ICS '78

# 一九七八年國際計算機會議論文集

## 第一册

# PROCEEDINGS OF INTERNATIONAL COMPUTER SYMPOSIUM 1978

VOLUME ONE



慶祝中央研究院成立五十週年

# PROCEEDINGS

# OF

# INTERNATIONAL COMPUTER SYMPOSIUM 1978

DECEMBER 18–20, 1978

NANKANG, TAIPEI, REPUBLIC OF CHINA

VOLUME ONE

TO CELEBRATE THE 50TH ANNIVERSARY

OF ACADEMIA SINICA

**ACADEMIA SINICA PUBLICATION**

# INTERNATIONAL COMPUTER SYMPOSIUM 1978

December 18–20, 1978

Nankang, Taipei, Republic of China

## HONORARY CHAIRMAN

DR. S. L. CHIEN

PRESIDENT, ACADEMIA SINICA, R.O.C.

## GENERAL CHAIRMAN

DR. JULIUS T. TOU

Member of Academia Sinica, R.O.C.

University of Florida, U.S.A.

## ADVISORS

| | |
|---|---|
| *L. A. CHEN* | Vice Minister of Education, R.O.C. |
| *HOWELL S. C. CHOU* | President, Computer Society of The Republic of China |
| *YAOHAN CHU* | Professor, University of Maryland, U.S.A. |
| *H.C. FANG* | President, Industry Technology Research Institute, R.O.C. |
| *I. T. HO* | Senior Engineer, IBM, U.S.A. |
| *N. H. KUO* | Dean, College of Engineering, National Chiao Tung University, R.O.C. |
| *K. C. LEE,* | Director, EDP Center, DGBAS, Executive Yuan, R.O.C. |
| *S. S. SHU* | Chairman, National Science Council, R.O.C. |
| *C. C. YU* | Dean, College of Engineering, National Taiwan University, R.O.C. |

Proceedings of International Computer Symposium 1978 (Vol. 1)

# DO NOT WRITE MORE AXIOMS THAN YOU HAVE TO

T.H.C. PEQUENO AND P.A.S. VELOSO
DEPTº INFORMÁTICA, PONTIFÍCIA UNIVERSIDADE CATÓLICA
22453 RIO DE JANEIRO, RJ, BRAZIL

## ABSTRACT

Abstract data types have been used as a powerful tool to construct elegant programs by factorization into a program manipulating data types and implementation of the data types in terms of selected representations. This requires the data types to be specified formally in a representation-independent manner, thus bringing about the problem of correctness of specification. The main difficulty resides in writing a set of correct axioms that is sufficient to completely characterize the data type. Here a methodology is presented to help solving this problem by guiding in the discovery of the axioms and by indicating when they are sufficient.

The method consists of electing a canonical form for the data type and then using it to describe the operations. Analysis of this description suggests candidates for axioms which are checked to be correct or modified. Once this process is over one is sure that no axioms are missing. A justification of the method for data types regarded as initial algebra specifiable by conditional axioms is outlined, based on the concept of canonical term algebra. Two data types - finite sets of natural numbers and traversable stack - are specified to illustrate the application of the method.

## 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[1]. The main difficulties in writing an axiomatic specification are: what axioms to write and when to stop writing them, i.e., if 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 have been used as 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 correctness proof of the program can also be factored in the proof of the program that manipulates the abstract data type and the proof of the correctness of the implementation of the data type. Both proofs require a formal specification of the data type[2]. The methodology presented consists of

Proceedings of International Computer Symposium 1978 (Vol. 1)

# DO NOT WRITE MORE AXIOMS THAN YOU HAVE TO

T.H.C. PEQUENO AND P.A.S. VELOSO
DEPTº INFORMÁTICA, PONTIFÍCIA UNIVERSIDADE CATÓLICA
22453 RIO DE JANEIRO, RJ, BRAZIL

## ABSTRACT

Abstract data types have been used as a powerful tool to construct elegant programs by factorization into a program manipulating data types and implementation of the data types in terms of selected representations. This requires the data types to be specified formally in a representation-independent manner, thus bringing about the problem of correctness of specification. The main difficulty resides in writing a set of correct axioms that is sufficient to completely characterize the data type. Here a methodology is presented to help solving this problem by guiding in the discovery of the axioms and by indicating when they are sufficient.

The method consists of electing a canonical form for the data type and then using it to describe the operations. Analysis of this description suggests candidates for axioms which are checked to be correct or modified. Once this process is over one is sure that no axioms are missing. A justification of the method for data types regarded as initial algebras specifiable by conditional axioms is outlined, based on the concept of canonical term algebra. Two data types - finite sets of natural numbers and traversable stack - are specified to illustrate the application of the method.

## 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[1]. The main difficulties in writing an axiomatic specification are: what axioms to write and when to stop writing them, i.e., if 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 have been used as 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 correctness proof of the program can also be factored in the proof of the program that manipulates the abstract data type and the proof of the correctness of the implementation of the data type. Both proofs require a formal specification of the data type[2].

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[3,4].

Two examples are presented to illustrate the methodology: finite sets of natural numbers and traversable stack (the latter having been the pivot of a recent controversy in SIGPLAN Notices).

## AN INTRODUCTORY EXAMPLE

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[5]. 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[5] in Fig. 1.

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 $succ^n(0)$, for some n. Notice that distinct terms represent distinct natural numbers. Thus these terms can be regarded as canonical representatives for nat.

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

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

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

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

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

The transformation of $eq(succ^m(0), succ^n(0))$ into true or false, according to the definition (2), can be done in two steps, as follows.

1. Decrease the number of succ's in both arguments simultaneously, while possible. This would be achieved by the axiom
   N1: eq(succ(i),succ(j)) = 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

   | | |
   |---|---|
   | $eq(0,0)$ | if $m = n$ |
   | $eq(succ^{(m-n)}(0),0)$ | if $m > n$ |
   | $eq(0,succ^{(n-m)}(0))$ | if $m < n$ |

2. Reduce the term obtained above to true or false by directly applying one of the following axioms
   N2: eq(0,0) = true
   N3: eq(succ(i),0) = false
   N4: eq(0,succ(j)) = 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

### 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 $\Sigma$ 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 $t_1, \ldots, t_n$, $\sigma$ being an operation in $\Sigma$.
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 valid (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[4,6]. Actually, steps 1 and 2 of the methodology guarantee that C is a canonical term algebra[3] and step 3 guarantees that C is isomorphic to the initial algebra in the category of all $\Sigma$-algebras satisfying those axioms (cf. theorem 5[3]).

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 there always exists some initial canonical term algebra (cf. theorem 4[3]).

### AN ILLUSTRATIVE EXAMPLE: SETS OF NATURAL NUMBERS

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

| | | |
|---|---|---|
| <u>0</u> | : | → <u>nat</u> |
| <u>succ</u> | : | <u>nat</u> → <u>nat</u> |
| <u>eq</u> | : | <u>nat</u> × <u>nat</u> → <u>bool</u> |
| { } | : | <u>nat</u> → <u>set</u> |
| $\phi$ | : | → <u>set</u> |
| <u>del</u> | : | <u>set</u> × <u>nat</u> → <u>set</u> |
| <u>U</u> | : | <u>set</u> × <u>set</u> → <u>set</u> |
| <u>has</u> | : | <u>set</u> × <u>nat</u> → <u>bool</u> |
| ¬ | : | <u>bool</u> → <u>bool</u> |
| <u>true</u> | : | → <u>bool</u> |
| <u>false</u> | : | → <u>bool</u> |

where nat, bool, succ, eq, ∪, true and false are the same as before. U, del, stand for union, delete and negation. { } 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 whether i belongs to s or not. The other operations have the usual meanings.

The ADJ-type diagram[4] in Fig. 2 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 $1 \le k,\ j \le n$ if $k > j$ then $i_k > i_j$. If n is zero then we agree that s is $\phi$.

For notational convenience we will write s as

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

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

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.

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

(4) $\underline{del}(\underline{U}^n(\phi d_1\ldots d_n),i)$

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

(5) $\underline{U}(\underline{U}^m(\phi d_1\ldots d_m),\underline{U}^n(\phi d_1'\ldots d_n'))=\underline{U}^k(\phi e_1\ldots e_k)$, where $<e_1\ldots e_k>$ is the merge without repetitions of $<d_1\ldots d_n>$ with $<d_1'\ldots d_n'>$.

(6) $\underline{has}(\underline{U}^n(\phi d_1\ldots d_n),i) =$

$$= \begin{cases} \underline{true} & \text{if } \{i\} = dj \text{ for some } 1\le j\le n \\ \underline{false} & \text{otherwise} \end{cases}$$

(7) $\neg$ (true) = false

(8) $\neg$ (false) = 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

$$\underline{U}(\underline{U}^m(\phi d_1\ldots d_m),\underline{U}^n(\phi d_1'\ldots d_n')) \qquad (i)$$

can be performed in four steps.

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

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

   To check the validity of S1 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 $\underline{U}(\phi,d)$ 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 (i) will produce

$$\underline{U}(\underline{U}^n(\underline{U}^m(\phi d_1\ldots d_m)\phi d_1'\ldots d_{m-1}')d_n')$$

which can be rewritten as

$$\underline{U}^{m+n+1}(\phi d_1\ldots d_n\phi d_1'\ldots d_m') \qquad \text{(ii)}$$

2. We need to eliminate the double occurrence of $\phi$ in (ii). The following axiom can do it

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

   The application of S2 to (ii) will produce

$$\underline{U}^{m+n}(\phi d_1\ldots d_n\ d_1'\ldots d_m') \qquad \text{(iii)}$$

3. In (iii) 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 (iii) 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: $\underline{U}(\underline{U}(s,d),d) = \underline{U}(s,d)$

One can compare the axioms S1 to S4 that we got here with the axioms set-1 through set-4 presented by ADJ[4] 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 $\underline{del}$ 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

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

   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" $\underline{del}$ 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 equation[3].

S6: has(s,i) = false → del(s,i) = s

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

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

S8: eq(i,j)=false →
has(U(s,{i}),j) = 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 has(ϕ,j) which is of course false

S9: has(ϕ,j) = false

Finally for ¬ we have the obvious axioms:

B1: ¬ true = false .

B2: ¬ false = true .

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

## 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[7,14]

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 $i \le t$. In general we require $0<i\le t$ 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 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 as in Fig. 3.

A configuration containing the elements $a_1,\dots,a_m$ 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 $0<i\le m$, 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 $\underline{downS}^n\ \underline{pushS}^m(a_1,\dots a_m)$ for some $0\le n<m$, with all $a_i$'s distinct and 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) $\underline{pushS}(\underline{errorS},a) = \underline{errorS}$

(a2) $\underline{pushS}(t,\underline{errorD}) = \underline{errorS}$

(a3) $\underline{downS}(\underline{errorS}) = \underline{errorS}$

(a4) $\underline{popS}(\underline{errorS}) = \underline{errorS}$

(a5) $\underline{returnS}(\underline{errorS}) = \underline{errorS}$

(a6) $\underline{readS}(\underline{errorS}) = \underline{errorD}$

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

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

(b1) $\underline{downS}(\underline{createS}) = \underline{errorS}$

(b2) $\underline{popS}(\underline{createS}) = \underline{errorS}$

(b3) $\underline{returnS}(\underline{createS}) = \underline{createS}$

(b4) $\underline{readS}(\underline{createS}) = \underline{errorD}$

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

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

(c2) $\underline{downS}[\underline{downS}^n\underline{pushS}^m(a_1,\ldots a_m)] = \begin{cases} \underline{downS}^{n+1}\underline{pushS}^m(a_1,\ldots a_m) & \text{if } n+1<m \\ \underline{errorS} & \text{if } n+1=m \end{cases}$

(c3) $\underline{popS}[\underline{downS}^n\underline{pushS}^m(a_1,\ldots,a_m)] = \begin{cases} \underline{downS}^n\underline{pushS}^{m-1}(a_1,\ldots,a_{m-1}) & \text{if } n<m-1 \\ \underline{errorS} & \text{if } n=m-1 \end{cases}$

(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 (b1),...,(b4).

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

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

$pushS[downS^n pushS^m(a_1,\ldots,a_n),a] \cdot$
$downS(pushS[downS^{n-1} pushS^m(a_1,\ldots,a_m),a]) =$
$=\ldots= downS^n pushS[pushS^m(a_1,\ldots,a_m),a]$
since at the $i^{th}$ step we have the term
$downS^i pushS[downS^{n-i} pushS^m(a_1,\ldots,a_m),a]$
to which (C1) is still applicable as $i<n$.

(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 $downS[downS^n pushS^{n+1}(a_1,\ldots,a_n,a_{n+1})]$ into errorS.

We can apply C1 n times to get, calling $\underline{a} = (a_2,\ldots,a_{n+1})$

$downS^n downS\ pushS^n(pushS(createS,a_1),\underline{a}) =$
$= downS^n pushS^n(downS\ pushS(createS,a_1),\underline{a})$

This suggests the axiom

C2: $downS[pushS(createS,a)] = errorS$

the validity of which is immediate. By applying C2, we have altogether

$downS^{n+1} pushS^{n+1}(a_1,\ldots,a_n,a_{n+1}) =$
$= downS^n pushS^n(errorS,\underline{a})$
$= downS^n(errorS)$ (by n applications of A1)
$= errorS$ (by n applications of A3)

(3) Effect of popS

The similarity between (c3) and (c2) suggests

C3: $popS[downS(t)] = downS[popS(t)]$,

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

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

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

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

An application of C4, now leads to

$popS[downS^n pushS^m(a_1,\ldots,a_{m-1},a_m)] =$
$= downS^n pushS^{m-1}(a_1,\ldots,a_{m-1})$,
which is what we want if $n<m-1$. If $n = m-1$, then this reduces to errorS as in (2).

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

Sucessive 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

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

the validity of which being easy to be ascertained.

(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, with $\underline{a}^j = (a_1,\ldots,a_j)$

$readS[downS^n pushS^m(a_1,\ldots,a_m)] =$
$= readS[popS\ downS^{n-1} pushS^m(\underline{a}^m)]$ (by C7)
$= readS[downS^{n-1} popS\ pushS^m(\underline{a}^m)]$ (by C3 repeatedly)
$= readS[downS^{n-1}\ pushS^{m-1}(\underline{a}^{n-1})]$ (by C4)
..............................
$= readS[pushS^{m-n}(\underline{a}^{m-n})]$ (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 returnS by means of C6.

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(errorS,errorD) (by C6)
= errorS (by A1 or A2)

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

## CONCLUSION

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 valid 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 on canonical term algebras[3,4]. 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 canonical term algebra 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"[15].
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

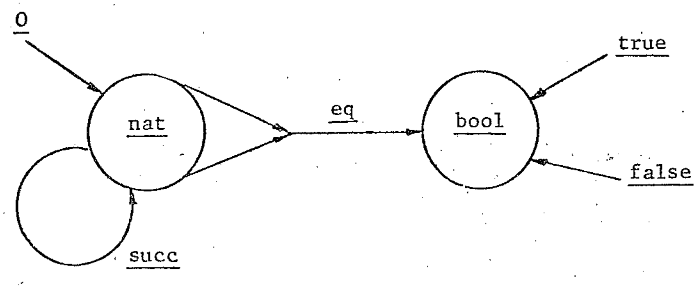


Fig. 1: Natural numbers with equality

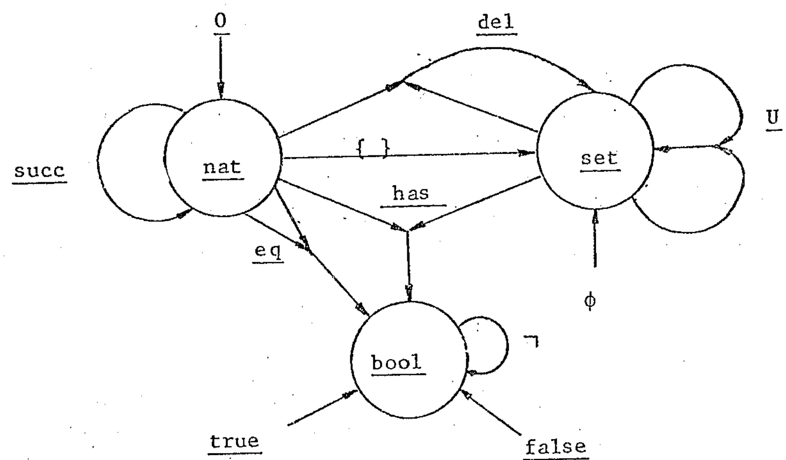


Fig. 2: Finite sets of naturals

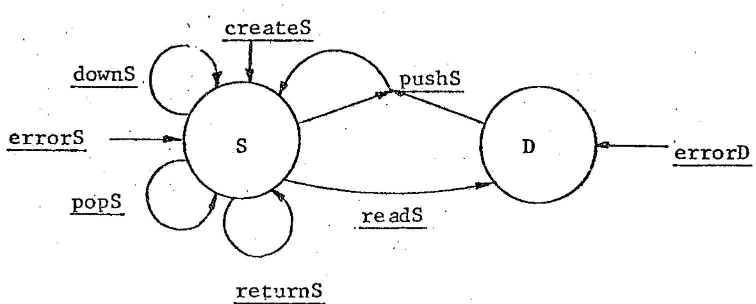


Fig. 3: Traversable stack

good hints on how to proceed, either by decomposing it into simpler transformations or by suggesting the candidate axiom. Here two features should be stressed. First, the validity check in case of failure 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.

REFERENCES

1. B.H.Liskov and S.N.Zilles,"Specification Techniques for data abstractions", IEEE Trans. on Software Engin., vol SE-1,pp 7-19, Mar. 1975.
2. J.V.Guttag, "Abstract data types and the development of data structures", Comm. of the ACM, vol. 20, pp. 396-404, June 1977.
3. J.W.Thatcher, E.G.Wagner and J.B.Wright, "Specification of abstract data types using conditional axioms"(extended abstract), IBM Res. Rept. RC 6214, Yorktown Heights, NY, Sept. 1976.
4. 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 RC 6487,Yorktown Heights, NY, Oct. 1976.
5. 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, pp. 89-93.
6. G.Grätzer, "Universal Algebra", Princeton, N.J., D.van Nostrand, 1968.
7. M.E.Majster, "Limits of the 'algebraic' specification of abstract data types", SIGPLAN Notices, vol. 12, pp 37-42, Oct. 1977.
8. J.J.Martin, "Critique of Mila E.Majster's paper 'Limits of the 'algebraic' specification of abstract data types'", SIGPLAN Notices, vol 12, pp 28-29, Dec. 1977.
9. M.E.Majster, "Letter to the editor", SIGPLAN Notices, vol, 13, pp 8-10, Jan 1978.
10. N.Hilfinger, "Letter to the editor", SIGPLAN Notices, vol. 13, pp.11-12, Jan 1978.
11. M.E.Majster, "Comment on a note by J.J. Martin", SIGPLAN Notices, vol. 13, pp. 22-23, Apr. 1978.
12. D.W. Jones, "A note on some limits of the algebraic specification method", SIGPLAN Notices, vol. 13, pp 64-67, Apr. 1978.
13. P.A. Subrahmanyan, "On a finite axiomatization of the data type L", SIGPLAN Notices, vol 13, pp 80-84, Apr. 1978.
14. T.A. Linden, "Specifying data types by restriction", Software Engeneering Notes, vol. 3, pp 7-13, Apr. 1978.
15. M.R. Levy, "Some remarks on abstract data types", SIGPLAN Notices, vol. 12, pp. 126-128, July 1977.