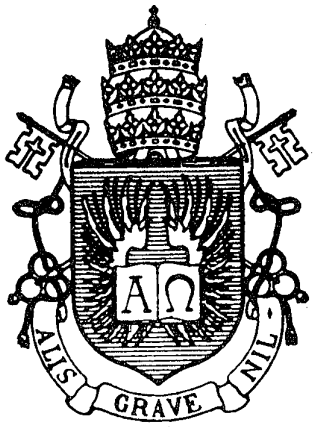


PUC

Series: Monografias em Ciência da Computação

Nº 15/83

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

by

Paulo A. S. Veloso
T. S. E. Maibaum

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Series: Monografias em Ciência da Computação, Nº 15/83

Series Editor: A. L. Furtado June, 1983

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

by

Paulo A. S. Veloso

T. S. E. Maibaum +

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

+ Departament of Computing, Imperial College of Science and Technology; London, U.K:

ABSTRACT

This paper outlines a logical approach to abstract data types , which is motivated by, and more adequate for, the practice of programming. Abstract data types are specified as axiomatic theories and notions concerning the former are captured by syntac - tical concepts concerning the latter. The basic concepts of na - mability, conservative extensions and interpretations of theories explain implementation, refinement and parameterization. Being simple, natural and flexible this approach is quite appropriate for program development and certification.

Key words: abstract data types,
axiomatic theories,
incomplete specifications,
program development,
program certification,
stepwise refinement,
implementation, parameterization,
interpretation,
conservative extension,
namability

R E S U M O

Este trabalho esboça um enfoque lógico para tipos abstratos de dados, que é mais adequado para a prática da programação, sendo por ela motivado. Tipos abstratos de dados são especificados por meio de teorias axiomáticas de modo que noções sobre aqueles são captadas por conceitos sintáticos envolvendo estas. Os conceitos básicos de nomeabilidade, extensões conservativas e interpretações de teorias explicam implementação, refinamento e parametrização. Por sua simplicidade, naturalidade e flexibilidade este enfoque é bastante adequado ao desenvolvimento e à certificação de programas.

Palavras chave : tipos abstratos de dados,
teorias axiomáticas,
especificações incompletas,
desenvolvimento de programas,
certificação de programas,
refinamentos sucessivos,
implementação,
parametrização,
interpretação,
extensão conservativa,
nomeabilidade

C O N T E S T S

1. INTRODUCTION

This paper outlines and illustrates a logical approach to the concept of abstract data type (ADT, for short). This approach, based on mathematical logic, is directly motivated by and more adequate to the practice of programming.

It is now widely accepted that abstraction provides a useful tool for the programmer. In particular, the usage of ADT's gives the program an elegant structure and enables a natural factorization of the various programming tasks: specification, development, documentation, verification, testing, etc.

In order to exploit abstraction, ADT's must be formally specified so as to be representation independent. For this purpose several approaches have been proposed: axiomatic, algebraic, state machines, etc. [H'72,GTW'78,G'77,GH'78,WPPDB'80,LZ'78]. However, each one of these approaches presents, together with its advantages, some annoying inconveniences. With some over-simplification one might say that these inconveniences amount to difficulty in applying these approaches.

The approach proposed here aims at alleviating such inconveniences. More specifically it offers the advantages of

- being simpler and closer to the usual practice of programming;
- being adequate for stepwise development and certification of programs;
- permitting incomplete specifications;
- dealing in a flexible way with error/exceptions;
- having simple and natural notions of parameterization and implementation.

The main features of our approach are

- namability: each object of an ADT must have a name in order to be referred to by a program;
- non-uniqueness: some details can (and in some cases should) be left open, which allows more freedom;

- logic: the language of first-order predicate logic is formal, flexible and employed for program verification.

Thus, our ADT's form a class of finitely generated models specified by a first-order theory. This indicates that our approach, though independently developed, bears some similarity to that of the Munich group [WPPDB'80]. It differs however in two aspects: technical and practical. The main technical distinguishing features of the approach proposed here are

- few technical concepts: conservative extension and interpretation between theories [Sh'67];
- a complete proof theory, which enables formal deductions without having to rely on reasoning in terms of models.

Some practical distinguishing features of our approach will become apparent in the sequel.

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) $\neg \underline{zero} \approx \underline{succ}(n)$

(2) $\underline{succ}(m) \approx \underline{succ}(n) \rightarrow m \approx n$

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

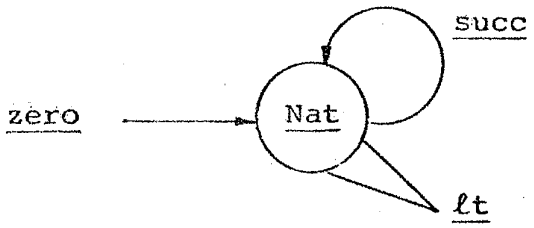
(N) $(\forall n{:}\underline{Nat})[n \approx \underline{zero} \vee n \approx \underline{succ}(\underline{zero}) \vee \ldots$

$\ldots \vee n \approx \underline{succ}(\ldots \underline{succ}(\underline{zero}) \ldots) \vee \ldots]$

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 $\approx$ is realized 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 $A/\approx^A$ where $\approx^A$ is the realization of $\approx$ in A) is isomorphic to N. (We shall not require that $\approx$ be realized always as identity for reasons to be clarified in the sequel).

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) $\ell t(succ(m),succ(n)) \leftrightarrow \ell t(m,n)$

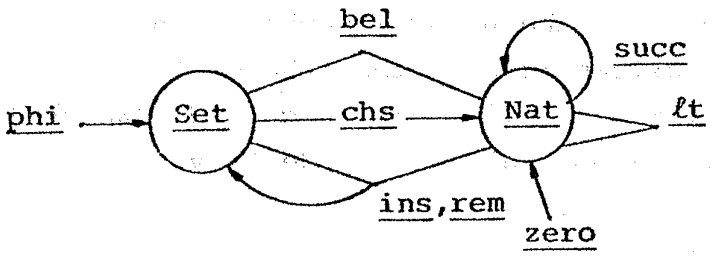
(4) $\ell t(zero,succ(n))$

(5) $\neg \ell t(zero,zero)$

(6) $\neg \ell t(succ(m),zero)$

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) $[(\forall n{:}\mathrm{Nat})\ \underline{bel}(n,s) \leftrightarrow \underline{bel}(n,t)] \rightarrow s \approx t$

(8) $\neg \underline{bel}(n,\underline{phi})$

(9) $\underline{bel}(m,\underline{ins}(s,n)) \leftrightarrow m \approx n \vee \underline{bel}(m,s)$

(10) $\underline{bel}(m,\underline{rem}(s,n)) \leftrightarrow \neg m \approx n \wedge \underline{bel}(m,s)$

(11) $\neg s \approx \underline{phi} \rightarrow \underline{bel}(\underline{chs}(s),s)$

Axiom (7) can be regarded (as its converse is a consequence of the underlying axioms for $\approx$) as defining $\approx$ (short for $\approx_{\underline{Set}}$) in terms of bel (onging). That is part of the reason why we do not require $\approx$ to be realized 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 a "nondeterministic" 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).

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

$$t = \underline{ins}(\underline{ins}(\underline{phi},\underline{succ}(\underline{zero})),\underline{zero})$$

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

$\underline{bel}(\underline{succ}(\underline{zero}),t)$
$\neg t \approx \underline{phi}$
$t \approx \underline{ins}(\underline{ins}(\underline{phi},\underline{zero}),\underline{succ}(\underline{zero}))$
$\underline{rem}(t,\underline{succ}(\underline{zero})) \approx \underline{ins}(\underline{phi},\underline{zero})$

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

$$\underline{chs}(t) \approx \underline{zero} \vee \underline{chs}(t) \approx \underline{succ}(\underline{zero})$$

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 supposed to be a non-deterministic choice. It would be premature at this level of specification to describe exactly how such an element is to be picked up. 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.

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

$$(\forall s{:}\underline{Set}) \bigvee_{t \in T} (s \approx t)$$

where T is an enumeration of all ground terms of sort Set. As a consequence we 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].

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 Σ. For each sort $s \in S$ we assume a binary predicate symbol $\approx_s$ in L. The machinery of the logic of namability has for each sort $s \in S$

. the usual equality axioms;

. a namability axiom $(\forall x{:}s) \bigvee_{t \in T} (x \approx_s t)$

where T is an enumeration of the ground terms of sort s; in addition to the usual logic axioms.

The realizations of a specification are those many-sorted structures of the language L that satisfy the specification Σ where for each sort $s \in S$

. $\approx_s$ is realized as a congruence

. the domain of sort s is namable, i.e., every object is denoted by a ground term of this sort.

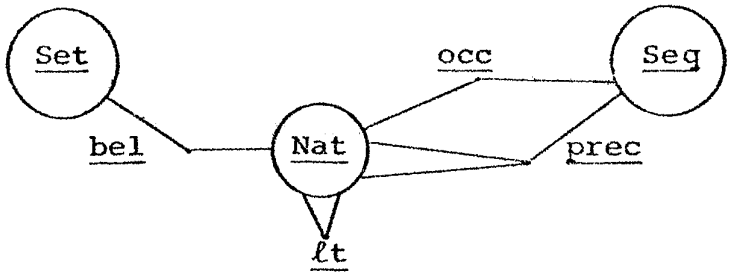
The logic of namability is complete in the usual sense that the provable sentences of a specification are exactly those holding in all the realizations of the specification.

3. PROGRAM DEVELOPMENT AND VERIFICATION

The proposed approach aims at being particularly 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) $\underline{\text{is-sort}}(q,t) \leftrightarrow \underline{\text{ordered}}(q) \wedge \underline{\text{same}}(q,t)$

Here the language consists of the following sorts and predicates



(the intended realization 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) $\underline{\text{ordered}}(q) \leftrightarrow (\forall m,n{:}\underline{\text{Nat}})[\underline{\ell t}(m,n) \wedge \wedge \underline{\text{occ}}(m,q) \wedge \underline{\text{occ}}(n,q) \rightarrow \underline{\text{prec}}(m,n,q)]$

(14) $\underline{\text{same}}(q,t) \leftrightarrow (\forall n{:}\underline{\text{Nat}})[\underline{\text{occ}}(n,q) \leftrightarrow \underline{\text{bel}}(n,t)]$

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

(15) $\neg\underline{\text{occ}}(n,\underline{\text{lmbd}})$

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

(16) $\underline{ordered}(q) \rightarrow \underline{ordered}(\underline{ordins}(q,n))$

(17) $\underline{occ}(m,\underline{ordins}(q,n)) \leftrightarrow m \simeq n \vee \underline{occ}(m,q)$

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 formalize this intuition it is useful to extend the specification of our ADT by the following definition

(18) $\underline{is\text{-}transf}(q,t,s_o) \leftrightarrow (\forall n{:}\underline{Nat})[\underline{bel}(n,s_o) \leftrightarrow$
$\leftrightarrow \underline{bel}(n,t) \vee \underline{occ}(n,q)]$

We are thus led to the following abstract program

$t{:=}s_o$; $q{:=}\underline{lmbd}$;
{$\underline{ordered}(q) \wedge \underline{is\text{-}transf}(q,t,s_o)$}
while $\neg t \simeq \underline{phi}$ **do**
  $n{:=}\ \underline{chs}(t)$ {$\underline{bel}(n,t)$};
  $q{:=}\ \underline{ordins}(q,n)$;
  $t{:=}\ \underline{rem}(t,n)$ {$\neg\underline{bel}(n,t)$}
end

We have already annotated the program with the loop invariant

$\underline{ordered}(q) \wedge \underline{is\text{-}transf}(q,t,s_o)$

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

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

(19) $\underline{\text{ordered}}(\underline{\text{lmbd}})$

(20) $\underline{\text{ordered}}(q) \rightarrow \underline{\text{ordered}}(\underline{\text{ordins}}(q,n))$

(21) $\underline{\text{is-transf}}(\underline{\text{lmbd}}, s_o, s_o)$

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

(23) $\underline{\text{is-transf}}(q,t,s_o) \wedge t \simeq \underline{\text{phi}} \rightarrow \underline{\text{same}}(q,t)$

These follow easily from the preceding axioms: (19) follows from (13) and (15); (20) is (16); (21) follows from (15) and (18); (22) follows from (10), (17) and (18); finally (23)follows from (8), (14) and (18).

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 $\prec$ of "being a subterm". Indeed we can show, using the normal form of section 2,

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

which suffices to guarantee termination.

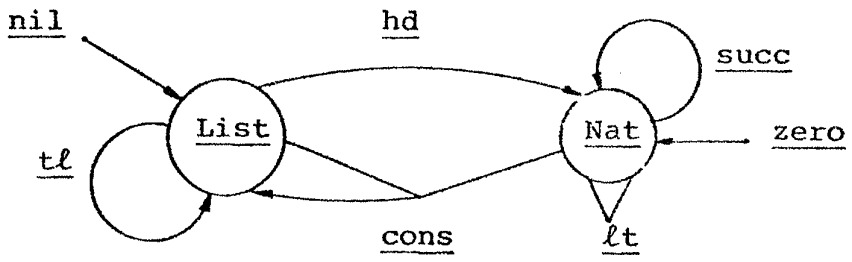
An advantage of the proposed approach is exactly this: we often can employ the syntactical well-founded relation of being a subterm in order to prove termination, rather than having to create a special well founded set for almost each program.

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 lists 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

(24) $\neg \underline{nil} \approx \underline{cons}(n,x)$

(25) $\underline{cons}(m,x) \approx \underline{cons}(n,y) \rightarrow x \approx y \wedge m = n$

(26) $\underline{hd}(\underline{cons}(m,x)) \approx m$

(27) $\underline{t\ell}(\underline{cons}(m,x)) \approx x$

(28) $\underline{t\ell}(\underline{nil}) \approx \underline{nil}$

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 is deciding which lists will represent sets. Our intuition tells us we need only those lists with non-repeated occurrences of elements. So, we extend LIST of NAT with the definition

(29) $\underline{\text{set-rep}}(x) \leftrightarrow (\forall n{:}\underline{\text{Nat}})[\underline{\text{is-in}}(n,x) \rightarrow \underline{\text{once}}(n,x)]$

where

(30) $\underline{\text{is-in}}(n,x) \leftrightarrow \neg x \approx \underline{\text{nil}} \wedge [\underline{\text{hd}}(x) \approx n \vee \underline{\text{is-in}}(n,\underline{t\ell}(x))]$

(31) $\underline{\text{once}}(n,x) \leftrightarrow \neg x \approx \underline{\text{nil}} \wedge [\underline{\text{hd}}(x) \approx n \wedge \neg\underline{\text{is-in}}(n,\underline{t\ell}(x))] \vee$
$\vee [\neg\underline{\text{hd}}(x) \approx n \wedge \underline{\text{once}}(n,\underline{t\ell}(x))]$

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.

(32) $\underline{\text{phi}}' \approx \underline{\text{nil}}$

(33) $\underline{\text{ins}}'(x,m) \approx y \leftrightarrow [\underline{\text{is-in}}(m,x) \wedge y \approx x] \vee$
$\vee [\neg\underline{\text{is-in}}(m,x) \wedge y \approx \underline{\text{cons}}(m,x)]$

(34) $\underline{\text{rem}}'(x,m) \approx y \leftrightarrow [\neg\underline{\text{is-in}}(m,x) \wedge y \approx x] \vee$
$\vee [\underline{\text{is-in}}(m,x) \wedge y \approx \underline{t\ell}(x)]$

(35) $\underline{\text{chs}}'(x) \approx m \rightarrow \underline{\text{is-in}}(m,x)$

(36) $\underline{\text{bel}}'(m,x) \leftrightarrow \underline{\text{is-in}}(m,x)$

(37) $x \approx'_{\underline{\text{Set}}} y \leftrightarrow (\forall n{:}\underline{\text{Nat}})[\underline{\text{is-in}}(n,x) \leftrightarrow \underline{\text{is-in}}(n,y)]$

Notice that $\approx$, not being considered a logical symbol realized as identity, undergoes the same treatment as the other symbols (we employ here $\approx'_{\underline{\text{Set}}}$ 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 (we can view them as input-output specifications of

procedures that will eventually realize them in a programming language). 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.

Now for the sort Seq it is natural to use a list as representing a sequence. So the corresponding representation predicate is trivial,namely

(38) $\underline{\text{seq-rep}}(x) \leftrightarrow x \simeq x$

Similarly for equality between sequences

(39) $x \simeq'_{\underline{\text{Seq}}} y \leftrightarrow x \simeq y$

For the constant, operation and predicate symbols of sort Seq, we introduce

(40) $\underline{\text{lmbd}}' \simeq \underline{\text{nil}}$

(41) $\underline{\text{ordins}}'(x,m) \simeq y \rightarrow \underline{\text{ordered}}'(y) \wedge$
$\wedge (\forall n{:}\underline{\text{Nat}})\,(\underline{\text{is-in}}(n,y) \leftrightarrow n \simeq m \vee \underline{\text{is-in}}(n,x))$

(42) $\underline{\text{occ}}'(m,x) \leftrightarrow \underline{\text{is-in}}(m,x)$

(43) $\underline{\text{prec}}'(m,n,x) \leftrightarrow \neg x \simeq \underline{\text{nil}} \wedge [(\underline{\text{hd}}(x) \simeq m \wedge \underline{\text{is-in}}(n,\underline{t\ell}(x))) \vee$
$\vee \underline{\text{prec}}'(m,n,\underline{t\ell}(x))]$

Notice that we do not have to worry about symbols like same, etc., that were introduced by definition.

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

By adding axioms (29) through (43) to LIST of NAT we have built a conservative extension of the latter. Call this extension LIST of NAT by SORT of NAT. Now, each sentence of the language of SORT of NAT can be translated into a corresponding one, its primed and relativized version. For instance consider axiom (10), which written with explicit leading universal quantifiers is

$$(\forall s{:}\underline{Set})(\forall m,n{:}\underline{Nat})[\underline{bel}(m,\underline{rem}(s,n))\leftrightarrow\neg m\simeq_{\underline{Nat}} n \vee \underline{bel}(m,s)]$$

Its translation is

$$(44)\quad (\forall x{:}\underline{List})(\forall i,j{:}\underline{Nat})\{\underline{set\text{-}rep}(x)\wedge\underline{nat\text{-}rep}(i)\wedge\underline{nat\text{-}rep}(j)\rightarrow \rightarrow[\underline{bel}'(i,\underline{rem}'(x,j))\leftrightarrow\neg i\simeq'_{\underline{Nat}} j \vee \underline{bel}'(i,x)]\}$$

[In fact, in the axioms (29) through (43) all quantifiers should be relativized to the corresponding representation predicate. For instance, (29) should read

$$\underline{set\text{-}rep}(x)\leftrightarrow(\forall i{:}\underline{Nat})\{\underline{nat\text{-}rep}(i)\rightarrow[\underline{is\text{-}in}(i,x)\rightarrow\underline{once}(i,x)]\}$$

Similarly, symbols for equality in SORT of NAT should be understood as the corresponding primed versions. So, (41) should read

$$(\forall x,y{:}\underline{List})(\forall i{:}\underline{Nat})\{\underline{ordins}'(x,i)\simeq y\rightarrow(\forall j{:}\underline{Nat})[\underline{nat\text{-}rep}(j)\rightarrow \rightarrow(\underline{is\text{-}in}(i,y)\leftrightarrow i\simeq'_{\underline{Nat}} j \vee \underline{is\text{-}in}(i,x))]\}$$

We have preferred to leave these details implicit for the sake of simplicity.]

Now, in order to guarantee the correctness of the implementation (and of our program for sorting) we have to verify that each realization of LIST of NAT induces a realization of SORT of NAT. This can be done as follows. Firstly for each axiom of SORT of NAT we verify that its translation is a theorem of LIST of NAT by SORT of NAT. For instance, (44) follows from LIST of NAT plus (30), (33) and (36) together with the definition of $\simeq'_{\underline{Nat}}$. Secondly, we verify closure of the relativization predicates under the corresponding primed operations. For instance,

$$(\forall x{:}\underline{List})(\forall i{:}\underline{Nat})[\underline{set\text{-}rep}(x)\wedge\underline{nat\text{-}rep}(i)\rightarrow \rightarrow\underline{set\text{-}rep}(\underline{ins}'(x,i))]$$

follows from LIST of NAT plus (29), (30), (31) and (33), together with the definition of $\simeq'_{\underline{Nat}}$ Thirdly, we have to verify that the translation of the underlying equality and namability axioms. For instance, we have to verify that

$$(\forall x,y{:}\underline{\text{List}})(\forall i{:}\underline{\text{Nat}})\{\underline{\text{set-rep}}(x) \wedge \underline{\text{set-rep}}(y) \wedge \underline{\text{nat-rep}}(i) \rightarrow$$
$$\rightarrow [x \approx'_{\underline{\text{Set}}} y \rightarrow \underline{\text{ins}}'(x,i) \approx'_{\underline{\text{Set}}} \underline{\text{ins}}'(y,i)]\}$$

(which states the substitutivity of $\approx'_{\underline{\text{Set}}}$ with respect to $\underline{\text{ins}}'$) and

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

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 implementation of SORT of NAT by LIST of NAT.

In general, our notion of implementation is a slight generalization 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,Σ_A) by an ADT C presented by (L_C,Σ_C) consists of a conservative extension A by C, obtained by adding to (L_A,Σ_A) partial specifications for the primed symbols of A and the relativization predicates for the sorts of L_A, together with an interpretation of the theory Σ_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-output specifications for the procedures he writes to realize the corresponding symbols of A in terms of the symbols of C. (Notice, in particular that the test for equality in A is now realized by a procedure in C). Also, the relativization predicates correspond to the representation invariants [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 our-

selves 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 (35) only to pick an element of a list, whereas the latter is partly specified by (41) only to insert an element into a list preserving the relative order. Suppose we decide to refine chs' to pick the least element of 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

(45) $\underline{chs}'(x) \approx' i \rightarrow \neg x \approx' \underline{nil}' \wedge \underline{occ}'(i,x) \wedge (\forall j{:}\underline{Nat})(\underline{occ}'(j,x) \rightarrow \rightarrow i \approx' j \vee \underline{\ell t}'(i,j))]$

(46) $\underline{ordins}'(x,i) \approx' \underline{cons}(i,x)$

(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 into LIST of NAT by SORT of NAT (conservatively) extended by (45) with chs" in lieu of chs' and (46) with ordins" in lieu of ordins', where the interpretation is the identity but for

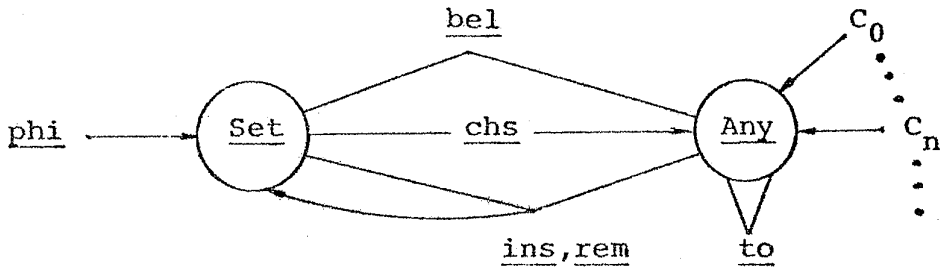
$\underline{chs}' \mapsto \underline{chs}''$ and $\underline{ordins}' \mapsto \underline{ordins}''$

5. PARAMETERIZATION 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, the idea is that we have a case of parameterization.

In order to illustrate the main ideas of our approach to parameterization let us consider the simple case of SET of NAT. The idea is that SET of NAT can be obtained from the parameterized ADT SET of ANY by substituting NAT for the parameter ANY. Now, what is SET of ANY? Well, SET of ANY 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 ANY is



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

(47) $(\forall i:\underline{Any})\neg \underline{to}(i,i)$

(48) $(\forall i,j,k:\underline{Any})\ \underline{to}(i,j) \wedge \underline{to}(j,k) \rightarrow \underline{to}(i,k)$

(49) $(\forall i,j:\underline{Any})\ \underline{to}(i,j) \vee i \approx j \vee \underline{to}(j,i)$

We still have the underlying axioms for equality and for namability. Only notice that the namability axiom for sort Any has the form

$$(\forall i:\underline{Any})(i \approx C_0 \vee i \approx C_1 \vee \ldots \vee i \approx C_n \vee \ldots)$$

This is the only axiom mentioning the constant symbols $C_o,\ldots,$ $C_n,\ldots$. 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 realization. Their only role is naming the elements of a domain of sort Any. That is why we regard Any as a parameter, subject to the only constraint of having a total order to. In particular the only interesting results derivable from this specification are those one might call results concerning sets per se.

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

$$\underline{Any} \mapsto \underline{Nat}$$

and of symbols

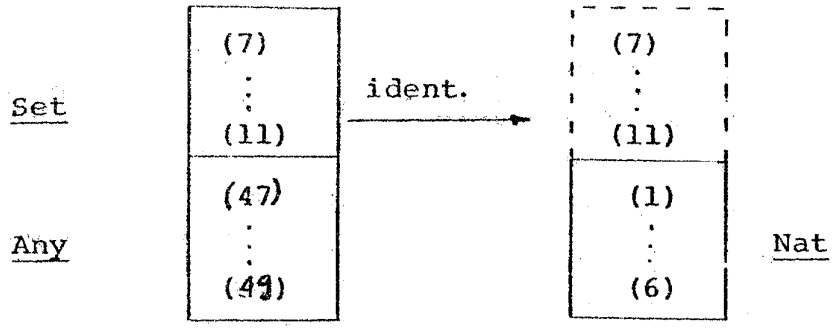
$\underline{to} \mapsto \underline{\ell t}$

$C_0 \mapsto \underline{zero}$

$C_1 \mapsto \underline{succ}(\underline{zero})$

.....

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 SET of ANY on top of that of NAT. Thus, this mapping will translate identically axioms (7) through (11). These axioms together with those of NAT, (1) through (6), will give the specification of SET of NAT. Pictorically



Notice that the translations of axioms (47) through (49), as well as of the equality and namability axioms of sort Any, are theorems of NAT. Thus, the outcome is an interpretation of theories, of SET of ANY into SET of NAT. Hence, all the results proved about SET of ANY translate into provable properties of SET of NAT.

Similarly, we have the parameterized ADT's SORT of ANY and LIST of ANY. Application of assignment p will build the expected ADT's together with the corresponding interpretations

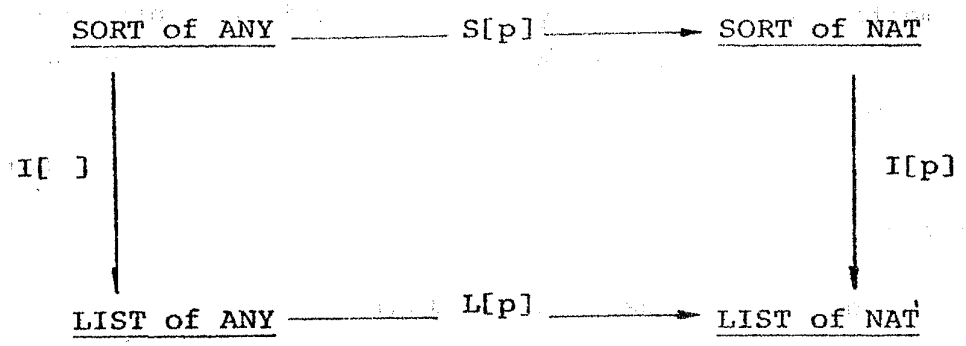
S[p] of SORT of ANY into SORT of NAT

and

L[p] of LIST of ANY into LIST of NAT

In the previous section we gave an implementation of SORT of NAT into LIST of NAT. Now, consider the implementation I[-], which is the same, i.e. consists of (29) through (43), except that it is the identity on sort Any and its symbols to, $C_0, C_1 \ldots$. This is a "parameterized" implementation of SORT of ANY into LIST of ANY.

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



The nice practical consequence is that we have the freedom to develop our program for sorting in a parameterized fashion and specialize the parameters to NAT when we please.

6. CONCLUSION

We have outlined an approach to ADT's, based on logic, which is a natural formalization 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 (formalized 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;
- parameterization 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 consequance 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 propagation 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 and certification.

REFERENCES

[D'77] J.Darlington - A synthesis of several sorting algorithms. Imperiall College of Science and Technology, Department of Computing; London, 1977.

[E'72] H.B.Enderton - A Mathematical Introduction to Logic. Academic Press, New York, 1972.

[G'77] J.V.Guttag - Abstract data types and the development of data structures. Comm. ACM, vol.20 (nº 6), p.396-404, June 1977.

[GH'78] J.V.Guttag and J.J.Horning - The algebraic specification of abstract data types. Acta Informatica, vol.10 (nº 1), p. 27-52, 1978.

[GTW'78] J.A.Goguen; J.W.Thatcher and E.G.Wagner - An initial algebra approach to the specification, correctness and implementation of abstract data types; in R.T.Yeh (ed) Current Trends in Programming Methodology, vol.IV; Prentice-Hall, Englewood Cliffs, 1978.

[H'72] C.A.R.Hoare - Proof of correctness of data representations. Acta Informatica, vol. 4, p. 271-281, 1972.

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

[LZ'77] B.Liskov and S.Zilles - An introduction to formal specifications of data abstractions; in R.T.Yeh (ed) Current Trends in Programming Methodology, vol. I; Prentice-Hall, Englewood Cliffs, 1977.

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

[Sh'67] J.R.Shoenfield - Mathematical Logic. Addison Wesley, Reading, 1967.

[VP'78] P.A.S.Veloso and T.H.C.Pequeno - Interpretations between many-sorted theories. 2nd Brazilian Colloquium on Logic, Campinas, 1978.

[WPPDB'80] M.Wirsing; P.Pepper; H.Partsch; W.Dosch; M.Broy - On hierarchies of abstract data types. Technische Univ. München, Inst. Informatik, 1980.