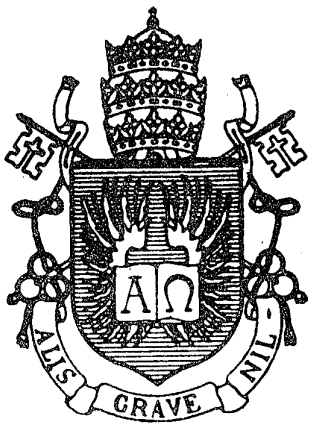


PUC

Series : Monografias em Ciência da Computação

No. 4/85

PROGRAMME DEVELOPMENT AS THEORY MANIPULATIONS

Paulo A. S. Veloso

Departamento de Informatica

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. 4/85 May 1985

Series Editor : Paulo A. S. Veloso

PROGRAMME DEVELOPMENT AS THEORY MANIPULATIONS *

Paulo A. S. Veloso

* Research partly sponsored by FINEP, CNPq and the Alvey Directorate (UK) .

RESUMO

O processo de desenvolvimento de programas consiste de manipulações em teorias. Apresentam-se argumentos para esta posição, não só como sendo uma descrição de um fato ("o que ocorre"), mas também como um ponto de vista ("o que deveria ocorrer"). A razão para isto é a observação que programas não manipulam diretamente objetos do mundo real, mas apenas suas representações simbólicas. Argumenta-se que o processo de desenvolvimento de programas (por meio de tipos abstratos de dados) é melhor descrito, explicado e entendido desta maneira sintática, sem recorrer a outras entidades. Dando suporte a essa posição temos uma teoria de tipos abstratos de dados e uma metodologia para desenvolvimento de programas por refinamentos sucessivos. A teoria, baseada em alguns conceitos lógicos (extensões conservativas e interpretações de teorias), fornece um tratamento simples e natural de especificações (talvez incompletas), implementação, parametrização e erros enquanto que a metodologia é orientada para a construção, sem muito esforço, de programas confiáveis e verificáveis. Parece-nos que ambos modelam o que (bons) programadores fazem (ou deveriam fazer).

Palavras chave : Desenvolvimento de programas,
tipos abstratos de dados,
refinamentos sucessivos,
metodologia de programação,
axiomas lógicos,
especificação formal,
implementação,
parametrização,
tratamento de erros,
extensão conservativa,
interpretação de teorias,
manipulações em teorias.

ABSTRACT

The process of programme development consists of manipulation of theories. A case is presented for this position, both as describing a fact ("what does happen") and as proposing a viewpoint ("what sould happen"). Its rationale is the realisation that programmes do not manipulate directly real-world objects but only their symbolic representations. The process of programme development (by means of abstract data types) is argued to be best described, explained and understood in this syntactical manner without resource to other entities. Supporting this position are both a theory of abstract data types and a methodolgy for programme development by stepwise refinements. The former, based on a few logical concepts (conservative extensions and interpretations of theories), offers a natural and simple treatment of (incomplete) specifications, implementation, parameterisation and errors. The latter is oriented towards the smooth construction of reliable, verifiable programmes. We feel that both model what (good) programmers (should) do.

Key words : Programme development,
abstract data types,
stepwise refinements,
programming methodology,
logical axioms,
formal specification,
implementation,
parameterisation,
error treatment,
conservative extension,
interpretation of theories,
theory manipulations.

ACKNOWLEDEGEMENTS

Research reported herein is a logical consequence of joint work with T. S. E. Maibaum and M. R. Sadler, both from the Dept. of Computing, Imperial College of Science and Technology, London, UK. In fact, a shorter version of this report - with T. S. E. Maibaum and M. R. Sadler as coauthors - is to be presented at the Third International Workshop on Software Specification and Design. Partial financial support from the Brazilian agencies CNPq and FINEP as well as the British Alvey Directorate is gratefully acknowledged. Discussions with F. E. P. Pessoa were very helpful.

CONTENTS

1. INTRODUCTION

The process of programme development consists of the manipulation of theories. This assertion can be taken as a fact (describing what does happen) or as a viewpoint (explaining what should happen) . The purpose of this paper is to argue for this position. In doing so we will put forward some arguments oriented towards convincing of the truthfulness of this fact, as well as other arguments that point out the usefulness of this viewpoint, since both kinds of arguments are closely related. The main point, insofar as practical aspects are concerned, is the fact that this viewpoint does shed some light on issues related to programme specification and development by means of stepwise refinements. In particular, it establishes clear distinctions amongst specification of desired behaviour, design decisions and programme text, which enable orthogonal developments, throughout the whole process [MVS].

Among the advantages derived from our viewpoint we mention being simpler and close to the actual programming practice as well as adequate for stepwise development and certification, permitting incomplete specifications, having simple and natural notions of implementation and parameterisation, and flexibility for dealing with errors and exceptions.

The structure of this paper is as follows. In the next section we mainly argue that the process of programme

specification and development has to do with syntactical objects and should be described and understood without resource to other entities. Section 3 introduces a simple example to illustrate a methodology for programme development, whereas section 4 advances an argument similar to that of section 2 for the data abstactions involved in the process, concentrating on their specification, much as axioms specify theories. Then, sections 5 and 6 show how these ideas can be carried further to the implementation phase, by providing and illustrating a description of the process of implementation of abstract data types in terms of syntactical manipulations of their specifications, which is argued to mimic what a programmer does, whereas section 7 deals with parameterised data abstractions by showing how parameter passing can be similarly described. The role played by exceptions and their handling in this framework is examined in section 8 and the concluding section emphasises the benefits for the process of programme development accrued from this viewpoint . We shall carry along the simple example introduced in section 3 in order to illustrate the main ideas.

We try to make the presentation as independent as possible of the particular kind of formalism [Sa] or institution [GB] employed, in the spirit of linguistic system [MT]. However, in some points, we shall resort to a terminolgy reminiscent of that of (first-order) mathematical logic [Sh].

2. PROGRAMMING : SINTAX VS. SEMANTICS

An important point is the realisation that programming has to do with syntactical objects rather than with their semantical counterparts. That a programme is a syntactical entity hardly needs any elaboration : a programme text is a syntactical description of the transformation effected by it, the function it computes [Kn]. This viewpoint is further stressed by the common practice of regarding an abstract programme manipulating an ADT (short for abstract data type) as a programme schema [Ma]. Of course, the very purpose of a programme is to transform input data into output results. But in doing so, a programme does not manipulate directly any real-world objects, only their symbolic representations [LT], which are syntactical (rather than semantical) objects. For instance, a programme to compute the greatest common divisor of two numbers will actually manipulate numerals, say, representing numbers, rather than the numbers themselves.

Let us now see how these considerations fit into the process of programme development by means of ADT's. The starting point of this process is a specification of the desired I/O (short for input-output) behaviour, which is given by a syntactical entity, written in a non-procedural formalism, for instance, a set of logical formulae. The end product consists of (the text of) an abstract programme manipulating an ADT together with (the text

of) a module implementing this ADT on more concrete ones. The ADT specification describes - in a non-procedural manner, say by means of axioms - the relevant properties of the data abstractions involved. Now, verifying that the abstract programme does satisfy the I/O specification consists of syntactical manipulations employing the ADT specification and, say, verification conditions [Mal. Similarly, the correctness of the implementation module is verified by syntactical processes, yielding the correctness of the entire programme.

Of course, the very idea of data abstraction involves hiding representation details. Conceptually, this means dealing with abstract structures or, more aptly, with their behaviour. And this is the point we wish to stress : the important aspects of the behaviour of these abstract structures can be captured in non-procedural specifications, say by axioms, which are syntactical descriptions of their relevant properties.

So, the objects manipulated by programmes are actually syntactical entities described by their properties as captured in non-procedural (say, axiomatical) specifications, at various levels of abstraction. But, what about the programmes themselves, or rather their behaviour ? After all, programming languages, besides syntax, do have semantics. However, notice that the so-called axiomatic semantics, a la Hoare, consist of axioms and rules of inference, thereby ascribing meaning to programmes entirely by syntactico-deductive means. And even the denotational methods give descriptions of functions, rather than functions proper.

3. METHODOLOGY FOR PROGRAMME DEVELOPMENT

In order to make the discussion more concrete we now introduce our simple running example : the problem of constructing a programme for sorting, say, into ascending order. We can describe it as follows. We want a program P manipulating objects from an ADT called SORT of IOD, which has three sorts, Iod, Set, and Seq specified as (having as their realisations) a totally ordered domain, the domain of the finite sets of elements from this domain, and that of the finite sequences of such elements, respectively. (We will introduce some of the operation and predicate symbols of this ADT as they are needed .) The desired I/O behaviour for program P is that, upon receiving an input t_in from sort Set, it will terminate producing an output q_out of sort Seq, such that is-sort(t_in,q_out) holds, where this predicate is defined by

(1) $\quad$ is-sort(t,q) <--> is-incr-ord(q)&is-same(t,q)

We omit the formulae defining the predicate symbols above so as to mean, respectively, that "q is increasingly ordered" and "t and q have the same elements".

A simple and natural way to sort a set t into a sequence q consists of repeatedly removing an element from t and inserting it into its proper place in q . This idea can be captured in the invariant inductive assertion

(2) $\underline{is\text{-}transf}(t,q) \longleftrightarrow (\forall d{:}\underline{Tod})[\ \underline{bel}(d,t\_in) \longleftrightarrow (\underline{bel}(d,t) \vee \underline{occ}(d,q))\ ]$

Here we are using the primitive predicate symbols bel and occ of the ADT SORT of TOD whose meanings are formally specified to be "element belonging to set" and "element occurring in sequence".

Now we can write a first version of our abstact program AP, already annotated :

```
var t , t_in : Set ;
var q , q_out : Seq ;
var  d : Tod ;
t := t_in ; q := lmbd ;
  { is-transf(t,q) & is-incr-ord(q) }
while ¬empty(t) do
    d := chs(t) ;
    q := ord-ins(q,d) ;
    t := rem(t,d)
end ;
q_out := q
  { is-sort(t_in,q_out) }
```

This program employs some new symbols, which are not in the ADT SORT of TOD and deserve some explanation, namely chs and ord-ins. Their intended meanings are, respectively, a nondeterministic choice of an element from a set and ordered insertion of an element into a sequence.

In the next section we will deal with the formal specification of the behaviour of these operations. For the moment, let us recall that we are arguing that the process of programme development by means of ADT's can be described, explained and understood - and more fruitly so - entirely by syntactico-axiomatical means, without resource to other entities. In this connexion two points are worth mentioning. Firstly, the process can - and should - be iterated, and we will come to this point in section 6. Secondly, it does involve some "insight" in taking good design decisions and we are not claiming to be able to automate this "creative" aspect (even though our framework appears to be a reasonable starting point for trying to semi-automate these aspects interactively). What we wish to stress here is the feature that these design decisions can be recorded in formulae which participate in the process, a point to be elaborated upon in the sequel.

In particular, notice that we wrote the above AP by relying on our intuitive understanding of the intended realisations of symbols such as chs and ord-ins. This will be somewhat alleviated when we provide formal specification for their behaviours. Here we are employing our version of the methodology of incomplete specifications for programme development (proposed by Lucena and Pequeno [LP,PL]), which can be summarised as follows :

- given an I/O specification and an ADT (as SORT of TOD);
- extend this ADT by postulating new operation and predicate symbols appropriate to the problem at hand;

- write an abstract programme relying on the intuitive understanding of the postulated behaviour of these new symbols;
- specify formally the behaviour of the new symbols just enough to be able to prove the correctness of the abstract programme with respect to the specification (using this goal as a guide in formulating the axioms);
- iterate the process, if necessary, for instance to write the implementation modules.

We just mention that one advantage of specifying as little as possible of the behaviour of the new symbols is that it makes easier to ensure that the extension performed is conservative (as defined in section 6). Other advantages are, of course, easing the specification task and allowing more freedom for subsequent decisions.

4. SPECIFYING ABSTRACT DATA TYPES

In the preceding section we have written an abstract programme to solve our example problem of sorting. This programme uses some symbols not present in the original ADT SORT of TOD; in fact we relied on our intuitive understanding of their postulated behaviour in order to write the programme. We now provide formal specification for these new symbols. However, instead of trying to formalise directly the intended behaviour of these operations, we first write verification conditions [Ma] for the above program and then specify these operations just enough so as to guarantee the correctness of the program.

One verification condition for partial correctness is

(3) $\neg$empty(t) ---> { is-incr-ord(q)&is-transf(t,q)--->
[d=chs(t) ---> is-incr-ord(ord-ins(q,d)
&is-transf(rem(t,d),ord-ins(q,d))] }

Now we can suggest some axioms that are sufficient to prove this verification condition. A natural suggestion would consist of the following three axioms

(4) (∀t:Set) [$\neg$empty(t) ---> bel(chs(t),t)]

(5) (∀q:Seq)(∀d:Tod) [is-incr-ord(q) --->
is-incr-ord(ord-ins(q,d)]

(6) $(\forall q{:}\underline{Seq})(\forall d,d'{:}\underline{Tod})\ [\ \underline{occ}(d,\underline{ord\text{-}ins}(q,d')) \leftrightarrow (\ d=d' \vee \underline{occ}(d,q)\ )\ ]$

since the specification of SORT of TOD already includes

$$(\forall d{:}\underline{Tod})[(\underline{bel}(d,t)\leftrightarrow\underline{bel}(d,t'))\rightarrow t=t']$$

$$(\forall d{:}\underline{Tod})[\underline{bel}(d,\underline{rem}(t,d'))\leftrightarrow(\neg d=d'\&\underline{bel}(d,t))]$$

Similarly, we can obtain the other axioms so that the partial correctness of the abstract programme AP can be proved. For termination, we can employ, for instance, the method of well-founded sets [Ma], which will generate some further axioms. (Examples of such axioms are namability axioms [MVS].) Then our abstract programme will be supported by an ADT, let us call it EXT SORT of TOD, now formally specified, which is an extension of the original SORT of TOD , actually a conservative one (cf. section 6).

Some comments are in order. The specification of the behaviour of the new symbols does not have to provide definitions for them; partial or incomplete definitions - that just constrain the possible realisations - can be quite sufficient, and in some cases more appropriate . An example for this point is the case of chs, whose intended meaning is a (possibly nondeterministic) choice of an element from a set. At this level of development it is not at all clear how such a choice should be effected, thus a complete specification of the behaviour of chs is not easy nor necessary. Also, our programme never attempts to apply chs to the empty set, thus the meaning of the term chs(phi) can be left

open. The only axiom in our specification involving chs is (4), which states exactly what is required in order to guarantee the correctness of the programme. Notice in particular that the specification need not be strong enough to enable us to decide whether a term like chs(ins(ins(phi,d),d')) is equal to d or d' .

5. IMPLEMENTING ABSTRACT DATA TYPES

We now have an abstract programme, called AP, manipulating an ADT EXT SORT of TOD formally specified so as to guarantee the correctness of AP with respect to the given I/O specification. We now must provide an implementation for this ADT. For the sake of the argument let us say that we are to implement it on the ADT LIST of TOD , with sorts List and Tod , and the usual list operations nil , cons , tl , and hd .

Now, how does a programmer proceed in designing an implementation as the one required above ? He has to represent the objects, operations and predicates of the more abstract data type in terms of the more concrete one. In order to represent the objects, he assigns to each sort - Tod, Set, and Seq - of the ADT EXT SORT of TOD a corresponding sort of LIST of TOD (or perhaps a "combination" thereof) , say

Tod ⟼ Tod , Set ⟼ List , Seq ⟼ List .

However, not every list will be used to represent a set; so he identifies within the sort List a part that will be used to represent the objects of Set . Now, in order to represent the operation and predicate symbols of EXT SORT of TOD the programmer writes for each such symbol a procedure, using as primitives the symbols of LIST of TOD, which is supposed to define it in terms of the available concrete primitives (perhaps together with some

auxiliary procedures and symbols) . Thus the implementation is described by a cluster-like module written in a programming language supporting such features.

We can factor out the details of the particular programming language employed to write such modules by focussing on their specifications [VPM]. Then verifying the correctness of the implementation can be naturally split into two separate tasks :

(a) verify that the module satisfies its I/O specification (this is basically the usual task of programme verification [Gu]);
and

(b) verify that the specification of EXT SORT of TOD is satisfied by deriving it from the specification of LIST of TOD together with that of the module.

It is to this latter task that we now turn to. Before this, however, notice that this description is well in accordance with the following methodology to develop an implementation : first, design the the I/O specification of the module, then programme it.

Let us take a closer look at a possible representation of an operation like chs by a procedure on LIST of TOD. Such a procedure may be regarded as a complete definition of chs in terms of the available primitives. On the other hand, its I/O specification does not have to provide all the details; in particular it does not have to enable us to predict the exact values returned in each case; properties of such values may be a sufficient specification. Thus, we regard this I/O specification

as a partial, incomplete, but sufficient, "definition" of chs. Similarly, the representation part of the implementation module gives rise to axioms capturing (some of) the so-called representation invariants [Gu], by means of relativisation predicates.

In the case of our running example, we want to implement the ADT EXT SORT of TOD on LIST of TOD. The relativisation predicate for sort Set is set-rep, which is required to satisfy the following natural axiom

$$(\forall l{:}\underline{list})[\underline{set\text{-}rep}(l) \leftrightarrow (\forall d{:}\underline{Tod})[\underline{tod\text{-}rep}(d) \rightarrow (\underline{occ}^*(d,l) \rightarrow \underline{once}(d,l))]]$$

where tod-rep is the relativisation predicate for sort Tod, occ* is a new predicate symbol (to correspond to occ) and once is an auxiliary predicate symbol, required to satisfy some axioms which we omit. Corresponding to the operation symbol ord-ins of EXT SORT of TOD we add to LIST of TOD an operation symbol ord-ins* declared to be of functionality (List,Tod)--->List and the axiom

$$\underline{ord\text{-}ins}^*(l,d)=l' \rightarrow \underline{is\text{-}incr\text{-}ord}^*(l') \,\&\, (\forall d'{:}\underline{Tod})[\underline{occ}^*(d',l') \leftrightarrow (d'=d \vee \underline{occ}^*(d',l'))]$$

Similarly, for chs we have

$$\underline{chs}^*(l)=d \rightarrow \underline{occ}^*(d,l)$$

Notice that these axioms are not classical explicit definitions of the new symbols (those with *). These and other axioms -

omitted here - are added to the specification of LIST of TOD to extend it to a new ADT, call it LIST of TOD by EXT SORT of TOD , the only constraint being the conservativeness of this extension, as elaborated upon in the next section.

An interpretation of the ADT EXT SORT of TOD in the "more concrete" ADT LIST of TOD by EXT SORT of TOD is given by the assignment of sorts from the former to the latter, as above, of relativisation predicates in the latter to the sorts of the former, and an assignment of symbols of the former to corresponding *-symbols of the latter. This - together with a mapping to translate variables properly - induces a translation of formulae from the former to the latter. For instance axiom (4) is translated into

$$(\forall t^*{:}\underline{List})\ [\ \underline{set\text{-}rep}(t^*) \longrightarrow$$
$$(\ \neg\underline{empty}^*(t^*) \longrightarrow \underline{bel}^*(\underline{chs}^*(t^*),t^*)\ )\ ]$$

For the relativisation predicate of sort Set, for instance, we have the non-triviality condition

$$(\exists l{:}\underline{List})\ \underline{set\text{-}rep}(l)$$

and closure conditions such as

$$(\forall l{:}\underline{List})(\forall d{:}\underline{Tod})\ [\ \underline{set\text{-}rep}(l)\&\ \underline{tod\text{-}rep}(d) \longrightarrow$$
$$\underline{set\text{-}rep}(\underline{rem}^*(l,d))\ ]$$

6. IMPLEMENTATION AS THEORY MANIPULATIONS

With the preceding section as motivation, we are now ready to describe, in general, the process of implementation of ADT's in terms of manipulations on their specifications.

Let us assume that we wish to implement an ADT A on another "more concrete" one C . We first expand the language of C, by adding some new symbols, including some to correspond to those of A, and then extend the specification of C by adding axioms which specify properties of these new symbols among themselves as well as with respect to the old ones. This gives rise to the specification of a new ADT, call it B , extending C . In order to guarantee consistency we require that this extension from C to B be conservative [Sh], i. e. any formula deduced in B involving only symbols of C must already be deducible in C (semantically, this means that any realisation of C can be expanded to a - not necessarily unique - realisation of B). Finally, the matching of the symbols of A to those of B is effected by the logical concept of interpretation of theories [Sh]. This amounts to an assignment from symbols of A to corresponding ones in B and relativisation predicates in B for the sorts of A . This (together with a translation of variables) induces a translation of formulae of A into formulae of B, which is required to map axioms of A to theorems of B and to ensure that the relativisation predicates are nontrivial and closed under the the translations of the

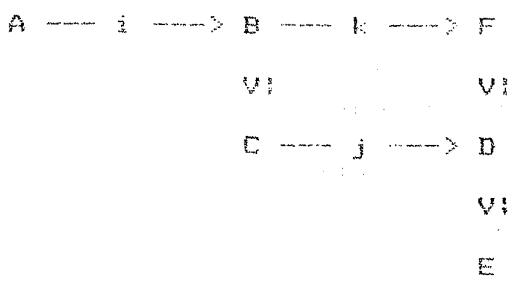
operations of A . The semantical effect of these conditions is to guarantee that every realisation of B induces a - not necessarily unique - realisation of A . Summing up, an implementation of A on C consists of an interpretation of A into a conservative extension of C ; thus every realisation of C gives rise to some realisation of A [VPM].

Some interesting points connected with this notion of implementation are worth mentioning : flexibility, orthogonality and composablity, besides naturalness. Firstly, let us comment on the flexibility. We have been treating '=' as logical identity for simplicity only. It is quite simple and natural to treat equality as a nonlogical symbol, "open to interpretation", by introducing in lieu of '=' a binary predicate symbol equal_s for each sort s , together with axioms imposing that their realisations are to be congruence relations. (This is especially natural for sorts consisting of "structured data", when equality is usually treated in a "componentwise" manner.) This will mean that in interpreting a specification into another one equal_s will now be translated into some symbol of the latter specification, which adds some extra flexibility to the notion.

Orthogonality of refinements is another important benefit obtained with this notion of implementation, stemming from the fact that we concentrate on its logical aspects. Indeed, recall that when we implement A on C we add to C formulae rather than programmes. These formulae record the design decisions taken in the implementation, not yet their actual coding into a programme text. Thus, we achieve orthogonality : the process of actually

coding modules is independent of - and can proceed in parallel with - the process of further (logical) refinements, say in implementing C on another ADT. A simple example of this feature is provided by "families of programmes" [Pa]. The successive refinements record the various design decisions, permitting the development of several algorithms, say for sorting, naturally classified into families according to the strategies employed [Da,VL].

Composability is a third dividend : the process can be easily iterated [MSV]. An implementation of A on C composes naturally with one of C on E to yield a composite implementation of A on E , as illustrated in the diagram below where the horizontal arrows denote interpretations and the 'V!'s denote conservative extensions



Here it is worthwhile noting that this composition mimics exactly what a programmer does in simply putting together the component modules to obtain a composite implementation, which argues for the naturalness of our formulation. This is possible because we require only that F be a conservative extension of E where we can interpret A , rather than insisting that F be minimal in some sense (as, say, a pushout [Eh]), which would force us to

eliminate the mediating language.

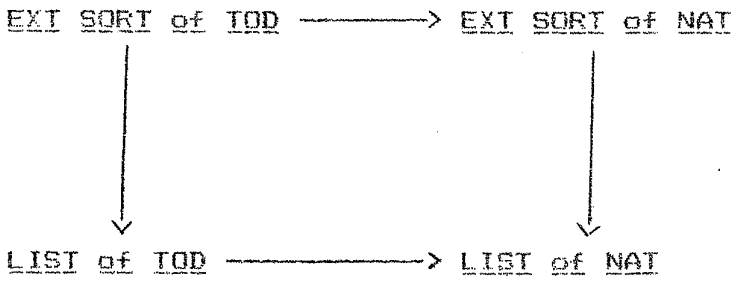
7. PARAMETERISATION AS THEORY MANIPULATIONS

In the preceding sections we have illustrated by means of a simple example how our ideas can be applied , obtaining a programme to sort into ascending order sequences represented by lists. The exact nature of the elements of these lists is unimportant, as long as they come from a totally ordered domain, which was the rationale for naming the various ADT's involved as "...of TOD". In this sense, these are parameterised ADT's.

Suppose now that we wish to apply this programme for sorting sequences of, say, naturals, given as an ADT NAT . Thus we want to substitute the (real) argument NAT for the (formal) parameter TOD so that the parameterised ADT's "...of TOD" become "...of NAT". This involves replacing symbols of TOD, such as the symbol lt for the ordering by some symbol lt* in NAT (or in some conservative extension thereof). Of course this replacement should be such that the properties of TOD required in the development are satisfied. But this is what is demanded of an implementation. This suggests that we can regard the process of specialising an ADT such as SORT of TOD to SORT of NAT by parameter substitution (TOD ---> NAT) quite naturally as an implementation of the former on the latter. It is a special kind of implementation, though, in that we generally want the relativisation predicates to hold for all the objects of its sort.

Thus the distinction between implementation and (correct) parameter passing is somewhat blurred at this level, since both are mechanisms for specialisation of ADT's. And this is not only advantageous but also natural. For, consider the following sequence of specialisations : SORT of TOD is first specialised to SORT of TODLE, where TODLE is the ADT "totally ordered domain with least element", and the latter is then specialised to SORT of NAT. This sequence of specialisations could be made even longer by interpolating gradually some properties of the ordering of the naturals, such as "discreteness", "having no last element", etc.

One advantage of this formulation of parameterised ADT's is that parameter passing clearly composes. Moreover, implementation and parameter passing commute (under some mild "orthogonality" restrictions) [MVS1], as indicated in the diagram below, where the horizontal arrows correspond to parameter passing and the vertical ones to implementation



The nice practical consequence is that we have the freedom to develop our programme for sorting, say, in a parameterised fashion and specialise it to natural numbers when we please.

Another, at first somewhat puzzling, consequence is the fact that we can regard any ADT as a parameterised one, for any symbol can be viewed as a open to a possible later replacement, hence as a (formal) parameter.

8. TO ERR OR NOT TO ERR

An important extra benefit accrued from regarding ADT's as (possibly incompletely specified) theories is not exaggerating the importance of errors. Prototypical cases of errors in ADT's arise in asking for the contents of an empty object, e. g. the top of an empty stack or the head of the empty list. Let us consider the latter; of course there is no "natural" value to assign to hd(nil) . Some approaches for dealing with this problem involve partial operations or errors [Gu,Go]. However, notice that asking for the value of such term is a semantical question, in that it only makes sense once we are given a realisation of the ADT. Another alternative, more adequately mirroring the feeling of "absence of a natural value to be assigned" is our proposal to leave the value of hd(nil) open. Let us clarify it. Our specifications do not have to enable us to compute a unique primitive term (of sort Tod) to be the value of hd(nil). Rather, we can have several (non-isomorphic) realisations of the ADT LIST of TOD differing on the value they assign to hd(nil). Notice, however, that in a given realisation the value of hd(nil) is not to be undefined nor is it to be an "abnormal" error element.

Our proposal, couched in semantical terms, embodies a non-uniqueness viewpoint : an ADT is a class of (not necessarily isomorphic) realisations. Syntactically, however, this amounts

to dealing with specifications that do not have to be complete (as those of [Go]) and not even sufficiently complete (as those of [Gu]) . Notice that this causes no extra conceptual difficulty, quite on the contrary, it does simplify the task of obtaining a formal specification, witness our methodology in section 3.

The preceding discussion is intended to support our proposal from the "theoretical" point of view. Now, from the viewpoint of programming two considerations might be offered in its support. First, the usual formulations with error propagation do not adequately model practice : a programme does remain forever in an error state; rather, upon entering an error state, a programme has its execution aborted or an exception-handling routine is called. Second, good programmes do not by themselves enter into error states. Indeed, even if perhaps somewhat utopically, a good programmer should not ask for the head of a list without first testing whether it is empty or not. (Notice that it was the presence of this test that allowed us to leave chs(phi) open in section 4.) Of course this test may be absent during some phases of tuning. Then, what happens if a programme inadvertently tries to compute hd(nil) or chs(phi) ? Something will be produced, maybe not predictable by our ADT specification, but certainly not in conflict with it; for the result is specified at a later refinement stage, when it is more natural.

An important point to notice, however, is that our approach downplays the importance of errors but does not prohibit dealing with them. Indeed, at some stage of the development it may be

desirable to take a term whose value was originally left open and to equate it to some special (error) constant. And in this case we may or may not wish to include error propagation and/or handling. As long as we have a conservative extension of the original specification we can take the course we see fit. The design decision that hd(nil) is going to be at this level of refinement, say O_3 is not very different from that of implementing a previously nondeterministic choice as now choosing, say, the first element. The point is, once again, that we can postpone a design decision until a stage where the consequences of taking it become clearer.

9. CONCLUDING REMARKS

We have argued that the process of programme development can be best described, explained and understood as consisting of the manipulation of theories. The starting point for this is the realisation that programmes manipulate, not semantical objects directly, but only symbolic representations thereof. This is especially clear in the task of programme verification, which consists of axiomatico-deductive derivation of the verification conditions. Semantical considerations do have a role to play in the process, mainly in its "creative" phases. However, even this is somewhat alleviated by the methodology we employ for program development : given an ADT C and an I/O specification; postulate an ADT A and write an abstract programme AP on A based on the intuitive understanding of A; specify A formally to prove the correctness of the AP (using the verification conditions as a guide); implement A on C , again by applying the methodology.

An important feature of this viewpoint is that it concentrates on the non-procedural (logical) aspects, enabling design decisions to be recorded into formulae which participate in the process. Thus, at each stage we have two possible next steps, namely further logical refinements and actual coding, and they can proceed in parallel due to their orthogonality.

This approach has an appealing theoretical simplicity, using just two basic logical concepts - conservative extension and

interpretation of theories - in order to deal with incomplete specifications and provide a smooth and uniform treatment of implementation, parameterisation and errors. From the practical side, we feel that this approach is closer to the practice of programme development by stepwise refinements in that it mimics what good programmers (should) do.

REFERENCES

[Da] J. Darlington, "A synthesis of several sorting algorithms", Acta Informatica, vol. 11, no. 1, pp.1-30, 1978.

[Eh] H. D. Ehrich, "On the theory of specification, implementation and parameterization of abstract data types", J. ACM, vol. 29, no. 1, pp.206-227, 1982.

[Go] J. A. Goguen, "Abstract errors for abstract data types", E. J. Neuhold (ed.) Formal Description of Programming Concepts, Amsterdam : North-Holand, 1978 (pp. 491-525).

[GB] J. A. Goguen and R. M. Burstall, "Introducing institutions", Logics of Programs, Berlin : Springer-Verlag, 1984 (pp.221- 256).

[Gu] J. V. Guttag, "Abstract data types and the development of data structures", Comm. ACM, vol. 20, no. 6, pp.396-404, 1977.

[Kn] D. E. Knuth, "Algorithm and program : information and data", [letter to the editor], Comm. ACM, vol. 9, no.9, p.654, 1966 [repr. in Comm. ACM, vol. 20, no. 1, p.56, 1983].

[LP] C. J. Lucena and T. H. C. Pequeno, "A view of the program derivation process based on incompletely specified data types", Res. Rept. MCC 25/77, Pont. Univ. Catolica,

Rio de Janeiro, 1977.

[LT] H. Ledgard and R. W. Taylor, "Two views on data abstraction", Comm. ACM, vol. 20, no. 6, pp.382-384 , 1977.

[MSV] T. S. E. Maibaum, M. R. Sadler and P. A. S. Veloso, "Logical specification and implementation", Proc. 4th Conf. on Foundations of Software Technology and Theoretical Computer Science, Bangalore : Tata Inst., 1984.

[MT] T. S. E. Maibaum and W. M. Turski, "On what exactly is going on when software is develpoed step-by-step", Proc. 7th Internat. Conf. on Sotware Engineering, Los Angeles : IEEE Computer Science Press, pp.528-533, 1984.

[MVS] T. S. E. Maibaum, P. A. S. Veloso and M. R. Sadler, "A theory of abstract data types for program development : bridging the gap ?", Joint Conf. on Theory and Practice of Software Development, Berlin, 1985.

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

[Pa] D. L. Parnas, "Designing sotware for ease of extension and contraction", IEEE Trans. on Software Engineering, vol. 5, no. 2, pp.128-138, 1979.

[PL] T. H. C. Pequeno and C. J. P. Lucena, "An approach to data type specification and its use in program verification", Inform. Proc. Letters, vol. 18, no. 2, pp. 98-103, 1979.

[Sa] M. R. Sadler, "Mapping out specifications", [position paper], Alvey Workshop on Formal Aspects of Specifications, Swindon, 1984.

[Sh] J. R. Shoenfield, Mathematical Logic. Reading, MA : Addison-Wesley, 1967.

[VL] P. A. S. Veloso and M. A. Lopes, "Sorting by divide-and-conquer data types : an example of problem-solving", Anales de la VII Conf. Latinoamericana de Informatica, Caracas, 1980 (pp 345-354).

[VPM] P. A. S. Veloso, F. E. P. Pessoa and T. S. E. Maibaum, "Theory of abstract datra types for programming : a logical approach" [in Portuguese], Anales de la VII Conf. Latinoamericana de Informatica, Lima, 1982 (pp.423- 430).