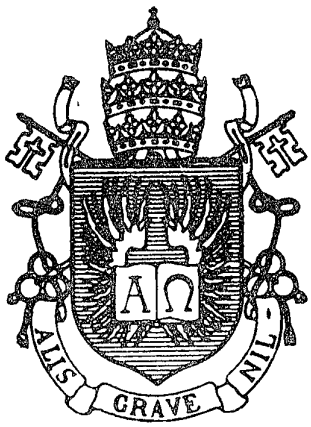

PUC

Series : Monografias em Ciência da Computação

No. 6/85

ON THE ROLE OF (DATA) ABSTRACTION IN PROGRAM DEVELOPMENT AND PROBLEM SOLVING

Paulo A. S. Veloso

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. 6/85 August 1985

Series Editor : Paulo A. S. Veloso

ON THE ROLE OF (DATA) ABSTRACTION IN PROGRAM DEVELOPMENT AND PROBLEM SOLVING *

Paulo A. S. Veloso

* Research partly sponsored by FINEP and CNPq.

ABSTRACT

Abstraction has played an extremely important role in the process of program development and should play an even more important one in that of problem solving. Program development by means of stepwise refinements can be more confidently carried out by employing abstract data types (ADT's, for short). This allows the programmer to concentrate on the problem by employing (abstract) operations close to it rather than worrying about their detailed representations from the very beginning. An even more flexible variation employs incompletely specified ADT's, which frees the programmer from the need to provide complete sspecifications for the data abstractions involved. A further step of abstraction leads to the concepts of problem and problem-solving method. A concrete problem (CP, for short) is a two-sorted mathematical structure consisting of two domains, one of (input) data and one of (output) results, and a binary relation between them, the problem condition; a solution is a function assigning results to data so as to satisfy the problem condition. An abstract problem (AP, for short) is a class of CP's specified by their properties, say by axioms. In solving a problem one generally defines a solution for it by extending its language, which leads to the idea of PSM (short for problem-solving method) as an extension mechanism for defining functions. A PSM is a module-like linguistic construct, operating on an ADT, with a designated main procedure. Thus, AP's and PSM's are regarded as ADT's and applying a PSM to an AP amounts to implementation of the corresponding ADT's, in that the PSM when interpreted on any CP of the AP defines a solution for it. Among the PSM's that have already been precisely formulated in this manner we mention reduction, divide-and-conquer, the greedy method, dynamic programming, etc. This framework appears to be adequate to a precise investigation of problem-solving methods aiming at a better understanding of their applicability and power.

Key words : Abstraction, program development, problem solving, abstract data type, theory of problems, problem-solving method, specification, implementation, correctness, methodology.

RESUMO

A abstração tem desempenhado um papel extremamente importante no processo de desenvolvimento de programas, um papel mais importante ainda lhe estando reservado no de resolução de problemas. O desenvolvimento de programas por refinamentos sucessivos pode ser efetuado com mais confiança quando são empregados tipos abstratos de dados (abreviado TAD's). Isto permite que o programador se concentre no problema, empregando operações (abstratas) próximas a ele, em vez de se preocupar com detalhes de suas representações desde o princípio. Uma variante mais flexível ainda emprega especificações incompletas, o que libera o programador da necessidade de fornecer especificações completas para as abstrações de dados envolvidas. Com mais um passo de abstração chega-se aos conceitos de problema e de método de resolução de problemas. Um problema concreto (abreviado PC) é uma estrutura matemática consistindo de dois domínios, um de dados (de entrada) e outro de resultados (de saída), e de uma relação binária entre eles, a condição do problema, e uma solução é uma função atribuíndo resultados a dados de maneira a satisfazer a condição do problema. Um problema abstrato (abreviado PA) é uma classe de PC's especificada por suas propriedades, por exemplo, por meio de axiomas. Ao se resolver um problema, geralmente se define uma solução para ele por meio de extensões de sua linguagem, o que nos leva a idéia de MRP (abreviatura para método de resolução de problemas) como um mecanismo de extensão para definir funções. Um MRP é um mecanismo lingüístico, semelhante a um módulo, operando em um TAD, com um procedimento principal distinguido. Assim, tanto PA's como MRP's são encarados como TAD's ; a aplicação de um MRP a um PA consistindo da implementação dos TAD's correspondentes, de modo que o MRP, quando interpretado em qualquer PC do PA , define uma solução para ele. Dentre os MRP's que já foram precisamente formulados dessa maneira, pode-se mencionar : redução, divisão-e-conquista, método guloso, programacao dinâmica, etc. Este contexto parece ser adequado para uma investigação precisa de métodos de resolução de problemas com o objetivo de uma melhor compreensão de sua aplicabilidade e poder.

Palavras chaves : Abstração, desenvolvimento de programas, resolução de problemas, tipo abstrato de dados, teoria de problemas, método de resolução de problemas, especificação, implementação, correção, metodologia.

CONTENTS

1. INTRODUCTION AND OVERVIEW

This paper discusses the crucial role played by (data) abstraction in the processes of problem solving and program development. The basic idea of data abstraction, namely abstracting away from representation details, is pushed further with the aim of providing precise definitions to capture our intuitions about the somewhat vague notions of problem and problem-solving method. These definitions are couched in terms of abstract data types, thereby providing a framework for their precise investigation as well as for the design of methodologies.

The basic motivation for the research reported herein is critical analysis aiming at a " rational reconstruction " of our intuitions about some extremely important, but somewhat vague, notions. An analogy with mathematical logic might clarify our standpoint. In mathematical logic one has precise definitions for the notions of proof and theorem, however one does not necessarily learn the art of proving theorems by studying mathematical logic. Analogously, we expect to provide a framework where problems and problem-solving methods can be rigorously investigated, and teaching the art of solving problems is not our main goal. Of course, when dealing with problems, it is hard to forget about the urge to solve them. So, we will have something to say concerning problem-solving methodologies.

The structure of this paper is as follows. In the next

section we review the role of (data) abstraction in the process of program development by stepwise refinements, which is carried further in section 3 where incompletely specified private abstract data types are argued to be more adequate for this process. Section 4 presents the main general concepts pertaining to abstract data types, which are employed to formulate the concepts of problem and solution in section 5. Then, section 6 introduces the concept of problem-solving method, exemplifying it with decomposition and reduction. In section 7 we illustrate how our formulations can be methodically applied to program development and problem solving. Finally, section 8 presents some concluding remarks and prospects for future research.

2. THE ROLE OF ABSTRACTION IN PROGRAM DEVELOPMENT

Abstraction plays an extremely important role in program development. It permits the process of program development by means of stepwise refinements to be more confidently carried out. This is particularly true if ADT's (short for abstract data types) are employed.

Let us take a closer look at the process of program development with ADT's [LZ'74]. The basic idea behind ADT's is allowing the programmer to concentrate on the problem itself employing operations close to it and to his/her intuition. This frees the programmer from worrying about the details of data representation right at the very outset of the process. The programmer first writes an abstract program operating on an ADT, and makes sure that this program is correct, by using for this purpose the ADT spec (short for specification). [Thus the properties stated in the ADT spec play at this stage the role of lemmas for the proof of correctness of the abstract program.] At a later stage the programmer chooses a representation for the data objects and writes procedures to realize the previous abstract operations on the data so represented, making sure that the properties given in the ADT spec are indeed satisfied by this implementation. [Thus at the implementation phase the properties given by the ADT spec, previously used as lemmas, are actually verified.]

The program obtained by usage of ADT's is naturally factored into an abstract program, operating on an ADT, and an implementation module, realizing the ADT in terms of more concrete ADT's. This factorization has two nice consequences. Firstly, the process can be iterated, which is well in accordance with the idea of stepwise refinements : the implementation module itself, for instance, can also be factored into components. Secondly, this factorization induces naturally a corresponding factorization in each programming task : design, coding, verification, testing, documentation, etc. For instance, the verification of the program is naturally decomposed into the verification of the abstract program and the verification of the implementation module, which are completely independent of each other and can be carried out separately.

In employing ADT's a crucial role is played by their specifications. In fact, the properties of the ADT's have to be given in a manner completely independent of any particular representation. This is absolutely necessary if one is going to take full advantage of the idea of abstraction : the "communication" between the abstract program and the implementation module, which encapsulates the representation, is provided only by the ADT spec. The connection between the abstract program and the implementation module should be in accordance with Parnas's " information hiding " idea [GJ'82] : the abstract program should employ only abstract operations as provided by the ADT and should not hinge on any representation detail hidden in the implementation module. Thus, the ADT spec

should provide information relevant to understanding the behavior of the abstract operations employed in the abstract program without relying on any particular representation, which is available only inside the implementation module.

Let us examine a very simple example : sorting sequences into ascending order. In writing an abstract program to sort sequences we should not have to worry about how the sequences are actually (going to be) represented. All that we need is some repertoire of abstract operations to manipulate abstract data objects, the behavior of such operations being given in the ADT spec, which enable us to write a correct abstract program to sort those abstract data, later to be represented as sequences. Moreover, it is clear in this case of sorting that the nature of the elements of the sequences is completely irrelevant as long as they come from a linearly ordered domain.

In order to actually present an ADT spec we need a specification formalism [VF'85]. One of the most widely employed such formalisms is the algebraic one [Gu'77], where the spec consists of (conditional) equations, especially the initial algebra approach [GTW'78]. In this approach, the behavior of our ADT of sequences to be sorted would be specified by means of (conditional) equations which tell us basically that some pairs of terms, denoting sequences of operations, have the same effect. In the initial algebra approach this spec is to be complete, in other approaches a sufficiently complete spec will be enough. But in either case we must specify the effect of, say, reading an element from an empty sequence, even if our abstract program,

wisely enough, never attempts to do this. So we will have to overspecify our ADT as far as the abstract program is concerned. Moreover, we shall be dealing with parameterized ADT's in order to capture the idea that the elements of the sequences are not fixed beforehand.

The two features of the algebraic approach presented above illustrate a burden placed on the specifier by the consideration of specifications which provide too much information. Such spec's may be appropriate for public (general purpose) ADT's, which are specified independently of any particular program. But, as far as program development is concerned, we may say that this view falls short of exploiting the full benefits of data abstraction.

3. PRIVATE ABSTRACTIONS IN PROGRAM DEVELOPMENT

We have seen in the preceding section that the usage of public (general purpose) ADT's puts a heavy burden on the specifier due to its tendency to overspecify the properties needed for a particular program. A natural way to try to alleviate this situation is the consideration of private (special purpose) ADT's [VPM'82], in the sense that they are developed for a particular application rather than in isolation. For, in the case of a private ADT an incomplete spec can be enough to guarantee the correctness of the corresponding abstract program.

We shall illustrate the usage of incompletely specified private ADT's by means of the example of sorting. Let us see what kind of data abstraction is naturally appropriate for the problem at hand. We may regard it as an ADT with three sorts, one for the input data, another one for the output results, and a third one, Tod , to play the role of a totally ordered domain of elements constituting the sequences to be sorted. Also, as the input sequences are not necessarily sorted, we may disregard their ordering and consider them more like sets than like sequences; thus we call the input sort Set and the output sort Seq .

We want a program that will receive an input st_in of sort Set and will produce an output sq_out of sort Seq satisfying the input-output specification is-sort(st_in,sq_out) . Now, a simple

and natural way to sort a set st into a sequence sq consists of repeatedly removing elements from st and inserting them into their proper places in sq until st is exhausted. This idea can be captured in the following invariant inductive assertion

$$\underline{is\text{-}transf}(\underline{st},\underline{sq}) \;\langle\text{---}\rangle\; (\forall e{:}\underline{Tod})\; [\; \underline{bel}(e,\underline{st\_in}) \;\langle\text{---}\rangle\; \langle\text{---}\rangle\; (\; \underline{bel}(e,\underline{st}) \vee \underline{occ}(e,\underline{sq})\;)\;]$$

Here st_in stands for the given input set and the binary predicate symbols bel and occ have their meanings formally specified to be, respectively, " element belonging to a set " and " element occurring in a sequence ".

We can now write a first version of our abstract program, already annotated

```
in st_in : Set { "input data" } ;
out sq_out : Seq { "output result" } ;
var st : Set ; sq : Seq ; el : Tod ;
  st := st_in ; sq := lmbd ;
     { is-transf(st,sq) & is-incr-ord(sq) }
  while ¬empty(st) do
     el := chs(st) { bel(el,st) } ;
     sq := ord-ins(sq,el) { occ(el,sq) & is-incr-ord(sq) } ;
     st := rem(st,el) { ¬bel(el,st) }
  end while ;
  sq_out := sq
      { is-sort(st_in,sq_out) }
```

The operation and predicate symbols underlined in the above program text correspond to those of our ADT (in particular lmbd denotes the null sequence) and are to be formally specified just enough so as to guarantee the correctness of the program. Two such symbols deserving special mention are ord-ins and chs , whose intended meanings are, respectively, " ordered insertion of an element into its proper place in a sequence " and " nondeterministic choice of an element in a set ". Our program never tries to apply chs to an empty set, so the spec for our ADT will contain as the sole axiom mentioning chs the following one

(∀s:Set) [¬empty(s) ---> bel(chs(s),s)]

Notice that this axiom specifies the behavior of chs as a nondeterministic choice of an element from a nonempty set, giving absolutely no information concerning the effect of applying chs on an empty set. The point is that this is all we need to know about chs in order to prove the verification conditions for the partial correctness [Ma'74] of the program.

Thus, the usage of incompletely specified private ADT's frees us from overspecifying our ADT vis a vis its abstract program.

4. SPECIFICATION AND IMPLEMENTATION

This section presents the general concepts pertaining to ADT's, which will be employed later also in dealing with problems and problem-solving methods. In doing so we try to be as much as possible independent of the particular formalism employed to write the specifications and implementations.

An ADT is a class of structures, its realizations, sharing a common language. A language L of an ADT consists of the following symbols : a nonempty set St of sorts, together with sets Op and Pr of, respectively, operation and predicate symbols, which are categorized into "visible" (denoted by V) and "hidden" (denoted by H). Thus, for instance, VOp is the set of visible operation symbols and HPr is the set of hidden predicate symbols, and Op = VOp U HOp . A specification (or spec, for short) of an ADT with language L is a set Ax of axioms involving the symbols of L , written procedurally and/or non-procedurally, in an appropriate formalism, allowing for, say, procedure declarations and logic-like formulas. Thus, an ADT spec is a presentation of a theory (not necessarily expressed in first-order logic). A realization of an ADT spec is a structure for the language L satisfying all the properties expressed in the set Ax of axioms. So, we shall be dealing with the so-called loose or incomplete specifications [PL'79,VPM'82], which give more room for the incorporation of the design decisions taken in the subsequent refinement steps.

Notice that the operation and predicate symbols of (the language of) an ADT are divided into two categories, namely visible and hidden ones. The rationale for this is as follows. Visible symbols may appear in program texts, where they are interpreted as calls to procedures whose declarations will eventualy provide their definitions. On the other hand, hidden symbols are not supposed to appear in program texts, they are auxiliary symbols appearing only in the spec; as such they are to be eventually defined as well, but not necessarily by means of procedures. These definitions will be effected by an extension construct. Now, there are three natural ways to extend an ADT, namely, by domain or sort definitions, by procedural definitions (say, by programs), and by non-procedural definitions (say, by logical formulas), and these match quite naturally with the sorts, visible and hidden symbols, respectively, of the ADT language.

An extension construct (EC, for short) is a generalized cluster-like module [LZ'74], the text of which consists of

* domain definitions for the sort symbols (say by means of type declarations as in Pascal);
* procedural definitions for the visible symbols (say by means of procedure declarations);
* non-procedural definitions for the hidden symbols (say by means of logical formulas).

We emphasize that the text of an EC may also involve some other symbols not explicitly defined therein. Thus, an EC is a language mechanism for defining new symbols in terms of some others, supposedly defined elsewhere. We say that an EC E operates on an ADT T iff

* every symbol occurring in E but not explicitly defined in it is a corresponding symbol of T ;
* the axioms of the spec of T guarantee the strong termination of the procedural definitions of E , in the sense that on any realization of T the interpreted procedural definitions terminate for all inputs (thus defining total functions and predicates on the appropriate domains).

The language of such an EC E operating on an ADT T is the union of the languages of E and T . Due to the above conditions, every realization of the ADT T has a (unique) expansion to a structure for this extended language. Thus, we may regard this process as providing a new ADT, called the extension of T by E .

We are now ready to consider the process of implementing an ADT on another one, supposedly more "concrete". An implementation of an ADT T on an ADT T' is an EC operating on T' which defines all the symbols of T (in terms of those of T') so that, with these definitions plugged in, every axiom of T is derivable from the spec of T' . In this case, every realization of T' expands, via the EC, to a realization of T , which is the intuitive idea behind the concept of implementation.

5. CONCRETE AND ABSTRACT PROBLEMS AND THEIR SOLUTIONS

We are now ready to consider the notion of problem. Of course, everybody can recognize a problem when seeing one, but , in general, what is a problem ? Polya [Po'71] suggests the following three questions to be asked in approaching a problem : " what are the data ? ", " what are the possible results ? ", and " what are the problem conditions ? ". We can take these three questions as a guide in formulating a precise definition for the vague, intuitive idea of problem.

Thus, we define a concrete problem (CP, for short) as [VV'81] a (two-sorted) mathematical structure $C = \langle D,R,q \rangle$, where D and R are nonempty sets, called the domains of, respectively, (input) data and (output) results; and q is a binary relation from D to R , called the (problem) requirement or condition. A solution for the CP C is a total function $f : D \longrightarrow R$, the graph of wich is included in q , i. e., for all $d \in D$ we have $\langle d,f(d) \rangle \in q$.

As an example the illustrate this definition, consider the problem of sorting sequences of, say, integers. We can formulate it as a CP as follows. Its data and result domains consist of the finite sequences of integers, and its requirement is the relation q such that $\langle d,r \rangle \in q$ iff r is an ordered sequence consisting of the same elements as d . In this problem, the integers play the role of a parameter. We may as well consider the whole class of

sorting problems as consisting of all the CP's like the above one when the the set of elements is allowed to range over totally ordered domains. This leads to the consideration of abstract problems.

An abstract problem (AP, for short) is an ADT A , the language of which has the following symbols : sort symbols D and R for the domains of, respectively, input data and output results, and a hidden binary predicate symbol g from sort D to sort R . The realizations of an AP are the corresponding CP's. Thus, a solution for an AP should be a description of a function which, when interpreted on each one of its CP's, defines a solution for it. This leads us to the concept of function schema.

A function schema (FS, for short) is an extension construct with a designated main function procedure f . Notice that if such an FS operates on an ADT T then on every realization of T the main procedure f defines a total function. So, we define a solution for an AP A as an FS operating on the ADT A such that for every realization C of A the function defined on C by the main procedure f is actually a solution for the CP C .

Continuing with our example of sorting, its formulation as an AP would involve an ADT, the spec of which states that the domains corresponding to the sorts D and R consist of the finite sequences of elements from some domain Tod which is totally ordered, together with an axiom defining the predicate symbol g as, for instance,

$$(\forall d{:}\underline{D})(\forall r{:}\underline{R})\ [\ g(d,r) \longleftrightarrow \underline{is\text{-}sort}(d,r)\ ]$$

Thus, the realizations of this AP would be CP's having the form < Seq[Tod] , Seq[Tod] , is-sort > . Now, a solution for this AP would be an FS consisting of a sorting algorithm, for instance, mergesort.

6. PROBLEM-SOLVING METHODS AND THEIR APPLICATION

We now address the question of what is a problem-solving method (PSM, for short). An immediate answer is that a PSM is a method to solve problems, which, being circular-like, is not very informative. So, let us look at an example, again sorting. A sorting algorithm, as mergesort, might be regarded as a method to solve sorting problems. However, such an algorithm is, in fact, a PSM already applied to a problem. We would like to separate the PSM from the (abstract) problem where it is applied, in order to study the former by itself in isolation, much as we have already defined problems and solutions. These intuitive ideas and desiderata lead us to the following definitions, which are intended to be precise formulations for our intuitions.

A problem-solving method (PSM, for short) M is a function schema (which, we recall, is an extension construct with a designated main procedure $\underline{f}$) operating on an ADT, called its underlying ADT. An application of a PSM M to an abstract problem A is an implementation of the ADT underlying M on the ADT of A . Notice that if the PSM M is applied to the AP A then, in view of the definition of implementation, we can derive the property

$$(\forall d{:}\underline{D})\, g(d, \underline{f}(d))$$

This means that this PSM is a method which gives a solution for every concrete problem realizing the AP A .

We now want to see to what extent these definitions actually capture our intuitive ideas about the corresponding vague notions that we have. One way to do this is by showing how some usual situations can be formulated within the proposed framework. Thus, we now examine some examples of problem-solving methods or strategies in order to substantiate our claim that these definitions can really be accepted as precise " rational reconstructions " of our intuitions.

A very useful and powerful strategy to solve problems is "divide-and-conquer" [HS'78]. Its basic idea consists of repeatedly splitting problem instances (or data) into successively simpler (or smaller) ones until obtaining instances that can be solved directly and obtaining a result for the original data by successive recombinations of the results obtained for the subproblem instances generated in the process. The PSM embodying this strategy will be called decomposition. Given a positive natural number n , by an n-ary decomposition [Ve'80] we mean the PSM Dc_n consisting of the following FS

$$\underline{f} = (\lambda d{:}\underline{D})\ [\ \text{if } \underline{smpl}(d) \text{ then } \underline{drct}(d) \text{ else } \underline{rcmbn}(\underline{f}(\underline{splt\_1}(d)),\ldots,\underline{f}(\underline{splt\_n}(d)))\ ]$$

whose underlying ADT has sorts D and R , the symbols appearing in the above FS as visible ones, and the binary predicate symbols smllr (from D to D) and g (from D to R) as hidden ones. Its spec consists of the following axioms

(1) $(\forall d{:}\underline{D})\ [\ \underline{smpl}(d) \longrightarrow \underline{g}(d, \underline{drct}(d))\ ]$

(2) $(\forall d{:}\underline{D})(\forall r\_1,\ldots,r\_n{:}\underline{R})\ \{\ \neg\underline{smpl}(d)\ \text{---}>$

$\text{--}>\ [\ \underline{g}(\underline{splt\_1}(d),r\_1)\ \&\ldots\&\ \underline{g}(\underline{splt\_n}(d),r\_n)\ \text{---}>$

$\text{---}>\ \underline{g}(d,\underline{rcmbn}(r\_1,\ldots,r\_n))\ ]\ \}$

(3) $(\forall d{:}\underline{D})\ \{\ \neg\underline{smpl}(d)\ \text{---}>$

$\text{---}>\ [\ \underline{smllr}(\underline{splt\_1}(d),d)\ \&\ldots\&\ \underline{smllr}(\underline{splt\_n}(d),d)\ ]\ \}$

(4) $\underline{smllr}$ is well-founded (i. e. it has no infinite descending chain [Ma'74]) .

It is not difficult to see that these 4 axioms guarantee the total correctness [Ma'74] of the program schema expressed by the above FS with respect to the input-output spec g(d,r) [the first two ensure partial correctness whereas the other two ensure (strong) termination].

Another important problem-solving strategy can be expressed by " Here is a problem related to yours and solved before " [Po'71], and is illustrated by the Cartesian method of reducing geometric problems to algebraic ones. We formulate this strategy as a PSM $\underline{reduction}$ **Rd** consisting of the FS

$$\underline{f} = (\lambda d{:}\underline{D})\ [\ \underline{rtrv}(\underline{aux}(\underline{trnsl}(d)))\ ]$$

operating on an ADT with the four sorts $\underline{D}$, $\underline{R}$, $\underline{E}$, and $\underline{S}$, two hidden binary predicate symbols $\underline{g}$ (from $\underline{D}$ to $\underline{R}$) and $\underline{p}$ (from $\underline{E}$ to $\underline{S}$) , and the three operation symbols occurring in the above FS as visible ones. Its spec consists of the following two axioms

$$(\forall d{:}\underline{D})(\forall s{:}\underline{S})\ [\ \underline{p}(\underline{trnsl}(d),s)\ \text{---}>\ \underline{g}(d,\underline{rtrv}(s))\ ]$$

$$(\forall e{:}\underline{E})\ \underline{p}(d, \underline{aux}(e))$$

The basic intuition captured by this PSM is the following one. The (realizations of the) problem to be solved can be viewed as $\langle D,R,q\rangle$ whereas $\langle E,S,p\rangle$ may be regarded as (the realizations of) an auxiliary problem already solved by some FS aux . The reduction itself from the former to the latter is effected by the two maps given by trnsl and rtrv to translate data and retrieve results, respectively, between the two problems.

The formulation of other problem-solving methods, such as the greedy method, dynamic programming, etc. in terms of ADT's along lines similar to the ones presented here can be found in [Wa'84], together with examples of application.

7. SOME ILLUSTRATIVE EXAMPLES OF APPLICATION

Let us now illustrate how the preceding formulations can actually be applied to the processes of problem solving and program development. For this purpose, we employ, once again, the problem of sorting sequences into ascending order, the formulation of which as an abstract problem A has been sketched in section 5.

We shall try to solve this problem by applying the "divide-and-conquer" strategy with decomposition into two subproblem instances. According to our formulations, this means obtaining an implementation of the ADT underlying the PSM Dc_2 on the ADT of the AP A . One such implementation is provided by the following extension construct

splt_1 = (λd:D) [" the first half of d "]

splt_2 = (λd:D) [" the second half of d "]

rcmbn = (λr_1,r_2:R) [" the merger of r_1 with r_2 "]

drct = (λd:D) [d]

smpl = (λd:D) [" d has length at most 1 "]

(∀d,d':D) [smllr(d,d') <---> length(d) < length(d')]

It is easy to see that, with these definitions plugged into, the four axioms of the spec of Dc_2 [which are the binary versions

of axioms (1) through (4) of section 6] are actually derivable from properties of sequences present in the spec of the AP A . Morever, the FS of Dc_2 with these definitions actually plugged into is clearly the familiar mergesort algorithm [Kn'73]. Another possible implementation of the ADT underlying Dc_2 on A is obtained by replacing the first three definitions of the preceding EC by the following ones

$\underline{\text{splt\_1}}$ = (λd:$\underline{D}$) [" the sequence consisting solely of the minimum element in d "]

$\underline{\text{splt\_2}}$ = (λd:$\underline{D}$) [" the sequence obtained from d by the removal of its minimum element "]

$\underline{\text{rcmbn}}$ = (λr_1,r_2:$\underline{R}$) [" the concatenation of r_1 with r_2 "]

the other definitions remaining as before. By plugging these definitions into the FS of Dc_2 we obtain the familiar straight selection algorithm for sorting [Kn'73].

As a final illustration we just outline how the solution of the problem of sorting by the tree-sort algorithm can be described in our framework. The first step is a reduction of the original problem to that of inserting sequences into ordered binary trees. Thus we have to implement the ADT underlying the PSM Rd on an ADT of the original AP A . For this purpose we provide an EC containing the definitions of two new sorts $\underline{E}$ = $\underline{\text{Tree[Tod]}}\times\underline{\text{Seq[Tod]}}$ and $\underline{S}$ = $\underline{\text{Tree[Tod]}}$ together with a hidden binary predicate symbol $\underline{\text{is-merger}}$ (to play the role of the requirement $\underline{p}$ of the auxiliary problem) and the three visible operation symbols of Rd . Now, it is quite easy to write a

procedure declaration defining trnsl , for instance, trnsl = (λd:D) [< null_tree , d >] . Those for rtrv and aux generate two new problems, respectively, traversing a binary tree in inorder and merging a sequence into a binary ordered tree, which can be solved by the divide-and-conquer strategy by means of appropriate implementations of the ADT underlying Dc_1 . The implementation of the latter, for instance, generates another problem, namely that of inserting a single element into a binary ordered tree, which can again be solved by decomposition.

8. CONCLUDING REMARKS

The main idea presented in this paper is that, by pushing further the idea of (data) abstraction in program development, one can give precise formulations for the vague, but crucial, notions of problem, problem-solving strategy and the appication of the latter to the former, all of them in terms of abstract data types (ADT's, for short).

By a critical analysis of the role of (data) abstraction in program development, we are led to the distinction between public (general purpose) and private (special purpose) ADT's, the possibly incomplete specifications of the latter permitting fuller exploitation of the benefits of abstraction. Thus, a (private) abstract data type is considered as a class of structures, their characterizing properties being given by a specification written in an appropriate formalism. Such ADT's can be extended by an extension construct, which is a generalized cluster-like module, upon which the definition of implementation is based. The definition of a concrete problem as a mathematical structure consisting of data, results and requirement formalizes some ideas of Polya [Po'71] on Heuristics, whereas an abstract problem is viewed as an ADT whose realizations are concrete problems. A problem-solving method is defined, using the concept of extension construct, as basically a function schema (which is a generalized program schema) operating on an ADT, the former indicating how a solution is obtained by the method and the

specification of the latter guaranteeing the total correctness of the procedure. The application of a problem-solving method to an abstract problem is defined in terms of implementation of the corresponding ADT's.

These formulations were argued to capture our basic intuitions behind these somewhat vague, but extremely fruitful, notions, which was illustrated by showing how they can be methodically applied to problem solving and program development. In particular, they can be applied modularly and in incremental steps, corresponding to stepwise refinements.

A problem-solving strategy embodies some general knowledge concerning how to go about solving problems from some wide class. This knowledge is incorporated in the corresponding problem-solving method in two parts, namely the procedural aspects in the function schema and the declarative aspects in the specification of the underlying ADT. One might envision a library of problem-solving methods, each one of them proven correct and well documented so as to indicate its domain of applicability. For, these precise formulations, besides being " rational reconstructions " of our intuitions, provide a framework for the precise investigation and comparison of these concepts.

REFERENCES

[GJ'82] C. Ghezzi, M. Jazayeri - Programming Language Concepts; Wiley, New York, 1982.

[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 R. T. Yeh (ed.) Current Trends in Programming Methodology, vol. IV, Prentice-Hall, Englewood Cliffs, 1978.

[Gu'77] J. V. Guttag - Abstract data types and the development of data structures; Comm. ACM, vol. 20 (no. 6), 1977.

[HS'78] E. Horowitz, S. Sahni - Fundamentals of Computer Algorithms; Computer Sci. Press, Potomac, 1978.

[Kn'73] D. E. Knuth - The Art of Computer Programming, vol. 3 : sorting and searching; Addison-Wesley, Reading, 1973.

[LZ'74] B. Liskov, S. Zilles - Programming with abstract data types; SIGPLAN Notices, vol. 4 (no. 4), 1974.

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

[PL'79] T. H. Pequeno, C. J. Lucena - An approach for data type specification and its use in program verification; Inform. Proc. Letters, vol. 8 (no. 2), 1979.

[Po'71] G. Polya - How to Solve it : a new aspect of the mathematical method; Princeton Univ. Press, Princeton, 1971.

[Ve'80] P. A. S. Veloso - Divide-and-conquer via data types; Proc. VII Conf. Latinoamericana de Informatica, Caracas, 1980.

[VF'85] P. A. S. Veloso, A. L. Furtado - Towards simpler and yet complete formal specifications; in A. Sernadas, J. Bubenko Jr., A. Olive (eds.) Information Systems : theoretical and formal aspects, North-Holland, Amsterdam, 1985.

[VPM'82] P. A. S. Veloso, F. E. P. Pessoa, T. S. E. Maibaum - Theory of abstract data types for programming : a logical approach (in Portuguese); Proc. IX Conf. Latinoamericana de Informatica, Lima, 1982

[VV'81] P. A. S. Veloso, S. R. M. Veloso - Decomposition and reduction : applicability, soundness, completeness; in R. Trappl, J. Klir, F. Pichler (eds.) Progress in Cybernetics and Systems Research, vol. VIII, Hemisphere, Washington DC, 1981.

[Wa'84] C. F. E. M. Waga - Methods for Solving Problems (in Portuguese); M. Sc. diss., Dept. Informatica, Pont. Univ. Catolica, Rio de Janeiro, 1984.