(x,j)) <-> i $='_{\underline{Nat}}$j v <u>bel</u>'(i,x)]}

The use of relativisation predicates with the translation of quantified formulae reflects the idea that properties of sets, when translated, are meant to hold only for lists which really represent sets.

Now, in order to guarantee the correctness of the implementation (and of our program for sorting) we have to verify that each realisation of <u>LIST of NAT</u> induces a realisation of <u>SORT of NAT</u>. This can be done as follows. Firstly for each axiom of <u>SORT of NAT</u> we verify that its translation is a theorem of <u>LIST of NAT by SORT of NAT</u>. For instance, (35) follows from <u>LIST of NAT</u> plus (25), (28) and (32) together with the definition of $='_{\underline{Nat}}$. Secondly, we verify closure of the relativisation predicates under the corresponding primed operations. For instance,

(∀x:<u>List</u>) (∀i:<u>Nat</u>) [<u>set-rep</u>(x) & <u>nat-rep</u>(i) -> <u>set-rep</u>(<u>ins</u>'(x,i))]

follows from <u>LIST of NAT</u> plus (24), (25), (26) and (28), together with the definition of $='_{\underline{Nat}}$. Thirdly, we have to verify the translation of the underlying equality and namability axioms. For instance, we have to verify that

(∀x,y:<u>List</u>) (∀ i:<u>Nat</u>) {<u>set-rep</u>(x) & <u>set-rep</u>(y) & <u>nat-rep</u>(i)
 ->[x $='_{\underline{Set}}$ y -> <u>ins</u>'(x,i) $='_{\underline{Set}}$ <u>ins</u>'(y,i)]}

(which states the substitutivity of $='_{\underline{Set}}$ with respect to <u>ins</u>') and

$$(\forall\ x{:}\underline{List})\ [\underline{set\text{-}rep}(x) \rightarrow \bigvee_{t \text{ in } T} (x\ ='_{\underline{Set}}\ t')]$$

where t' denotes the translation of t in an enumeration T as before (which states the namability of set-rep by primed Set operations).

After these verifications we have a correct implemenation of SORT of NAT by LIST of NAT.

In general, our notion of implementation is a slight generalisation of the familiar logical concept of interpretation between theories [E'72,Sh'67,VP'78]. A (correct) implementation of an ADT A presented by (L_A, G_A) by an ADT C presented by (L_C, G_C) consists of a conservative extension A by C, obtained by adding to (L_A, G_A) partial specifications for the primed symbols of A and the relativisation predicates for the sorts of L_A, together with an interpretation of the theory A into the theory of A by C.

This notion appears to capture what a programmer does when implementing A by C: the partial specifications for the primed symbols correspond to the input-ouput specifications for the procedures he writes to realise the corresponding symbols of A in terms of the symbols of C. (Notice, in particular, that the test for equality in A is now realised by a procedure in C). Also, the relativisation predicates correspond to the representation invariants of [G'77]. The abstraction mapping or representation function is implicitly given by the interpretation, which is both a conceptual and technical advantage.

One should notice that with this implementation we have a proven correct sorting program receiving sets of naturals represented by lists of naturals and outputting the corresponding sorted lists. But we have not yet completely committed ourselves to a particular sorting algorithm, because chs' and ordins' are still only partly specified. (We are committed only to the families of algorithms of sorting by selection or by insertion [Kn'75,D'77]).

In order to illustrate refinements as interpretations let us consider refining chs' and ordins'. The former is partly specified by (30) only to pick an element of a list, whereas the latter is partly specified by (33) only to insert an element into a list preserving the relative order. Suppose we decide to refine chs' to pick the least element occuring in a list and accordingly ordins' to insert an element at the head of a list.

This refinement step can be described as the (non-conservative) extension of LIST of NAT by SORT of NAT by means of the following axioms

(36) $\underline{chs}'(x) ='i \rightarrow \sim x ='\underline{nil}' \ \&\ \underline{occ}'(i,x)\ \&\ (\forall\ j{:}\underline{Nat})\ (\underline{occ}'(j,x) \rightarrow i =' j \vee \underline{lt}'(i,j))]$

(37) $\underline{ordins}'(x,i) =' \underline{cons}(i,x)$

(Notice that this still does not assign a value to chs(phi), which we do not need for our program).

Alternatively this refinement can be regarded as a simple implementation of LIST of NAT by SORT of NAT contensions into LIST of NAT by SORT of NAT (conservatively) extended by (36) with chs" in lieu of chs' and (37) with ordins" in lieu of ordins', where the interpretation is the identity but for

chs' -> chs" and ordins -> ordins"

We can now illustrate how such implementations compose. Suppose we have the ADT A implementated in the ADT B by means of the conservative extension C of B and the interpretation of A into C. Similarly for B interpreted in E which is a conservative extension of D. Diagrammatically,

$$\begin{tikzcd}
\underline{A} \arrow[r, dashed, "I1"] & \underline{C} & \\
& \underline{B} \arrow[u, "e1"'] \arrow[r, dashed, "I2"] & \underline{E} \\
& & \underline{D} \arrow[u, "e2"']
\end{tikzcd}$$

where I1 and I2 are interpretations and e1 and e2, conservative extensions. Then there exists F such that

$$\begin{tikzcd}
\underline{C} \arrow[r, dashed, "I2"] & \underline{F} \\
\underline{B} \arrow[u, "e1"] \arrow[r, dashed, "I2"'] & \underline{E} \arrow[u, "I2(e1)"']
\end{tikzcd}$$

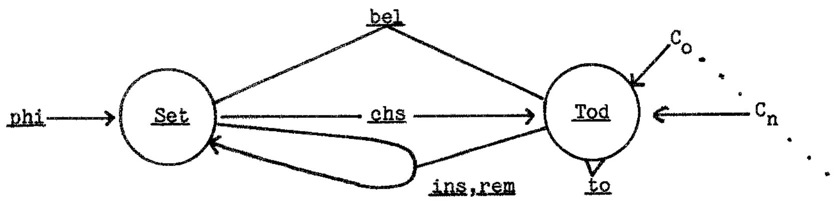
is a pushout (i.e. there is a least F). F is essentially E with the translations of the formulae defining e1 by I2 (extended as identity to the symbols introduced in e1). So the extension e1 is just carried forward (in translation) and all its consequences are still (translated) consequences). Then the composition is represented by composing the conservative extensions e2 and I2(e2) and composing the interpretations I2 and I1. The methodological point to note is that F, I2, and I2(e1) are automatically constructable and the software engineer need not actually worry himself about it!

5. Parameterisation As Interpretation

If we look back at our abstract program P for sorting we see that it does not depend heavily on the exact nature of the elements. In fact, we used more properties of sets and sequences (and, in the implementation, lists) than properties of the natural numbers, the usage of the latter being confined to a small corner. Of course, we have a case of parameterisation.

In order to illustrate the main ideas of our approach to parameterisation let us consider the simple case of SET of NAT. The idea is that SET of NAT can be obtained from the parameterised ADT SET of TOD by substituting NAT for the parameter TOD. Now, what is SET of TOD? Well, SET of TOD should be the same as SET of NAT but with the nature of the elements left completely open (except for the fact that it has a "less than" ordering, since we intend to use it for sorting). So, as far as sets are concerned, we should have the same specification as before.

To be more precise, the language of SET of TOD is



The axioms are those concerning the set symbols i.e. (7) through (11) plus the following (stating that <u>to</u> is to be realised as a total ordering relation).

(38) $(\forall\, i{:}\underline{Tod})\ \sim \underline{to}\ (i,i)$

(39) $(\forall\, i,j,k{:}\underline{Tod})\ \underline{to}\ (i,j)\ \&\ \underline{to}\ (j,k) \rightarrow \underline{to}\ (i,k)$

(40) $(\forall\, i,j{:}\underline{Tod})\ \underline{to}\ (i,j)\ \vee\ i = j\ \vee\ \underline{to}\ (j,i)$

We still have the underlying axioms for equality and for namability. Only notice that the namability axiom for sort <u>Tod</u> has the form

$(\forall\, i{:}\underline{Any})\ (i = C_o \vee i = C_1 \vee \dots \vee i = C_n \vee \dots)$

This is the only axiom mentioning the constant symbols C_o, ..., C_n, ... As there are no axioms jointly mentioning some C_n with some other symbol, the C_n's are not constrained to any particular value in a realisation. Their only rôle is naming the elements of a domain of sort <u>Tod</u>. That is why we regard <u>Tod</u> as a parameter, subject to the only constraint of having a total order <u>to</u>. In particular the only interesting results derivable from this specification are those one might call results concerning sets per se and total orders, in general.

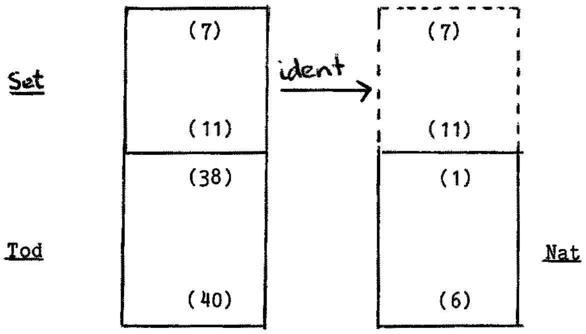
Now, how do we pass parameters in order to obtain <u>SET of NAT</u> from <u>SET of TOD</u>? This is performed by the assignment p, of sorts

$\underline{Tod} \mapsto \underline{Nat}$

and of symbols

$\underline{to} \mapsto \underline{lt}$
$C_o \mapsto \underline{zero}$
$C_1 \mapsto \underline{succ}(\underline{zero})$
....

We extend this assignment p to be the identity on the remaining symbols, so that it builds a copy of this part of the language of <u>SET of TOD</u> on top of that of <u>NAT</u>. Thus, this mapping will translate identically axioms (7) through (11). These axioms together with those of <u>NAT</u>, (1) through (6), will give the specification of <u>SET of NAT</u>. Pictorially



Notice that the translations of axioms (38) through (40), as well as of the equality and namability axioms of sort <u>Tod</u>, are theorems of <u>NAT</u>. Thus, the outcome is an interpretation of theories, of <u>SET of TOD</u> into <u>SET of NAT</u>. Hence, all the results proved about <u>SET of TOD</u> translate into provable properties of <u>SET of NAT</u>.

Similarly, we have the parameterised ADT's <u>SORT of TOD</u> and <u>LIST of TOD</u>. Application of assignment p will build the expected ADT's together with the corresponding interpretations

S[p] of <u>SORT of TOD</u> into <u>SORT of NAT</u>

and

L[p] of <u>LIST of TOD</u> into <u>LIST of NAT</u>

In the previous section we gave an implementation of <u>SORT of NAT</u> into <u>LIST of NAT</u>. Now, we consider the implementation I[-], which is the same except that it is the identity on sort <u>Tod</u> and its symbols <u>to</u>, C_0, C_1 ... . This is a "parameterised" implementation of <u>SORT of TOD</u> into <u>LIST of TOD</u>.

In a natural way, the parameter assignment p coupled with this "parameterised" implementation I[-] defines the original implementation I[p] of <u>SORT of NAT</u> into <u>LIST of NAT</u>. Furthermore, the following diagram (where the horizontal interpretations come from parameter passing and the vertical ones correspond to implementations) commutes.

$$\begin{tikzcd}
\underline{\text{SORT of TOD}} \arrow[r, "S[p]"] \arrow[d, "I[-]"'] & \underline{\text{SORT of NAT}} \arrow[d, "I[p]"] \\
\underline{\text{LIST of TOD}} \arrow[r, "L[p]"] & \underline{\text{LIST of NAT}}
\end{tikzcd}$$

The nice practical consequence is that we have the freedom to develop our program for sorting in a parameterised fashion and specialise the parameters to <u>NAT</u> (or any other suitable data type) when we please.

6. <u>Conclusion</u>

We have outlined an approach to ADT's, based on logic which is a natural formalisation of what programmers (should) do.

The key idea is that an ADT is (specified by) a (many-sorted) logical theory presented by axioms. Thus, notions concerning ADT's are captured by syntactical concepts concerning their theories. In particular,

- the properties of an ADT are (formalised as) the theorems deduced from its axioms
- an implementation of an ADT by another ADT is an interpretation of the theory of the former into a conservative extension of the theory of the latter
- a refinement is an extension, which is a simple implementation
- parameterisation is also an interpretation.

Thus we need only the familiar logical concepts of (conservative) extension and interpretation. In general, the formulas of an extension are input-output specifications of procedures.

As an illustration of the technical simplicity of our theory, we have the fact that parameter passing commutes with implementations. In most approaches this result has a somewhat elaborate proof. Here it is an immediate consequence of a simple but important result, namely the composability of implementations.

Flexibility is another important asset. We are free to specify just as much as we want or need. In particular our specifications can be incomplete, sufficiently complete or even complete in the algebraic sense [G'77, GH'78, GTW'78]. This flexibility is very convenient in dealing with errors or undefined values. For instance, consider the case of hd(nil). We can decide to leave it unspecified but defined. Or we can decide to have an error constant to be the value of hd(nil) with the further choice of either specifying error propogation or leaving it open. In any case we have the well-founded relation of "being a subterm" at hand to use in proofs of termination.

Finally, the usage of simple logical concepts together with the flexibility and naturalness of this approach make it quite adequate for program development.

References

[B'77]	J. Barwise,ed: Handbook of Mathematical Logic, Studies in Logic and the Foundations of Mathematics, Vol.90, North Holland, 1977.
[BG'79]	R.M. Burstall, J.A. Goguen: The Semantics of CLEAR, A Specification Language, (as in D'79)
[BG'81]	R.M. Burstall, J.A. Goguen: An Informal Introduction to Specifications using CLEAR, in: "The Correctness Problem in Computer Science", eds. R.S. Boyer, J.S. Moore, Academic Press, 1981
[BW'82]	M. Broy, M. Wirsing: Partial Abstract Types, Acta Informatica, Vol.
[D'77]	J. Darlington: A Synthesis of Several Sorting Algorithms, Imperial College of Science and Technology, Department of Computing, London, 1977
[D'79]	B. Domolski: An Example of Hierarchical Program Specification, Proc. of 1979 Copenhagen Winter School on Abstract Software Specifications, LNCS86, Springer-Verlag
[E'72]	H.B. Enderton: A Mathematical Introduction to Logic, Academic Press, New York, 1972
[Eh'82]	H-D. Ehrich: On the Theory of Specification, Implementation and Parameterisation of Abstract Data Types, JACM, Vol. 29, No. 1, 1982
[EK'82]	H. Ehrig, H-J. Kreowski: Parameter Passing Commutes with Implementation of Parameterised Data Types, 9th ICALP, LNCS 140, Springer-Verlag
[EKMP'80]	H. Ehrig, H-J. Kreowski, B. Mahr, P. Padawitz: Compound Algebraic Implementations: an Approach to Stepwise Refinement of Software Systems, 9th MFCS, LNCS88, Springer-Verlag, 1980
[G'77]	J.V. Guttag: Abstract Data Types and the Development of Data Structures, Comm. ACM, Vol. 20, No. 6, pp. 396-404, June 1977
[G'80]	J.V. Guttag: Notes on Type Abstraction (Version 2), IEEE TSE, Vol. 6, No. 1, 1980
[Go'83]	M. Gogolla: Algebraic Specifications with Partially Ordered Sorts, Tech.Report 169, Abt. Informatik, U. of Dortmund, 1983.
[Ga'83]	H. Ganzinger: Parameterised Specifications: Parameter Passing and Implementation, ACM, TOPLAS, Vol. 5, No. 3, 1983
[GH'78]	J.V. Guttag and J.J. Horning: The Algebraic Specification of Abstract Data Types, Acta Informatica, Vol. 10, No. 1, pp. 27-52, 1978
[GHM'78]	J.V. Guttag, E. Horowitz, D.R. Musser: The Design of Data Type Specifications, in "Current Trends in Programming Methodology, Vol. IV", Ed. R.T. Yeh, Prentice Hall, 1978
[GTW'78]	J.A. Goguen, J.W. Thatcher, E.G. Wagner: An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, in "Current Trends in Programming Methodology, Vol. IV", Ed. R.T. Yeh, Prentice Hall, Englewood Cliffs, 1978
[H'80]	U.L. Hupbach: Abstract Implementation of Abstract Data Types, 9th MFCS, LNCS88, Springer-Verlag, 1980
[He'80]	P. Henderson: Functional Programming: Application and Implementation, Prentice-Hall, 1980
[H'72]	C.A.R. Hoare: Proof of Correctness of Data Representations, Acta Informatica, Vol. 4, pp. 271-281, 1972

[J'80] C.B. Jones: Software Development: a Rigorous Approach, Prentice-Hall, London, 1980

[Kn'75] D.E. Knuth: The Art of Computer Programming, Vol. 3, Addison Wesley, Reading, 1975

[L'79] B. Liskov: Modular Program Construction Using Abstractions, (as in D'79]

[LZ'77] B. Liskov, S. Zilles: An Introduction to Formal Specifications of Data Abstractions, in "Current Trends in Programming Methodology, Vol. I", Ed. R.T. Yeh, Prentice-Hall, Englewood Cliffs, 1977

[Ma'74] Z. Manna: The Mathematical Theory of Computation, McGraw-Hill, New York, 1974

[MV'81] T.S.E. Maibaum, P.A.S. Veloso: A Logical Approach to Abstract Data Types, Technical Report, Department of Computing, Imperial College, and Departamento de Informatica, PUC/RJ, 1981

[MSV'83] T.S.E. Maibaum, M.R. Sadler, P.A.S. Veloso: Logical Specification and Implementation, Technical Report, Department of Computing, Imperial College, 1983

[MSV'83a] T.S.E. Maibaum, M.R. Sadler, P.A.S. Veloso: A Straightforward Approach to Parameterised Specifications, Technical Report, Department of Computing, Imperial College, 1983

[P,84] A. Poigne: Another Look at Parameterisation Using Suborts, MFCS84, LNCS176,1984.

[SM'84] M.R. Sadler, T.S.E. Maibaum: The Logic of Namability, In preparation

[SW'82] D. Sanella, M. Wirsing: Implementation of Parameterised Specifications, 9th ICALP, LNCS140, Springer-Verlag, 1982

[Sh'67] J.R. Schoenfield: Mathematical Logic, Addison Wesley, Reading, 1967

[T'78] W.M. Turski: Computer Programming Methodology, Heyden, London, 1978

[VP'78] P.A.S. Veloso, T.H.C. Pequeno: Interpretations between Many-Sorted Theories, 2nd Brasilian Colloquium on Logic, Campinas, 1978

[WB'82] M. Wirsing, M. Broy: An Analysis of Semantic Models for Algebraic Specifications, in "Theoretical Foundations of Programming Methodology", eds. M. Broy, G. Schmidt, Reidel, Dordrecht, 1982

[WPPDB'80] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy: On Hierarchies of Abstract Data Types, Technische Univ., Munchen, Inst. Informatik, 1980

[WPPDB'83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy: On Hierarchies of Abstract Data Types, Acta Informatica, Vol. 20, Fasc. 1, pp. 1-33, 1983