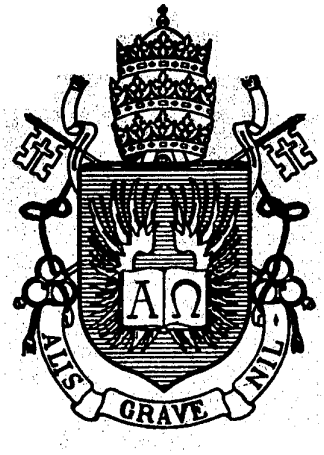

PUC

Series: Monografias em Ciência da Computação

Nº 14/78

AN APPROACH FOR DATA TYPE SPECIFICATION AND ITS USE IN PROGRAM VERIFICATION

by

Tarcísio H. Pequeno
Carlos J. Lucena

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 225 – CEP-22453
Rio de Janeiro – Brasil

Series: Monografias em Ciência da Computação

Nº 14/78

Series Editor: Michael F. Challis August, 1978

AN APPROACH FOR DATA TYPE SPECIFICATION AND ITS USE IN PROGRAM VERIFICATION*

by

Tarcísio H. Pequeno
Carlos J. Lucena

*This research has been supported in part by CNPq - contract TC2222.0060/77, and FINEP - contract CT-370.

For copies contract

Rosane T.L. Castilho
Head, Setor de Documentação e Informação
Deptº de Informática - PUC/RJ
Rua Marquês de São Vicente, 209 - GAVEA
22453 - Rio de Janeiro - RJ - Brasil

ABSTRACT:

This work presents an approach for the specification of data types and data representation. The approach was motivated by the problem of verifying programs that handle abstract data types. The ideas are presented in this paper through the specification and analysis of a simple sorting program.

KEYWORDS:

Abstract data types, formal specification, correctness of data representations, program verification.

RESUMO:

Uma abordagem à especificação de tipos de dados e à correção de representações de dados é apresentada. Essa abordagem foi motivada pelo problema da verificação de programas que manipulam tipos de dados abstratos. As ideias são apresentada nesta monografia via a especificação e análise de um pequeno programa de classificação.

PALAVRAS CHAVES:

Tipos de dados abstratos, especificação formal, correção de representações de dados, verificação de programas.

1 - INTRODUCTION

Abstract data types [1] have been used as an important tool in the development of programs. It allows the programmer to write programs in terms of the data types which he considers most adequate for the solution of a given problem. The methodology implied by the use of abstract data types leads to the development of an elegant solution to programming problems. The solution can be divided into two parts: design of a program expressed in terms of abstract data types, usually called an abstract program [2], and the choice and definition of data representations for the abstract program's types. The proof of correctness of the solution is therefore divided into two parts: the proof of correctness of the abstract program based on the properties of the abstract data types and the proof of correctness of the data representation. This last proof consists of the verification of whether the defined representations do model the abstract types. To enable this verification a formal specification of the abstract data type is required.

Various approaches have been proposed in the literature for the specification of abstract data types. Several of them are surveyed [3]. In particular, the so-called algebraic[4,5,6] and axiomatic [7,8] methods have played an important role in the studies of correctness of data representations. In the algebraic approach the type is regarded as an algebra defined by a set of equations involving its operations. In the axiomatic approach the type is seen as a theory whose models are possible representations for the type. Such a theory is defined by a set of axioms that establish relationship among the functional and predicative symbols and the constants used in the description of the type.

One of the problems involved in the use of the above specification methods has to do with the determination of the necessary properties for the complete characterization of the data type. In case of the axiomatic methods there is the problem of establishing a set of axioms that needs to be satisfied by all the models of the type and only by those models. In the

present work we suggest an approach to circumvent the above problem in the proof of programs that handle abstract data types. We present also a method for the proof of correctness of the representations of the data types.

Since initially we are interested in properties of the abstract program the axioms we write for the data type are only powerful enough to allow us to verify these properties. In other words, the type that is used to solve the proposed problem is not completely specified. The class of models defined by these (incomplete) axioms will, in general, be much larger than the class of models of the type which is ultimately intended to solve the problem.

As a later stage the operations of the type are defined in terms of a data representation. At this point the additional properties which complete the specification are introduced, although not at the abstract level.

The specification and verification method to be presented here is based on interpretations between theories [9]. Its theoretical backgraound is discussed in detail in [10]. Among other things, the correctness approach to be presented here eliminates the need for the use of a mapping function from the space of the data representation to the abstract data space. Such a function is required, for instance, in the method proposed in [2].

2 - PROGRAM DESIGN

The proposed method will be discussed in detail through a case study. The problem selected for the illustration of the method is a thoroughly well known problem: the sorting in ascending order of a non-repeating sequence of elements. We will use the notation of the first order predicate calculus and will assume from the reader some familiarity with concepts such as axiom, theorem, theory, model, etc. [14] from mathematical logic.

2.1 - Specification and Design of the Abstract Program

We shall begin by proposing a data type that seems adequate at the abstract level for the problem solution. With the type in hand we will formulate an abstract program which deals with objects of the type through the type's defined operations.

For the classical problem of sorting a sequence of elements we chose to use an abstract type called sequence.

The type will be described through a language L, L = <same, good, change>, for which the symbols same and good are predicative symbols with arities 2 and 1 respectively and change is a functional symbol with arity 1.

Intuitively, these symbols can be interpreted in the following manner:

a. change is an operation over sequences; since we wish to transform the initial sequence into a sorted sequence, change will be required to change the order of the elements according to an adequate discipline.

b. same is a predicate that tells us whether the input sequence is the "same" as the output sequence except for a different ordering of its elements.

c. good is a predicate which tells us if the elements of the sequence are arranged in ascending order.

The program specification can then be expressed through the following inductive expression [2]:

$$\{T\}\ P(x,z)\ \{same(x,z) \wedge good(z)\}$$

The assertion {T} indicates that there are no restrictions with respect to the input. P(x,z) is an abstract program

in which x is an input variable and z is an output variable.

To be able to prove the correctness of P(x,z) we need a more precise specification of our type. This specification will be independent of any particular representation of the type. The term "abstract" in the expression "abstract data type" tries to capture this notion of independence from the representation. The specification can be expressed as a set of axioms written in the language L presented above. At this point we can contrast our approach with the usual manner in which abstract data types are used. What is usually done is to give at this point a set of axioms capable of defining completely the data type. We, on the contrary, write at this point only axioms which are necessary for the proof of the abstract program or when we are concerned with program synthesis [11], we write only axioms which are necessary for the derivation of the program.

At this point we have a number of alternatives to design and check our program. We could, for instance, propose a program that supposedly solves the problem and then analyse it to try to discover which axioms are necessary for its proof. Alternatively, we could state a number of reasonable axioms and then try to derive the program from them [11]. A third alternative would be to formulate the program and the axioms in a more or less independent manner and then try to prove the first with respect to the second. This is the approach in the present case study.

A possible version for the abstract program reads as follows:

```
{T}
y1 ← x
y2 ← change (y1)
while ¬ y1 = y2
   do y1 ← y2
      y2 ← change (y1)
   od
z ← y2
end
{same (x,z) ∧ good (z)}
```

Note that this program is abstract also in the sense that it is representative of a whole class of programs · not all of which need be sorting programs.

We now propose the axioms which specify the type used in the abstract program.

The predicate same is clearly at least an equivalence relation and therefore:

A1. $\forall x$[same (x,x)]

A2. $\forall x,y,z$ [same (x,y) $\wedge$ same (y,z) $\rightarrow$ same (x,z)]

A3. $\forall x,y$ [same (x,y) $\rightarrow$ same (y,x)]

The following axiom establishes the fact that change preserves same (same is a congruence relation for change):

A4. $\forall x$ [same (x, change (x))]

The axiom that finishes the presentation of the type sequence is in fact a definition of good in terms of change:

A5. $\forall x$ [good (x) $\leftrightarrow$ x = change (x)]

2.2 - Correctness of the Abstract Program

For the purpose of proving the program in Floyd's style [12], we present an annotated version of the program.

```
{same (x,x)}
y1 ← x
{same (x,y1)}
y2 ← change (y1)
{same (x,y2) ∧ y2 = change (y1)}
while ¬ y1 = y2
    do y1 ← y2
       y2 ← change (y1)
{same (x,y2) ∧ y2 = change (y1)}
    od
{same (x,y2) ∧ y2 = change (y1) ∧ y1 = y2}
z ← y2
end
{same (x,z) ∧ good (z)}
```

The partial correctness of the above program can be proved in a formal system using axioms A1 to A5 above and the inference rules for the control structures which were used [7]. The proof is very simple and is left to the reader.

To prove the total correctness we will suppose the existence of a function called dist, from the set of sequences into the set of naturals, which has the following property:

A6. $\forall x\ [\neg$ good $(x) \rightarrow$ dist$(x) >$ dist (change $(x))]$

The existence of such a function is sufficient to guarantee the termination of the program [13]. Note that the addition of A6 still does not completely specify what we normally think of as the data type "sequence". This condition will have to be satisfied by the data type representation.

3 - SPECIFICATION AND DESIGN OF THE DATA REPRESENTATION

To proceed with the design of the program we must now choose a representation for the data type. This, of course, can be done in several levels. In our example we will use only two levels - the abstract type above and its representation using arrays.

We will initially represent the data type through a theory in the following many sorted language:

$L = \langle \underline{Ind}, \underline{len}, \underline{swap}, +, >, \leq, \geq, <, \underline{error}, 1 \rangle$

The symbols are of sorts which we will denote, respectively, A, N, AxN and D. They must be interpreted in the following manner:

a. A is the set of all the arrays.

b. N is the set of natural numbers.

c. AxN is the Cartesian product of A and N.

d. D is the domain of the elements of the array.

Ind, len, swap and +, are the following functions:

a. Ind: AxN → D; retrieves an element of a given array by means of the array index if the index is between 1 and the lenght of the array, otherwise the value of the function is error. We will write x_i, istead of Ind (x,i).

b. len: A → N; determines the lenght of an array.

c. swap: AxN → AxN; swap (x,i) interchanges the values of x_i and x_{i+1} in case they are not ordered and

maintains their values otherwise. Additionally, swap increments the value of i by one.

d. +; is the usual addition over the natural numbers.

The predicate $\geq_D$ has in D a meaning which is consistent with the term "ascending order" used in the problem definition. Finally, error and 1 are respectively constants in D and N.

The theory will be defined by a set of axioms that establish the properties of the symbols adopted in the language. The sequences will be represented by arrays that satisfy the following predicate entitled the relativization predicate for the sort A:

$$P_A(x) \leftrightarrow \forall i \neg\exists j[j \geq 1 \wedge j \leq \underline{len}(x) \wedge \neg j=i \wedge x_j=x_i] \wedge \underline{len}(x)>1$$

P_A establishes that the sequences of the abstract type will be represented by arrays of length greater than one without repeated elements. The other sorts in the many-sorted language do not have any restrictions, except for the product AxN for which the restriction P_A also holds.

We can now define the symbols same, good and change used to define sequence in terms of the language introduced above:

B1. $$\underline{same}(x,y) \leftrightarrow \forall i[i \geq 1 \wedge i \leq \underline{len}(x) \rightarrow (\exists j[j \geq 1 \wedge j \leq \underline{len}(x) \wedge x_i=y_i] \wedge \forall k[k \geq 1 \wedge k \leq \underline{len}(x) \wedge y_i=x_k])]$$

B2. $$\underline{good}(x) \leftrightarrow \forall i[i \geq 1 \wedge i < \underline{len}(x) \rightarrow x_{i+1} \geq_D x_i]$$

B3. $$\underline{change}(x)=y \leftrightarrow <y, \underline{len}(x)> = \underline{swap}^{\underline{len}(x)-1}(<x,1>) \quad (*)$$

---

* This formula is not strictly expressed in the 1st order language used before because of the power used in swap. Nevertheless, it can be considered as a schema for formulas of the type

$$<y, \underline{len}(x)> = \underline{swap}(\underline{swap}(\ldots \underline{swap}(<x,1>))\ldots)),$$

one for each length of x.

If we have correct definitions then the theory of the arrays extended by B1 to B3 defines a class of models that can be converted into a sub-class of the models of A1 to A6. In fact if the definitions capture the informal meaning of the operations this class is going to be a sub-class of the class of the models of sequence.

The symbols swap and len, whose meanings were described intuitively, may in turn be expressed in terms of the other symbols of the language.

i. $\underline{swap}\ (\langle x,i\rangle)=\langle y,j\rangle \leftrightarrow (x_i \geq x_{i+1} \rightarrow (y_i = x_{i+1} \wedge y_{i+1} =$
$= x_i \wedge \forall k[(\neg k=i \wedge \neg k=i+1) \rightarrow x_k = y_k])$
$\wedge (\neg x_i \geq x_{i+1} \rightarrow x=y) \wedge j=i+1$

ii. $\underline{len}(x)=i \leftrightarrow x_i = \underline{error} \wedge \forall j[j>i \rightarrow x_j = \underline{error}]$

In summary, the type has been expressed in a language with the following symbols:

$$\langle \underline{Ind}, +, \geq, \leq, \geq, <, \underline{error}, 1\rangle$$

The definitions of same and good specify an actual sorting method. In fact, the definitions of change and swap introduce an algorithm for the solution of the problem: in the present case the well-known algorithm usually referred to as bubble-sort. A whole family of sorting programs could be designed following the same approach by using different definitions of same, good and change.

4 - CORRECTNESS OF THE DATA REPRESENTATIONS

To prove the correctness of the representation of a data type means to prove that the implementation of the representation is also an implementation of the data type. If we consider the type and its representation as theories we are

able to define the term correctness precisely. We say that a representation for a data type is correct if and only if every model of the representation is a model of the data type. Note that the inverse is not required; that is, a data type may have models which are not models of the representation.

The proof of correctness of the representation can be done in two steps. The first step consists of showing that the closure properties for each sort may be deduced in the extended theory in which the type has been represented. The closure properties for a sort S whose relativization is P_S, are the following:

i. $\exists x[P_S(x)]$, which assures that the predicate can be satisfied in S.

ii. For every constant a from sort S, $P_S(a)$.

iii. For every functional symbol f from sort S and arity $S_1, S_2 \ldots, S_n$ with relativization predicates $P_{S_1}, P_{S_2}, \ldots, P_{S_n}$:

$$\forall x_1, x_2, \ldots, x_n [P_{S_1}(x_1) \wedge \ldots \wedge P_{S_n}(x_n) \rightarrow P_S(f(x_1, x_2, \ldots, x_n))]$$

Thus, in the given example we need to prove, among other things that:

$$\forall x[P_A(x) \rightarrow P_A(\underline{change}(x))]$$

that is, that the subset of A defined by P_A is closed with respect to change.

The second step for the proof of the representation consists of showing that the axioms of the data type, relativized for the representation, may be deduced in the extended theory in which the type was represented. We need to deduce, for instance, that:

$$\forall x[P_A(x) \rightarrow \underline{same}(x, \underline{change}(x))].$$

input variables of a sub-routine form the input assertion whereas the definition constitutes the output assertion.

5 - CONCLUSIONS

The present paper introduced the following two ideas:

a. An axiomatic approach for the specification of abstract data types. The abstract data type is seen as a theory defined by a set of axioms. The representation is extended by the definition of the symbols of the first theory. The methodology is based on the Theory of Definition [9] and its theoretical aspects are discussed in detail in [10].

b. A method for the proof of programs that handle abstract data types which avoids the problem of determining a complete specification for the type at the program level. The idea consists of writing only powerful enough axioms for the proof of the program at the abstract program level. The specification of the program is completed when a representation is chosen for the abstract data type.

We have illustrated the above results through the development of a sample programming example.

ACKNOWLEDGEMENT: The authors are grateful to R.L. Carvalho e P.A. Veloso for suggestions and criticisms and to T. Maibaum for valuable comments on the paper.

In the example given above these proofs can be easily developed. Formal proofs would require that the axioms of the theory used for the representation be given explicitly.

Finally, to guarantee the termination of the program we extend the theory by defining a function dist and prove that it has the following property:

$$\forall x[\neg \underline{good}(x) \rightarrow \underline{dist}(x) > \underline{dist}(\underline{change}(x))]$$

Let us state the following definition for dist:

$$\begin{aligned}\underline{dist}(x) = n \leftrightarrow & \forall i,j[n<i \wedge i<j \wedge j \leq \underline{len}(x) \rightarrow x_{j_0} \geq x_i] \wedge \\ & \forall i,j[i \geq 1 \wedge i \leq n \wedge j > n \wedge j \leq \underline{len}(x) \rightarrow x_{j_0} \geq x_i] \wedge \\ & \exists k[k \geq 1 \wedge k \wedge n \wedge \forall i,j[k<i \wedge 1<j \wedge \\ & \wedge j \leq \underline{len}(x) \rightarrow x_{j_0} \geq x_i] \wedge \forall i,j,[i \geq 1 \wedge i \leq k \\ & \wedge j > k \wedge j \leq \underline{len}(k) \rightarrow x_{j_0} \geq x_i]]\end{aligned}$$

Intuitively dist(x) is the length of x minus the length of a subsequence formed by the last elements of x which are already ordered. The definition says that all the elements above position n are ordered, that these elements are greater than the other elements of x and that there does not exist any position below n for which the above is true. Note that the greatest value of dist(x) is the length of x and the smallest, when the sequence is ordered, is zero. We need to prove that when the sequence is not ordered one application of change produces an ordered subsequence which is longer than the preceding subsequence. The proof is done by induction on the size of the sequence.

The described representation could be directly encoded through a cluster-like mechanism [1], in a programming language that supports an array mechanism with indexing and comparison and the type D with the predicate $\geq$. The definitions of the operations may be used to prove the sub-routines that will implement them or even for the derivation of such sub-routines. The relativization predicates of the sorts corresponding to the

REFERENCES

[1] Liskov, B.; Zilles, S.N. Programming with Abstract Data Types. Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices, 9, 4 (April 1974) p. 50-59.

[2] Hoare, C.A.R. Proof of Correctness of Data Representations. Acta Informatica, vol. 1 (1972), p. 271-281.

[3] Liskov, B.; Zilles S.N. Specification Techniques for Data Abstractions. IEEE Transactions on Software Engineering, vol. 1, SE-1, nº 1 (March 1975), p. 8-18.

[4] Zilles, S. Algebraic Specifications for Data Types. Project MAC Progress Report for 1973-74.

[5] Guttag, J. Abstract Data Types and the Development of Data Structures. CACM, vol. 20, nº 6 (June 1977) p.396-404.

[6] Goguen, J.A.; Thatcher, J.W.; Wagner,E.G.; Wright, J.B.. Abstract Data Types as Initial Algebras and Correctness of Data Representations. Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structures, (May 1975).

[7] Hoare, C.A.R. An Axiomatic Basis for Computer Programming. CACM, (October 1969) p. 576-580.

[8] Standish, T.A. Data Structures: An Axiomatic Approach. BBN Report, nº 2639, Bolt, Beranek and Newman, Cambridge, Mass., 1973.

[9] Tarski, A. Some Methodological Investigations on the Definability of Concepts in Logic, Semantics, Mathematics. Academic Press, Oxford (1956).

[10] Veloso, P.A.; Pequeno, T.H.C. Interpretations Between Many-Sorted Theories. Proceedings of II Encontro Brasileiro de Lógica, Campinas, Brazil, July 1978.

[11] Lucena, C.J.P.; Pequeno, T.H.C. A View of the Program Derivation Process Based on Incompletely Defined Data Types: A Case Study. Technical Report, Departamento de Informática, PUC-RJ, nº 25, 1977.

[12] Floyd, R.W. Assigning Meanings to Programs. Mathematical Aspects of Computer Science, vol. XIX, American Mathematical Society, Providence, Rhode Island, 1967, p. 19-32.

[13] Manna, Z. Mathematical Theory of Computation. McGraw-Hill Book Company, New York, 1974.

[14] Shoenfield, J.R. Mathematical Logic, Addison Wesley, Reading, Mass. 1969.