ABSTRACT IMPLEMENTATIONS AND CORRECTNESS PROOFS

Gilles BERNOT, Michel BIDOIT, Christine CHOPPY

Laboratoire de Recherche en Informatique
Bât 490, Université PARIS-SUD
F-91405 ORSAY CEDEX
FRANCE

ABSTRACT
In this paper, we present a new semantics for the implementation of abstract data types. This semantics leads to a simple, exhaustive description of the abstract implementation correctness criteria. These correctness criteria are expressed in terms of *sufficient completeness* and *hierarchical consistency*. Thus, correctness proofs of abstract implementations can always be handled using classical tools such as *theorem proving* methods, *structural induction* methods or *syntactical methods* (e.g. fair presentations). The main idea of our approach is the use of intermediate “concrete sorts”, which synthesize the available values used by implementation. Moreover, we show that the *composition* of several correct abstract implementations is always correct. This provides a formal foundation for a methodology of program development by stepwise refinement.

1. INTRODUCTION

For about ten years [LZ 75, Gut 75, ADJ 76], the formalism of abstract data types has been considered a major tool for writing hierarchical and modular specifications. Algebraic specifications provide the user with legible and relevant properties concerning the specified data structure. In particular, an abstract specification does not necessarily reflect the “concrete” implementation of the described data structure. But then, we have often to prove that the concrete implementation is *correct* according to our abstract specification. The following example shows the difference between “*abstract*” and “*concrete*” specifications.

Example 1
Let us specify the stacks of natural numbers. *STACK(NAT)* is specified as follows :

$$\begin{array}{rcl} pop(empty) & = & empty \\ pop(push(n,X)) & = & X \\ top(empty) & = & 0 \\ top(push(n,X)) & = & n \end{array}$$

But this data structure is often implemented by means of arrays. A stack is then characterized by an array, which contains the elements of the stack, and an integer, which is the height of the stack :

$$\begin{array}{rcl} empty & = & <t,0> \\ push(n,<t,i>) & = & <t[i]:=n,succ(i)> \\ pop(<t,0>) & = & <t,0> \\ pop(<t,succ(i)>) & = & <t,i> \\ top(<t,0>) & = & 0 \\ top(<t,succ(i)>) & = & t[i] \end{array}$$

The first element pushed onto the stack is then *t[0]* ; and the index *i* points to the place where the next element will be pushed.

Our problem is to prove that the second set of axioms *simulates* the data structure described by the first one. Correctness proofs of abstract implementations can be done by using the notions of *representation invariants* and *equality representation* [GHM 76, Gau 80]. For instance, the equality representation of Example 1 can be stated by :

<t,i> = <t',i'> iff i=i' and t[j]=t'[j] for all j=0..i

Unfortunately, this equality representation must be specified by the user, and nothing proves that it is correct. In particular, if we specify an equality representation where "everything is true", then every implementation will be correct. Since 1980, several works have formalized the notion of *simulation* [EKP 80, EKMP 80, SW 82] ; all these works give *pure semantical* correctness criteria (such as existence of a morphism between two algebras). Unfortunately, pure semantical correctness criteria do not provide the specifier with *theorem proving* methods or *structural induction* methods. It is therefore necessary to complete the abstract data type framework with an abstract implementation formalism which is able to provide the user with "simple" correctness proof criteria. These criteria are mainly *sufficient completeness* and *hierarchical consistency*.

In this paper, we present a new formalism of abstract implementation. This formalism leads in a natural way to an exhaustive description of the abstract implementation correctness criteria. These correctness criteria can be checked via classical methods since they are expressed in terms of sufficient completeness and hierarchical consistency. This approach is especially powerful, since it is then always possible to prove the correctness of an implementation via theorem proving methods. Moreover, we prove that our formalism is compatible with *enrichment* and that the *composition* of two correct implementations always gives a correct result. Our formalism allows use of *positive conditional axioms*. We will show that this feature imposes an explicit specification of the equality representation, but that it also facilitates the specification process. In particular our abstract implementation formalism can easily be extended to the algebraic data types with exception handling features [Ber 85].

The next section explains the classical problems related to abstract implementation. Section 3 describes the main ideas of our formalism which solve these problems. Sections 4 through 6 describe our abstract implementation formalism. In Section 7, we show how correctness proofs of abstract implementation can be handled. And finally, we prove that abstract implementations cope with *enrichment* (Section 8), and *composition* (Section 9). We assume that the reader is familiar with elementary results of category theory and abstract data type theory.

2. PROBLEMS RAISED BY ABSTRACT IMPLEMENTATION

Abstract implementations can be specified in two main ways : with an *abstraction* function, or with a *representation* function.

2.1. Abstraction

The abstraction takes already implemented objects (e.g. arrays and natural numbers), and returns "abstract" objects (e.g. stacks). This is done by means of an *abstraction operation* (e.g. *A: ARRAY NAT* → *STACK*). For instance, we obtain the axioms of the implementation of stacks by substituting *A(t,i)* for *<t,i>* in Example 1. Another example is the following :

Example 2

Natural numbers can be implemented by means of integers as follows :

$$\begin{aligned} 0_{\mathbf{N}} &= A(0_{\mathbf{Z}}) \\ succ_{\mathbf{N}}(A(z)) &= A(succ_{\mathbf{Z}}(z)) \\ eq?_{\mathbf{N}}(A(z), A(z')) &= eq?_{\mathbf{Z}}(z, z') \end{aligned}$$

where *A: INT* → *NAT* is the abstraction operation.

Unfortunately, abstraction operations create too many abstract objects. For instance, *A(create,4)* does not implement any stack, since if the height of a stack is equal to 4, then the four first ranges of the

corresponding array must be initialized. In the same way, *A(-1)* does not implement any natural number.
As shown in [EKMP 80], this fact prevents the specifier from carrying out simple correctness proofs by theorem proving methods. For instance, one of the proofs needed by implementation is the consistency of the implementation. This means that two distinct abstract objects must be implemented by two distinct concrete objects. The only formal concept of abstract data types which can handle such a condition is *hierarchical consistency*. Thus, it is necessary to put together the specification of our implementation (Example 2) and the abstract specification to be implemented (*NAT*). Then, we obtain a specification that contains both the abstract implementation and the specification to be implemented, and we can check whether this specification is hierarchically consistent over *NAT*. *NAT* is specified as follows :

$$\begin{array}{rcl} eq?_{\mathbf{N}}(0_{\mathbf{N}}, 0_{\mathbf{N}}) & = & True \\ eq?_{\mathbf{N}}(0_{\mathbf{N}}, succ_{\mathbf{N}}(m)) & = & False \\ eq?_{\mathbf{N}}(succ_{\mathbf{N}}(n), 0_{\mathbf{N}}) & = & False \\ eq?_{\mathbf{N}}(succ_{\mathbf{N}}(n), succ_{\mathbf{N}}(m)) & = & eq?_{\mathbf{N}}(n, m) \end{array}$$

But then, we obtain : $True = eq?_{\mathbf{N}}(0_{\mathbf{N}}, 0_{\mathbf{N}}) = eq?_{\mathbf{N}}(0_{\mathbf{N}}, succ_{\mathbf{N}}(A(-1))) = False$. Consequently, we cannot prove the consistency of our implementation this way.

2.2. Representation

The aim of a representation is to provide a composition of already implemented operations (e.g. those of *NAT* and *ARRAY*) for every operation to be implemented (e.g. *empty, push, pop, top*). For instance, the representation associated with Example 1 is specified as follows :

$$\begin{array}{rcl} \rho(empty) & = & <t,0> \\ \rho(push(n, <t, i>)) & = & <t[i]:=n, succ(i)> \\ \rho(pop(<t, 0>)) & = & <t,0> \\ \rho(pop(<t, succ(i)>)) & = & <t,i> \\ \rho(top(<t, 0>)) & = & 0 \\ \rho(top(<t, succ(i)>)) & = & t[i] \end{array}$$

where ρ is the *representation* function.
Since representation only gives a representation for each operation to be implemented, it should not create undesirable abstract values. Unfortunately, it is very difficult to give an algebraic meaning to such axioms. This is due to the fact that "<_ ,_ >" has no real algebraic definition. If we consider <_ ,_ > as an operation, then its arity is necessarily : $<,>: ARRAY\ NAT \rightarrow STACK$ because it takes an array and a natural number, and returns a stack (as we apply *pop* to <*t,i*>). Consequently, the arity of <_ ,_ > is the same as the arity of the abstraction operation. Thus, the function ρ is useless (equal to the identity), since the operation $<,>$ can simply be used as an abstraction operation, which simplifies the specification of abstract implementation. Nevertheless, we will show how our formalism uses both ρ and A , by means of an intermediate "product sort".

2.3. Presentations and implementations

Assume that the *STACK* data structure is already implemented by means of *ARRAY* and *NAT*. The user of this data structure will probably specify a presentation over the *STACK* specification (presentations over *STACK* can be viewed as abstract programs). But the user should never have to know how the implementation is done. In other words, (s)he knows the abstract specification of *STACK*, but not the specification of the implementation. Thus, every proof concerning this enrichment is done w.r.t. the abstract specification of *STACK*, but not w.r.t. the abstract implementation. Nothing proves that the composition of our implementation and the new enrichment gives the expected results. A particular subproblem of this is the composition of several implementations. All correctness proofs of the second implementation are handled w.r.t. to the abstract specification of the first implemented data structure, but they are not done w.r.t. to the concrete specification of the first implementation. In our framework, an enrichment of an abstract implementation always gives the expected result. This feature was not provided for in any of the works previously put forward.

In order to achieve this goal, we need an *explicit* specification of the equality representation in the implementation : when we enrich the implementation of *STACK*, the associated presentation will probably contain some axioms of the form :

$$X = Y \quad \Rightarrow \quad \ldots$$

We may have : $X = empty$ and $Y = pop(push(x,empty))$. The implementations of X and Y are then *<create,0>* and *<create[0]:=x,0>* . If the designer of the implementation says nothing about "*when two distinct pairs implement the same stack*", our enrichment viewed through the implementation will not be correct, since several occurrences of these axioms are not taken into account. Thus, it is necessary to specify the *equality representation* in the implementation, in order to handle conditional axioms. We will show that equality representation is also a useful tool for correctness proofs.

3. PRESENTATION OF OUR FORMALISM

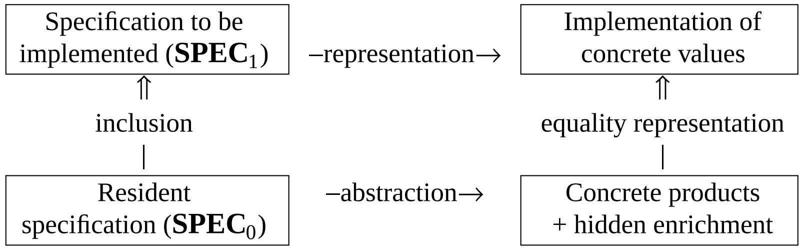
Our situation is described as follows :

- □ The already implemented data structure (e.g. *ARRAY* and *NAT*) is specified by $\mathbf{SPEC}_0 = <\mathbf{S}_0, \mathbf{\Sigma}_0, \mathbf{A}_0>$, where $\mathbf{S}_0$ is a set of sorts, $\mathbf{\Sigma}_0$ is a set of operations with arity in $\mathbf{S}_0$, and $\mathbf{A}_0$ is a set of *positive conditional* axioms over the signature $<\mathbf{S}_0, \mathbf{\Sigma}_0>$. $\mathbf{SPEC}_0$ is called the *resident* specification.
- □ We want to implement an enrichment (e.g. *STACK*) of the already implemented data structure. This enrichment is described by a specification $\mathbf{SPEC}_1 = <\mathbf{S}_1, \mathbf{\Sigma}_1, \mathbf{A}_1>$ which contains $\mathbf{SPEC}_0$, and is persistent over $\mathbf{SPEC}_0$. $\mathbf{SPEC}_1$ is the *abstract specification* of the data structure obtained after the implementation is done (*STACK+ARRAY+NAT*).

Our implementation will be made in five steps :

- □ The first step describes the representation. For each (abstract) sort of $\mathbf{SPEC}_1$ (e.g. *STACK*), there is a *concrete* sort which represents it ($\overline{STACK}$) ; $\overline{STACK}$ will be the product sort "Array×Natural". For each (abstract) operation of $\mathbf{SPEC}_1$ (e.g. *empty*, *push*, *pop*, *top*), there is a *concrete* operation which is its *actual implementation* ($\overline{empty}$, $\overline{push}$, $\overline{pop}$, $\overline{top}$). These concrete operations work on the concrete sorts (e.g. $\overline{STACK}$) instead of working on the abstract sorts to be implemented (*STACK*).
- □ The second step synthesizes the *concrete values* used by implementation. These concrete values are synthesized by means of abstraction operations. For instance, $\underline{A_{STACK}}: ARRAY\ \ NAT \rightarrow \overline{STACK}$ is the abstraction operation that synthesizes the product sort $\overline{STACK}$ (*ARRAY*×*NAT*), associated with $STACK \in \mathbf{S}_1$.
- □ The third step is only a convenient (hidden) enrichment of the previously synthesized data structure. This *hidden component* of the implementation was first introduced in [EKP 80]. It allows us to add hidden operations which are useful to specify the implementation. For instance, if the resident specification of integers (Example 2) does not contain the operation $eq?_{\mathbf{Z}}$, then it is very useful to define it in the hidden component before specifying the main part of the implementation.
- □ The fourth step recursively specifies the actual implementation of the concrete operations, on the concrete sorts. This step is handled by means of (conditional) axioms, as in previous examples.
- □ The last step specifies the equality representation. It will be specified by means of a set of (conditional) axioms. Thus, our last step specifies the implementation of the *classes* (or equivalently *values*) to be implemented.

This approach can be pictured as follows :



Our abstract implementation is described on three different levels :

□ the *formal definition* only contains the information which the specifier must provide in order to define the implementation

□ the *associated syntax* is automatically deduced from the formal definition ; it gives an algebraic specification for the implementation

□ the *associated semantics* is automatically deduced from the syntax ; it describes the models (algebras) of the implementation.

The distinction between these three levels was first introduced by [EKP 80]. This distinction has been shown to be a firm basis to handle correctness proofs for implementations.

4. FORMAL DEFINITION

Definition 1

We define an *abstract implementation*, denoted by **IMPL**, as a tuple :

$$\mathbf{IMPL} = <\rho, \mathbf{\Sigma}_{ABS}, \mathbf{H}, \mathbf{A}_{OP}, \mathbf{A}_{EQ}>$$

where :

□ ρ is the signature isomorphism defined as follows :

• for each abstract sort to be implemented, $s \in \mathbf{S}_1$, there is an associated “concrete sort”, $\bar{s}$. We denote the set of concrete sorts by $\mathbf{S}_{ABS}$ (since it will be synthesized by the abstraction operations [*]). Thus, $\mathbf{S}_{ABS}$ is a copy of $\mathbf{S}_1$.

• for each operation to be implemented ($\in \mathbf{\Sigma}_1$), op: $s_1 \cdots s_n \rightarrow s_{n+1}$, there is a “concrete operation”, $\overline{op}$: $\overline{s_1} \cdots \overline{s_n} \rightarrow \overline{s_{n+1}}$, where $\overline{s_i}$ is the concrete sort associated with s_i. We denote the set of concrete operations by $\mathbf{\Sigma}_{OP}$.

ρ is the signature isomorphism from $<\mathbf{S}_1, \mathbf{\Sigma}_1>$ to $<\mathbf{S}_{ABS}, \mathbf{\Sigma}_{OP}>$. ρ is called *representation signature isomorphism*, or simply *representation*, since it gives the actual representation of each sort (resp. operation) to be implemented. For instance, ρ sends the sort *NAT* to $\overline{NAT}$, *STACK* to $\overline{STACK}$, *push: NAT STACK* → *STACK* to $\overline{push}$: $\overline{NAT}$ $\overline{STACK} \rightarrow \overline{STACK}$, and so on.

□ $\mathbf{\Sigma}_{ABS}$ is the set of *abstraction operations* : for each sort to be implemented, $s \in \mathbf{S}_1$, there is one abstraction operation, A_s: $r_1 \cdots r_m \rightarrow \bar{s}$, where all the r_i are sorts in $\mathbf{S}_0$. For instance, the abstraction operation associated with the sort *STACK* is : A_{STACK}: $ARRAY\ NAT \rightarrow \overline{STACK}$; the abstraction operation associated with *NAT* is a copy operation : A_{NAT}: $NAT \rightarrow \overline{NAT}$.

□ **H** is the *hidden component* of **IMPL**. $\mathbf{H} = <\mathbf{S}_H, \mathbf{\Sigma}_H, \mathbf{A}_H>$ is a presentation over $\mathbf{ABS} = \mathbf{SPEC}_0 + <\mathbf{S}_{ABS}, \mathbf{\Sigma}_{ABS}, \varnothing>$, which enriches the concrete data structure in order to facilitate the implementation. In our *STACK* by *ARRAY* example, **H** is empty.

□ $\mathbf{A}_{OP}$ is a set of positive conditional axioms over the signature $<\mathbf{S}_0 + \mathbf{S}_H + \mathbf{S}_{ABS}, \mathbf{\Sigma}_0 + \mathbf{\Sigma}_H + \mathbf{\Sigma}_{ABS} + \mathbf{\Sigma}_{OP}>$. It describes the actual implementation of the concrete operations $\overline{op}$. $\mathbf{A}_{OP}$ is the set of operation implementing axioms. These axioms are those specified for abstraction :

[*] in our formalism, *abstraction* functions return *concrete* values (!).

$$\begin{aligned}
\overline{empty} &= A_{STACK}(t,0)\\
\overline{push}(A_{NAT}(n), A_{STACK}(t,i)) &= A_{STACK}(t[i]{:=}\,n, succ(i))\\
\overline{pop}(A_{STACK}(t,0)) &= A_{STACK}(t,0)\\
\overline{pop}(A_{STACK}(t,succ(i))) &= A_{STACK}(t,i)\\
\overline{top}(A_{STACK}(t,0)) &= A_{NAT}(0)\\
\overline{top}(A_{STACK}(t,succ(i))) &= A_{NAT}(t[i])
\end{aligned}$$

□ $\mathbf{A}_{EQ}$ is a set of positive conditional axioms over the same signature. It defines the *equality representation*. For instance, the equality representation of our *STACK* by *ARRAY* example can be specified as follows [*] :

$$A_{STACK}(t,0) = A_{STACK}(t',0)$$
$$A_{STACK}(t,i) = A_{STACK}(t',i) \wedge t[i] = t'[i] \implies A_{STACK}(t, succi) = A_{STACK}(t', succi)$$

5. ASSOCIATED SYNTAX

The syntax associated with the formal definition of an abstract implementation is defined as follows :

EQ : $\mathbf{A}_{EQ}$
REP : $\mathbf{S}_1 - \mathbf{S}_0$, $\mathbf{\Sigma}_1 - \mathbf{\Sigma}_0$, $\mathbf{\Sigma}_{REP}$, $\mathbf{A}_{REP}$
OPimpl : $\mathbf{\Sigma}_{OP}$, $\mathbf{A}_{OP}$
H : $\mathbf{S}_{\mathrm{H}}$, $\mathbf{\Sigma}_{\mathrm{H}}$, $\mathbf{A}_{\mathrm{H}}$
ABS : $\mathbf{S}_{ABS}$, $\mathbf{\Sigma}_{ABS}$
$\mathbf{SPEC}_0$: $\mathbf{S}_0$, $\mathbf{\Sigma}_0$, $\mathbf{A}_0$

where **ABS** is a presentation over $\mathbf{SPEC}_0$, **H** is a presentation over $\mathbf{SPEC}_0$+**ABS**, and so on.

□ **ABS** is the abstraction component of the syntax. It describes the synthesis of the concrete sorts $\bar{s}$, by means of the abstraction operation arities ($A_s\colon r_1 \cdots r_n \to \bar{s}$).

□ **H** is the hidden component of the syntactical level. **H** is a presentation over the concrete specification $\mathbf{SPEC}_0$+**ABS**.

□ **OPimpl** is the operation implementing part of the syntax. It specifies the actual implementation of the concrete operations ($\overline{op} \in \mathbf{\Sigma}_{OP}$) working on the concrete sorts, by means of $\mathbf{A}_{OP}$.

□ **REP** is the representation component. It *explicitly* specifies (in the syntax) the effect of the representation signature isomorphism. We define $\mathbf{\Sigma}_{REP}$ and $\mathbf{A}_{REP}$ below.

□ **EQ** is the equality representation part of the syntax. It specifies when two distinct availables values (concrete values) represent the same abstract value.

H, $\mathbf{S}_{ABS}$, $\mathbf{\Sigma}_{ABS}$, $\mathbf{\Sigma}_{OP}$, $\mathbf{A}_{OP}$ and $\mathbf{A}_{EQ}$ are defined in Section 4. $\mathbf{\Sigma}_{REP}$ and $\mathbf{A}_{REP}$ are defined as follows :

□ $\mathbf{\Sigma}_{REP}$ is the set of *representation operations*. For each abstract sort, $s \in \mathbf{S}_1$, there is one representation operation : $\overline{\rho_s}\colon s \to \bar{s}$.

□ $\mathbf{A}_{REP}$ is the set of axioms which state that $\overline{\rho_s}$ extends the representation signature isomorphism ρ. This means that for all $\mathbf{\Sigma}_1$-terms, t, of sort s, $\overline{\rho_s}(t)$ is equal to the term deduced from t via ρ. Thus, for each operation to be implemented, $op \in \mathbf{\Sigma}_1$, $\mathbf{A}_{REP}$ contains the following axiom :

$$\overline{\rho_s}(op(x_1,\ldots,x_n)) = \rho(op)(\overline{\rho_{s_i}}(x_i),\ldots,\overline{\rho_{s_n}}(x_n)) \quad [*]$$

where s is the target sort of op, and s_i is the sort of x_i .

Moreover, $\mathbf{A}_{REP}$ contains the following axiom for each abstract sort, $s \in \mathbf{S}_1$:

$$\overline{\rho_s}(x) = \overline{\rho_s}(y) \implies x = y\ .$$

This axiom is explained as follows : our goal is to specify the data structure obtained after the

[*] In fact, $\mathbf{A}_{EQ}$ can be empty in this example, since $\mathbf{A}_{OP}$ already implies our two axioms. But this is particular to our example.

[*] $\rho(op)$ is equal to $\overline{op}$.

implementation is done. If two terms to be implemented, x and y, are represented by the same concrete values, then it is impossible to distinguish x from y. Thus their values are equal in the resulting data structure.

Example 3
In the *STACK* by *ARRAY* example, $\mathbf{A}_{REP}$ is deduced from the signature isomorphism ρ as follows :

$$
\begin{array}{rcl}
\overline{\rho_{STACK}}(empty) & = & \overline{empty} \\
\overline{\rho_{STACK}}(push(x, X)) & = & \overline{push}(\overline{\rho_{NAT}}(x), \overline{\rho_{STACK}}(X)) \\
\overline{\rho_{STACK}}(pop(X)) & = & \overline{pop}(\overline{\rho_{STACK}}(X)) \\
 & \cdots etc \cdots & \\
\overline{\rho_{STACK}}(X) = \overline{\rho_{STACK}}(Y) & \Rightarrow & X = Y \\
\overline{\rho_{NAT}}(m) = \overline{\rho_{NAT}}(n) & \Rightarrow & m = n \\
\overline{\rho_{ARRAY}}(t) = \overline{\rho_{ARRAY}}(t') & \Rightarrow & t = t'
\end{array}
$$

6. ASSOCIATED SEMANTICS

The semantics of our abstract implementation is the composition of two functors :

$$
\begin{array}{ccccc}
\mathrm{Alg}(\mathbf{SPEC}_0) & -F_{\mathbf{ABS+H+OPimpl+REP+EQ}}\rightarrow & \mathrm{Alg}(\mathbf{EQ}) & -U_{<\mathbf{S}_1,\mathbf{\Sigma}_1>}\rightarrow & \mathrm{Alg}(<\mathbf{S}_1, \mathbf{\Sigma}_1>) \\
T_{\mathbf{SPEC}_0} & \mapsto F_{\mathbf{ABS+H+OPimpl+REP+EQ}}\rightarrow & T_{\mathbf{EQ}} & \mapsto U_{<\mathbf{S}_1,\mathbf{\Sigma}_1>}\rightarrow & SEM_{\mathrm{IMPL}}
\end{array}
$$

where $F_{\mathbf{ABS+H+OPimpl+REP+EQ}}$ is the usual synthesis functor associated with the presentation **ABS+H+OPimpl+REP+EQ** over $\mathbf{SPEC}_0$; and $U_{<\mathbf{S}_1,\mathbf{\Sigma}_1>}$ is the usual forgetful functor.

More precisely, the intuitive meaning of this semantics can be divided as follows :

- □ $T_{\mathbf{SPEC}_0}$ describes the (abstract) resident data structure.
- □ $T_{\mathbf{ABS}}$ describes the concrete data structure synthesized from the resident one by means of the abstraction operations. $T_{\mathbf{ABS}}$ is the *available* structure which our abstract implementation can use.
- □ $T_{\mathbf{H}}$ describes the hidden enrichment of the concrete data structure and the resident abstract data structure.
- □ $T_{\mathbf{OPimpl}}$ handles the concrete implementation of the concrete operations ($\overline{op}$) over the previously synthesized concrete sorts.
- □ $T_{\mathbf{REP}}$ is the implementation of the abstract ground terms to be implemented. It contains both the abstract operations (*op*), and their concrete implementation ($\overline{op}$). The correspondance between *op* and $\overline{op}$ is made via the representation operations $\overline{\rho_s}$.
- □ $T_{\mathbf{EQ}}$ handles the *identification* of the concrete terms which represent the same abstract value.
- □ Notice that $T_{\mathbf{EQ}}$ contains all the sorts and operations used in our implementation. Thus, it is necessary to remove the hidden sorts and operations, the intermediate concrete sorts, the abstraction operations, and the concrete operations $\overline{op}$. This is done by means of a forgetful functor, and the *semantical result* is a $\mathbf{\Sigma}_1$-algebra, denoted by SEM_{IMPL}. Thus, SEM_{IMPL} is the “user view” of the implementation, since the user must not use the specific operations and sorts of the implementation.

7. CORRECTNESS PROOFS

The above semantics leads, in a natural way, to define abstract implementation correctness as follows : an abstract implementation is *correct* iff each operation to be implemented has a (complete) concrete representation, and the semantical result (SEM_{IMPL}) is isomorphic to the initial algebra to be implemented ($T_{\mathbf{SPEC}_1}$). These criteria are handled in four steps. The complete implementation of all operations to be implemented is called *operation-completeness*. The isomorphism between SEM_{IMPL} and $T_{\mathbf{SPEC}_1}$ is divided into three conditions. SEM_{IMPL} must be finitely generated over $\mathbf{\Sigma}_1$; this condition is the *data protection*. SEM_{IMPL} must be a $\mathbf{SPEC}_1$-algebra ; this condition is the *validity* of **IMPL**. SEM_{IMPL} must be an initial

$\mathbf{SPEC}_1$-algebra ; this condition is the *consistency* of **IMPL**.

7.1. Operation completeness

Operation completeness was first introduced by [EKP 80]. The fact that all abstract operations have a concrete implementation means that all $\mathbf{\Sigma}_1$-terms have an "available" representation. Thus, operation completeness is defined as follows :

Definition 2
IMPL is *op-complete* iff for all terms $t \in T_{\mathbf{\Sigma}_1}$, there is $\alpha \in T_{\mathbf{ABS}}$ such that $\overline{\rho_s}(t) = \alpha$ in $T_{\mathbf{REP}}$
Notice that op-completeness must be tested without any consideration of the equality representation. Thus, it is defined in $T_{\mathbf{REP}}$ and not in $T_{\mathbf{EQ}}$.

Op-completeness can be directly proved by structural induction. Moreover, we have the following theorem :

Theorem 1
If **OPimpl** is sufficiently complete over **ABS**, then **IMPL** is op-complete.

Proof : Since **REP** is always sufficiently complete over **OPimpl** (fair presentation, [Bid 82]), (**REP**+**OPimpl**) is also sufficiently complete over **ABS**. But the sufficient completeness of **REP** over **ABS** means that for each $(\mathbf{\Sigma}_1 + \mathbf{\Sigma}_{\mathrm{H}} + \mathbf{\Sigma}_{ABS} + \mathbf{\Sigma}_{OP} + \mathbf{\Sigma}_{REP})$-term, r, whose sort belongs to $(\mathbf{S}_0 + \mathbf{S}_{\mathrm{H}} + \mathbf{S}_{ABS})$, there is $\alpha \in T_{\mathbf{ABS}}$ such that $r = \alpha$ in $T_{\mathbf{REP}}$. In particular, this holds for all terms of the form $\overline{\rho_s}(t)$, as needed. □

Example 4
We prove that our implementation of *STACK* by *ARRAY* is op-complete, by structural induction.

- $\overline{\rho_{STACK}}(empty)$ is equal to $\overline{empty}$, which is equal to $\alpha = A_{STACK}(create, 0)$
- if x and X have concrete representations ($x = \alpha_1$ and $\overline{\rho_{STACK}}(X) = \alpha_2 = A_{STACK}(t, i)$), then $\overline{\rho_{STACK}}(push(x, X))$ do too :
  $$\overline{\rho_{STACK}}(push(x, X)) = \overline{push}(\alpha_1, A_{STACK}(t, i)) = A_{STACK}(t[i] := \alpha_1, succ(i)) .$$
- similar reasoning applies for *pop* and *top*.

7.2. Data protection

Theorem 2
If **H** is sufficiently complete over $\mathbf{SPEC}_0$, then SEM_{IMPL} is finitely generated over $\mathbf{\Sigma}_1$.

Proof : The syntax of our abstract implementation does not contain any operations with target sort in $\mathbf{S}_1 - \mathbf{S}_0$, except those of $\mathbf{\Sigma}_1$. Thus, SEM_{IMPL} is always finitely generated w.r.t. the sorts of $\mathbf{S}_1 - \mathbf{S}_0$. It suffices to prove that SEM_{IMPL} is finitely generated w.r.t. the sorts of $\mathbf{S}_0$. Consequently, Theorem 2 results from the fact that our abstract implementation syntax does not contain any operation with target sort in $\mathbf{S}_0$, except those of $\mathbf{\Sigma}_1$ and $\mathbf{\Sigma}_{\mathrm{H}}$. □

Definition 3
IMPL is *data protected* iff **H** is sufficiently complete over $\mathbf{SPEC}_0$. This means that the resident (abstract) sorts are protected through **IMPL** .

Data protection is then not difficult to prove, since it can be proved by structural induction or via syntactical tools (such as *fair presentations*, [Bid 82]). Our *STACK* by *ARRAY* example is clearly data protected, as **H** is empty.

7.3. Validity

Definition 4
IMPL is a *valid* abstract implementation iff for all $\mathbf{\Sigma}_1$-terms, *t* and *t'*, we have :

if $t=t'$ in $T_{\mathbf{SPEC}_1}$ then $t=t'$ in SEM_{IMPL}.

Theorem 3
If **IMPL** is data protected then the following conditions are equivalent :

- □ **IMPL** is a valid abstract implementation
- □ there is a $\mathbf{\Sigma}_1$-morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL}
- □ SEM_{IMPL} validates the axioms of $\mathbf{A}_1 = \mathbf{A}_0 + \mathbf{A}$
- □ SEM_{IMPL} validates the axioms of **A**
- □ $T_{\mathbf{EQ}}$ validates the axioms of **A**
- □ **ID** is hierarchically consistent over **EQ**

where **ID** is the presentation over **EQ** which contains the set of axioms **A**. Thus, **ID** contains all the specifications involved in our formalism (both the syntax of **IMPL** and $\mathbf{SPEC}_1$).

Proof : given in Appendix.

The main result is the equivalence between the validity of **IMPL** and the consistency of **ID** over **EQ**. This feature is entirely due to our intermediate product sorts and the equality representation explicitly specified via $\mathbf{A}_{EQ}$. This result facilitates the validity proofs, since then, they can always be handled by theorem proving methods.

Example 5
The validity of our abstract implementation of *STACK* is shown by proving that each *STACK*-axiom is a theorem of the syntax of **IMPL**. We prove here that *pop(push(x,X))* is equal to *X* in $T_{\mathbf{EQ}}$. Other axioms of *STACK* are proved in a straightforward manner, following the same method.
Since $\mathbf{A}_{REP}$ contains the axiom $\overline{\rho_{STACK}}(X) = \overline{\rho_{STACK}}(Y) \implies X = Y$, and since our implementation is op-complete, it suffices to show that $\overline{pop}(\overline{push}(x, A_{STACK}(t, i)))$ is equal to $A_{STACK}(t, i)$ in $T_{\mathbf{EQ}}$. From $\mathbf{A}_{OP}$, it results that $\overline{pop}(\overline{push}(x, A_{STACK}(t, i))) = A_{STACK}(t[i] := x, i)$. Moreover, from the equality representation ($\mathbf{A}_{EQ}$), it results that $A_{STACK}(t[i] := x, i) = A_{STACK}(t, i)$, which ends our proof.

7.4. Consistency

Definition 5
IMPL is *consistent* iff for all $\mathbf{\Sigma}_1$-terms, *t* and *t'*, we have :

if *t=t'* in SEM_{IMPL}, then *t=t'* in $T_{\mathbf{SPEC}_1}$.

Theorem 4
If **IMPL** is data protected and valid, then the following conditions are equivalent :

- □ for all *t* and *t'* in $T_{\mathbf{\Sigma}_1}$, if *t=t'* in $T_{\mathbf{EQ}}$ then *t=t'* in $T_{\mathbf{SPEC}_1}$
- □ **IMPL** is consistent
- □ the initial morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL} is a monomorphism
- □ SEM_{IMPL} is an initial $\mathbf{SPEC}_1$-algebra
- □ the initial morphism from $T_{\mathbf{SPEC}_1}$ to $U_{\mathbf{S}_1}(T_{\mathbf{ID}})$ is a monomorphism
- □ **ID** is hierarchically consistent over $\mathbf{SPEC}_1$

Proof : given in Appendix.

For the same reasons as Theorem 3, Theorem 4 facilitates the consistency proofs, since they can always be handled by theorem proving methods.

Example 6
The only axioms that can destroy the consistency of **ID** over $\mathbf{SPEC}_1$ are the axioms whose sort is in $\mathbf{S}_1$. These axioms are :

$$\overline{\rho_{STACK}}(X) = \overline{\rho_{STACK}}(Y) \implies X = Y$$
$$\overline{\rho_{NAT}}(m) = \overline{\rho_{NAT}}(n) \implies m = n$$
$$\overline{\rho_{ARRAY}}(t) = \overline{\rho_{ARRAY}}(t') \implies t = t'$$

These axioms lead to show that two abstract terms represented by the same concrete value (in $T_{\mathbf{EQ}}$), are equal. Thus, we must consider each axiom of $\mathbf{A}_{OP} \cup \mathbf{A}_{REP} \cup \mathbf{A}_{EQ}$, and prove that it does not create inconsistencies. Let us consider, for instance, the axiom

$$\overline{push}(A_{NAT}(x), A_{STACK}(t, i)) = A_{STACK}(t[i] := x, succ(i)) .$$

Since we work in the stack *values* (not in the stack ground terms), we can handle our proofs w.r.t. the normal forms of *STACK*. It is possible to prove, by structural induction, that $A_{\overline{STACK}}(t, i)$ represents the stack $push(t[i\text{-}1], push(\ldots, push(t[0], empty)\ldots))$. Then, our proof is clear, as $\overline{push}(x, \overline{\rho_{STACK}}(X))$ represents *push(x,X)*. Other axioms are handled in a similar manner, by using the normal forms.

Definition 6
IMPL is *correct* iff it is both op-complete, data protected, valid and consistent.

8. ABSTRACT IMPLEMENTATIONS AND ENRICHMENTS

Let $\mathbf{SPEC}_1$ be a specification implemented via **IMPL**. Let **P** be a presentation over $\mathbf{SPEC}_1$. We have shown (Section 2.3) that every proof concerning **P** is done w.r.t. $\mathbf{SPEC}_1$, but not w.r.t. the syntax of **IMPL**. The “concrete” implementation of **P**+$\mathbf{SPEC}_1$ is not specified by **P**+$\mathbf{SPEC}_1$. It is specified by **P**+**EQ**, where **EQ** is the whole syntax of the implementation of $\mathbf{SPEC}_1$. The following theorem proves that the user view of the concrete specification **P**+**EQ** is isomorphic to the data structure specified by **P**+$\mathbf{SPEC}_1$.

Theorem 5
If **IMPL** is a correct abstract implementation of $\mathbf{SPEC}_1$, then for all persistent presentations, **P**, over $\mathbf{SPEC}_1$, we have :

$$U_{<\Sigma_1+\Sigma_P>}(T_{\mathbf{EQ+P}}) \; = \; T_{\mathbf{SPEC}_1\mathbf{+P}}$$

Proof : given in Appendix.

This theorem proves that the presentation **P**, pushed together with the abstract implementation of $\mathbf{SPEC}_1$, always provides the user with the expected results.

9. COMPOSITION OF ABSTRACT IMPLEMENTATIONS

When we implement $\mathbf{SPEC}_1$ by means of $\mathbf{SPEC}_0$, the resident specification $\mathbf{SPEC}_0$ is often already implemented by means of a lower level specification. But all our correctness proofs are done w.r.t. the specification $\mathbf{SPEC}_0$, not w.r.t. the specification of the implementation of $\mathbf{SPEC}_0$. We prove in this section that the composition of two correct implementations always gives correct results. This feature is not provided in any work already put forward. The formalism of [SW 82] provides correct “vertical compositions”, but these vertical compositions do not solve our problem : all upper level implementation operations must be implemented by the lower level implementation. This results in a large amount of operations to be implemented by the lowest level implementation ; moreover, this implies that all the lower level implementations must be redefined each time we add a new implementation. Such a composition is incompatible with modular, structured implementation.
The following theorem proves that the user view, obtained by pushing two correct abstract implementations together, is always correct.

Theorem 6
Let $\mathbf{IMPL}_2$ be an abstract implementation of $\mathbf{SPEC}_2$ by means of $\mathbf{SPEC}_1$. Let $\mathbf{IMPL}_1$ be an abstract implementation of $\mathbf{SPEC}_1$ by means of $\mathbf{SPEC}_0$. Consider the specification **IMPL**(1,2) obtained from the syntax of $\mathbf{IMPL}_2$ by substituting the syntax of $\mathbf{IMPL}_1$ for $\mathbf{SPEC}_1$.

$$\mathbf{IMPL}(1,2) \; = \; \mathbf{SPEC}_0 + (\mathbf{H}_1 + \mathbf{ABS}_1 + \ldots + \mathbf{EQ}_1) + (\mathbf{H}_2 + \mathbf{ABS}_2 + \ldots + \mathbf{EQ}_2)$$

If $\mathbf{IMPL}_1$ and $\mathbf{IMPL}_2$ are both correct, then we have :

$$U_{<\mathbf{S}_2,\Sigma_2>}(T_{\mathbf{IMPL}(1,2)}) \; = \; T_{\mathbf{SPEC}_2}$$

Proof : Since $\mathbf{IMPL}_2$ is correct, $(\mathbf{H}_2 + \ldots + \mathbf{EQ}_2)$ is persistent over $\mathbf{SPEC}_1$. Thus, Theorem 5 proves that $U_{\mathbf{SPEC}_1+..+\mathbf{EQ}_2}(T_{\mathbf{IMPL}(1,2)}) = T_{\mathbf{EQ}_2}$.
In particular, $U_{<\mathbf{S}_2,\Sigma_2>}(T_{\mathbf{IMPL}(1,2)}) = U_{<\mathbf{S}_2,\Sigma_2>}(T_{\mathbf{EQ}_2}) = SEM_{\mathrm{IMPL}_2}$. Moreover, the correctness of $\mathbf{IMPL}_2$ implies that $SEM_{\mathrm{IMPL}_2} = T_{\mathbf{SPEC}_2}$, which ends our proof. □

This theorem can easily be extended to every (finite) number of implementations. Thus, it is possible to handle structured and modular abstract implementations. This provides a formal foundation for a methodology of program development by stepwise refinement.

10. CONCLUSION

The abstract implementation formalism described in this paper relies on three main ideas :

- □ Abstract implementation is done by means of intermediate concrete values, which are distinct from the abstract values to be implemented. These concrete sorts are synthesized by means of *abstraction operations*.
- □ The correspondance between the abstract sorts or operations to be implemented and the concrete sorts or operations is specified by means of a *representation signature isomorphism*.
- □ The *equality representation* is explicitly introduced into the abstract implementation, in order to handle conditional axioms.

The main results of this abstract implementation formalism are the following :

- □ It allows use of *positive conditional axioms*, which facilitates the specifications and increases the class of models taken into account.
- □ All correctness proof criteria for abstract implementation are "simple" ones (sufficient completeness, hierarchical consistency or fair presentations). This feature provides the specifier with *theorem proving* methods, *structural induction* methods or *syntactical* criteria.
- □ Abstract implementations cope with the notion of *enrichment*.
- □ The *composition* of several correct implementations always gives correct results. Thus, abstract implementations can be specified in a modular and structured way.

As a last remark, we want to emphasize the fact that the semantics of our abstract implementation is a functorial one. Thus it is not difficult to include the notion of *parameterization* into our formalism, since parameterization mainly relies on synthesis functors and pushouts (see [ADJ 80]).

ACKNOWLEDGEMENTS

This work is partially supported by CNRS GRECO de Programmation, ESPRIT Project METEOR and FOR-ME-TOO.

11. APPENDIX

This appendix contains the technical proofs omitted in the body of the article. The results being proved are restated for convenience of reference.

11.1. Proof of Theorem 3

Theorem 3

If **IMPL** is data protected then the following conditions are equivalent :

1) **IMPL** is a valid abstract implementation
2) there is a $\mathbf{\Sigma}_1$-morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL}
3) SEM_{IMPL} validates the axioms of $\mathbf{A}_1 = \mathbf{A}_0 + \mathbf{A}$
4) SEM_{IMPL} validates the axioms of $\mathbf{A}$
5) $T_{\mathbf{EQ}}$ validates the axioms of $\mathbf{A}$
6) **ID** is hierarchically consistent over **EQ**

where **ID**=**EQ**+<**A**>.

Proof :

[1 ⇔ 2] is clear : since $T_{\mathbf{SPEC}_1}$ is finitely generated over $\mathbf{\Sigma}_1$, there is a morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL} if and only if two $\mathbf{\Sigma}_1$-terms equal in $T_{\mathbf{SPEC}_1}$ are also equal in SEM_{IMPL}.

[2 ⇔ 3] results from the facts that SEM_{IMPL} is finitely generated over $\mathbf{\Sigma}_1$ and that $T_{\mathbf{SPEC}_1}$ is initial in $\mathbf{SPEC}_1$. Thus, there is a morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL} if and only if SEM_{IMPL} is a **SPEC**-algebra (i.e. SEM_{IMPL} validates $\mathbf{A}_1$).

[3 ⇔ 4] results from the fact that **EQ** contains $\mathbf{A}_0$. Thus, SEM_{IMPL} always validates $\mathbf{A}_0$.

[4 ⇔ 5] results from the fact that the axioms of **A** only concern the signature $<\mathbf{S}_1, \mathbf{\Sigma}_1>$, and

$SEM_{\text{IMPL}} = U_{<\mathbf{S}_1,\boldsymbol{\Sigma}_1>}(T_{\mathbf{EQ}})$.
[5 ⇔ 6] results from the fact that **ID** does not add new operations to **EQ** (**EQ=ID–A**). Thus, **ID** is hierarchically consistent over **EQ** if and only if $T_{\mathbf{EQ}}$ already validates the axioms of **A**. □

11.2. Proof of Theorem 4

Theorem 4
If **IMPL** is data protected and valid, then the following conditions are equivalent :

1) for all t and t' in $T_{\boldsymbol{\Sigma}_1}$, if $t=t'$ in $T_{\mathbf{EQ}}$ then $t=t'$ in $T_{\mathbf{SPEC}_1}$
2) **IMPL** is consistent
3) the initial morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL} is a monomorphism
4) SEM_{IMPL} is an initial $\mathbf{SPEC}_1$-algebra
5) the initial morphism from $T_{\mathbf{SPEC}_1}$ to $U_{\mathbf{S}_1}(T_{\mathbf{ID}})$ is a monomorphism
6) **ID** is hierarchically consistent over $\mathbf{SPEC}_1$

Proof :
[1 ⇔ 2] results from the fact that SEM_{IMPL} is equal to the part of $T_{\mathbf{EQ}}$ concerning the signature $<\mathbf{S}_1, \boldsymbol{\Sigma}_1>$.
[2 ⇔ 3] results from the fact that $T_{\mathbf{SPEC}_1}$ is finitely generated over $\boldsymbol{\Sigma}_1$, and from Definition 5. Notice that the initial morphism $T_{\mathbf{SPEC}_1} \rightarrow SEM_{\text{IMPL}}$ exists, from Theorem 3.
[3 ⇔ 4] results from the fact that SEM_{IMPL} is finitely generated over $\boldsymbol{\Sigma}_1$.
[3 ⇔ 5] results from $SEM_{\text{IMPL}} = U_{\mathbf{S}_1}(T_{\mathbf{EQ}})$, and from $T_{\mathbf{EQ}} = T_{\mathbf{ID}}$ (Theorem 3).
[5 ⇔ 6] is clear since the initial morphism $T_{\mathbf{SPEC}_1} \rightarrow U_{\mathbf{S}_1}(T_{\mathbf{ID}})$ is the unit of adjunction associated with the presentation **ID** over $\mathbf{SPEC}_1$. □

11.3. Proof of Theorem 5

Theorem 5
If **IMPL** is a correct abstract implementation of $\mathbf{SPEC}_1$, then for all persistent presentations, **P**, over $\mathbf{SPEC}_1$, we have :

$$U_{<\boldsymbol{\Sigma}_1+\boldsymbol{\Sigma}_P>}(T_{\mathbf{EQ+P}}) \quad = \quad T_{\mathbf{SPEC}_1\mathbf{+P}}$$

Proof : We recall the following classical lemma :

Lemma If $\mathbf{P}_a$ and $\mathbf{P}_b$ are two *persistent* presentations over a specification **SP** such that $< \mathbf{S}_a, \boldsymbol{\Sigma}_a > \cap < \mathbf{S}_b, \boldsymbol{\Sigma}_b >$ is empty, then $\mathbf{P}_b$ is still a persistent presentation over ($\mathbf{P}_a$+**SP**).
(proved in [Ber 85] with positive conditional axioms)

The correctness of **IMPL** implies that **ID** is persistent over $\mathbf{SPEC}_1$. Thus, our lemma proves that $T_{\mathbf{SPEC}_1\mathbf{+P}}$ is isomorphic to $U_{<\boldsymbol{\Sigma}_1+\boldsymbol{\Sigma}_P>}(T_{\mathbf{ID+P}})$.
Moreover, since **P** is sufficiently complete over the $\mathbf{S}_1$ part of $T_{\mathbf{ID+P}}$, Theorem 3 proves that $T_{\mathbf{EQ+P}}$ is isomorphic to $T_{\mathbf{ID+P}}$. Consequently, $U_{<\boldsymbol{\Sigma}_1+\boldsymbol{\Sigma}_P>}(T_{\mathbf{EQ+P}})$ is isomorphic to $T_{\mathbf{SPEC}_1\mathbf{+P}}$, as needed. □

12. REFERENCES

[ADJ 76] Goguen J., Thatcher J., Wagner E. : “An initial algebra approach to the specification, correctness, and implementation of abstract data types”, Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978 (also IBM Report RC 6487, Oct. 1976).

[ADJ 80] Ehrig H., Kreowski H., Thatcher J., Wagner J., Wright J. : “Parameterized data types in algebraic specification langages”, Proc. 7th ICALP, July 1980.

[Ber 85] Bernot G. : “Une sémantique algébrique pour une spécification différenciée des exceptions et des erreurs : application à l’implémentation et aux primitives de structuration des spécifications formelles”, Thèse de troisième cycle, Université de Paris-Sud, Orsay, 1985.

[Bid 82] Bidoit M. : “Algebraic data types: structured specifications and fair presentations”, Proc. of AFCET Symposium on Mathematics for Computer Science, Paris, March 1982.

[EKMP 80] Ehrig H., Kreowski H., Mahr B., Padawitz P. : “Algebraic implementation of abstract data types”, Theoretical Computer Science, Oct. 1980.

[EKP 80] Ehrig H., Kreowski H., Padawitz P. : “Algebraic implementation of abstract data types: concept, syntax, semantics and correctness”, Proc. ICALP, Springer-Verlag LNCS 85, 1980.

[Gau 80] Gaudel M.C. : “Génération et preuve de compilateurs basée sur une sémantique formelle des langages de programmation”, Thèse d’état, Nancy, 1980.

[GHM 76] Guttag J.V., Horowitz E., Musser D.R. : “Abstract data types and software validation”, C.A.C.M., Vol 21, n.12, 1978. (also USG ISI Report 76-48).

[Gut 75] Guttag J.V. : “The specification and application to programming”, Ph.D. Thesis, University of Toronto, 1975.

[LZ 75] Liskov B., Zilles S. : “Specification techniques for data abstractions”, IEEE Transactions on Software Engineering, Vol.SE-1 N 1, March 1975.

[SW 82] Sanella D., Wirsing M. : “Implementation of parameterized specifications”, Report CSR-103-82, Department of Computer Science, University of Edinburgh.