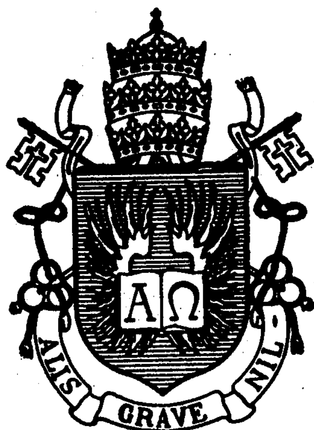


PUC

Series: Monografias em Ciência da Computação

Nº 10/79

DON'T WRITE MORE AXIOMS THAN YOU HAVE TO: A METHODOLOGY FOR THE COMPLETE AND CORRECT SPECIFICATION OF ABSTRACT DATA TYPES; WITH EXAMPLES

Paulo A. S. Veloso

Tarcísio H. C. Pequeno

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 10/79

Series Editor: Daniel A. Menascé May, 1979

DON'T WRITE MORE AXIOMS THAN YOU HAVE TO: A METHODOLOGY FOR THE COMPLETE AND CORRECT SPECIFICATION OF ABSTRACT DATA TYPES; WITH EXAMPLES*

Paulo A. S. Veloso
Tarcísio H. C. Pequeno

* Research partly sponsored by FINEP and CNPq.
This paper is a revised and expanded version of "Do not write more axioms than you have to", given at International Computing Symposium 1978, Nankang, China, Dec. 1978.

ABSTRACT:

A data type can be formally specified by a set of correct axioms that is sufficient to completely characterize it. This paper presents a methodology that indicates what axioms to write and when to stop writing them. Some data types are specified to illustrate the application of the method, which is based on the concept of canonical term algebra.

KEY WORDS:

Abstract data type, axiomatic specication, correctness proof, systematic specification, canonical terms, complete specification.

RESUMO:

Um tipo de dados pode ser formalmente especificado por um conjunto de axiomas que seja suficiente para caracterizá-lo completamente. Este trabalho apresenta uma metodologia que indica que axiomas escrever e quando parar de escrevê-los. São especificados alguns tipos de dados para ilustrar a aplicação do método, que é baseado no conceito de álgebra de termos canônicos.

PALAVRAS CHAVES:

Tipos de dados abstratos, especificação axiomática, certificação de correção, especificação sistemática, termos canônicos , especificação completa.

CONTENTS

1. INTRODUCTION 1

2. A PRELIMINARY EXAMPLE: NATURAL NUMBERS 3

3. A SIMPLE DATA TYPE: QUEUES 5

4. THE METHODOLOGY 11

5. AN ILLUSTRATIVE EXAMPLE: SETS OF NATURALS 13

6. A MORE CONVINCING EXAMPLE: TRAVERSABLE STACK 18

7. CONCLUSIONS 25

ACKNOWLEDGMENTS 27

REFERENCES 28

1. INTRODUCTION

Several methods have been proposed for the specification of a data type by presenting some of its basic properties (axioms) in a representation-independent manner [Liskov and Zilles 1975]. Some difficulties in writing an axiomatic specification are what axioms to write and when to stop writing them, i.e., whether the axioms written are sufficient to define the data type. Here we present a methodology that helps in both difficulties by guiding in the discovery of the axioms and by indicating when they are sufficient.

Abstract data types been used a powerful programming tool. Its use provides an elegant construction of the program by factoring it in two parts: a program that manipulates an abstract data type and an implementation of the data type in terms of some selected representation. The correcteness proof of the program can also be factored into a proof of the program that manipulates the abstract data type and a proof of the correcteness of the implementation of the data type. Both proofs require a formal specification of the data type [Guttag 1977].

The methodology presented consists of the choice of a canonical form for the data type and in the analysis of the effect of the application of each operation of the data type on this canonical form. This analysis suggests what axioms are needed and, once one has done it for all the operations, one can be sure that no more axioms are necessary.

For abstract data types regarded as initial algebras, using conditional equations as axioms, a formal justification of the methodology can be provided, based on the concept of canonical term algebra [Thatcher et al. 1976].

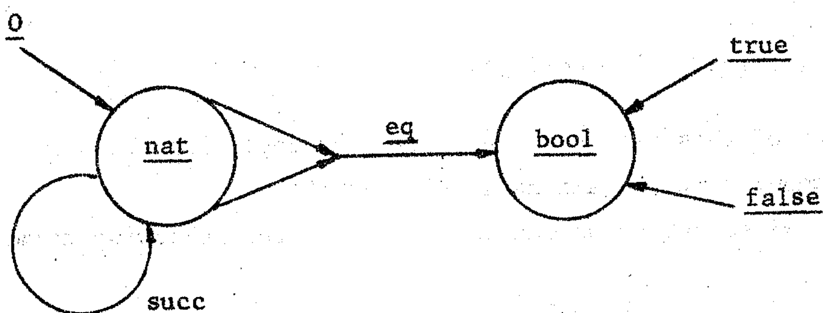
In the next sections two simple introductory examples - natural numbers and queues - are specified as a motivation for the method. The general methodology is described in section 4, which also includes a brief discussion of its application and an outline of its justification in terms of c.t.a.'s (canonical term algebras), Then, sections 5 and 6 illustrate the methodology by applying it

to specify two data types: finite sets of natural numbers and a version of traversable stack (where errors form a crucial part). Finally, we conclude with some remarks on the method and the suggestion of c.t.a.'s as a complementary specification form, precise yet intuitive.

2. A PRELIMINARY EXAMPLE : NATURAL NUMBERS

Suppose we want to specify a data type in a representation-independent manner. We are given its operations and an informal specification by means of a model. We are required to define the type using only its properties.

Let us consider the data type natural numbers with equality (Example 3 of [Goguen et al. 1975]). It consists of two sorts nat for the natural numbers and bool for the boolean values true and false. The operations are represented in the ADJ-like diagram below



The intended meanings of these operations are the usual ones, as suggested by their mnemonical names. This is going to be our informal model.

It is clear that each natural number can be represented as a finite number of applications (maybe zero) of succ to 0, i.e., by the term $\text{succ}^n(0)$, for some n. Notice that distinct terms represent distinct natural numbers. Thus these terms can be regarded as representatives for **nat**, in a "canonical" way.

We are now able to give a more precise specification of the operations by describing their effects on these canonical terms. Namely

$$\text{succ}[\text{succ}^n(0)] = \text{succ}^{(n+1)}(0) \tag{1}$$

$$\text{eq}[\text{succ}^m(0),\text{succ}^n(0)] = \begin{cases} \text{true} & \text{if } m = n \\ \text{false} & \text{if } m \neq n \end{cases} \tag{2}$$

We are going to view axioms as rules to transform the lefthand sides of the above definitions into the required righthand sides.

In the first definition the lefthand side is already in the desired form, thus requiring no axioms.

The transformation of $\underline{eq}(\underline{succ}^m(\underline{0}),\underline{succ}^n(\underline{0}))$ into <u>true</u> or <u>false</u>, according to the definition (2), can be done in two steps, as follows.

1. Decrease the number of <u>succ</u>'s in both arguments simultaneously, while possible.

   This would be achieved by the axiom

   N1: $\underline{eq}(\underline{succ}(i),\underline{succ}(j)) = \underline{eq}(i,j)$

   The validity of this axiom can be checked by replacing the variables i and j by canonical terms and using (1) and (2).

   By applying N1 as far as we can we get one of the following terms

   $\underline{eq}(\underline{0},\underline{0})$ if $m = n$

   $\underline{eq}(\underline{succ}^{(m-n)}(\underline{0}),\underline{0})$ if $m > n$

   $\underline{eq}(\underline{0},\underline{succ}^{(n-m)}(\underline{0}))$ if $m < n$

2. Reduce the term obtained above to <u>true</u> or <u>false</u> by directly applying one of the following axioms

   N2: $\underline{eq}(\underline{0},\underline{0}) = \underline{true}$

   N3: $\underline{eq}(\underline{succ}(i),\underline{0}) = \underline{false}$

   N4: $\underline{eq}(\underline{0},\underline{succ}(j)) = \underline{false}$

   We can be sure that we do not need more axioms because we were able to reduce any term to its canonical representative. Thus, N1 through N4 give a complete specification for the data type.

3. A SIMPLE DATA TYPE : QUEUE

The preceding example involved a very simple data type of a somewhat mathematical nature. We shall now consider a simple data type more akin to programming,namely queue of D (where D is some already specified sort of data to be stored in the queue, say, integers).

We may describe a queue configuration as a linear array of elements from D with two pointers front and rear.

The syntactical specification of the operations is given as follows.

$$\underline{createQ} : \quad \rightarrow Q$$

$$\underline{remQ} : Q \rightarrow Q$$

$$\underline{putQ} : Q \times D \rightarrow Q$$

$$\underline{getQ} : Q \mapsto D$$

assuming

$$\text{error } D : \rightarrow D$$

The intended effect of each operation can be informally described as follows

- createQ creates an empty queue, with front = rear = 0;
- putQ adds a data d to the rear of the queue, which increases its length by one;
- remQ removes the front element of the queue, thereby shortening its length by one;
- getQ reads the front element of the queue without altering it, if possible; otherwise the result is errorD (a distinguished element in D).

The above description will be our informal model. For simplicity sake, we decided to ignore the possibility of putting the error signal errorD back into the queue.

From the above description it is clear that a queue configuration containing the data $d_1,d_2,\ldots,d_n$, with d_1 in the front and d_n at the rear, can be obtained from the empty queue by successively introducing the data

$d_1, \ldots, d_n$ by means of putQ operations. So it can be represented as

putQ(...(putQ (putQ(createQ, d_1), d_2)..., d_n), which we shall abbreviated as
$\underline{putQ}^n(d_1\ d_2\ \ldots\ d_n)$.
We shall include the case n=0 by agreeing that $\underline{putQ}^0 = \underline{createQ}$.

Thus, any element of the sort Q can be uniquely represented as a term $\underline{putQ}^n(d_1, \ldots d_n)$ for some unique $n \geq 0$ and $d_1, \ldots d_n \in D$. (Recall that we are ignoring error propagation). So, we shall consider this as a canonical form for Q.

We are now in position to give a precise specification of the operations. We shall describe its effect on a canonical term, as follows

(q1) $\underline{putQ}[\underline{putQ}^n(d_1, \ldots, d_n), d] = \underline{putQ}^{n+1}(d_1, \ldots, d_n, d)$

(q2) $\underline{remQ}[\underline{putQ}^n\ (d_1, \ldots, d_n)] = \begin{cases} \underline{putQ}^{n-1}(d_2, \ldots, d_n) & \text{if} \quad n > 0 \\ \underline{createQ} & \text{if} \quad n = 0 \end{cases}$

(q3) $\underline{getQ}[\underline{putQ}^n(d_1, \ldots, d_n)] = \begin{cases} d_1 & \text{if} \quad n > 0 \\ \underline{errorD} & \text{if} \quad n = 0 \end{cases}$

This specification was obtained by translating the preceding informal description. Notice that the case n=0 of (q2) says remQ(createQ) is createQ . Some people would rather say that this results in an error. Our informal description was vague about this point, thereby allowing either choice. Incidentally, one of the advantages of specifying the operations by their effects on all the canonical terms is pinpointing possible ambiguities so common in informal verbal descriptions.

We shall now write correct axioms that enable us to convert the lefthand side of each specification above into the corresponding righthand side.

As the righthand side of (q1) is already in canonical form, no axiom is needed for this case.

So, let us consider (q2). The case $n > 0$ tell us to delete the leftmost data in the canonical term. By noticing that the strings $d_1\ d_2 \ldots d_n$ and $d_2\ \ldots\ d_n$ are almost identical, one is led to set $\underline{d} = d_2 \ldots d_n$.

Then, by (q1)

$\underline{putQ}^n(d_1,d_2, \ldots, d_n) = \underline{putQ}^{n-1}(\underline{putQ}(\underline{createQ}, d_1), \underline{d})$

by using a natural extension of our convention. Now, we can rewrite the specification for $n > 0$ as

$\underline{remQ}\ [\underline{putQ}^{n-1}\ (\underline{putQ}(\underline{createQ}, d_1)\ ,\ \underline{d}\ )] = \underline{putQ}^{n-1}(\underline{createQ}, \underline{d})$.

A natural candidate axiom to achieve the above transformation is

$\underline{remQ}\ [\ \underline{putQ}\ (q,d)\ ] = \underline{putQ}\ [\ \underline{remQ}\ (q),\ d]$

In order to check it for correctness we replace the variable q of sort Q by the corresponding canonical term. The lhs becomes $\underline{remQ}\ [\underline{putQ}\ (\underline{putQ}^n\ (d_1,\ldots,d_n),d)]$, which by (q1) is $\underline{remQ}\ [\underline{putQ}^{n+1}(d_1,\ldots,d_n,d)]$,

whence by (q2), we get

$lhs = \underline{putQ}^n(d_2,\ldots,d_n,d)$

Now, the rhs becomes, upon replacement, $\underline{putQ}\ [\underline{remQ}\ (\underline{putQ}^n(d_1,\ldots,d_n)),d]$. which by (q2) is, as $n \geq 0$, then either $\underline{putQ}(\underline{createQ},d)$, if $n=0$, or $\underline{putQ}^{n-1}(d_2,\ldots,d_n\ d)$, if $n > 0$.

So, we get from (q1)

$$rhs = \begin{cases} \underline{putQ}^1(d) & \text{if} \quad n = 0 \\ \underline{putQ}^n(d_2,\ldots,d_n d) & \text{if} \quad n > 0 \end{cases}$$

Notice that we have by the preceding computation

$$lhs = \begin{cases} \underline{createQ} & \text{if} \quad n = 0 \\ \underline{putQ}^n\ (d_2,\ldots,d_n) & \text{if} \quad n > 0 \end{cases}$$

Thus, lhs = rhs if and only if $n > 0$. So the above candidate axiom is correct only under the condition $q \neq \underline{createQ}$. Thus, we are led to the modified version

Q1: $q \neq \underline{createQ} \rightarrow remQ\ [\underline{putQ}\ (q,d)] = \underline{putQ}\ [\underline{remQ}(q),\ d]$

the correctness of which has already been established by the very argument above (which led to its formulation).

Now, let us check whether this axiom enables us to perform the transformation required by (q2) in case $n > 0$.

Starting with remQ $[\text{putQ}^n (d_1,\ldots,d_n)]$, we may rewrite it by (q1), as

$$\text{remQ}\ [\text{putQ}\ (\text{putQ}^{n-1}\ (d_1,\ldots,d_{n-1}),\ d_n)]$$

Here if we call $q = \text{putQ}^{n-1}\ (d_1,\ldots,d_{n-1})$ our notational comunication indicates that $q \neq$ createD in case $n > 1$. Then Q1 is applicable, yielding

$$\text{putQ}\ [\text{remQ}\ (\text{putQ}^{n-1}(d_1,\ldots,d_{n-1})),\ d_n]$$

If $n > 2$ we may apply again Q1, as above.

After (n-1) applications of Q1, we obtain

$$\text{putQ}(\ldots.\ \text{putQ}(\text{remQ}(\text{putQ}^1(d_1)),d_2),\ldots,\ d_n) \qquad \text{(I)}$$

Now, Q1 is no longer applicable, since

$\text{putQ}^1(d_1)$ abbreviates $\text{putQ}(\text{createQ},\ d_1)$. So we need an axiom specifying the effect of remQ on this term (which denotes a queue of lenght one). Since (q2) gives

$\text{remQ}\ [\text{putQ}^1\ (d_1)\ ] = \text{putQ}^{1-1} = \text{createQ}$,

a natural choice, clearly correct, is

Q2: $\text{remQ}\ [\ \text{putQ}\ (\text{createQ},\ d\ )\ ] = \text{createQ}$.

An application of Q2 to (I) gives

$\text{putQ}(\ldots..\ \text{putQ}(\text{remQ}\ [\text{putQ}^1(d_1)]\ ,d_2),\ldots,\ d_n) =$
$\text{putQ}(\ldots..\ \text{putQ}(\text{createQ},\ d_2),\ \ldots\ldots,\ d_n) =$
$\text{putQ}^{n-1}(d_2,\ldots,d_n)$,

as desired.

We still have to consider the case $n = 0$ of (q2), which is already in the form of an equation

Q3: $\text{remQ}(\text{createQ}) = \text{createQ}$

Now, for (q3), which specifies the effect of the read-out operation, one might start by noticing its similarity with (q2), which removes the first element, rather than reading it (both regardless of the others).

This observation suggests the following versions of Q1 and Q2 for getQ

Q4: $q \neq$ createQ $\rightarrow$ getQ [putQ(q,d)] = getQ(q)

Q5: getQ [putQ(createQ, d)] = d

easily checked to be correct.

To see that Q4 and Q5 do perform the transformation required by (q3) when $n > 0$, notice that getQ [$\text{putQ}^n(d_1,\ldots,d_n)$] =

= getQ [putQ($\text{putQ}^{n-1}(d_1,\ldots,d_{n-1})$, d_n)] (by notation)

= getQ [$\text{putQ}^{n-1}(d_1,\ldots,d_{n-1})$] (by Q4)

Repeating these transformations while getQ applies to a queue of lenght greater than 1, we arrive at getQ [$\text{putQ}^1(d_1)$] =

= getQ [putQ (createD, d_1)] (by notation)

= d_1 (by Q5)

For the case $n = 0$, we have

Q6 : getQ(createQ) = errorD

Now, we have checked that axioms Q1,...Q6 are correct with respect to (q1),(q2),(q3), and transform their lhs's to the corresponding rhs's. Thus, these axioms provide a sufficient complete abstract specification for the data type queue of D.

Once we have Q1,...Q6, we can try to improve the form of the axioms, easily and safely. For instance, Q1 does not have to be a conditional axiom, as we may clearly replace it by

Q1' : remQ [putQ(putQ(q,d'), d)] =

= putQ [remQ(putQ(q,d')), d] =

and, similarly Q4 can be replaced by

Q4': getQ [putQ(putQ (q,d'),d)] =

= getQ (putQ(q,d'))

Alternatively one might prefer to keep Q1 and Q4 as they are and replace Q2 and Q5 by

Q2': q = createQ $\rightarrow$ remQ[putQ(q,d)] = q

and

Q5' : q = createQ → getQ [putQ (q,d)] = d

which form "balanced Pairs", in a way with Q1 and Q4, respectively.

Finally, one might still prefer to merge these "balanced pairs" into single axioms using conditional terms. So, Q1 and Q2' could be merged into

Q1.5 : remQ [putQ(q,d)] = if q=createQ then q else putQ [remQ(q),d]

Similarly Q4 and Q5' would be replaced by

Q4.5: getQ [putQ(q,d)] = if q=createQ then d

else getQ (q)

4. THE METHODOLOGY

The above method can be generalized to a methodology, which can be used to give an axiomatic specification for a data type. The syntax of the data type is supposed to be given by a set Σ of operations. Its semantics is given (formally or informally) by some other method, for instance, by means of a model.

The methodology consists of the following steps.

1. Elect a canonical form, i.e., a set C of terms such that every element of the data type is uniquely represented in C and whenever $\sigma t_1 \ldots t_n$ is in C then so are $t_1, \ldots, t_n$, σ being an operation in Σ.
2. Translate the given specification into a specification of the operations in terms of the canonical form of 1.
3. For each operation $\sigma \in \Sigma$, write axioms to transform each term of the form $\sigma c_1 \ldots c_n$, where $c_1, \ldots, c_n$ are canonical representatives, into the appropriate canonical representative given by 2.

In many cases we can perform 3 by steps using the following heuristics

3.1) devise a simpler transformation that "approximates" the desired transformation;

3.2) write "candidate axioms" to perform the simpler transformation (which often suggests some candidates);

3.3) check that these candidate axioms

   a) are correct (by using the given specification or the one given by 2),

   b) indeed perform the desired transformation.

This methodology can be formally justified for data types that can be regarded as (many-sorted) algebras in which every element is the value of a variable-free term. A detailed proof would require some algebraic tools (cf. [Grätzer 1968, Goguen et al. 1977]). Actually, steps 1 and 2 of the methodology guarantee that C is a canonical term algebra in the sense of [Thatcher et al. 1977, p. 11] and step 3 guarantees the hypotheses of their theorem 5, whence C is isomorphic to the initial algebra in the category of all Σ-algebras satisfying those axioms.

A few remarks about the methodology are in order. Firstly, we can treat the various sorts modularly. Secondly, the usefulness of the methodology hinges on the selection of a convenient canonical form (in fact, this is the most creative part), even though theorem 4 of [Thatcher et al. 1977] guarantees that there always exists some initial canonical term algebra.

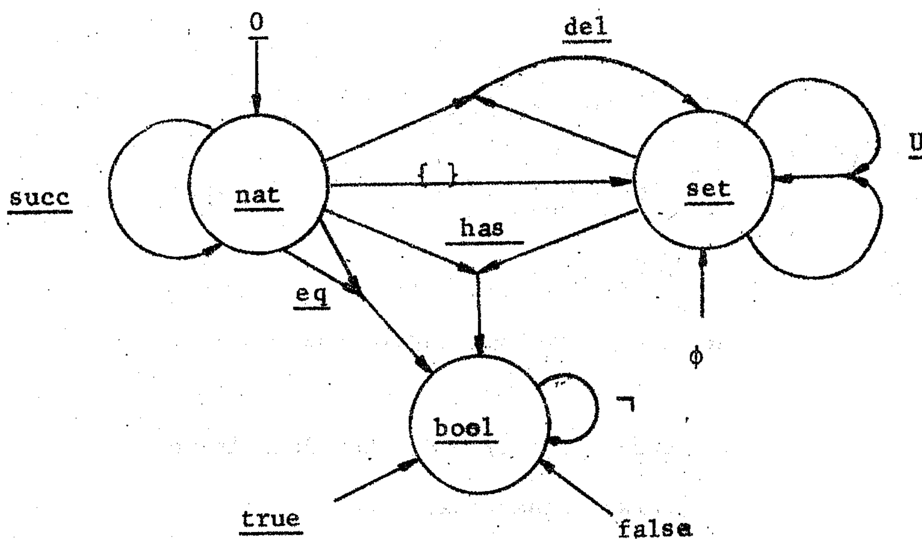
5. AN ILLUSTRATIVE EXAMPLE: SETS OF NATURALS

To illustrate the method described let us consider a data type consisting of three sorts: natural numbers, sets and boolean values with the following operations:

0	:	→ nat
succ	:	nat → nat
eq	:	nat × nat → bool
{ }	:	nat → set
ϕ	:	→ set
del	:	set × nat → set
U	:	set × set → set
has	:	set × nat → bool
¬	:	bool → bool
true	:	→ bool
false	:	→ bool

where nat, bool, succ, eq, 0, true and false are the same as before. U, del, ¬ stand for union, delete and not. { } gets a singleton from a natural number (we will use {i}, instead of { }(i)). del(s,i) gives s minus {i}, if i belongs to s, and gives s, otherwise. has (s,i) verifies if i belongs to s or not. The other operations have the usual meanings.

The ADJ-type diagram below, [Goguen et al. 1976], represents the data type.



To follow the method we begin by choosing canonical forms for the sorts involved. An element b of the sort bool has an obvious form that is

$$b = \begin{cases} \underline{true} \\ \underline{false} \end{cases}$$

For the sort nat we will use the form $\underline{succ}^n\underline{0}$ as before.

Finally for an element s of sort set we will adopt the form

$$s = \underline{U}(\ldots(\underline{U}(\phi,\{i_1\}),\{i_2\}),\ldots),\{i_n\})$$

where for all $1 \leq k,j \leq n$ if $k > j$ then $i_k > i_j$. If n is zero then we agree that s is ϕ.

For notational convenience we will write s as

$$\underline{U}^n(\phi d_1 \ldots d_n) \quad ,$$

where $d_j = \{i_j\}$ for $1 \leq j \leq n$.

Before proceeding with the method one must convince oneself that there is a one-to-one correspondence between the expressions of the form above and the finite sets of natural numbers, to be sure that it is in fact a canonical form.

The second step of the method is to give a specification of the operations in terms of the canonical forms. For succ and eq this was done before so we will do it for the other operations.

$$\{i\} = U^1(\phi,\{i\})$$

$$\underline{del}(\underline{U}^n(\phi d_1 \ldots d_n),i) = \begin{cases} \underline{U}^{n-1}(\phi d_1 \ldots d_{j-1} d_{j+1} \ldots d_n) & \text{if } d_j = \{i\} \text{ for some } 1 \leq j \leq n \\ \underline{U}^n(\phi d_1 \ldots d_n) & \text{otherwise} \end{cases}$$

$$\underline{U}(\underline{U}^m(\phi d_1 \ldots d_m),\underline{U}^n(\phi d'_1 \ldots d'_n)) = \underline{U}^k(\phi e_1 \ldots e_k),$$

where $\langle e_1 \ldots e_k \rangle$ is the merge without repetitions of $\langle d_1 \ldots d_n \rangle$ with $\langle d'_1 \ldots d'_n \rangle$.

$$\underline{has}(\underline{U}^n(\phi d_1 \ldots d_n),i) = \begin{cases} \underline{true} & \text{if } \{i\} = dj \text{ for some } 1 \leq j \leq n \\ \underline{false} & \text{otherwise.} \end{cases}$$

$$\neg(\underline{\text{true}}) = \underline{\text{false}}$$

and

$$\neg(\underline{\text{false}}) = \underline{\text{true}}$$

We proceed now by imagining the transformations necessary to convert the terms on the lefthand sides according to their definitions and by writing suitable axioms to do it. This is already done for succ and eq, so we will do it for the other operations. Let us begin with union. The transformation on

$$\underline{U}(\underline{U}^m(\phi d_1 \ldots d_m), \underline{U}^n(\phi d'_1 \ldots d'_n)) \qquad (\alpha)$$

can be performed in four steps.

1. The symbol "$\underline{U}$" must appear at the begining of the term. The following axiom can move an internal "$\underline{U}$" to the begining

$$\text{S1:} \quad \underline{U}(s_1, \underline{U}(s_2, d)) = \underline{U}(\underline{U}(s_1, s_2), d)$$

To check S1 for correctness, let us substitute canonical representatives for s_1 and s_2 on both sides of S1. On the lefthand side we get

$$\underline{U}(\underline{U}^m(\phi d_1 \ldots d_m), \underline{U}(\underline{U}^n(\phi, d'_1 \ldots d'_n), d))$$

By the definition of {} we can substitute $U(\phi, d)$ for d. By applying the definition of union to $\underline{U}(\underline{U}^n(\phi, d'_1 \ldots d'_n), \underline{U}(\phi d))$ and then to the entire term we get

$$\underline{U}^k(\phi e_1 \ldots e_k)$$

where $<e_1 \ldots e_k>$ is the merge, without repetitions, of $<d_1 \ldots d_m>$, $<d'_1 \ldots d'_n>$ and d.

The substitution into the righthand side yields

$$\underline{U}(\underline{U}(\underline{U}^m(\phi d_1 \ldots d_m), \underline{U}^n(\phi d'_1 \ldots d'_n)), d)$$

We can again substitute $\underline{U}(\phi, d)$ for d and apply the definition of union to $\underline{U}(\underline{U}^m(\phi d_1 \ldots d_m), \underline{U}^n(\phi d'_1 \ldots d'_n))$ and then to the entire term to get the same result as before.

The validity checks of the axioms along this example can be done in a similar way and are left to the reader.

The application n times of S1 to (α) will produce

$$\underline{U}(\underline{U}^n(\underline{U}^m(\phi d_1 \ldots d_m) \phi d'_1 \ldots d'_{n-1}\ d'_n) \qquad (\beta)$$

2. We need to eliminate the double occurrence of ϕ in (β). The following axiom can do it

   S2: $\underline{U}(s,\phi) = s$

   The application of S2 to (β) will produce

   $$\underline{U}^{m+n}(\phi d_1 \ldots d_m\ d'_1 \ldots d'_n) \qquad (\gamma)$$

3. In (γ) the singletons d_i and d'_i may not be in the desired order so we must be able to interchange them. The following axiom allows us to do it

   S3: $\underline{U}(\underline{U}(s,d_1),d_2) = \underline{U}(\underline{U}(s,d_2),d_1)$

4. Convenient applications of S3 can put the singletons of (γ) in the correct order but some of them may appear twice because some d_i may be equal to some d'_j. To eliminate these repetitions we can apply

   S4: $\underline{U}(\underline{U}(s,d),d) = \underline{U}(s,d)$

The reader can compare the axioms S1 to S4 that we got here with the axioms set-1 through set-4 presented in [Goguen et al. 1976] to conclude that our axioms are one by one a bit weaker than theirs, but for S2, which is set-1. At a first glance it is surprising the fact that the two systems of axioms have the same power (which they do, as both are complete). This happens because our axioms are "more independent" so to speak, than theirs. The reader is asked to try as an exercise to prove set-1 through set-4 from S1 through S4.

To discover the transformations on $\underline{del}(\underline{U}^n(\phi d_1 \ldots d_n),i)$ to conform the definition of del we will examine two cases:

1. There is a j, $1 \le j \le n$ such that $d_j = \{i\}$. In this case we have to eliminate d_j. We can use S3 to move d_j to the right and get

   $$\underline{del}(\underline{U}^n(\phi d_1 \ldots d_{j-k} d_{j+1} \ldots d_n d_j),i)$$

   Now d_j can be eliminated by the following axiom

   S5: $\underline{del}(\underline{U}(s,\{i\}),i) = \underline{del}(s,i)$

   By applying S5 we get

   $$\underline{del}(\underline{U}^{n-1}(\phi d_1 \ldots d_{j-1} d_{j+1} \ldots d_n),i)$$

   So we have reduced the first case to the second one.

2. There is no d_j such that $\{i\} = d_j$. In this case what we would like to do is just to "erase" del and i of the expression. The following equation

does just that

$\underline{del}(s,i) = s$

But unfortunately it cannot be an axiom since it is not valid because it obviously fails when i belongs to s. This difficulty can be overcome by using a conditional axiom as in [Thatcher et al. 1976]

S6: $\underline{has}(s,i) = \underline{false} \rightarrow \underline{del}(s,i) = s$

To get true or false from $\underline{has}(\underline{U}^n(\phi d_1 \ldots d_n),i)$ we can apply one of the following axioms, as the case may be

S7: $\underline{has}(\underline{U}(s,\{i\}),i) = \underline{true}$

S8: $\underline{eq}(i,j) = \underline{false} \rightarrow \underline{has}(\underline{U}(s,\{i\}),j) = \underline{has}(s,j)$

In the first case we are done. In the second case we can reapply S8 until we reach the first case or $\underline{has}(\phi,j)$ which is of course false

S9: $\underline{has}(\phi,j) = \underline{false}$

Finally for $\neg$ we have the obvious axioms:

B1: $\neg\, \underline{true} = \underline{false}$.

B2: $\neg\, \underline{false} = \underline{true}$.

We have written all the axioms that we need since we analysed all the operations, except { }, but note that the value of {i} can be obtained directly from S2.

The attentive reader may have noticed some oversimplifications, or slight inaccuracies.This was done in this example for didactical purposes. In the next example we intend to present the details in amore careful and thorough manner.

6. A MORE CONVINCING EXAMPLE; TRAVERSABLE STACK

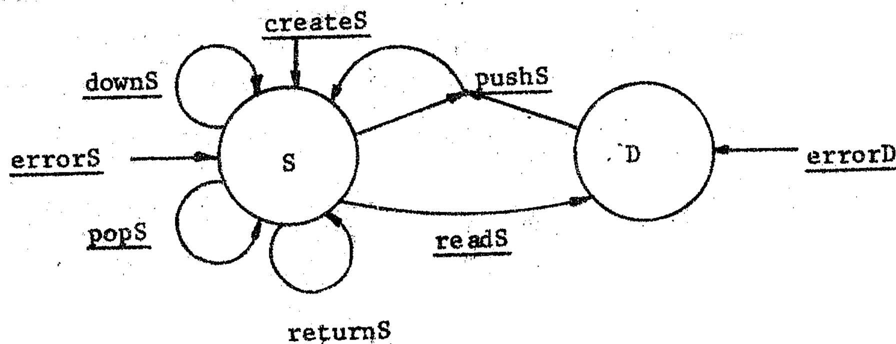
A traversable stack is similar to an ordinary pushdown stack but it has the added ability that readout is not restricted to the topmost position. A version of traversable stack has played a key role in a recent controversy about the limitations of algebraic specification techniques (cf. [Majster 1977,1978], [Martin 1977], [Subrahmanyan 1978].

Our version of traversable stack of D, where D is some already specified sort, say, integers, may be described informally as follows. A configuration of a traversable stack of D is a linear array of elements of D together with 2 pointers, one to the top position t, and an inner one which may point to any position $i \leq t$. In general we require $0 < i \leq t$ except for the empty stack, which has $i = t = 0$.

The operations are

- createS, which creates an empty stack with both pointers set to 0;
- pushS, which pushes an element of D on top of a stack, increasing both pointers by one;
- downS, the effect of which is to move the inner pointer one step toward the bottom by decrementing i by one, if possible; otherwise it gives errorS;
- popS, which removes the top element, decreasing both pointers by one, if possible; otherwise it gives errorS;
- returnS, which resets the inner pointer to the top;
- readS, to read out the content of the cell pointed by the pointer i, if possible; otherwise giving errorD (a distinguished element in D);
- errorS, the error condition of stack.

The syntactical specification of the type is then



A configuration containing the elements $a_1,\ldots,a_m$ of D, in this order, can be obtained from the empty stack createS via a sequence of m pushS's. This gives both pointers at m. If the inner pointer is to have value i, with $0<i\leq m$, we must then apply $n = m-i$ downS's.

Thus, any configuration can be represented, in a unique way, as

(a) errorS, or

(b) createS, or

(c) downS(... downS(pushS(... pushS(createS,a_1),...,a_m))...) ,
which we abbreviate as $\text{downS}^n\ \text{pushS}^m(a_1,\ldots,a_m)$, for some $0\leq n<m$, with all a_i's from D but different from errorD.

This should be clear from the above informal description, which suggested it.

We now describe the effect of each operation on the canonical representatives.

a) We generally assume that errors propagate without bothering to say it explicitly in the informal description. So

(a1) pushS(errorS,a) = errorS

(a2) pushS(t,errorD) = errorS

(a3) downS(errorS) = errorS

(a4) popS(errorS) = errorS

(a5) returnS(errorS) = errorS

(a6) readS(errorS) = errorD

b) The effect of each operation on createS is, as suggested by the informal description, as follows

(b0) pushS(createS,a) = $\text{pushS}^1(a)$

(b1) downS(createS) = errorS

(b2) popS(createS) = errorS

(b3) returnS(createS) = createS

(b4) readS(createS) = errorD

c) The informal description suggests the following specification of the effects of the operations on a nontrivial term $\text{downS}^n\text{pushS}^m(a_1,\ldots,a_m)$ with $0\leq n<m$

(c1) $\text{pushS}[\text{downS}^n\text{pushS}^m(a_1,\ldots,a_m),a] = \text{downS}^n\text{pushS}^{m+1}(a_1,\ldots,a_m,a)$

(c2) $\text{downS}[\text{downS}^n\text{pushS}^m(a_1,\ldots,a_m)] =$

$$= \begin{cases} \text{downS}^{n+1}\text{pushS}^m(a_1,\ldots,a_m) & \text{if } n+1<m \\ \text{errorS} & \text{if } n+1=m \end{cases}$$

(c3) $\underline{popS}[\underline{downS}^n \underline{pushS}^m(a_1,\ldots,a_m)] =$

$$= \begin{cases} \underline{downS}^n \underline{pushS}^{m-1}(a_1,\ldots,a_{m-1}) & \text{if } n < m-1 \\ \underline{createS} & \text{if } n = m-1 = 0 \\ \underline{errorS} & \text{if } n = m-1 > 0 \end{cases}$$

(c4) $\underline{returnS}[\underline{downS}^n \underline{pushS}^m(a_1,\ldots,a_m)] = \underline{pushS}^m(a_1,\ldots,a_m)$

(c5) $\underline{readS}[\underline{downS}^n \underline{pushS}^m(a_1,\ldots,a_m)] = a_{m-n}$.

In order to describe the transformations on canonical terms specified before, we let a be a variable of sort D and t be a variable of sort S.

A) errorS

The specifications (a1) through (a6) are already in the required form, thus giving 6 axioms

(A1),...,(A6) : error propagation, corresponding to (a1),...,(a6).

B) createS

Similarly, (b1) through (b4) have the required form and we need no axiom for (b0), thus we have 4 axioms

(B1),...,(B4):effect on empty stack, corresponding to $(b_1),\ldots,(b_4)$.

C) $\underline{downS}^n \underline{pushS}^m(a_1,\ldots,a_m)$ with $0 \leq n < m$

(1) Effect of pushS

The specification (c1) requires the most recent pushS to be moved inside, over the downS's, if any. This suggests an equation to the effect that pushS and downS commute, e.g. pushS[downS(t),a] = downS[pushS(t,a)]. Let us check it.

Replacing t by errorS or a by errorD, we clearly get errorS on both sides. The same holds if the value of t is createS. Now, let t denote $\underline{downS}^n \underline{pushS}^m(a_1,\ldots,a_m)$ with $0 \leq n < m$. The righthand side gives, by (c1) and (c2)

$$\underline{downS}(\underline{pushS}[\underline{downS}^n \underline{pushS}^m(a_1,\ldots,a_m),a)] = \underline{downS}^{n+1} \underline{pushS}^{m+1}(a_1,\ldots,a_n,a)$$

whereas the lefthand side gives the same result, by (c2), (c1) and (a1), only if n+1<m, i.e., if the downS causes no error. We are thus led to reformulate the above axiom as a conditional one

C1: $\underline{downS}(t) \neq \underline{errorS} \rightarrow \underline{pushS}[\underline{downS}(t),a] = \underline{downS}[\underline{pushS}(t,a)]$

We have just checked that this axiom is valid. It remains to check that is strong enough to perform the transformation required by (c1). But, this is clear as we can apply C1 n times to get

$$\underline{pushS}[\underline{downS}^n \underline{pushS}^m(a_1,\ldots a_n),a] = \underline{downS}(\underline{pushS}[\underline{downS}^{n-1}\underline{pushS}^m(a_1,\ldots,a_m),a]) = = \ldots = \underline{downS}^n\underline{pushS}[\underline{pushS}^m(a_1,\ldots a_m),a]$$

since at the ith step we have the term

$$\underline{downS}^i\underline{pushS}[\underline{downS}^{n-i}\underline{pushS}^m(a_1,\ldots,a_m),a]$$

to which (C1) is still applicable if $i < n$.

(2) Effect of downS

The specification (c2) requires no transformation when $n+1 < m$, otherwise a transformation into errorS is called for. So, let us assume $n+1 = m$ and try to transform $\underline{downS}[\underline{downS}^n\underline{pushS}^{n+1}(a_1,\ldots,a_n,a_{n+1})]$ into errorS.

By comparing (c2) and (c3) we see that downS produces an error exactly when popS gives errorS or createS. This suggests the axiom

C2: $\underline{popS}(t) = \underline{errorS} \vee \underline{popS}(t) = \underline{createS} \rightarrow \underline{downS}(t) = \underline{errorS}$

which reduces our problem to the effect of popS.

(3) Effect of popS

The similarity between (c3) and (c2) suggests

C3: $\underline{popS}[\underline{downS}(t)] = \underline{downS}[\underline{popS}(t)]$,

the validity of which can be checked as before. We thus can get, by n applications of C3

$$\underline{popS}[\underline{downS}^n\underline{pushS}^m(a_1,\ldots,a_m)] = \underline{downS}^n\underline{popS}\ \underline{pushS}^m(a_1,\ldots,a_m)$$

To get from here to the terms specified by (C3), it is natural to use $\underline{popS}[\underline{pushS}(t,a)] = t$, which is easily checked to be valid provided that $a \neq \underline{errorD}$. So we add

C4: $a \neq \underline{errorD} \rightarrow \underline{popS}[\underline{pushS}(t,a)] = t$.

An application of C4, now leads to

$$\underline{popS}[\underline{downS}^n\underline{pushS}^m(a_1,\ldots,a_{m-1},a_m)] = \underline{downS}^n\underline{pushS}^{m-1}(a_1,\ldots,a_{m-1}) ,$$

which is what we want if $n<m-1$ or if $n=m-1=0$ (for then we get createS).
In case $n=m-1>0$ we have downS^n pushS^n $(a_1,\ldots,a_n)$,
which should reduce to errorS. As $n>0$, it will, according to C2, depending
on popS downS^{n-1} pushS$^n(a_1,\ldots,a_n)$, which is by $(n-1)$ applications of C3,
downS^{n-1} popS pushS$^{n-1}(a_1,\ldots,a_n)$ =,
= downS^{n-1} popS pushS pushS^{n-1} $(a_1,\ldots,a_{n-1})$.
So, C4 brings us to examine downS^{n-1} pushS$^{n-1}(a_1,\ldots,a_{n-1})$.
Continuing this inductively, we are led to downS^1 pushS$^1(a_1)$.
Since, by C4, popS pushS^1 (a_1) = createS, we get altogether
downS^n pushS^n $(a_1,\ldots,a_n)$ = errorS, as desired.

(4) Effect of returnS

In order to make the returnS cancel all the downS's it is natural to use returnS[downS(t)] = returnS(t), which is easily seen to be correct under the proviso downS(t) $\neq$ errorS. So, we add

C5: downS(t) $\neq$ errorS $\rightarrow$ returnS[downS(t)] = returnS(t)

Successive applications of C5 lead to
returnS[downS^{n}pushS$^m(a_1,\ldots,a_m)$] = returnS pushS$^m(a_1,\ldots,a_m)$
from where we obtain the desired result by means of n applications of

C6: returnS[pushS(t,a)] = pushS returns(t)

the correctness of which being easy to be ascertained, and one of B3.

(5) Effect of readS

The specification (c5) does not depend on $a_{m-n+1},\ldots,a_m$, which could have been popped. Indeed
readS[downS^{n}pushS$^m(a_1,\ldots,a_m)$] = a_{m-n} =
= readS[pushS$^{m-n}(a_1,\ldots,a_{m-n})$] = readS[popS^{n}pushS$^m(a_1,\ldots,a_m)$] (by c3)
This suggests the axiom

C7: readS[downS(t)] = readS[popS(t)]

which is easily checked to be valid. We thus have

readS[downS^{n}pushS^m($a_1,\ldots,a_m$)] =
= readS[popS downS^{n-1}pushS^m($a_1,\ldots,a_m$)] (by C7)
= readS[downS^{n-1}popS pushS^m($a_1,\ldots,a_m$)] (by C3 repeatedly)
= readS[downS^{n-1}pushS^{m-1}($a_1,\ldots,a_{m-1}$)] (by C4)
....
= readS[pushS^{m-n}($a_1,\ldots,a_{m-n}$)] (by repeating the above cycle)

to obtain a_{m-n} from here it seems natural to use readS[pushS(t,a)] = a, which of course is not valid if t contains downS's. This can be overcome by using instead, readS(returnS[pushS(t,a)]) = a, which is correct unless t happens to be errorS. We are thus led to

C8: t ≠ errorS → readS(returnS[pushS(t,a)]) = a

which is valid and may be applied to the above term after the introduction of a return by means of C6 (m-n) times, after B3.

We now have a sufficiently complete specification for our data type. Notice that we have not tried to write strong axioms, quite on the contrary. Also, we did not worry about independence: some axioms may be obtainable from others(in fact, this is the case in the current example). We think it is a good policy first to concentrate on writing a correct complete specification, only afterwards should one try to improve in some other aspects, as independence for instance.

In this case one might notice that

returnS(errorS) = returnS[pushS(errorS,errorD)] (by A1 or A2)
= pushS[returnS(errorS), errorD] = errorS (by C6 and A2).

Thus A5 could be removed if one wished to reduce the number of axioms.

As another example, one might observe that, in reducing downS^n pushS^n ($a_1,\ldots,a_n$), with n>0, to errorS in (2) and (3), what was actually used was the reduction of pops^n pushS^n($a_1,\ldots,a_n$) to createS. So C2 might be replaced by the simpler version

C2': popS(t) = createS → downS(t) = errorS

Notice that popS^n pushS$^n(a_1,...,a_m)$, with $n>m$, still reduces to popS^{n-m}(createS) = errorS by A4, and similarly with downS instead of popS, by A3.

7. CONCLUSIONS

We have described and illustrated a methodology to write a correct and complete axiomatic specification for a given data type. The method may be summarized as follows. First, elect a set C of (canonical) representatives. Second, use them to specify the operations. Third, write correct axioms to guarantee that C is "closed" under the operations (in the sense that the result is transformable into C). It is apparent that the method does require some insight but we think it provides good guidelines together with hints. Its main advantage appears to be that it shows when to stop writing axioms.

The justification of the method is based on results of [Goguen et al. 1976] and [Thatcher et al. 1976] on canonical term algebras. These results were derived to prove the correctness of a given specification. Here we use these tools to obtain a specification.

The first step of the method, the election of a canonical form, is the most critical one, requiring some good insight into the data type. For, the selection of a nice form will make the remaining steps smooth, whereas an unlucky one can make them cumbersome and obscure. Of course, the known existence of some initial c.t.a. is no great help here. This difficulty can be alleviated by supplying a canonical form together with the given data type. This demand is in accordance with the suggestion that "a very high level (set theoretic) operational model should accompany the equational description of the data type, as an aid to the intuitive understanding of the type" [M. Levy 1977, p. 128]. In this connection we would like to add that a canonical term algebra consisting of a canonical form together with the operations specified on the representatives can be a very good aid to understanding the data type. It has the advantage of being a formal specification without any variables ranging over the type being specified, besides giving a good idea about how the type operates.

The third step of the method may also require some ingenuity. But the very outlook of the transformation to be performed gives good hints on how to proceed, either by decomposing it into simpler transformations or by suggesting the candidate axioms. Here two features should be stressed. First, a failure in a correctness check generally suggests some minor modifications on the candidate to make it into an axiom. Also, if one tries

to take care to put into the axioms just what is required for the transformations one gets a complete system with individually weak axioms. This contrasts with the axiom systems usually found in the literature.

We have been trying this method on several examples and find it very helpful. Also, it helped us in detecting mistakes in published specifications of well-known examples.

ACKNOWLEDGEMENTS

The authors are grateful to D. Kapur, from MIT, for pointing out some errors in an earlier version of this paper and to M.A. Ardis, from the University of Maryland, for supplying some recent literature.

REFERENCES:

R.L. de Carvalho, A.L. Furtado and A. Pereda B. - A relational model towards the synthesis of data strucures; Tech. Rept. , Depto. de Informática-PUC/RJ , nº 17, Nov. 1977.

J.A. Goguen, J.W. Thatcher and E.G. Wagner - An initial algebra approach to the specification, correctness and implementation of abstract data types; IBM Res. Rept. RC6487, Yorktown Heights, NY, Oct. 1976.

J.A. Goguen, J.W. Thatcher,E.G.Wagner and J.B. Wright- Abstract data types as initial algebras and the correctness of data representations; Proc. Conf. on Computer Graphics, Pattern Recognition and Data Structures, May 1975, p. 89-93.

G.Grätzer - Universal algebra; D. van Nostrand, Princeton, NJ, 1968.

J.V. Guttag - Abstract data types and the development of data structures; Comm of the ACM, vol. 20 (nº6), June 1977, p. 306-404.

J.V. Guttag and J.J. Horning - The algebraic specification of abstract data types; Acta Informatica, vol. 10, 1978, p.27-52.

N. Hilfinger - Letter to the editor; SIGPLAN NOTICES ,vol. 13 (Nº1) , Jan. 1978, p. 11-12.

D.W. Jones - A note on some limits of the algebraic specification method; SIGPLAN Notices, vol. 13(nº4), Apr. 1978, p.64-67.

M.R. Levy - Some remarks on abstract data types; SIGPLAN Notices , vol. 12 (nº 7) July 1977, p. 126-128.

T.A. Linden - Specifying data types by restrictions; Software Engineering Notes, vol. 3, (Nº4), Apr. 1978, p. 7-13.

B.H. Liskov and S.N. Zilles - Specification techniques for data abstractions; IEEE Trans. on Software Engin, vol. SE-1(Nº1) , Mar. 1975, p.7-19.

C.J. P. Lucena and T.H.C.Pequeno -A view of the program derivation process based on incompletely defined data types: a case study; Tech. Rept., Depto. Informática, PUC/RJ, Nº 25, Dec. 1977.

M.E.Majster. Limits of the "algebraic" specification of abstract data types; SIGPLAN Notices, vol. 12(nº 10), Oct. 1977, p.37-42.

M.E.Majster. Letter to the editor; SIGPLAN Notices, vol. 13(nº1), Jan. 1978, p. 8-10.

J.J. Martin. Critique of Mila E. Majster's paper "Limits of the "algebraic" specification of abstract data types"; SIGPLAN Notices, vol. 12(Nº 12), Dec. 1977, p. 28-29

T.H.C.Pequeno and P.A.S. Veloso - Do not write more axioms than you have to; Proc. International Computing Symp. 1978, vol. 1, Academia Sinica, Taipei, China, Dec. 1978, p. 487-498.

J.R. Shoenfield - Mathematical logic: Addison Wesley, Reading, Mass, 1969.

P.A. Subrahmanyan - On a finite axiomatization of the data type L; SIGPLAN Notices, vol. 13 (Nº 4), Apr. 1978, p.80-84.

J.W. Thatcher, E.G.Wagner and J.B. Wright - Specification of abstract data types using conditional axioms (Extended abstract); IBM Res. Rept. RC6214, Yorktown Heights, NY, Sept. 1976.

J.W. Thatcher, E.G. Wagner and J.B. Wright - Data type specification: parametrization and the power of specification techniques; SIGACT 10th. Annual Symp. Theory of Computing, San Diego, Calif. May 1978.

P.A.S.Veloso - Traversable stack with fewer errors; SIGLAN Notices, vol. 14 (Nº2), Fev. 1979, p. 55-59.