INFORMATION PROCESSING 80, S.H. Lavington (ed.)
North-Holland Publishing Company

PROVING IMPLEMENTATIONS CORRECT – TWO ALTERNATIVE APPROACHES

Hans-Dieter EHRICH and Udo LIPECK
University of Dortmund, Department of Informatics
Dortmund, Federal Republic of Germany

Motivated by the desire to prove implementations correct, we present two approaches to making the fundamental notion of implementation as a relationship between data types precise. The first one captures the idea of simulation, while the second one, being more constructive in nature, formalizes the idea of constructing on a given basis in distinguished steps something isomorphic to a given target structure. Both approaches give rise to corresponding proof methods which are demonstrated in examples. The approaches are shown to be equivalent as far as the existence of implementations is concerned.

1. INTRODUCTION

The essential task of programming is to bridge the conceptual distance between the facilities of a programming language and an abstractly formulated solution for a user's problem, based on a certain application. Therefore programs are designed via some intermediate levels of (bottom-up) abstraction or (top-down) refinement {2}. Between these levels simpler programming steps apply to "realize" or "implement" the concepts of a higher level by those of the next lower one.

The basic components of such a hierarchical design are formed by modules {11} having a specified behaviour and communicating with other modules via specified interfaces. Intuitively speaking, a module is a unit built up from data, operations, and processes. A simple example of a module is a procedure describing a function in the mathematical sense. More complex modules are data types {6,9}, consisting of a collection of data objects together with a set of operations examining and manipulating these data. Such modules are supported by quite a lot of programming languages, e.g. SIMULA, CLU, ALPHARD, MODULA, ADA. Even more sophisticated modules include processes defined in terms of the operations and some control structures. For the time being, we exclude processes from our considerations and concentrate on data type modules.

When designing a software system, it is most important that higher level modules are implemented correctly by lower level modules. To this end, it is necessary to develop more formal and rigorous methods to investigate what "implementation" means precisely, independently of any programming language. Obviously, a precise notion of implementations requires a precise notion of a module. For equationally defined abstract data types, there have been several approaches to defining implementation as a relationship between data types or their specifications {1,3-4,5,8}, providing a basis for developing verification methods for proving implementations correct. That work has been influenced by Guttag's ideas {6,7}. A more general concept of data type and implementation is due to Mayoh {1o}.

In this paper, we discuss two alternatives for making the notions of data type and implementation precise. The first one is based on Mayoh's approach. The other one, which is based on the approaches of the authors {4,8}, takes a more constructive viewpoint. While a great deal of the literature cited above uses a rather sophisticated algebraic machinery, our objective in this paper is to use as simple and elementary mathematics as possible in order to expose the fundamental structure of implementation.

2. FUNCTIONAL DATA TYPES

One of the central ideas in modular programming is that the external use of a module may depend only on its specified behaviour, not on internal peculiarities of a particular realization. If the module is a procedure, its behaviour is readily expressed by its input-output function. If the module, however, has an "internal memory", e.g. some files, its behaviour at a certain moment depends on its history, i.e. on some (or all) previous inputs, starting from the moment the module was created. In such cases the behaviour (assumed to be deterministic) can be characterized by a function mapping input histories to outputs.

Therefore, in a very general sense, module behaviour can be understood as a function whose arguments typically possess some structure. Without assuming a specific structure for the time being, we call the arguments "expressions" and the outputs "values".

Following Mayoh {1o}, we define a (functional) data type as the behaviour of a module.

Definition 2.1:

A functional data type is a total surjective function

$$\delta : E \to V$$

(called "evaluation") from a nonempty set of expressions E to a set of values V.

(Whenever it will not cause confusion, we will omit the adjective 'functional'.)

This definition reflects the intuition that each expression possesses a value (perhaps 'undefined'), and each value appears as the result of evaluating some expression. Since E is nonempty, V cannot be empty either.

Example 2.2:

Consider, as an example of a module, queues with their characteristic operations add (to the rear), remove (at the front) and front. Let B be a set of entries, and let b_0 be a distinguished entry (playing the role of a default option). Then, queue values are defined to be strings over B, formally $V_q = B^*$, and expressions E_q are defined inductively as follows:

(i) empty is a queue expression; each $b \in B$ is an entry expression.

(ii) If q is a queue expression and b is an entry expression, then add b to q and remove q are queue expressions, and front of q is an entry expression.

(iii) E_q contains exactly the queue and entry expressions constructable by (i) and (ii).

The evaluation $\delta_q: E_q \to V_q$ is given by: (of course, $\delta_q(b)=b$ for $b \in B$)

$$\delta_q(\text{empty}) = \varepsilon \quad \text{(the empty string)}$$

$$\delta_q(\text{add b to q}) = \delta_q(q) \circ \delta_q(b)$$

$$\delta_q(\text{remove q}) = \begin{cases} x & \text{if } \delta_q(q) = b \circ x \\ & \text{for some } x \in B^*,\ b \in B \\ \varepsilon & \text{otherwise} \end{cases}$$

$$\delta_q(\text{front of q}) = \begin{cases} b & \text{if } \delta_q(q) = b \circ x \\ & \text{for some } x \in B^*,\ b \in B \\ b_0 & \text{otherwise} \end{cases}$$

Example 2.3:

We give one more example that will be used later for implementing queues, namely (idealized) arrays. Let B, b_0 given as in ex. 2.2. Then, values are defined to be functions from $\mathbb{N}$ (the natural numbers including 0) to B (where only a finite number of arguments has a function value $\neq b_0$), representing infinite arrays with indices 0,1,2,..., and elements from B as the result of array accesses. Formally, $V_a = B \cup B^{\mathbb{N}}$.

Expressions E_a are defined inductively:

(i) new is an array expression; each $b \in B$ is an entry expression.

(ii) If a is an array expression, i is a natural number, and b is an entry expression, then a[i]:=b is an array expression and a[i] is an entry expression.

(iii) E_a contains exactly the array and entry expressions constructable by (i) and (ii).

The evaluation $\delta_a: E_a \to V_a$ is defined as follows:

$$\delta_a(\text{new}) = \lambda x[b_0] \quad \text{(the constant function with value } b_0\text{)}$$

$$\delta_a(a[i]:=b) = \lambda x \begin{bmatrix} \delta_a(a)\ (x) & \text{if } x \neq i \\ \delta_a(b) & \text{otherwise} \end{bmatrix}$$

$$\delta_a(a[i]) = \delta_a(a)\ (i)$$

3. IMPLEMENTATION AS SIMULATION

Let $\delta_t: E_t \to V_t$ be the data type to be implemented, called the target data type, and let $\delta_b: E_b \to V_b$ be that used for implementation, called the base data type. A very natural notion of implementation is due to Mayoh [10]. It reflects the idea that δ_b simulates the behaviour of δ_t in the sense that each target expression is encoded as a base expression and evaluated by δ_b. This value is then decoded to a target value such that it is the same as that resulting from evaluating the original expression directly by δ_t. These conditions are summarized as follows:

Definition 3.1:

δ_b implements δ_t iff there are a function encode: $E_t \to E_b$ and a partial function decode: $V_b \to V_t$ such that the diagram in fig. 1 commutes, i.e.:

$$\text{decode}\ (\delta_b(\text{encode}(e))) = \delta_t(e) \quad \text{for all } e \in E_t$$

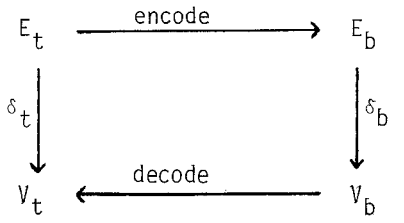

Figure 1

From the commutativity of the diagram, it follows that encode is a total and decode is a surjective function.

Defining implementations as simulations as above suggests a conceptually simple method for correctness proofs: given δ_t and δ_b, we must specify the encode and decode functions and prove commutativity of the diagram.

Example 3.2:

We want to implement queues by arrays with two pointers, front and rear. The queue data type is described in ex. 2.2. For implementation, we need a data type derived from the arrays (ex. 2.3) as follows: Values are triples (front, rear, array), formally $V_b = \mathbb{N} \times \mathbb{N} \times V_a$.

Expressions are all triples consisting of natural number expressions in the first two components and an array expression in the last. Formally, $E_b = E_{\mathbb{N}} \times E_{\mathbb{N}} \times E_a$, where $E_{\mathbb{N}}$ is the set of expressions made up from natural numbers and the operations + and -. Then $\delta_b(\langle m,n,a\rangle) := (m,n,\delta_a(a))$ defines the evaluation.

Intuitively, a queue is represented by an interval in the array, bounded by front and rear. This idea leads to the following inductive definition of the encode function:

$$\text{encode}(\underline{\text{empty}}) = \langle 0 , 0 , \underline{\text{new}}\rangle$$

Let $\text{encode}(q) = \langle f , r , a\rangle$.

$$\text{encode}(\underline{\text{add}}\ b\ \underline{\text{to}}\ q) = \langle f , r+1 , a[r]:= b\rangle$$

$$\text{encode}(\underline{\text{remove}}\ q) = \begin{cases} \langle 0 , 0 , \underline{\text{new}}\rangle & \text{if } f+1 = r \text{ or } f = r = 0 \\ \langle f+1 , r , a\rangle & \text{otherwise} \end{cases}$$

$$\text{encode}(\underline{\text{front of}}\ q) = \begin{cases} b_0 & \text{if } f = r = 0 \\ a[f] & \text{otherwise} \end{cases}$$

The domain of the decode function is the set

$$\{(f,r,a) \mid f=r=0 \text{ or } f < r\}$$

and decode is defined by:

$$\text{decode}((f,r,a)) := b_1 \circ b_2 \circ \ldots \circ b_{r-f} \in B^*$$

where $b_i = a(f-1+i)$ $(i=1,\ldots,r-f)$

Thus the position of such an interval in the array and the entries outside that interval are ignored.

In order to prove that (encode, decode) is an implementation, we must show:

$$\text{decode}(\delta_b(\text{encode}(q))) = \delta_q(q) \quad \text{for all } q \in E_q .$$

This can be done by induction over the expressions in E_q. We carry out the case $q = \underline{\text{add}}\ b\ \underline{\text{to}}\ q'$. The remaining cases are straightforward and left to the reader.

$q = \underline{\text{add}}\ b\ \underline{\text{to}}\ q'$:

Let $\text{encode}(q') = \langle f , r , a\rangle$. By induction, we assume that $\text{decode}(\delta_b(\text{encode}(q'))) = \delta_q(q')$.

Then we have:

$$\text{decode}(\delta_b(\text{encode}(\underline{\text{add}}\ b\ \underline{\text{to}}\ q')))$$
$$= \text{decode}(\delta_b(\langle f , r+1 , a[r]:=b\rangle))$$
$$= \text{decode}((f , r+1 , \delta_a(a[r]:=b)))$$
$$= \text{decode}((f , r+1 , \lambda x \begin{bmatrix} \delta_a(a)(x) & \text{if } x \neq r \\ b & \text{otherwise} \end{bmatrix}))$$
$$= b_1 \circ b_2 \circ \ldots \circ b_{r-f} \circ b$$
where $b_i = \delta_a(a)(f-1+i)$ $(i=1,\ldots,r-f)$
$$= \text{decode}((f,r,\delta_a(a))) \circ b$$
$$= \text{decode}(\delta_b(\langle f,r,a\rangle)) \circ b$$
$$= \text{decode}(\delta_b(\text{encode}(q'))) \circ b$$
$$= \delta_q(q') \circ b \quad \text{by assumption}$$
$$= \delta_q(\underline{\text{add}}\ b\ \underline{\text{to}}\ q')$$

□

4. ABSTRACT DATA TYPES

The above method for describing modules and verifying implementations requires us to deal with the values involved explicitly. For this purpose, certain mathematical models like functions (for arrays), strings (for queues) etc. are usually used. Giving full details about values, however, is not always desirable for abstract design and often practically impossible in a precise way, particularly for complex systems. The main disadvantage is that many technical peculiarities of the model chosen must be considered in addition to the intended and essential properties of a data type.

Hence, the behaviour characteristics ought to be defined independently of any explicit value structure, i.e. in an "abstract" way (cf. {1,6}). This suggests that we should describe the results of evaluations relatively, i.e. by relating the expressions to each other: two expressions are equivalent iff they evaluate to the same value.

A formal basis fitting this view can be derived from our definition 2.1 of functional data types. Because of surjectivity, the underlying function δ induces an equivalence relation on the expressions as follows:

$$e_1 \equiv_\delta e_2 \text{ iff } \delta(e_1)=\delta(e_2) \text{ for } e_1,e_2 \in E$$

That means: the values are in bijective correspondence with the equivalence classes of expressions. - With respect to other, structured (algebraic or order-theoretic) approaches to data types (where δ resp. the induced relation has to satisfy certain compatibility conditions), we call the equivalence relation a 'congruence'.

Definition 4.1:

An abstract data type

$$D = (E , \equiv)$$

is given by a set of expressions E and a congruence $\equiv$ on E.

Definition 4.2:

The (functional) data types presented by an abstract data type $D = (E,\equiv)$ are those mappings $\delta: E \to V$ with induced congruence equal to $\equiv$. - These are not considered to be essentially different from $\delta_D: E \to V_D$, where V_D is $E/\equiv$, the set of congruence classes, and δ_D maps each expression to its equivalence class.

For practical applications, we need finite specifications of all equivalences among expressions. Equational systems provide a well-known technique for doing that, having the advantage of being theoretically investigated by many researchers in various fields. If the expressions are constant terms built up from a set of operation symbols (without variables), equations between terms with variables can be set up, from which exactly all valid equations between expressions can be generated canonically.

Queues and arrays as in example 2.2 and 2.3 can be specified by the following finite sets of equations:

Example 4.3: Queues:

Let b and q be variables to be substituted by entry and queue expressions respectively.

```
remove empty              = empty
remove (add b to q)       =
        if q = empty then empty
          else add b to remove q fi
front of empty            = b0
front of (add b to q)     =
        if q = empty then b
          else front of q fi
```

Example 4.4: Arrays:

Here i/j, $b/b_1/b_2$, and a are variables for numbers, entries, and arrays.

```
new[i]:=b0                = new
(a[i]:=b1)[j]:=b2         =
        if i=j then a[i]:=b2
               else (a[j]:=b2)[i]:=b1 fi
new[j]                    = b0
(a[i]:=b)[j]              =
        if i=j then b else a[j] fi
```

5. IMPLEMENTATION AS STEPWISE CONSTRUCTION

Let D_b and D_t be abstract data types, called "base" and "target". We are now going to describe how an implementation of the target can be constructed from the base. Intuitively speaking, D_b plays the role of a (virtual) "machine", and to implement means to "program" this machine such that the programs behave in some sense like the target. More precisely, the implementation constitutes an abstract data type D_{ip} that is isomorphic to D_t. It is not quite obvious how to capture the idea of programming as a relationship between D_b and D_{ip}. Intuition and experience suggest that this relationship is subdivided into three steps, in coincidence with the approaches taken in {5,8}, and related to the ideas in {1}.

Firstly, the base is enriched by additional expressions, the "programs", designed to correspond to target expressions. They are defined somehow in terms of the base expressions. Thus a realization data type is obtained, where both base and new expressions can be applied.

Secondly, the enriched base is restricted to a part forming the actual representation of the target. The remaining expressions of the realization are hidden for the user of the implementation.

Thirdly, exactly this representation is interpreted as target. Actually, this step is subdivided into two steps: representation expressions standing for the same target entities are identified such that the resultant abstract data type is isomorphic to the target.

As a technical tool for talking about relationships between abstract data types, we introduce the notion of morphism as a structure preserving map, by analogy to homomorphisms in algebra.

Definition 5.1:

Let $D_i = (E_i, \equiv_i)$, i=0,1, be abstract data types. A morphism $f: D_0 \to D_1$ from D_0 to D_1 is given by a function $f': E_0 \to E_1$ such that

$$e \equiv_0 \bar{e} \Rightarrow f'(e) \equiv_1 f'(\bar{e}) \quad \text{for all } e,\bar{e} \in E_0.$$

Morphisms of abstract data types have obvious counterparts on the level of functional data types, as indicated by the following proposition.

Proposition 5.2:

A function $f': E_0 \to E_1$ determines a morphism $f: D_0 \to D_1$ iff, for arbitrary data types $\delta_i: E_i \to V_i$ presented by D_i (i=0,1), there is a function $f'': V_0 \to V_1$ such that

$$f'' \circ \delta_0 = \delta_1 \circ f'.$$

The easy proof is left to the reader. Such pairs (f',f'') are called morphisms of (functional) data types.

Those implementation steps between abstract data types outlined above can be clarified by special morphisms.

Definition 5.3:

$f: D_0 \to D_1$ is an isomorphism iff $f': E_0 \to E_1$ is bijective, and the inverse mapping $f'^{-1}: E_1 \to E_0$ determines a morphism, too.

This means that not only the expressions, but also the congruence classes are in one-to-one correspondence.

Definition 5.4:

$f: D_0 \to D_1$ is an enrichment iff f' is injective, and we have

(i) for each $e \in E_0$, there is an expression $\bar{e} \in E_1$ such that $e \equiv_1 f'(\bar{e})$

(ii) for all $e,\bar{e} \in E_0$: $f'(e) \equiv_1 f'(\bar{e})$ iff $e \equiv_0 \bar{e}$.

Enriching an abstract data type means adding expressions such that each new expression must be defined as equivalent to some old one (completeness), and that no new equivalences are introduced on the old expressions (consistency).

Definition 5.5:

$f: D_0 \to D_1$ is a restriction iff f' is injective, and for all $e,\bar{e} \in E_0$: $e \equiv_0 \bar{e}$ iff $f'(e) \equiv_1 f'(\bar{e})$.

Definition 5.6:

$f: D_0 \to D_1$ is an identification iff f' is bijective.

All that can happen with an identification is that nonequivalent expressions in D_0 are mapped to equivalent expressions in D_1 and thus identified. The expression sets are the same, up to unique renaming.

Now we are in a position to give a precise definition of implementation according to our above argument.

Definition 5.7:

An implementation of a target D_t by a base D_0 is a quadruple I = (er,rs,id,i) of morphisms as depicted in the following diagram, where er is an enrichment, rs is a restriction, id is an identification, and i is an isomorphism.

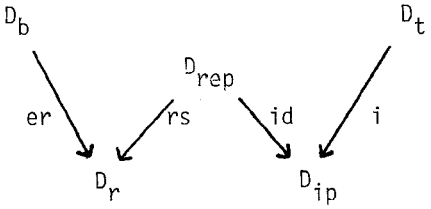

Figure 2

D_r, D_{rep}, and D_{ip} are called the realization, representation, resp. implementation data type.

This situation of implementation between abstract data types can equivalently be stated on the level of data types as evaluation functions.

Proposition 5.8:

For the respective morphisms according to prop. 5.2 the following conditions must hold:

(i) er' is injective, er" is bijective.

(ii) rs' is injective, rs" is injective.

(iii) id' is bijective, id" is surjective.

(iv) i' is bijective, i" is bijective.

On the basis of this definition of implementation, correctness proofs for implementations have to establish the existence of the respective morphisms with their required properties. That rs is a restriction and id is an identification is usually evident. The key points to check are er being an enrichment, i.e. er being complete and consistent, and i being an isomorphism.

Example 5.9:

Subsequently, we give an implementation of queues by arrays in the sense of definition 5.7 (cf. also ex. 3.2). These are specified equationally as abstract data types in examples 4.3 and 4.4 .

The accurate base abstract data type describes triples consisting of an array and two natural numbers. It is obtained from arrays by adding triple expressions <m,n,a>, where m,n are natural number expressions and a is an array expression, natural number expressions $t.proj_1$, $t.proj_2$, where t is a triple expression, and array expressions $t.proj_3$. The specification is completed by the equations

$$\langle t_1,t_2,t_3\rangle.proj_1 = t_i \quad (i=1,2,3)$$
$$\langle t.proj_1, t.proj_2, t.proj_3\rangle = t$$

First we have to enrich the base to an appropriate realization D_r . For mnemonic purposes, $proj_1$,$proj_2$,$proj_3$ are renamed by f,r,arr (the numbers are to play the roles of pointers to front and rear). The following expressions are added:

- init, addrear(t,b), removefront(t) as new triple expressions,
- and front(t) as new entry expressions.

Here, t and b may be replaced by arbitrary triple resp. entry expressions. These new expressions are related to the base expressions by the following equations:

init = <0,0, new>

addrear(t,b) = <t.f, t.r+1, t.arr[t.r]:=b>

removefront(t) =
  if t.r-t.f≤1 then <0,0, new>
    else <t.f+1, t.r, t.arr> fi

front(t) =
  if t.f=t.r then b_0
    else t.arr[t.f] fi

This specification obviously is complete and consistent w.r.t. the base.

Now we have to choose a restriction of D_r that is to be used as representation (D_{rep}) of queues (our target). It consists of all those expressions constructable with init, addrear, removefront, front, and all $b \in B$, together with their inherited congruence.

The next step is to interpret representation expressions as target expressions. To this end, we consider the bijective renaming given by associating init with empty, addrear with add...to... , removefront with remove, front with front of, and each entry $b \in B$ with itself. Let $\phi: E_{rep} \to E_q$ be this mapping.

According to our definition of implementation, we describe the interpretation in two substeps namely an identification and a subsequent isomorphism. The identification is readily obtained by setting $E_{ip} = E_{rep}$, and simply

adding the target equations from example 4.3 -mapped by δ^{-1}- to D_{rep}, thus getting D_{ip}.

Then we have a bijective mapping $i=\phi^{-1}: E_q \to E_{ip}$. That i is an abstract data type morphism is evident from the construction. In order to prove isomorphism, we must check whether we have not identified too much. Technically speaking, i^{-1} must be a morphism, too, or equivalently: $e \not\equiv_q \bar{e} \Rightarrow i(e) \not\equiv_{ip} i(\bar{e})$ for all $e,\bar{e} \in E_q$.

We give the main arguments of the proof: For queue expressions $e,\bar{e}$ there are queue expressions $g,\bar{g}$ with $g \equiv_q e$ and $\bar{g} \equiv_q \bar{e}$, such that g and $\bar{g}$ are terms in the operations empty and add...to... alone, and we have $e \equiv_q \bar{e}$ iff $g=\bar{g}$, i.e. g and $\bar{g}$ are equal as expressions. This can be proven by induction. Evidently, $g=\bar{g}$ iff $i(g)=i(\bar{g})$, since i is bijective. By induction using base and realization equations, we can show that $i(g) \equiv_{rep} i(\bar{g})$ enforces $i(g)=i(\bar{g})$, and adding target equations has no effects on these expressions, i.e. $i(g) \equiv_{ip} i(\bar{g})$ iff $i(g) \equiv_{rep} i(\bar{g})$. Consequently, $e \not\equiv_q \bar{e} \Rightarrow g \neq \bar{g} \Rightarrow i(g) \neq i(\bar{g}) \Rightarrow i(e) \equiv_{ip} i(g) \not\equiv_{ip} i(\bar{g}) \equiv i(\bar{e})$. □

6. EQUIVALENCE OF THE APPROACHES

With respect to functional data types, both notions of implementation cover the same cases of base - target pairs. The following equivalence proof demonstrates how one sort of implementation can be transformed into the other one.

Theorem 6.1

Let D_b, D_t be abstract data types and δ_b, δ_t functional data types presented by D_b, D_t. Then, there exists an implementation I = (encode, decode) of δ_t by δ_b (according to def. 3.1) if and only if there is an implementation $\hat{I}$ = (er, rs, id, i) of D_t by D_b (acc. def. 5.7).

Proof: Because of space limitations, we can only give a short sketch. The situations for I and $\hat{I}$ are depicted in figures 1 and 3, respectively. (Cf. prop. 5.8).

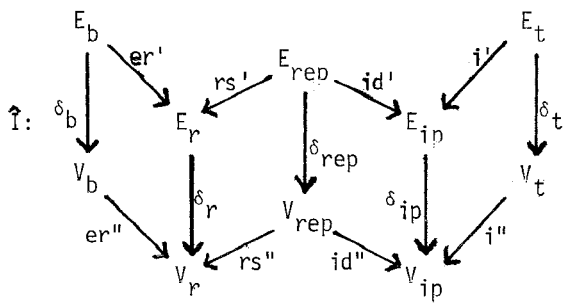

Figure 3

"⇒": Given I, we set

$\delta_{rep} := \delta_b \circ \text{encode}: E_t \to V_b$

$\delta_r := \delta_b + \delta_{rep}: E_b + E_t \to V_b$

$\delta_{ip} := \delta_t: E_t \to V_t$

and id" := decode. (The other maps are trivial)

"⇐": Given $\hat{I}$, we set

encode := $(\delta_r \circ er')^{-1} \circ \delta_r \circ (rs' \circ id'^{-1} \circ i'): E_t \to E_b$

decode := $(is")^{-1} \circ id" \circ (rs")^{-1} \circ er": V_b \to V_t$

The required conditions can be shown straightforward. □

This result gives some support to our belief that the formalizations presented here really meet the intuitive concept of implementation.

REFERENCES

{1} ADJ (J.A. Goguen, J.W. Thatcher, and E.G. Wagner), An initial algebra approach to the specification, correctness, and implementation of abstract data types, Current trents in programming methodology, vol. IV: Data Structuring (R.T. Yeh, ed.), Prentice Hall, Englewood Cliffs, 1978, 80 - 149.

{2} O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press, London, 1972

{3} H.-D. Ehrich, Extensions and implementations of abstract data type specifications, Proceedings 7th MFCS Symposium, Lecture Notes in Computer Science, vol. 64, Springer-Verlag, Berlin, 1978, 155 - 164.

{4} H.-D. Ehrich, On the theory of specification, implementation, and parametrization of abstract data types, Report 82/79, Abteilung Informatik, Universität Dortmund 1979

{5} H. Ehrig, H.-J. Kreowski, and P. Padawitz, Algebraische Implementierung abstrakter Datentypen, Report 79-3, Fachbereich Informatik (20), Techn. Univ. Berlin, March 1979.

{6} J.V. Guttag, The specification and application to programming of abstract data types, Technical Report CSRG-59, University of Toronto, September 1975.

{7} J.V. Guttag, E. Horowitz, and D.R. Musser, Abstract data types and software validation, Communications of the ACM, vol. 21, no. 12, Dec. 1978, 1048 - 1063.

{8} U. Lipeck, Zum Begriff der Implementierung in der Theorie der abstrakten Datentypen, Diploma Thesis, Abteilung Informatik, Universität Dortmund, March 1979 (German).

{9} B.H. Liskov and S.N. Zilles, Specification techniques for data abstractions, IEEE Transactions on Software Engineering, vol. SE-1, March 1975, 7 - 19.

{10} B.H. Mayoh, Data types as functions, Report DAIMI PB-89, Computer Science Department, Aarhus University, July 1978.

{11} D.L. Parnas, A technique for module specification with examples, Communications of the ACM, vol. 15, no. 5, May 1972, 330 - 336.