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 \( \tau \), consists of:

(i) a list of constructors of type \( \tau \), such as (for \( \tau = \text{Stack} \)):

- \text{emptystack} : \text{Stack}
- \text{push} : \text{Stack} \times \text{Integer} \rightarrow \text{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.\( \tau \) which defines a subset of the term language; this predicate may be viewed as the "invariant" of the type;

(iii) a predicate noted Eq.\( \tau \) 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 \( \lambda \)-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. \( \omega \)) and a maximal element representing the error value (viz. \( \Omega \)).

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:

  \[
  \text{Is.cons}(t) = \begin{cases} 
  \text{true} & \text{if the leftmost constructor in the term } t \text{ is cons;} \\
  \text{false} & \text{otherwise;} 
  \end{cases}
  \]
(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
   \[\text{Is.Set} = [\alpha M. [\lambda s \in \text{Set}. \text{if Is.emptyset}(s)
      \text{then true}
      \text{else if Memberof}(s[1], s[2])
      \text{then } \omega
      \text{else } M(s[1]) ]]\]

(iii) Equivalence relation
   \[\text{Eq.Set} = [\lambda s_1, s_2 \in \text{Set}. \text{if Subset}(s_1, s_2)
      \text{then } \text{Subset}(s_2, s_1) \text{ else false}]\]

(iv) External functions
   \[\text{Emptyset} = \text{emptyset}\]
   \[\text{Insert} = [\lambda s \in \text{Set}, i \in \text{Integer}.
      \text{if Memberof}(s, i) \text{ then } s \text{ else insert}(s, i)]\]
   \[\text{Delete} = [\alpha M. [\lambda s \in \text{Set}, i \in \text{Integer}.
      \text{if Is.emptyset}(s)
      \text{then } \text{emptyset}
      \text{else if } s[2] = i
      \text{then } s[1]
      \text{else insert}(M(s[1], i), s[2]) ]]\]
   \[\text{Memberof} = [\alpha M. [\lambda s \in \text{Set}, i \in \text{Integer}.
      \text{if Is.emptyset}(s)
      \text{then false}
      \text{else if } s[2] = i
      \text{then true}
      \text{else } M(s[1], i)]\]
   \[\text{Subset} = [\alpha M. [\lambda s_1, s_2 \in \text{Set}.
      \text{if Is.emptyset}(s_1)
      \text{then true}
      \text{else if Memberof}(s_2, s_1[2])
      \text{then } M(s_1[1], s_2)
      \text{else false }]\]

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 \( \text{cons}(u,v) \) then

\[
\begin{align*}
t[1] &= u \\
t[2] &= v
\end{align*}
\]

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

2.2 The data type defined

The data type \( \tau \) 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 \( \mathcal{T} \) on the subset of the term language defined by \( \mathcal{L}_\tau \);
- an element ERROR;
- an element UNDEFINED.

To each external function \( F \) is associated an operation \( F_{\text{op}} \) in the following way. Suppose \( F \) maps terms of type \( \tau_1, \ldots, \tau_n \) into terms of type \( \tau_{n+1}, \ n \geq 0 \); 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 \( F_{\text{op}} \) maps the carrier set of \( \tau_1, \ldots, \tau_n \) into the carrier set of \( \tau_{n+1} \) and its value is defined by:

\[
F_{\text{op}} (\varphi(t_1), \ldots, \varphi(t_n)) = \varphi(F(t_1, \ldots, t_n))
\]

Note that the definition of \( F_{\text{op}} \) 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 \( A_\sigma \) be the algebra defined by a set of specifications containing a specification of the data type \( \sigma \). Let \( A_\tau \) be defined by the same set of specifications except that the specification of the data type \( \sigma \) is replaced by a specification of the data type \( \tau \). The data types \( \sigma \) and \( \tau \) are called equivalent if the algebras \( A_\sigma \) and \( A_\tau \) are isomorphic.

When the data type \( \tau \) is felt to be more "elementary" than \( \sigma \) (e.g. because it is easy to write efficient programs for its external functions) one also says that \( \tau \) is an implementation of \( \sigma \).

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 \( A_\sigma \) and \( A_\tau \) are both error-free and total it is sufficient to consider a weaker notion. Let \( A_\sigma' \) and \( A_\tau' \) be the subalgebras of \( A_\sigma \) and \( A_\tau \) obtained by deleting the elements ERROR and UNDEFINED. Then \( \sigma \) and \( \tau \) are called weakly equivalent if \( A_\sigma' \) and
A\textsuperscript{t} \text{ are isomorphic. A \textit{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\textsubscript{\textsigma} and A\textsubscript{\texttau} be defined as in Section 3.2.

Let moreover
\[
\text{RP}: \texttau \rightarrow \textsigma
\]
be a function mapping the terms of type \texttau into terms of type \textsigma; RP is called a \textit{representation function}.

The data type \texttau is a \textit{weak} implementation of the data type \textsigma if the following three conditions are satisfied:

(i) for all terms \text{d} of type \texttau:
\[
\text{if } \text{Is.}\texttau(d) = \text{true} \quad \text{then } \text{Is.}\textsigma(\text{RP}(d)) = \text{true}
\]

(ii) for all terms \text{d}_1, \text{d}_2 of type \texttau:
\[
\text{if } \text{Is.}\texttau(d_1) = \text{Is.}\texttau(d_2) = \text{true} \quad \text{then } \text{Eq.}\texttau(d_1, d_2) = \text{Eq.}\textsigma(\text{RP}(d_1), \text{RP}(d_2))
\]

(iii) there exists a one-to-one correspondence between the external functions of \textsigma and \texttau; more precisely, to each external function
\[
\text{F}: \rho_1 \times \ldots \times \rho_{n} \rightarrow \rho_{n+1}, \quad n > 0
\]
of \textsigma corresponds the external function
\[
\text{Im.}\text{F}: \rho'_1 \times \ldots \times \rho'_{n} \rightarrow \rho'_{n+1}
\]
of \texttau with for each \text{i}, \text{1} \leq \text{i} \leq \text{n+1}:
\[
\rho'_i = \begin{cases} 
\texttau & \text{if } \rho_i = \textsigma \\
\rho_i & \text{otherwise}
\end{cases}
\]
moreover, for all terms $d_i$ of type $\rho_i$, $1 \leq i \leq n$:

$$
\begin{align*}
\text{if } \text{Is.}\rho_i(d_i) &= \text{true} \quad \text{for all } i, 1 \leq i \leq n \\
\text{then } Eq.\sigma(F(d_1',\ldots,d_n'), \text{RP(Im.F}(d_1,\ldots,d_n))) &= \text{true} \\
& \quad \text{if } \rho_{n+1} = \sigma \\
Eq.\rho_{n+1}(F(d_1',\ldots,d_n'), \text{Im.F}(d_1,\ldots,d_n)) &= \text{true} \\
& \quad \text{if } \rho_{n+1} \neq \sigma
\end{align*}
$$

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

$$
\begin{align*}
d_i' &= \text{RP}(d_i) \text{ if } \rho_i = \sigma \\
d_i &\text{ else}
\end{align*}
$$

That these conditions imply $\tau$ to be a weak implementation of $\sigma$ is formally proved in [Lo 80c]. Intuitively, (i) expresses that $\text{RP}$ maps "allowed" terms into "allowed" terms; (ii) expresses that two terms of $\tau$ are equivalent iff the corresponding terms of $\sigma$ are equivalent or, more loosely, that $\text{RP}$ corresponds to an injective mapping of the equivalence classes of $\tau$ into those of $\sigma$. 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

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

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.

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

hence (b) is equivalent with

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

this condition is illustrated by Figure 3 (B) and expresses that the external functions $\text{Im.F}$ of $\tau$ "simulate" the corresponding functions $F$ of $\sigma$ 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

\[ IM: \sigma \rightarrow \tau \]

rather than a representation function

\[ RP: \tau \rightarrow \sigma \]

puts no problem: due to the symmetry of the definitions it is sufficient to permute \( \sigma \) and \( \tau \) 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 \( \text{pair} \) - as indicated in Figure 5. The specifications of the data type \( \text{Vector} \) 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 (*)

\[
RP = [\alpha M. [\lambda m \epsilon \text{Imstack} \\
if \ m[2] = 0 \then \text{emptystack} \\
else \text{push}(RP(\text{pair}(m[1], m[2]-1), \\
Read(m[1],m[2])))]]
\]

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

```plaintext
emptyvector : \rightarrow Vector
write: Vector \times Integer \times Integer \rightarrow Vector
```

(ii) Acceptor function

```plaintext
Is.Vector has the constant value true
```

(iii) Equivalence relation

```plaintext
Eq.Vector = [\lambda v_1,v_2 \in Vector. 
  if Subvector (v_1,v_2) 
  then Subvector (v_2,v_1) else false ]
```

(iv) External function

```plaintext
Emptyvector = emptyvector 
Write = [\lambda v \in Vector, i,j \in Integer. 
  write (v,i,j) ] 
Read = [\alpha M.[\lambda v \in Vector, i \in Integer. 
  if Is.emptyvector (v) 
  then 0 else if i = v[2] 
  then v[3] else M(v[1],i) ] ] 
Defined = [\alpha M.[\lambda v \in Vector, i \in Integer. 
  if Is.emptyvector (v) 
  then false else if i = v[2] 
  then true else M(v[1],i) ] ]
```

(v) Auxiliary function

```plaintext
Subvector = [\alpha M.[\lambda v_1,v_2 \in 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

\[ \text{pair} : \text{Vector} \times \text{Integer} \rightarrow \text{Imstack} \]

(ii) Acceptor function

\[ \text{Is.Imstack} = [\lambda m \in \text{Imstack}. \]
\[ \quad \text{if } m[2] > 0 \text{ then true else false } ] \]

(iii) Equivalence relation

\[ \text{Eq.Imstack} = [\alpha M. [\lambda m_1, m_2 \in \text{Imstack}. \]
\[ \quad \text{if } m_1[2] = m_2[2] \]
\[ \quad \text{then if } m_1[2] = 0 \]
\[ \quad \text{then true } \]
\[ \quad \text{else if Read (m_1[1], m_1[2]) = Read (m_2[1], m_2[2]) } \]
\[ \quad \text{then M(pair (m_1[1], m_1[2]-1), } \]
\[ \quad \quad \text{pair (m_2[1], m_2[2]-1)) } \]
\[ \quad \text{else false } \]
\[ \quad \text{else false } ] \]

(iv) External functions

\[ \text{Imemptystack} = \text{pair}(\text{emptyvector}, 0) \]
\[ \text{Impush} = [\lambda m \in \text{Imstack}, i \in \text{Integer}. \]
\[ \quad \text{pair (Write (m[1], m[2]+1, i), m[2]+1)) } \]
\[ \text{Impop} = [\lambda m \in \text{Imstack}. \]
\[ \quad \text{if } m[2] = 0 \text{ then m else pair (m[1], m[2]-1)) } \]
\[ \text{Imtop} = [\lambda m \in \text{Imstack}. \]
\[ \quad \text{if } m[2] = 0 \text{ then 0 else Read (m[1], m[2])) } \]
\[ \text{Imisnew} = [\lambda m \in \text{Imstack}. \text{ if } m[2] = 0 \text{ then true else false } ] \]

FIGURE 5: The data type \text{Imstack} which is intended to be an implementation of the data type \text{Stack}. Intuitively the data type consists of a vector and a "pointer". According to (iii) \text{pair (v}_1, i_1) \text{ and pair (v}_2, i_2) \text{ are equivalent of either } i_1 = i_2 = 0 \text{ or } (i_1 = i_2) \text{ and } (i_1, i_2 \ "point" \ to the same value in v}_1, v_2 \text{ and } \text{pair (v}_1, i_1-1) \text{ and pair (v}_2, i_2-2 \text{ are equivalent).}
Intuitively the value of \( \text{RP} \) is the empty stack when the pointer is zero; otherwise the value is the stack consisting of the elements \( v[1], \ldots, v[k] \) 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 \( \sigma = \text{Stack} \) and \( \tau = \text{Imstack} \).

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

The condition (ii) is:

\[
\text{if Is.Imstack}(m_1) = \text{Is.Imstack}(m_2) = \text{true} \\
\text{then Eq.Imstack}(m_1, m_2) = \text{Eq.Stack}(\text{RP}(m_1), \text{RP}(m_2))
\]

The condition (iii) consists of:

(a) \( \text{Eq.Stack}(\text{Emptystack}, \text{RP(Imemptystack)}) = \text{true} \)

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

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

(d) \( \text{if Is.Imstack}(m) = \text{true} \) \\
\text{(d) if Is.Imstack}(m) = \text{true} \\
\text{then Top(RP(m)) = Imtop(m)} \) (*)

(e) \( \text{if Is.Imstack}(m) = \text{true} \) \\
\text{then Isnew(RP(m)) = Isnew(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:

\[
\text{if } i > 0 \\
\text{then Top(\text{RP(pair(v,i))}) = Imtop(pair(v,i)).}
\]

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

(**) using the infix operator "=" for EqBOOLEAN
The case \( i = 0 \) leads to:

\[
\text{Top} \left( \text{emptystack} \right) = 0
\]

which holds by the definition of \( \text{Top} \); the case \( i > 0 \) leads to

\[
\text{Top} \left( \text{push} \left( \text{RP} \left( \ldots, \text{Read}(v,i) \right) \right) \right) = \text{Read}(v,i)
\]

which again holds by the definition of \( \text{Top} \).

### 5.3 The case of an implementation function

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

\[
\text{IM} = [\lambda s \in \text{Stack}. \ \
\text{pair} \left( \text{Construct}(s), \text{Depth}(s) \right) ]
\]

in which \( \text{Construct} \) and \( \text{Depth} \) are (auxiliary) functions:

\[
\text{Depth} = [\alpha M. [\lambda s \in \text{Stack}. \ \
\text{if} \ Is.\text{emptystack}(s) \ \text{then} \ 0 \ \
\text{else} \ M(s[1]) + 1 ] ]
\]

\[
\text{Construct} = [\alpha M. [\lambda s \in \text{Stack}. \ \
\text{if} \ Is.\text{emptystack}(s) \ \text{then} \ \text{emptyvector} \ \
\text{else} \ \text{write} \left( \text{Construct}(s[1]), \ \
\text{Depth}(s), s[2] \right) ] ]
\]

Informally, \( \text{Depth} \) determines the number of elements of the stack and \( \text{Construct} \) constructs a vector for a given stack.

Let us consider the correctness conditions (i) to (iii) of Section 4.1 with \( \tau = \text{Imstack} \), \( \tau = \text{Stack} \) and \( \text{IM} \) instead of \( \text{RP} \).

The condition (i) is:

\[
\text{Is.\text{Imstack}}(\text{IM}(s)) = \text{true} \tag{a}
\]

for all terms \( s \) of type \( \text{Stack} \)

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

\[
\text{Eq.\text{Imstack}}(\text{IM}(s), \text{IM}(s)) = \text{true} ;
\]

this condition holds because \( \text{Eq.\text{Imstack}} \) is an equivalence relation \((*)\).

\((*)\) Remember the assumption that the verification conditions of the specification of \( \text{Imstack} \) have been checked; these conditions imply that \( \text{Eq.\text{Imstack}} \) is effectively an equivalence relation.
The condition (iii) consists of: for all terms s and i of type Stack and Integer respectively:

\[
\begin{align*}
\text{Eq.}\text{Im}\text{stack}(\text{Im}\text{emptystack}, \text{IM}\text{(Emptystack)}) &= \text{true} \\
\text{Eq.}\text{Im}\text{stack}(\text{Impush}\text{(IM}(s),i), \text{IM}(\text{Push}(s,i))) &= \text{true} \\
\text{Eq.}\text{Im}\text{stack}(\text{Impop}\text{(IM}(s)), \text{IM}(\text{Pop}(s))) &= \text{true} \\
\text{Im}\text{top}(\text{IM}(s)) &= \text{Top}(s) \\
\text{Im}\text{is}\text{new}(\text{IM}(s)) &= \text{Is}\text{new}(s)
\end{align*}
\]

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

References


[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)

