Implementations of abstract data types and their correctness proofs*

by

Jacques Loeckx

A 80/13

*submitted for publication

November 1980

Fachbereich 10 - Informatik
Universität des Saarlandes
6600 Saarbrücken
West Germany

1. Introduction

Algorithmic specifications of abstract data types have been introduced in [Lo 80b]. While being strongly related to the algebraic specification method used by e.g. [GH 78a, GHM 78b, Mu 80], the algorithmic specification method is more general and treats undefined and error values in a "natural" way; moreover it is felt to be easier to use for the specification of non-trivial data types.

The purpose of the present report is to introduce the notion of implementation of a data type in the framework of algorithmic specifications and to present "correctness conditions" for such implementations. These conditions are "symmetric" in that they indistinctly allow the use of a "representation function" (as in [SWL 77], [GHM 78b] or [EKP 80]) or of an "implementation function" (as in [ADJ 78], [Ga 79] or [Su 79]). It should be noted that the ideas proposed are also applicable to the implementation of data types with undefined or error values.

The present report does not contain formal developments nor extended proof examples. For a formal justification of the correctness conditions the reader is referred to [Lo 80c]; for a detailed description of some proofs - which, by the way, were performed mechanically with the AFFIRM-system [Mu 80] - the reader is referred to [Lo 80a].

Section 2 presents an overview of the algorithmic specification method. Section 3 introduces the notion of an implementation and presents the correctness conditions. Section 4 contains an example treated first by a representation function and then by an implementation function.

2. The algorithmic specification method

2.1 Algorithmic specifications

According to [Lo 80b] an algorithmic specification of an abstract data type, say τ, consists of :

(i) a list of *constructors* of type τ, such as (for τ = *Stack*):

emptystack: → *Stack*

push : *Stack* × *Integer* → *Stack*

these constructors define a *term language* similar to the carrier set of the word algebra of [ADJ 78]; note that a constructor is a purely syntactical object which is not to be interpreted as a function;

(ii) a predicate noted Is.τ which defines a subset of the term language; this predicate may be viewed as the "invariant" of the the type;

(iii) a predicate noted Eq.τ which defines an equivalence relation in the term language; this predicate may be viewed as defining the equality for the type;

(iv) a list of *external* (or: *user*) functions such as Push or Pop;

(v) a possibly empty list of *auxiliary* (or:*hidden)* functions which are intended to be inaccessible to the user.

Examples are in Figure 1, 2, 4 and 5.

The different functions introduced in a specification are essentially defined as recursive programs [Ma 74]; but in order to dispose of a clear theoretical basis the formalism used is that of (pure) LCF [Mi 72]. Essentially this formalism makes use of the λ-notation; moreover, if t is an expression and M a function variable, $[\alpha M.t]$ denotes the minimal fixpoint of $[\lambda M.t]$. In order to be applicable on term languages each term language is viewed as a flat lattice with a minimal element representing the undefined value (viz. ω) and a maximal element representing the error value (viz. Ω).

In the definition of these functions use may be made of the following "basic" functions defined over the term language:

- for each constructor, say cons , a function Is.cons defined as follows:

$$\text{Is.cons}(t) = \begin{cases} \underline{\text{true}} & \text{if the leftmost constructor in the term t is cons;} \\ \underline{\text{false}} & \text{otherwise;} \end{cases}$$

(i) Constructors

emptystack : → *Stack*

push: *Stack* × *Integer* → *Stack*

(ii) Acceptor function

Is.Stack has the constant value true

(iii) Equivalence relation

Eq.Stack is the syntactical equality (in the term language of type *Stack*)

(iv) External functions

```
Emptystack = emptystack
Push = [λs∈Stack, i∈Integer. push(s,i)]
Pop = [λs∈Stack.
         if Is.push(s) then s[1]
                       else emptystack]
Top = [λs∈Stack.
         if Is.push(s) then s[2]
                       else 0]
Isnew = [λs∈Stack
         if Is.push(s) then false
                       else true ]
```

(v) There are no auxiliary functions.

FIGURE 1: The specification of the data type *Stack*. The data type *Integer* with the 0-ary external function 0 is assumed to have been specified previously. Note that Emptystack is a 0-ary external function (i.e. a constant) and emptystack a 0-ary constructor (i.e. a term). Note also that, according to the specification, "popping" or "topping" an empty stack does not lead to an error but to an empty stack and the number 0 respectively.

```
(i) Constructors
           emptyset : → Set
           insert: Set  × Integer → Set
(ii) Acceptor function
           Is.Set = [αM.[λs∈Set. if Is.emptyset(s)
                      then true
                      else if Memberof (s[1], s[2])
                           then ω
                           else M(s[1]) ]]
(iii) Equivalence relation
           Eq.Set = [λs1,s2∈Set.
                      if Subset(s1,s2)
                      then Subset(s2,s1) else false]
iv) External functions
           Emptyset = emptyset
           Insert = [λs∈Set,i∈Integer.
                      if Memberof(s,i) then s else insert(s,i)]
           Delete = [αM.[λs∈Set,i∈Integer.
                      if Is.emptyset(s)
                      then emptyset
                      else if  s[2] = i
                           then s[1]
                           else insert(M(s[1],i),s[2])]]
           Memberof = [αM.[λs∈Set,i∈Integer.
                        if Is.emptyset(s)
                        then false
                        else if s[2] = i
                             then true
                             else M(s[1],i)]]
           Subset = [αM.[λs1,s2∈Set.
                      if Is.emptyset(s1)
                      then true
                      else if Memberof(s2,s1[2])
                           then M(s1[1],s2)
                           else false      ]]
```

FIGURE 2: The specification of the data type *Set*; the data type *Integer* is assumed to have been specified previously. Note that Is.Set avoids the occurrence of duplicates in the term language and that Eq.Set identifies sets which differ only by the order of occurrence of their elements.

- a "projector function" which extracts an "argument" of a constructor; the value of this function is denoted by the array notation; for instance, if t is a term of the form cons(u,v) then

$$t[1] = u$$
$$t[2] = v$$

For more precision and more details the reader is referred to [Lo 80b].

2.2 The data type defined

The data type τ defined by an algorithmic specification consists of a carrier set and a set of operations.

The carrier set is the set containing the following elements:

- the equivalence classes induced by Eq.τ on the subset of the term language defined by Is.τ;
- an element ERROR;
- an element UNDEFINED.

To each external function F is associated an operation F_{op} in the following way. Suppose F maps terms of type $\tau_1,\ldots,\tau_n$ into terms of type τ_{n+1}, $n \geq 0$; let $\varphi(t)$ denote

- the equivalence class of t, if t is a term
- ERROR, if $t = \Omega$
- UNDEFINED, if $t = \omega$;

then the corresponding operation F_{op} maps the carrier set of $\tau_1,\ldots,\tau_n$ into the carrier set of τ_{n+1} and its value is defined by:

$$F_{op}(\varphi(t_1),\ldots,\varphi(t_n)) = \varphi(F(t_1,\ldots,t_n))$$

Note that the definition of F_{op} is consistent only if the external function F satisfies certain *verification conditions*, e.g. that equivalent arguments lead to equivalent values. More details and a study of these conditions - which, by the way, are similar to those in [GHM 78b] - may be found in [Lo 80b].

Note that a data type together with the data types it makes use of (i.e. the data types which are at a "hierarchically lower level") constitutes a heterogeneous algebra.

2.3 Accessible data types

Consider the algebra defined by a set of specifications. An element of a carrier set is called *accessible* if it may be obtained as the value of an expression built with operations.

The algebra is called *surjective* if all elements of the carrier sets - except possibly ERROR and UNDEFINED - are accessible; it is called *error-free (total)* if ERROR (UNDEFINED) is not accessible.

In the sequel only sets of specifications defining surjective algebras will be considered.

3. Implementations

3.1 Definition

Let A_σ be the algebra defined by a set of specifications containing a specification of the data type σ. Let A_τ be defined by the same set of specifications except that the specification of the data type σ is replaced by a specification of the data type τ. The data types σ and τ are called *equivalent* if the algebras A_σ and A_τ are isomorphic. When the data type τ is felt to be more "elementary" than σ (e.g. because it is easy to write efficient programs for its external functions) one also says that τ is an *implementation* of σ.

In spite of its symmetric character this definition corresponds to the intuitive notion of an implementation; the main point is that the isomorphism is on the level of the algebras, not on the level of the external functions. By the way, this notion of implementation and the correctness conditions which will be deduced from it in Section 4 are very similar to those of [GHM 78b].

3.2 A special case

When the algebras A_σ and A_τ are both error-free and total it is sufficient to consider a weaker notion. Let A'_σ and A'_τ be the subalgebras of A_σ and A_τ obtained by deleting the elements ERROR and UNDEFINED. Then σ and τ are called *weakly equivalent* if A'_σ and

A'_τ are isomorphic. A *weak implementation* is defined similarly.

For reasons of simplicity we will limit ourselves to this special case. The general case is treated in [Lo 80c]; it merely differs by the fact that for each correctness condition the cases "ERROR" and "UNDEFINED" have to be treated separately.

4. The correctness conditions

4.1 The case of representation function

Let the algebras A_σ and A_τ be defined as in Section 3.2 . Let moreover

$$RP: \underline{\tau} \to \underline{\sigma}$$

be a function mapping the terms of type τ into terms of type σ; RP is called a *representation function*.

The data type τ is a weak implementation of the data type σ if the following three conditions are satisfied:

(i) for all terms d of type τ:

$$\text{if } Is.\tau(d) = \underline{\text{true}}$$
$$\text{then } Is.\sigma(RP(d)) = \underline{\text{true}}$$

(ii) for all terms d_1, d_2 of type τ:

$$\text{if } Is.\tau(d_1) = Is.\tau(d_2) = \underline{\text{true}}$$
$$\text{then } Eq.\tau(d_1,d_2) = Eq.\sigma(RP(d_1),RP(d_2))$$

(iii) there exists a one-to-one correspondence between the external functions of σ and τ; more precisely, to each external function

$$F: \underline{\rho}_1 \times \ldots \times \underline{\rho}_n \to \underline{\rho}_{n+1} \quad , \quad n \geq 0$$

of σ corresponds the external function

$$Im.F : \underline{\rho}'_1 \times \ldots \times \underline{\rho}'_n \to \underline{\rho}'_{n+1}$$

of τ with for each i, $1 \leq i \leq n+1$:

$$\rho'_i = \begin{cases} \tau & \text{if } \rho_i = \sigma \\ \rho_i & \text{otherwise} \end{cases}$$

moreover, for all terms d_i of type ρ_i, $1 \le i \le n$:

$$\text{if } Is.\rho_i(d_i) = \underline{true} \quad \text{for all } i,\ 1 \le i \le n$$

$$\text{then} \left| \begin{array}{l} Eq.\sigma(F(d_1',\dots,d_n'),\ RP(Im.F(d_1,\dots,d_n))) = \underline{true} \\ \qquad\qquad\qquad\qquad \text{if } \rho_{n+1} = \sigma \\ Eq.\rho_{n+1}(F(d_1',\dots,d_n'),\ Im.F(d_1,\dots,d_n)) = \underline{true} \\ \qquad\qquad\qquad\qquad \text{if } \rho_{n+1} \neq \sigma \end{array} \right. \qquad (a)$$

where for each i, $1 \le i \le n$:

$$d_i' = \left| \begin{array}{l} RP(d_i) \text{ if } \rho_i = \sigma \\ d_i \text{ else} \end{array} \right.$$

That these conditions imply τ to be a weak implementation of σ is formally proved in [Lo 80c]. Intuitively, (i) expresses that RP maps "allowed" terms into "allowed" terms; (ii) expresses that two terms of τ are equivalent iff the corresponding terms of σ are equivalent or, more loosely, that RP corresponds to an injective mapping of the equivalence classes of τ into those of σ. In order to interpret (iii) consider the special case $n = 1$, $\rho_1 = \rho_2 = \sigma$; in that case the condition (a) of (iii) becomes

$$Eq.\sigma(F(RP(d)),\ RP(Im.F(d))) = \underline{true} \qquad (b)$$

and is illustrated by Figure 3 (A); now as F is supposed to satisfy the verification conditions, equivalent arguments lead to equivalent values (see Section 2.2), i.e.

$$\text{if } Eq.\sigma(c,RP(d)) = \underline{true}$$
$$\text{then } Eq.\sigma(F(c),F(RP(d))) = true\ ;$$

hence (b) is equivalent with

$$\text{if } Eq.\sigma(c,RP(d)) = \underline{true}$$
$$\text{then } Eq.\sigma(F(c),RP(Im.F(d))) = \underline{true};$$

this condition is illustrated by Figure 3 (B) and expresses that the external functions Im.F of τ "simulate" the corresponding functions F of σ up to equivalence.

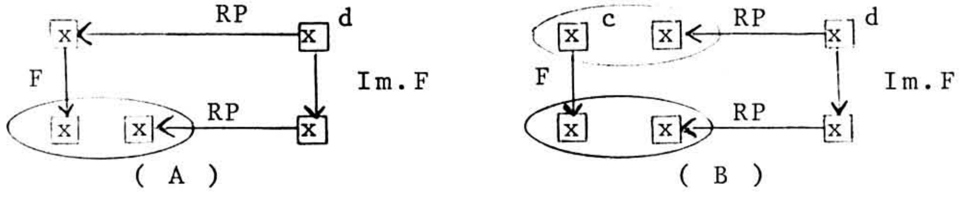

FIGURE 3: Illustration of the correctness condition (iii); points in the same "circle" are equivalent.

4.2 The case of an implementation function

The use of an implementation function

$$\text{IM}: \underline{\sigma} \rightarrow \underline{\tau}$$

rather than a representation function

$$\text{RP}: \underline{\tau} \rightarrow \underline{\sigma}$$

puts no problem: due to the symmetry of the definitions it is sufficient to permute σ and τ in the conditions of Section 4.1 .

5. An example

5.1 The data types

The implementation to be proved correct is that of a stack by a vector and a pointer.

More precisely, the data type to be implemented is that of Figure 1. The data type which constitutes the implementation consists of a vector and an integer which are "melted" into a single data type by a constructor called *pair* - as indicated in Figure 5. The specifications of the data type *Vector* are in Figure 4.

It is assumed that for all specifications the verification conditions mentioned in Section 2.2 have been checked.

5.2 The correctness conditions

Consider the representation function (*)

```
RP = [αM.[λm∈Imstack
          if m[2] = 0
          then emptystack
          else push(RP(pair(m[1], m[2]-1),
                    Read(m[1],m[2]))      ]]
```

(*) As in the figures use is made of the decimal notation and the infix notation for usual (external) functions of the type *Integer*; moreover Eq.Integer is replaced by the infix predicate "=".

```
(i) Constructors
            emptyvector : → Vector
            write: Vector × Integer × Integer → Vector
(ii) Acceptor function
            Is.Vector has the constant value true
(iii) Equivalence relation
            Eq.Vector = [λv1,v2∈Vector.
               if Subvector (v1,v2)
               then Subvector (v2,v1) else false ]
(iv) External function
            Emptyvector = emptyvector
            Write = [λv∈Vector, i,j∈Integer.
                                  write (v,i,j) ]
            Read = [αM.[λv∈Vector, i∈Integer.
                if Is.emptyvector (v)
                then 0 else if i = v[2]
                            then v[3] else M(v[1],i) ]]
            Defined = [αM.[λv∈Vector, i∈Integer.
                if Is.emptyvector (v)
                then false else if i = v[2]
                                then true else M(v[1],i) ]]
(v) Auxiliary function
            Subvector = [αM.[λ$v_1$,$v_2$∈Vector.
                if Is.emptyvector ($v_1$)
                then true else if Defined ($v_2$,$v_1$[2])
                               then if Read ($v_2$,$v_1$[2]) = $v_1$[3]
                                    then M($v_1$[1], $v_2$)
                                    else false
                               else false       ]]
```

FIGURE 4: The data type *Vector*. Note that "overwritten" values are not erased; note also that reading a not yet initialized error component leads to a read result "0" (rather than "error").

```
(i) Constructor
          pair : Vector × Integer → Imstack
(ii) Acceptor function
          Is.Imstack = [λm∈Imstack.
              if m[2] ≥ 0 then true else ω]
(iii) Equivalence relation
          Eq.Imstack = [αM.[λ$m_1$,$m_2$∈Imstack.
              if $m_1$[2] = $m_2$[2]
              then if $m_1$[2] = 0
                   then true
                   else if Read ($m_1$[1],$m_1$[2]) = Read ($m_2$[1],$m_2$[2])
                        then M(pair ($m_1$[1],$m_1$[2]-1),
                               pair ($m_2$[1],$m_2$[2]-1))
                        else false
              else false ]]
(iv) External functions
          Imemptystack = pair(emptyvector, 0)
          Impush = [λm∈Imstack, i∈Integer.
                     pair (Write (m[1],m[2]+1,i),m[2]+1)]
          Impop = [λm∈Imstack.
                     if m[2] = 0 then m else pair (m[1],m[2]-1)]
          Imtop  = [λm∈Imstack.
                     if m[2] = 0 then 0 else Read (m[1],m[2])]
          Imisnew = [λm∈Imstack. if m[2]= 0 then true else false ]
```

FIGURE 5: The data type *Imstack* which is intended to be an implementation of the data type *Stack*. Intuitively the data type consists of a vector and a "pointer". According to (iii) pair (v_1,i_1) and pair (v_2,i_2) are equivalent of either $i_1 = i_2 = 0$ or $(i_1 = i_2)$ and (i_1,i_2 "point" to the same value in v_1,v_2) and (pair (v_1,i_1-1) and pair (v_2,i_2-2) are equivalent).

Intuitively the value of RP is the empty stack when the pointer is zero; otherwise the value is the stack consisting of the elements v[1],...,v[k] where v is the vector and k the value of the pointer.

Let us consider the correctness conditions (i) to (iii) of Section 4.2 with σ = *Stack* and τ = *Imstack*.

The condition (i) trivially holds because Is.Stack has the constant value true.

The condition (ii) is :

```
        if Is.Imstack(m₁) = Is.Imstack(m₂) = true
        then Eq.Imstack(m₁,m₂) = Eq.Stack(RP(m₁),RP(m₂))
```

The condition (iii) consists of :

```
(a) Eq.Stack (Emptystack,RP(Imemptystack)) = true

(b) if Is.Imstack(m) = Is.Integer(i) = true
    then Eq.Stack (Push(RP(m),i),
                   RP(Impush(m,i))) = true

(c) if Is.Imstack(m) = true
    then Eq.Stack (Pop(RP(m)), RP(Impop(m))) = true

(d) if Is.Imstack(m) = true                                     (*)
    then Top(RP(m)) = Imtop(m)

(e) if Is.Imstack(m) = true                                     (**)
    then Isnew(RP(m)) = Imisnew(m)
```

These conditions have been proved mechanically with the help of the AFFIRM-system; the proofs are in [Lo 80a]. Essentially a proof consists in replacing functions such as Is.Imstack, Impush or Push by their definition and, if necessary, to apply structural induction on the term language. As a trivial example consider the proof of (d); by the definition of Is.Imstack one has to prove:

```
        if i ≥ 0
        then Top(RP(pair(v,i)) = Imtop(pair(v,i)).
```

(*) using the infix operator "=" for Eq.Integer

(**) using the infix operator "=" for Eq.Boolean

The case i = 0 leads to :

Top(emptystack) = 0

which holds by the definition of Top; the case i > 0 leads to

Top(push(RP(...), Read(v,i))) = Read(v,i)

which again holds by the definition of Top.

5.3 The case of an implementation function

The same correctness proof can be performed with the help of an implementation function such as

```
IM = [λs∈Stack.
          pair (Construct(s), Depth(s))]
```

in which Construct and Depth are (auxiliary) functions :

```
Depth = [αM.[λs∈Stack.
            if Is.emptystack(s) then 0
                                else M(s[1]) + 1]]
Construct = [αM.[λs∈Stack.
    if Is.emtystack(s) then emptyvector
                       else write (Construct(s[1]),
                                   Depth(s), s[2]) ]]
```

Informally, Depth determines the number of elements of the stack and Construct constructs a vector for a given stack.

Let us consider the correctness conditions (i) to (iii) of Section 4.1 with ɿ = *Imstack*, τ = *Stack* and IM instead of RP.

The condition (i) is :

Is.Imstack(IM(s)) = true (a)
for all terms s of type *Stack*

Due to the definition of Is.Stack and Eq.Stack the condition (ii) may be written

Eq.Imstack(IM(s), IM(s)) = true ;

this condition holds because Eq.Imstack is an equivalence relation (*).

(*) Remember the assumption that the verification conditions of the specification of *Imstack* have been checked; these conditions imply that Eq.Imstack is effectively an equivalence relation.

The condition (iii) consists of : for all terms s and i of type Stack and Integer respectively:

Eq.Imstack(Imemptystack, IM(Emptystack)) = <u>true</u> (b)
Eq.Imstack(Impush(IM(s),i),IM(Push(s,i))) = <u>true</u> (c)
Eq.Imstack(Impop(IM(s)), IM(Pop(s))) = <u>true</u> (d)
Imtop(IM(s)) = Top(s) (e)
Imisnew(IM(s)) = Isnew(s) (f)

The conditions (a) to (f) may be proved as in Section 5.2.

References

[ADJ 78] J.A. Goguen, J.W. Thatcher, E.G. Wagner, "An initial algebra approach to the specification, correctness and implementation of abstract data types" in "Current Trends in Programming Methodology IV" (R.Yeh, ed.), pp. 80-149, Prentice-Hall, 1978

[EKP 80] H. Ehrig, H.J. Kreowski, P. Padawitz, "Algebraic Implementation of Abstract Data Types : Concept, Syntax, Semantics and Correctness", Lecture Notes in Computer Science 85, Springer-Verlag, 1980, pp. 142-156

[Ga 79] M.C. Gaudel, "Algebraic specification of abstract data types", Internal Report 360, IRIA, Le Chesnay (August 1979)

[GH 78a] J.V. Guttag, J.J. Horning, "The algebraic specifications of abstract data types", Acta Informatica 10, 1, pp. 27-52 (1978).

[GHM 78b] J.V. Guttag, E. Horowitz, D.R. Musser, "Abstract data types and software validation", Comm. ACM 21, 12, pp. 1048-1064 (1978)

[Lo 80a] J. Loeckx, "Proving properties of algorithmic specifications of abstract data types in AFFIRM", AFFIRM-Memo-29-JL,USC-ISI, Marina del Rey, 1980

[Lo 80b] J. Loeckx, "Algorithmic specifications of abstract data types", Internal Report A 80/12, Fachbereich 10, Universität des Saarlandes, Saarbrücken, 1980 (submitted for publication)

[Lo 80c] J. Loeckx, "Some properties of implementations of abstract data types, Internal Report A 80/14, Fachbereich 10, Universität des Saarlandes, Saarbrücken, 1980

[Ma 74] Z. Manna, "Mathematical theory of computation", McGraw-Hill, 1974.

[Mi 72] R. Milner, "Implementation and applications of Scott's logic for computable functions", Proc. ACM Conf. on Proving Assertions about Programs, SIGPLAN Notices 7, 1, pp. 1-6 (1972)

[Mu 80] D.R. Musser, "Abstract data type specification in the AFFIRM-system", IEEE Trans. on Softw. Eng., SE-6, 1, pp. 24-32 (1980)

[Su 79] P.A. Subrahmanyam, "On proving the correctness of data type implementations", Internal report, Dept. Comp. Sc., University of Utah, Sept. 1979

[SWL 77] M. Shaw, W.A. Wulf, R.L. London , "Abstraction and verification in ALPHARD : Defining and specifying iteration and generators", CACM 20, 8 (1977)