CORRECTNESS PROOFS FOR ABSTRACT IMPLEMENTATIONS

To appear in Information and Computation, Vol. 80, Num. 2, pp. 121-151, Feb. 1989.

Gilles BERNOT
Laboratoire d'Informatique,
Ecole Normale Supérieure,
45 Rue d'Ulm,
F-75230 PARIS CEDEX 05,
FRANCE

bitnet: bernot@frulm63
uucp: bernot@ens.ens.fr

ABSTRACT

New syntax and semantics for implementation of abstract data types are presented in this paper. This formalism 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 be handled using classical tools such as *term rewriting* methods, *structural induction* methods or *syntactical methods* (e.g. fair presentations). The main idea of this approach is a fundamental distinction between *descriptive* and *constructive* specifications, using both abstraction and representation functions. 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.

Key-words: abstract data types, abstraction, correctness proofs, implementation, initial model, mathematical programming, representation, theorem proving.

1 INTRODUCTION

For about twelve 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 structures. Nevertheless, as algebraic specifications give a *description* of the data structure *properties*, they should not provide the designer with a *constructive* specification of the corresponding *implementation*. To implement a data structure, the descriptive specification is not directly used. Rather, "resident" data structures (which have been previously implemented) are used. For instance, we implement a *STACK* data structure by means of *ARRAY*. The following example shows the difference between "*descriptive*" and "*constructive*" specifications:

Example 1 :
Let us specify stacks of natural numbers, *STACK(NAT)*, as follows:

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

This specification is *descriptive*, as it describes the basic properties of stacks. But this data structure is more efficiently 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.
Without leaving off the abstract data type formalism, a *constructive* specification of the *implementation* of *STACK(NAT)* using *ARRAY* and *NAT* can be done as follows:

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

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 (see [BBC 86] for a more realistic treatment of the exceptional cases *pop(empty)* and *top(empty)*).
However, we have to prove that the second set of (constructive) axioms is *correct* with respect to the data structure described by the first one.

It is well known that this need of establishing the correctness of an implementation with respect to the "designer's intentions" induces very difficult problems. The use of formal specifications (in particular algebraic specifications) is particularly fruitful, both for proving [Hoa 72, ADJ 78, EKP 80, EKMP 82, SW 82, San 87, Sch 87, GM 88 ...] or for testing [Bou 82, BCFG 86, Gau 86 ...] the correctness of an implementation. A natural idea is to describe the implementation problem in a homogeneous abstract specification framework; this leads to the concept of *abstract implementation*. We then hope that the usual proof techniques of abstract data types would facilitate correctness proofs of abstract implementations.

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:

$$< t, i > \;=\; < t', i' > \qquad \text{iff} \qquad i = i' \;\text{ and }\; t[j] = t'[j] \;\text{ for all }\; j = 0..i$$

Unfortunately, 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 implementation correctness [EKP 80, EKMP 82, SW 82, San 87, Sch 87] without using an explicit equality representation. 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 (e.g. structural induction). It is therefore necessary to complete the abstract data type framework with an abstract implementation formalism which provides the user with "simple" correctness proof criteria.

In this paper, a new formalism for abstract implementations is provided. 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 by means of *sufficient completeness* and *hierarchical consistency*. These two concepts are well known in classical abstract data types; thus we show that the *correctness* of abstract implementations does not require new concepts in the abstract data type field. This approach is especially powerful, since it is then possible to prove the correctness of an implementation using term rewriting techniques, structural induction etc. Moreover, this formalism is compatible with *enrichment*, and the *composition* of two correct implementations always gives a correct result. This new definition of abstract implementation allows for the use of *positive conditional axioms*. We will show that this feature requires an equality representation, which in turn facilitates the correctness proofs. Moreover, the adequacy of the specified equality representation will be implied by the correctness of the implementation; thus the difficulties raised by the equality representation, which have been pointed out above, will be solved in this framework. Finally, the semantical level is very simple because it only uses the classical forgetful and synthesis functors. Thus, this formalism can easily be extended, for instance to algebraic data types with exception handling features [Ber 86].

The next section presents 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. Finally, we prove that abstract implementations cope with *enrichment* and *composition* (Section 8). 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 are usually specified either with an *abstraction* function (presented in [Hoa 72]), or with a *representation* function ([ADJ 78], section 5.4.2).

2.1 THE ABSTRACTION FUNCTION

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

Example 2 :
Natural numbers can be implemented by means of integers as follows:

$$0_{I\!N} = A(0_{Z\!\!Z})$$
$$succ_{I\!N}(A(z)) = A(succ_{Z\!\!Z}(z))$$
$$eq?_{I\!N}(A(z), A(z')) = eq?_{Z\!\!Z}(z, z')$$

where *A: INT* $\rightarrow$ *NAT* is the abstraction operation.

The abstraction viewpoint is generalized and formalized in [EKP 80, EKMP 82], and is also underlying in [SW 82, San 87].

Unfortunately, abstraction operations synthesize too many objects in the sorts to be implemented. For instance, *A(create,4)* does not implement any stack, because 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 82], this fact prevents the specifier from carrying out simple correctness proofs by theorem proving methods. For instance, one of the proofs required is the implementation consistency: two objects which are distinct with respect to the descriptive specification must be implemented by two objects (synthesized by abstraction operations) which are distinct with respect to the constructive specification. The only formal concept of abstract data types which can handle such a condition is the *hierarchical consistency*. Thus, it is necessary to put together the constructive specification of our implementation (Example 2) and the descriptive specification to be implemented (NAT). We obtain a specification that contains both the (constructive) abstract implementation and the descriptive specification to be implemented, and we can check whether this specification is hierarchically consistent over *NAT*. *NAT* is specified as follows:

$$eq?_{I\!N}(0_{I\!N}, 0_{I\!N}) = True$$
$$eq?_{I\!N}(0_{I\!N}, succ_{I\!N}(m)) = False$$
$$eq?_{I\!N}(succ_{I\!N}(n), 0_{I\!N}) = False$$
$$eq?_{I\!N}(succ_{I\!N}(n), succ_{I\!N}(m)) = eq?_{I\!N}(n, m)$$

But we obtain: $True = eq?_{I\!N}(0_{I\!N}, 0_{I\!N}) = eq?_{I\!N}(0_{I\!N}, succ_{I\!N}(A(-1))) = False$. Consequently, although it is clearly correct, we cannot prove the consistency of our implementation this way.

2.2 THE REPRESENTATION FUNCTION

The aim of a representation is to provide a composition of previously 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:

$$\rho(empty) =< t, 0 >$$
$$\rho(push(n, < t, i >)) =< t[i] := n, succ(i) >$$
$$\rho(pop(< t, 0 >)) =< t, 0 >$$
$$\rho(pop(< t, succ(i) >)) =< t, i >$$
$$\rho(top(< t, 0 >)) = 0$$
$$\rho(top(< t, succ(i) >)) = t[i]$$

where ρ is the *representation* function.
Since representation only gives a representation for each operation to be implemented, it does not create undesirable values in the sorts to be implemented. 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. Considering <_,_> as an operation, its signature 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 signature of <_,_> is the same as the signature of the abstraction operation. Thus, the function ρ is useless (in fact ρ is the identity), because the operation $< , >$ can simply be used as an abstraction operation, which simplifies the previous specification.

A second way of looking at the representation may be to consider two representation operations:

$$\begin{array}{rccc} \rho_1: & STACK & \rightarrow & ARRAY \\ \rho_2: & STACK & \rightarrow & NAT \end{array}$$

But $\rho_1(empty)$ must be specified as a particular array. If we specify that $\rho_1(empty)$ can be equal to any array ($\rho_1(empty) = t$ together with $\rho_2(empty) = 0$, as in the abstraction case), then all arrays will be collapsed, which results in inconsistencies. Thus, breaking the representation operation into several operations is not powerful enough.
In fact, we will develop an abstract implementation formalism which uses both ρ and <_,_> . The problems mentioned above are avoided by means of intermediate "constructive sorts."

2.3 REUSABILITY ISSUES

Let us assume that the *stack* data structure is already implemented by means of *arrays* and *natural numbers*. A user of this data structure will probably include it in some other programs. At the specification level, this means that some presentations over *STACK* will be specified (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 *descriptive* specification of *STACK*, but (s)he does not know the *constructive* specification of its implementation. Thus, every proof concerning an enrichment is done with respect to the descriptive specification of *STACK*, but not with respect to the implementation specification. This does not prove that the composition of the *STACK implementation* with the new enrichment gives the expected result.
A particular subproblem is the composition of several implementations (i.e. implementations which reuse other implementations). All correctness proofs of the second implementation are handled with respect to the descriptive specification of the first implemented data structure; they are not done with respect to the constructive specification of this first implementation. *A priori*, the composition of the first implementation and the second one is not proved to be correct, even if these two implementations are separately proved correct.
In our framework, enrichments and compositions of correct abstract implementations always give the expected (semantical) results. This feature was not provided in any of the previous works in this area.

In order to achieve this goal, a specification of the equality representation must be included into the implementation (at least as soon as we want to enrich this implementation by a presentation containing *conditional* axioms).
For example, a presentation over *STACK* can contain a conditional axiom of the form:

$$pop(X) = empty \Longrightarrow M = N$$

We may have: $X = push(n,empty)$. The implementations of the terms *empty* and *pop(push(n,empty))* are then $<create,0>$ and $<create[0]:=n,0>$. These pairs are not equal, but the premise of this axiom must be satisfied. If the implementation cannot detect *when two distinct pairs implement the same stack*, then our enrichment viewed through the implementation will not be correct, since some instances of this axiom are not taken into account. Thus, it is necessary to include the equality representation into the implementation in order to handle conditional axioms of enrichments. We will show that equality representation is also a useful tool for correctness proofs.

3 OVERVIEW OF OUR FORMALISM

We have shown that the abstraction (A) has the advantage of *synthesizing* products of previously implemented sorts. Therefore, the inconsistencies described with ρ_1 and ρ_2 in section 2.2 are avoided. But abstraction leads to complicated correctness proofs because it adds some undesirable values in the sorts to be implemented. A *restriction* is necessary before defining implementation correctness. On the other hand, the representation (ρ) solves this problem, because it only returns the implementation of each values to be implemented. Intuitively, the image of ρ is just the result of the restriction. Representation automatically handles restriction. But we must face the difficulty of giving an algebraic syntax for representation.
In fact, we will take advantage of both abstraction and representation by using intermediate *constructive sorts*. Indeed, the main idea of the abstract implementation formalism described here is a systematic distinction between *descriptive* and *constructive* specifications or models. A *descriptive* specification or model only results from the abstract description of the known or required properties of the data structure under consideration. A *constructive* specification or model results from information or choices about its implementation.

Let us state the problem as follows:

- The previously implemented data structure (e.g. NAT and $ARRAY$) is specified by $\mathbf{SPEC_0} = (\mathbf{S_0}, \Sigma_\mathbf{0}, \mathbf{A_0})$, where $\mathbf{S_0}$ is a set of sorts, $\Sigma_\mathbf{0}$ 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 $(\mathbf{S_0}, \Sigma_\mathbf{0})$. $\mathbf{SPEC_0}$ is called the *resident* specification.
  Of course, $\mathbf{SPEC_0}$ is a *descriptive specification*. $\mathbf{SPEC_0}$ does not explain how resident sorts are implemented; it only describes "what properties we know" about the resident values. In particular the initial algebra $T_{\mathbf{SPEC_0}}$ is a "descriptive model" of resident values; $T_{\mathbf{SPEC_0}}$ does not necessarily reflect the constructive (previously completed) implementation of resident sorts.
- We want to implement a data structure described by $\mathbf{SPEC_1} = (\mathbf{S_1}, \Sigma_\mathbf{1}, \mathbf{A_1})$.
  $\mathbf{SPEC_1}$ is only a *descriptive specification* of "what properties we want to obtain" after the implementation is performed (e.g. NAT+$STACK$). In particular, the $\mathbf{SPEC_1}$-initial algebra $T_{\mathbf{SPEC_1}}$ is only a "reference model" (for correctness) which does not necessarily reflect the actual implementation semantics. $T_{\mathbf{SPEC_1}}$ is a *descriptive model* of the expected implementation result.
- Notice that $\mathbf{SPEC_0}$ and $\mathbf{SPEC_1}$ are not necessarily disjoint. For example, NAT is a specification included both in $\mathbf{SPEC_0}$=NAT+$ARRAY$ and in $\mathbf{SPEC_1}$=NAT+$STACK$. In the following, we assume that $\mathbf{SPEC_0}$ and $\mathbf{SPEC_1}$ are both *persistent* (i.e. hierarchically consistent and sufficiently complete) over the common specification $\mathbf{SP}$=($\mathbf{S}$,Σ,$\mathbf{A}$)=($\mathbf{S_0} \cap \mathbf{S_1}, \Sigma_\mathbf{0} \cap \Sigma_\mathbf{1}, \mathbf{A_0} \cap \mathbf{A_1}$).

The abstract implementation problem is to define a *constructive* specification of $\mathbf{SPEC_1}$ using $\mathbf{SPEC_0}$; and to provide the specifier with usable correctness criteria.

An *abstract implementation* will be performed in five steps, using intermediate *constructive sorts* containing *constructive values*:

- The first step describes the *representation*. For each (descriptive) sort of $\mathbf{SPEC_1}$ (e.g. $STACK$), there is a *constructive* sort which represents it ($\overline{STACK}$). Intuitively, $\overline{STACK}$ will be the product sort "Array×Natural." For each (descriptive) operation of $\mathbf{SPEC_1}$ (e.g. *empty*, *push*, *pop*, *top*), there is a *constructive* operation which is its *actual implementation* ($\overline{empty}$, $\overline{push}$, $\overline{pop}$, $\overline{top}$). These constructive operations work on the constructive sorts (e.g. $\overline{STACK}$) instead of directly working on the descriptive sorts to be implemented ($STACK$). Notice that there are also constructive operations and a constructive sort associated with NAT (as $NAT \subset \mathbf{SPEC_1}$). Since NAT has already been implemented ($\subset \mathbf{SPEC_0}$), $\overline{NAT}$ is simply a *copy* of NAT. Intuitively, this corresponds to the following fact: $\mathbf{SPEC_0}$ is a *descriptive* specification; we do not know the constructive (previously completed) implementation of NAT. Consequently, *by default*, we synthesize $\overline{NAT}$ as a copy of NAT.
- The second step synthesizes the *constructive values* used by the implementation. These constructive values are generated by means of *synthesis operations*. For example, the synthesis operation associated with $\overline{STACK}$ is the abstraction operation

$$< \_, \_ >_{STACK}\colon\ ARRAY\ NAT\ \rightarrow\ \overline{STACK}$$

  that synthesizes the product sort $\overline{STACK}$ ($ARRAY \times NAT$), associated with $STACK \in \mathbf{S_1}$. Moreover, the synthesis operation associated with $\overline{NAT}$ is simply (by default) a copy operation $< \_ >_{NAT}\colon\ NAT\ \rightarrow\ \overline{NAT}$.
- 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 $eq?_{\mathbb{Z}}$, then it is very useful to define it in the hidden component before specifying the main part of the implementation.
- The fourth step is the usual constructive specification of the implementation. It recursively specifies the actual implementation of each new constructive operation ($\overline{empty}$, $\overline{push}$,...) on the constructive sorts ($\overline{STACK}$). 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. Our last step specifies the implementation of the *classes* (or equivalently *values*) to be implemented; while the fourth step only specifies the implementation of *terms* to be implemented.

This new fundamental distinction between descriptive and constructive aspects will be reflected on three different levels: the *textual* level, the *presentation* level and the *semantical* level.

- the *textual level* (Section 4) only contains the informations that the specifier must provide in order to define the implementation
- the *presentation level* (Section 5) is automatically deduced from the textual level; it gives a complete algebraic specification for the implementation (which will be useful for correctness proofs)

- the *semantical level* (Section 6) is automatically deduced from the presentation level; it describes the models (algebras) of the implementation.

Similar levels have been first introduced by [EKP 80]. They have been shown to be a firm basis to define correctness for abstract implementations.

4 THE TEXTUAL LEVEL

Definition 1 *(Textual level)* **:**
We define an *abstract implementation of* $\mathbf{SPEC_1}$ *by* $\mathbf{SPEC_0}$, denoted by **IMPL**, as a tuple:

$$\mathbf{IMPL} = (\ \rho\ ,\ \Sigma_{\mathbf{SYNTH}}\ ,\ \mathbf{H}\ ,\ \mathbf{A_{OP}}\ ,\ \mathbf{A_{EQ}}\ )$$

where ρ is the *representation*, $\Sigma_{\mathbf{SYNTH}}$ is the set of *synthesis operations*, $\mathbf{H}$ is the *hidden component*, $\mathbf{A_{OP}}$ is the set of *constructive axioms*, and $\mathbf{A_{EQ}}$ is the *equality representation.*
These five parts are precisely defined in the following subsections.

4.1 THE REPRESENTATION

Definition 1.1 :
The *representation*, ρ, is the signature isomorphism defined as follows:

- for each descriptive sort to be implemented, $s \in \mathbf{S_1}$, there is an associated *constructive sort,* $\overline{s}$. We denote the set of constructive sorts by $\overline{\mathbf{S_1}}$ (actual constructive values of sort $\overline{s}$ will be generated by the synthesis operations). Thus, $\overline{\mathbf{S_1}}$ is a copy of $\mathbf{S_1}$. The constructive sort $\overline{s}$ implements s.
- for each operation to be implemented, $op: s_1 \cdots sn \rightarrow s_{n+1}\ (\in \Sigma_{\mathbf{1}})$, there is a *constructive operation,* $\overline{op}: \overline{s_1} \cdots \overline{s_n} \rightarrow \overline{s_{n+1}}$, where $\overline{s_i}$ is the constructive sort associated with s_i. We denote the set of constructive operations by $\overline{\Sigma_{\mathbf{1}}}$. The constructive operation $\overline{op}$ implements op.

ρ is the signature isomorphism from $(\mathbf{S_1}, \Sigma_{\mathbf{1}})$ to $(\overline{\mathbf{S_1}}, \overline{\Sigma_{\mathbf{1}}})$. ρ is called the *representation signature isomorphism*, or simply the *representation*, since it gives the constructive representation of each descriptive sort/operation to be implemented. For instance, ρ sends the sort NAT to $\overline{NAT}$, $STACK$ to $\overline{STACK}$, *push:* $NAT\ STACK \rightarrow STACK$ to $\overline{push}: \overline{NAT}\ \overline{STACK} \rightarrow \overline{STACK}$, and so on.

Remark 1 :
Notice that the representation ρ may seem useless. In practice, it is clear that we do not ask the specifier to explicitly characterize ρ . Nevertheless, on a theoretical viewpoint, it is necessary to precisely specify the correspondence between the descriptive signature to be implemented and its constructive implementation.

4.2 THE SYNTHESIS OPERATIONS

Definition 1.2 :
The set of *synthesis operations*, denoted by $\Sigma_{\mathbf{SYNTH}}$ is defined as follows: for each constructive

sort, $\overline{s} \in \overline{\mathbf{S_1}}$, there is a synthesis operation, $< ... >_s$: $r_1 \cdots r_m \rightarrow \overline{s}$, where all the ri are resident sorts in $\mathbf{S_0}$.

For instance, the synthesis operation associated with the sort $STACK$ is the "abstraction operation:" $< \_, \_ >_{STACK}$: $ARRAY\ NAT \rightarrow \overline{STACK}$; the synthesis operation associated with NAT is the "copy operation:" $< \_ >_{NAT}$: $NAT \rightarrow \overline{NAT}$.

Remark 2 :
The synthesis operation associated with each previously implemented sort of **SP** (e.g. NAT) will be a copy operation. Thus, in practice, we never ask the specifier to give the synthesis operation associated with these sorts. Nevertheless, this copy is useful, and necessary, when rigorously proving the correctness of an abstract implementation. Intuitively, the introduction of constructive sorts, together with the representation signature isomorphism, handles the *restriction* problem. This restriction must apply to all sorts to be implemented, including the sorts of **SP**. For example, *top(<create,4>)* could be a new value belonging to NAT, which must be removed before verifying the correctness of our $STACK$ implementation (as the pair *<create,4>* is not a reachable stack). With this abstract implementation formalism, $\overline{top}(< create, 4 >)$ will be of *constructive* sort $\overline{NAT}$, and thus, the *descriptive* sort NAT is preserved.

4.3 THE HIDDEN COMPONENT

Definition 1.3 :
The *hidden component* of **IMPL**, $\mathbf{H} = (\mathbf{S_H}, \Sigma_\mathbf{H}, \mathbf{A_H})$, is a presentation over $\mathbf{SORTimpl} = \mathbf{SPEC_0} + (\overline{\mathbf{S_1}}, \Sigma_{\mathbf{SYNTH}}, \emptyset)$ that enriches the synthesized data structures in order to facilitate the implementation.

In our $STACK$ by $ARRAY$ example, **H** is empty. An example of non empty hidden component is given in Example 3 (Section 4.5 below).

4.4 THE OPERATION-IMPLEMENTING AXIOMS

Definition 1.4 :
We denote by $\mathbf{A_{OP}}$ the set of constructive axioms of **IMPL**. $\mathbf{A_{OP}}$ is a set of positive conditional axioms over the signature $(\mathbf{S_0} + \mathbf{S_H} + \overline{\mathbf{S_1}}, \Sigma_\mathbf{0} + \Sigma_{\mathbf{SYNTH}} + \Sigma_\mathbf{H} + \overline{\Sigma_\mathbf{1}})$. It specifies the actual implementation of the constructive operations $\overline{op}$. $\mathbf{A_{OP}}$ is the set of *operation-implementing* axioms.

The axioms of $\mathbf{A_{OP}}$ are those specified for abstraction:

$$\overline{empty} =< t, 0 >_{STACK}$$
$$\overline{push}(< n >_{NAT}, < t, i >_{STACK}) =< t[i] := n, succ(i) >_{STACK}$$
$$\overline{pop}(< t, 0 >_{STACK}) =< t, 0 >_{STACK}$$
$$\overline{pop}(< t, succ(i) >_{STACK}) =< t, i >_{STACK}$$
$$\overline{top}(< t, 0 >_{STACK}) =< 0 >_{NAT}$$
$$\overline{top}(< t, succ(i) >_{STACK}) =< t[i] >_{NAT}$$

Of course, these axioms can always be automatically deduced from those of Example 1 (i.e. from axioms "without overlines").

4.5 THE EQUALITY REPRESENTATION

Definition 1.5 :
The *equality representation*, denoted by $\mathbf{A_{EQ}}$, is a set of positive conditional axioms which can use all the sorts and operations previously mentioned: $(\mathbf{S_0}+\mathbf{S_H}+\overline{\mathbf{S_1}}+\mathbf{S_1}, \Sigma_\mathbf{0}+\Sigma_\mathbf{H}+\Sigma_\mathbf{SYNTH}+\overline{\Sigma_\mathbf{1}}+\Sigma_\mathbf{1})$.

For instance, the equality representation of our *STACK* by *ARRAY* example can be specified as follows:

$$\begin{array}{c} < t, 0 >_{STACK} \quad = \quad < t', 0 >_{STACK} \\ < t, i >_{STACK} = < t', i >_{STACK} \ \textit{and}\ t[i] = t'[i] \implies < t, succ(i) >_{STACK} \quad = \quad < \\ t', succ(i) >_{STACK} \end{array}$$

(In fact, $\mathbf{A_{EQ}}$ can be empty in this example, since $\mathbf{A_{OP}}$ already implies our two axioms; but this is specific to the *STACK* example).

Let us specify another standard example: the implementation of *SET* by *STRING* (of natural numbers, for instance).

Example 3 *(textual implementation of SET by STRING)* **:**

The representation signature isomorphism ρ sends the descriptive sorts *BOOL*, *NAT* and *SET* to the constructive sorts $\overline{BOOL}$, $\overline{NAT}$ and $\overline{SET}$ respectively; and sends $\emptyset$ to $\overline{\emptyset}$, *ins* to $\overline{ins}$, $\in$ to $\overline{\in}$, as well as *True* to $\overline{True}$, *False* to $\overline{False}$, and so on.
The synthesis operations of $\Sigma_\mathbf{SYNTH}$ are:

$$\begin{array}{c} < \_ >_{SET} \ : \ STRING \ \rightarrow \ \overline{SET} \text{ “true synthesis operation”} \\ < \_ >_{NAT} \ : \ NAT \ \rightarrow \ \overline{NAT} \text{ “copy operation”} \\ < \_ >_{BOOL} \ : \ BOOL \ \rightarrow \ \overline{BOOL} \text{ “copy operation”} \end{array}$$

If *STRING* does not contain the operations *remove* and *occurs*, then **H** may specify them as hidden operations:

$$\begin{array}{c} remove(x, \lambda) \quad = \quad \lambda \\ remove(x, add(x, s)) \quad = \quad s \\ eq?(x, y) = False \quad \implies \quad remove(x, add(y, s)) \quad = \quad add(y, remove(x, s)) \\ occurs(x, \lambda) \quad = \quad False \\ occurs(x, add(x, s)) \quad = \quad True \\ eq?(x, y) = False \quad \implies \quad occurs(x, add(y, s)) \quad = \quad occurs(x, s) \end{array}$$

From the second axiom, *remove(x,s)* only removes the first occurrence of x in s. This does not matter because sets will only be represented by non-redundant strings.
The constructive axioms of $\mathbf{A_{OP}}$ are:

$$\begin{array}{c} \overline{\emptyset} = < \lambda >_{SET} \\ occurs(x, s) = True \implies \overline{ins}(< x >_{NAT}, < s >_{SET}) = < s >_{SET} \\ occurs(x, s) = False \implies \overline{ins}(< x >_{NAT}, < s >_{SET}) = < add(x, s) >_{SET} \\ \overline{del}(< x >_{NAT}, < s >_{SET}) = < remove(x, s) >_{SET} \\ < x >_{NAT} \ \overline{\in} \ < s >_{SET} = < occurs(x, s) >_{BOOL} \end{array}$$

And the equality representation $\mathbf{A_{EQ}}$ is given as follows:

$$< add(x, add(y, s)) >_{SET} \quad = \quad < add(y, add(x, s)) >_{SET}$$

Notice that this axiom does not create inconsistency on strings, because it applies to $\overline{SET}$.

5 THE PRESENTATION LEVEL

A *presentation* is automatically built from the textual level of an abstract implementation. This presentation is an enrichment of $\mathbf{SPEC_0}$. It is useful for proving the correctness of an implementation. Intuitively, all well known difficulties of abstract implementation are treated by the presentation level. These difficulties are mainly the *restriction* to reachable values, and the *identification* of several implementation values which represent the same object. In [EKP 80, EKMP 82, SW 82, San 87], these two problems are handled at the semantical level. This results in a rigorous definition of correctness, but does not provide the specifier with useful correctness proof tools (since correctness is mainly related to the existence of a morphism between two algebras). Here, the restriction problem is explicitly handled via the intermediate constructive sorts and the representation, while the identification problem is explicitly handled via the equality representation.

The *presentation level* associated with the textual level of an abstract implementation is defined as follows:

Figure 2 :

- $\mathbf{EQ : A_{EQ}}$
- $\mathbf{REP : S_1 - S_0}$, $\Sigma_\mathbf{1} - \Sigma_\mathbf{0}$, Σ_{REP} , $\mathbf{A_{REP}}$
- $\mathbf{OPimpl :} \ \overline{\Sigma_\mathbf{1}}$, $\mathbf{A_{OP}}$
- $\mathbf{H : S_H}$, $\Sigma_\mathbf{H}$, $\mathbf{A_H}$
- $\mathbf{SORTimpl :} \ \overline{\mathbf{S_1}}$, $\Sigma_{\mathbf{SYNTH}}$
- $\mathbf{SPEC_0}$: $\mathbf{S_0}$, $\Sigma_\mathbf{0}$, $\mathbf{A_0}$

where **SORTimpl** is a presentation over the specification $\mathbf{SPEC_0}$, **H** is a presentation over the specification $\mathbf{SPEC_0}$+**SORTimpl** (union of $\mathbf{SPEC_0}$ and **SORTimpl**), and so on.
These presentations can be explained as follows:

- $\mathbf{SPEC_0}$ is the *descriptive* specification of the *resident* (previously implemented) data structure.
- **SORTimpl** is the *synthesis* presentation. For each descriptive sort to be implemented $s \in \mathbf{S_1}$, the corresponding constructive sort $\overline{s} \in \overline{\mathbf{S_1}}$ is synthesized by means of the synthesis operations $< ... >_s$: $r_1 \cdots r_m \rightarrow \overline{s}$. Moreover, **SORTimpl** does not contain any axiom. Thus, **SORTimpl** "*implements the constructive sorts*" as free products, or copies, of resident sorts. The initial algebra $T_{\mathbf{SORTimpl}}$ contains the *available* constructive structure which our abstract implementation can use.
- **H** is the *hidden* presentation of the abstract implementation. **H** is a presentation over $\mathbf{SPEC_0}$+**SORTimpl**, as defined in previous section. It will facilitate the constructive specification of the abstract implementation by enriching the resident or available constructive specifications (cf. *remove* and *occurs* in Example 3).

- **OPimpl** is the *operation-implementing* part of the presentation level. It specifies how the constructive operations $\overline{op} \in \overline{\Sigma_1}$ (implementing the descriptive operations $op \in \Sigma_1$) work over the previously synthesized constructive sorts. This is done by means of the operation-implementing axioms $\mathbf{A_{OP}}$, as defined in the previous section. Thus, the initial algebra $T_{\mathbf{OPimpl}}$ handles the constructive implementation of the constructive operations ($\overline{op}$) over the synthesized sorts.
- **REP** is the *representation* presentation. It *explicitly* specifies (in the specification) the effect of the representation signature isomorphism (ρ). We will define Σ_{REP} and $\mathbf{A_{REP}}$ below.
  This presentation **REP** has two principal characteristics. First, it *syntactically* specifies the correspondence between the descriptive operations *op* and the constructive operations $\overline{op}$. Second, it explicitly handles the *restriction* part of the abstract implementation. Let us return to Example 2 (implementation of NAT using INT). With our new formalism, values such as $< -1 >_{NAT}$ are not of sort NAT; they belong to $\overline{NAT}$ which is a copy of INT. There is no NAT-term t such that $\rho(t)$ is equal to $< -1 >_{NAT}$.
- **EQ** is the *equality representation* part of the presentation level. It specifies when two distinct available constructive values represent the same descriptive value to be implemented. This is done via the set $\mathbf{A_{EQ}}$ of conditional axioms. In view of the definition of $\mathbf{A_{EQ}}$ given in previous section, **EQ** is a presentation over the signature ($\mathbf{S_0} + \mathbf{S_H} + \overline{\mathbf{S_1}} + \mathbf{S_1}, \Sigma_\mathbf{0} + \Sigma_\mathbf{H} + \Sigma_\mathbf{SYNTH} + \overline{\Sigma_1} + \Sigma_\mathbf{1}$); in particular **EQ** is a presentation over $\mathbf{SPEC_0}$+**H**+**SORTimpl**+**OPimpl**+**REP**. Thus, the initial algebra $T_{\mathbf{EQ}}$ handles the *identification* of constructive values which represent the same descriptive value to be implemented.

$\overline{\mathbf{S_1}}$, $\Sigma_\mathbf{SYNTH}$, $\mathbf{H}$, $\overline{\Sigma_1}$, $\mathbf{A_{OP}}$ and Aeq are already defined in Section 4. Σ_{REP} and $\mathbf{A_{REP}}$ are defined as follows:

- Σ_{REP} is the set of *representation operations*. For each descriptive sort to be implemented, $s \in \mathbf{S_1}$, there is a representation operation: $\overline{\rho_s} : s \rightarrow \overline{s}$.
- $\mathbf{A_{REP}}$ is the set of axioms which state that $\overline{\rho_s}$ extends the representation signature isomorphism ρ. This means that for each $\Sigma_\mathbf{1}$-ground-term t of sort s, $\overline{\rho_s}(t)$ is equal to the $\overline{\Sigma_\mathbf{1}}$-term deduced from t via ρ. Thus, for each operation to be implemented, $op \in \Sigma_\mathbf{1}$, $\mathbf{A_{REP}}$ contains the following axiom:

$$\overline{\rho_s}(op(x_1, \cdots, x_n)) \quad = \quad \rho(op)(\overline{\rho_{s_i}}(x_i), \cdots, \overline{\rho_{s_n}}(x_n))$$

  where s is the target sort of op, s_i is the sort of x_i , and $\rho(op)$ is equal to $\overline{op}$.
  Moreover, we have to specify that $\overline{\rho_s}$ and $< ... >_s$ both work as *copy* operations on the signature of **SP** (common specification). Thus, for each sort s of **SP**, $\mathbf{A_{REP}}$ contains the following axiom:

$$< x >_s \quad = \quad \overline{\rho_s}(x)$$

  (Such an axiom implies that $\overline{0} = < 0 >_{NAT}$ and $\overline{succ}(< n >_{NAT}) = < succ(n) >_{NAT}$ in $\overline{NAT}$).
  Finally, $\mathbf{A_{REP}}$ contains the following axiom for each descriptive sort $s \in \mathbf{S_1}$:

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

The intuitive meaning of this axiom is the following: if two (descriptive) terms to be implemented, x and y, are represented by the same constructive value ($\overline{\rho_s}(x) = \overline{\rho_s}(y)$), then they must be equal after the implementation is done (x=y). The reason why this constraint is required can be explained as follows: our goal is to describe the data structure that "the user thinks (s)he manipulates" after the implementation is done. If the terms x and y get the same representation, then the user of the implementation cannot distinguish x from y ; consequently, "(s)he thinks that x is equal to y." Such an amalgamation is exactly handled by the axiom specified above.

Notice that Σ_{REP} and $\mathbf{A_{REP}}$ are automatically deduced from the signature isomorphism ρ . Thus the presentation level is always automatically built from the textual definition of **IMPL** without help from the specifier.

Example 4 :
In the *STACK* by *ARRAY* example, $\mathbf{A_{REP}}$ is deduced from the signature isomorphism ρ as follows:

$$\overline{\rho_{STACK}}(empty) = \overline{empty}$$
$$\overline{\rho_{STACK}}(push(n, X)) = \overline{push}(\overline{\rho_{NAT}}(n), \overline{\rho_{STACK}}(X))$$
$$\overline{\rho_{STACK}}(pop(X)) = \overline{pop}(\overline{\rho_{STACK}}(X))$$
$$\cdots etc \cdots$$
$$\overline{\rho_{STACK}}(X) \;=\; \overline{\rho_{STACK}}(Y) \Longrightarrow X \;=\; Y$$
$$\overline{\rho_{NAT}}(m) \;=\; \overline{\rho_{NAT}}(n) \Longrightarrow m \;=\; n$$

Remark 3 :
This specification, from $\mathbf{SPEC_0}$ to **OPimpl**, is very close to the "syntactical level" of [EKP 80] or [EKMP 82]. Our formalism mainly adds the presentations **REP** and **EQ**. It can be shown that **REP** explicitly specifies the *Restriction* functor of the [EKMP 82] semantics; and when the abstract implementation is correct, **EQ** explicitly specifies the *Identification* functor of the [EKMP 82] semantics.

6 THE SEMANTICAL LEVEL

We have shown the following: $T_{\mathbf{SORTimpl}}$ contains all synthesized constructive sorts; $T_{\mathbf{OPimpl}}$ handles the constructive implementation of all constructive operations ($\overline{op}$); $T_{\mathbf{REP}}$ does not add unreachable values to the descriptive sorts to be implemented (thus *restriction* is already included in $T_{\mathbf{REP}}$); and $T_{\mathbf{EQ}}$ contains the *identification* of constructive values which implement the same descriptive value. Consequently, $T_{\mathbf{EQ}}$ is not far from the semantical result of the abstract implementation.
Notice that $T_{\mathbf{EQ}}$ contains all intermediate sorts and operations used by the abstract implementation. But the user of the new implemented data structure must not use the specific operations and sorts of the implementation. It is necessary to forget: the resident sorts and operations which are not in $\mathbf{SPEC_1}$, the hidden sorts and operations, the intermediate constructive sorts, the synthesis operations, and the constructive operations $\overline{op}$. Then, we get a new $\Sigma_\mathbf{1}$-algebra which contains only what the user "thinks (s)he manipulates." This "user view" algebra is called the semantical result of **IMPL** and is denoted by $SEM_{\mathbf{IMPL}}$.

The semantics of an abstract implementation **IMPL** is the composition of two functors:

$$Alg(\mathbf{SPEC_0}) - - F_{\mathbf{SORTimpl}+..+\mathbf{EQ}} \rightarrow Alg(\mathbf{EQ} + .. + \mathbf{SPEC_0}) - - U_{\Sigma_\mathbf{1}} \rightarrow$$
$$Alg(\mathbf{S_1}, \Sigma_\mathbf{1}) T_{\mathbf{SPEC_0}} \vdash F_{\mathbf{SORTimpl}+..+\mathbf{EQ}} \rightarrow T_{\mathbf{EQ}} \vdash U_{\Sigma_\mathbf{1}} \rightarrow SEM_{\mathbf{IMPL}}$$

$F_{\mathbf{SORTimpl+..+EQ}}$ is the usual synthesis functor associated with the presentation **SORTimpl + H + OPimpl + REP + EQ** over $\mathbf{SPEC_0}$ (left adjoint to the forgetful functor). U_{Σ_1} is the usual forgetful functor from Alg($\mathbf{SPEC_0}$+**SORTimpl**+**H**+**OPimpl**+**REP**+**EQ**) to Alg($\mathbf{S_1}, \Sigma_1, \emptyset$) .
So, $SEM_{\mathbf{IMPL}}$ describes the "*user view*" of the new implemented data structure. $SEM_{\mathbf{IMPL}}$ is the part of $T_{\mathbf{EQ}}$ corresponding to the descriptive sorts to be implemented ($\mathbf{S_1}$); and the only operations accessible to the user are those of Σ_1.

Notice that this semantical level is considerably simpler than the ones of [EKP 80, EKMP 82, SW 82...]. This reflects the fact that all abstract implementation problems (restriction and identification) are handled at the presentation level. Moreover, the next section shows that correctness criteria can be stated in a constructive manner, because restriction and identification are taken into account at the presentation level.
At first glance, our formalism may seem to be more restrictive than the [EKP 80, EKMP 82] formalism, because we require a specification of the equality representation. The [EKP 80, EKMP 82] formalism uses the equations of $\mathbf{A_1}$ in order to perform the Identification functor. In fact, when we choose $\mathbf{A_{EQ}} = \mathbf{A_1}$ in our formalism, we get exactly the same semantical result $SEM_{\mathbf{IMPL}}$. The interresting point here is that our semantics avoids the Restriction functor by using the representation ρ (another title of this paper could be "Implementation without Restriction" ...). All the difficulties encountered in [EKP 80, EKMP 82] are due to this "bad" Restriction functor.
Correctness proofs are considerably simpler, in the [EKMP 82] formalism, when the Identification functor can be performed before the Restriction functor (*IR* semantics instead of *RI* semantics). It can be shown that the *IR* semantics of [EKMP 82] is equivalent to the semantics obtained in our formalism by $\mathbf{A_{EQ}} = \rho(\mathbf{A_1}) = \overline{\mathbf{A_1}}$ (e.g. $\overline{pop}(\overline{push}(x, X)) = X$ ...etc); then, correctness can be directly checked at the constructive level.

7 CORRECTNESS PROOFS

Of course, we cannot accept an implementation which does not completely simulate "from the user point of view" the abstract data structure described by $\mathbf{SPEC_1}$. From the above semantics, this means that an abstract implementation must (at least) satisfy the following criteria:

- Each operation to be implemented ($\in \Sigma_1$) has a complete constructive representation (in the "product values" synthesized by $\Sigma_{\mathbf{SYNTH}}$).
- The *user view* of **IMPL** is isomorphic to the descriptive view associated with $\mathbf{SPEC_1}$. This means that $SEM_{\mathbf{IMPL}}$ must be isomorphic to $T_{\mathbf{SPEC_1}}$.

These two criteria are handled in four steps (by dividing the second one into three conditions):

- The complete implementation of all operations to be implemented is called *operation-completeness*.
- $SEM_{\mathbf{IMPL}}$ must be finitely generated over Σ_1. This means that $SEM_{\mathbf{IMPL}}$ is an object of the subcategory Gen($\mathbf{S_1}, \Sigma_1, \emptyset$) of Alg($\mathbf{S_1}, \Sigma_1, \emptyset$). This condition is called *data protection*.
- $SEM_{\mathbf{IMPL}}$ must be a $\mathbf{SPEC_1}$-algebra. This means that $SEM_{\mathbf{IMPL}}$ must validate the $\mathbf{SPEC_1}$-axioms ($\mathbf{A_1}$). This condition is called the *validity* of **IMPL**.
- Finally, among all finitely generated $\mathbf{SPEC_1}$-algebras, $SEM_{\mathbf{IMPL}}$ must be initial. This condition is called the *consistency* of **IMPL**. Now, $SEM_{\mathbf{IMPL}}$ is necessarily isomorphic to $T_{\mathbf{SPEC_1}}$ (unicity of the initial object).

An abstract implementation satisfying these four conditions is called *acceptable*. Some other correctness criteria will be added. For instance, acceptability only concerns the objects to be implemented; it does not ensure the protection of resident values.
Notice that the last acceptability condition reflects an initial view of abstract data types. Of course, the consistency condition can be modified according to a loose semantics "protecting some predefined specifications" [SW 83, Ber 87] by simply requiring consistency of $SEM_{\mathbf{IMPL}}$ with respect to these predefined specifications.

7.1 OPERATION COMPLETENESS

Operation completeness was first introduced by [EKP 80]. The fact that all operations to be implemented have a synthesized constructive representation means that all $\Sigma_{\mathbf{1}}$-terms have a synthesized constructive representation.

Definition 2 :
IMPL is *op-complete* if and only if for all terms $t \in T_{\Sigma_{\mathbf{1}}}$, there is $\alpha \in T_{\mathbf{SORTimpl}}$ such that $\overline{\rho_s}(t) = \alpha$ in $T_{\mathbf{REP}}$ (where s is the sort of t).

Notice that the operation-implementing axioms ($\mathbf{A_{OP}}$) must entirely (recursively) define the implementation of all operations. This must be done without any consideration of the equality representation (i.e. without using $\mathbf{A_{EQ}}$). For example, given a representation <*t,1*> of the term *push(n,empty)* , we must be able to directly apply *pop* : $\overline{pop}(< t, 1 >) = < t, 0 >$, without looking at the implementation of the term *empty*. Thus, op-completeness is defined in $T_{\mathbf{REP}}$ and not in $T_{\mathbf{EQ}}$.

The following theorem shows that op-completeness can always be checked at the constructive level (i.e. without explicitly using the representation).

Theorem 1 :
IMPL is *op-complete* if and only if for all terms $\overline{t} \in T_{\overline{\Sigma_{\mathbf{1}}}}$, there is $\alpha \in T_{\mathbf{SORTimpl}}$ such that $\overline{t} = \alpha$ in $T_{\mathbf{OPimpl}}$.

Proof :
From the specification of $\mathbf{A_{REP}}$, for each $\Sigma_{\mathbf{1}}$-term t, $\overline{\rho_s}(t)$ is equal to the $\overline{\Sigma_{\mathbf{1}}}$-term $\overline{t} = \rho(t)$. Consequently, if all $\overline{\Sigma_{\mathbf{1}}}$-terms have a synthesized value for **OPimpl**, then a fortiori all $\Sigma_{\mathbf{1}}$-terms have a synthesized representation for **REP**. Conversely, ρ is a surjective signature morphism, and **REP** is consistent over **OPimpl**. Thus, if for each $\Sigma_{\mathbf{1}}$-term t, the term $\overline{t} = \overline{\rho_s}(t)$ is equal to a synthesized value α in $T_{\mathbf{REP}}$, then each $\overline{\Sigma_{\mathbf{1}}}$-term $\overline{t}$ is equal to a synthesized value α in $T_{\mathbf{OPimpl}}$. □

Consequently, op-completeness is not difficult to check. It can be directly proved by structural induction over $\overline{\Sigma_{\mathbf{1}}}$. Moreover, we have the following result:

Corollary :
If **OPimpl** is sufficiently complete over **SORTimpl**, then **IMPL** is op-complete.

Proof :
Immediate, because **OPimpl** adds $\overline{\Sigma_{\mathbf{1}}}$ to **SORTimpl** and sufficient completeness means that the canonical adjunction morphism from $T_{\mathbf{SORTimpl}}$ to $T_{\mathbf{OPimpl}}$ is surjective. □

Sufficient completeness of **OPimpl** over **SORTimpl** is not needed in the general case. For instance, we may think of a *SET* by *STRING* implementation where $\overline{del}(< 'aaa' >_{SET})$ does not

return any string. Then, **OPimpl** will not be sufficiently complete over **SORTimpl**. Nevertheless, since $< 'aaa' >_{SET}$ is not a reachable value, this fact does not destroy op-completeness of **IMPL**. We only need for $\overline{del}$ to return a string when its argument is non redundant.
However, this corollary works in most examples. For instance, $\overline{del}(< 'aaa' >_{SET}) =< 'aa' >_{SET}$ in Example 3. Similar results were first given in [EKP 80].

Example 5 :
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 $\alpha = < create, 0 >_{STACK}$
- if x and X have constructive representations ($x = \alpha_1 =< n >_{NAT}$; and $\overline{\rho_{STACK}}(X) = \alpha_2 =< t, i >_{STACK}$), then so does $\overline{\rho_{STACK}}(push(x, X))$:

$$\overline{\rho_{STACK}}(push(x, X)) = \overline{push}(< n >_{NAT}, < t, i >_{STACK}) = < t[i] := n, succ(i) >_{STACK}$$

- similar reasonings apply for *pop* and *top*.

7.2 DATA PROTECTION

Definition 3 :
IMPL is *data protected* if and only if the semantical result $SEM_{\mathbf{IMPL}}$ is finitely generated over $\Sigma_{\mathbf{1}}$.

Theorem 2 :
If **H** is sufficiently complete over **SP**, then **IMPL** is data protected
(**SP**=(**S**,Σ,**A**) is the common specification between $\mathbf{SPEC_0}$ and $\mathbf{SPEC_1}$).

Proof :
The specification of an abstract implementation does not contain any operation with target sort in $\mathbf{S_1}$–**S**, except those of $\Sigma_{\mathbf{1}}$. Thus, it suffices to prove that $SEM_{\mathbf{IMPL}}$ is finitely generated with respect to the sorts of **S**. Since $SEM_{\mathbf{IMPL}}$ is included in $T_{\mathbf{EQ}}$, it suffices to prove that $T_{\mathbf{EQ}}$ is finitely generated with respect to $T_{\mathbf{SP}}$; i.e. that $\mathbf{EQ}+\mathbf{REP}+..+\mathbf{SPEC_0}$ is sufficiently complete over **SP**. Consequently, Theorem 2 results from the fact that the abstract implementation specification does not contain any operation with target sort in **S**, except those of $\Sigma_{\mathbf{1}}$ and $\Sigma_{\mathbf{H}}$. □

From the theoretical point of view, sufficient completeness of the hidden component is not required, since $\mathbf{A_{OP}}$ or $\mathbf{A_{EQ}}$ may complete the specification of some hidden operations. However it is clearly suitable from a methodological point of view, as $\mathbf{A_{EQ}}$ and $\mathbf{A_{OP}}$ have not to play this role.
Data protection is 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. Example 3 (*SET* by *STRING*) is also data protected because *remove* and *occurs* always return predefined strings or booleans ($\mathbf{A_H}$ is equivalent to a canonical rewriting system).

7.3 VALIDITY

Definition 4 :
IMPL is a *valid* abstract implementation if and only if for all $\Sigma_{\mathbf{1}}$-terms, t and t', we have:

if t=t' in $T_{\mathbf{SPEC_1}}$ then t=t' in $SEM_{\mathbf{IMPL}}$.

The following results prove that validity is equivalent to the fact that $SEM_{\mathbf{IMPL}}$ validates $\mathbf{SPEC_1}$; they also prove that validity can always be reduced to a hierarchical consistency property.

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

1) **IMPL** is a valid abstract implementation
2) there is a Σ_1-morphism from $T_{\mathbf{SPEC_1}}$ to $SEM_{\mathbf{IMPL}}$
3) $SEM_{\mathbf{IMPL}}$ validates the axioms of $\mathbf{A_1}$
4) $SEM_{\mathbf{IMPL}}$ validates the axioms of $\mathbf{A_1}$–$\mathbf{A}$
5) $T_{\mathbf{EQ}}$ validates the axioms of $\mathbf{A_1}$–$\mathbf{A}$
6) **ID** is hierarchically consistent over $\mathbf{EQ}$+$\mathbf{REP}$+..+$\mathbf{SPEC_0}$

where **ID** is the presentation over $\mathbf{EQ}$+..+$\mathbf{SPEC_0}$ which contains the set of axioms $\mathbf{A_1}$–$\mathbf{A}$. Thus, **ID**+$\mathbf{EQ}$+..+$\mathbf{SPEC_0}$ contains all the specifications involved in our formalism (both the specification associated with **IMPL** and the descriptive specification $\mathbf{SPEC_1}$).

Proof :
[1 $\Longleftrightarrow$ 2] is clear : since $T_{\mathbf{SPEC_1}}$ is finitely generated over Σ_1, there is a morphism from $T_{\mathbf{SPEC_1}}$ to $SEM_{\mathbf{IMPL}}$ if and only if two Σ_1-terms equal in $T_{\mathbf{SPEC_1}}$ are also equal in $SEM_{\mathbf{IMPL}}$.
[2 $\Longleftrightarrow$ 3] results from the facts that $SEM_{\mathbf{IMPL}}$ is finitely generated over Σ_1 and that $T_{\mathbf{SPEC_1}}$ is initial in $\mathbf{SPEC_1}$. Thus, there is a morphism from $T_{\mathbf{SPEC_1}}$ to $SEM_{\mathbf{IMPL}}$ if and only if $SEM_{\mathbf{IMPL}}$ is a $\mathbf{SPEC_1}$-algebra (i.e. $SEM_{\mathbf{IMPL}}$ validates $\mathbf{A_1}$).
[3 $\Longleftrightarrow$ 4] results from the fact that $\mathbf{EQ}$+..+$\mathbf{SPEC_0}$ contains $\mathbf{SPEC_0}$. In particular, it contains **SP**, thus it contains **A**. Consequently, $SEM_{\mathbf{IMPL}}$ always validates **A**.
[4 $\Longleftrightarrow$ 5] results from the facts that the axioms of $\mathbf{A_1}$–$\mathbf{A}$ only concern the signature $(\mathbf{S_1}, \Sigma_1)$, and $SEM_{\mathbf{IMPL}} = U_{\Sigma_1}(T_{\mathbf{EQ}})$.
[5 $\Longleftrightarrow$ 6] results from the fact that **ID** does not add new operations to $\mathbf{EQ}$+..+$\mathbf{SPEC_0}$ (**ID**=$\mathbf{A_1}$–$\mathbf{A}$). Thus, **ID** is hierarchically consistent over $\mathbf{EQ}$+..+$\mathbf{SPEC_0}$ if and only if $T_{\mathbf{EQ}}$ already validates the axioms of $\mathbf{A_1}$–$\mathbf{A}$. □

The main result is the equivalence between the validity of **IMPL** and the consistency of **ID** over $\mathbf{EQ}$+..+$\mathbf{SPEC_0}$. Thus, validity proofs can always be handled by "classical" methods. This feature is entirely due to our intermediate constructive sorts and the equality representation explicitly specified via *Aeq*.

Examples 6 :
The validity of our abstract implementation of *STACK* is shown by proving that each *STACK*-axiom is a theorem of the specification associated with **IMPL**. We prove here that *pop(push(n,X))* is equal to X in $T_{\mathbf{EQ}}$. The 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 $\overline{pop}(\overline{push}(< n >_{NAT}, < t, i >_{STACK}))$ is equal to $< t, i >_{STACK}$ in $T_{\mathbf{EQ}}$. From $\mathbf{A_{OP}}$, it results that $\overline{pop}(\overline{push}(< n >_{NAT}, < t, i >_{STACK}$)) $= < t[i] := n, i >_{STACK}$. Moreover, from the equality representation ($\mathbf{A_{EQ}}$), it results that $< t[i] := n, i >_{STACK} = < t, i >_{STACK}$, which ends our proof.

To prove that our implementation of *SET* by *STRING* is valid, it suffices to prove that each axiom of *SET* is true in $T_{\mathbf{EQ}}$. We will prove here that

$ins(x,ins(y,X)) = ins(y,ins(x,X))$ is true in $T_{\mathbf{EQ}}$.

Since $\mathbf{A_{REP}}$ contains the following axiom:

$$\overline{\rho_{SET}}(X) = \overline{\rho_{SET}}(Y) \quad \Longrightarrow \quad X = Y$$

and since our implementation is op-complete, it suffices to prove that

$$\overline{ins}(< n >_{NAT}, \overline{ins}(< m >_{NAT}, < s >_{SET})) \quad = \quad \overline{ins}(< m >_{NAT}, \overline{ins}(< n >_{NAT}, < s >_{SET}))$$

We have to distinguish 5 cases:

- n and m both occur in s ; then we get

$$< s >_{SET} \quad =? = \quad < s >_{SET}$$

- n occurs in s and m does not; then we get

$$< add(m, s) >_{SET} \quad =? = \quad < add(m, s) >_{SET}$$

- m occurs in s and n does not; then we get

$$< add(n, s) >_{SET} \quad =? = \quad < add(n, s) >_{SET}$$

- s does not contain n and m, but n and m are equals; then we get

$$< add(n, s) >_{SET} \quad =? = \quad < add(m, s) >_{SET} \text{ (with } n{=}m)$$

- s does not contain n and m, and n and m are distinct; then we get

$$< add(n, add(m, s)) >_{SET} \quad =? = \quad < add(m, add(n, s)) >_{SET}$$

The four first equalities are trivial. The last one results from the equality representation.

7.4 CONSISTENCY

Definition 5 :
IMPL is *consistent* if and only if for all $\Sigma_{\mathbf{1}}$-terms, t and t', we have:

$$\text{if } t{=}t' \text{ in } SEM_{\mathbf{IMPL}}, \text{ then } t{=}t' \text{ in } T_{\mathbf{SPEC_1}}.$$

The following results prove that consistency is equivalent to the fact that $SEM_{\mathbf{IMPL}}$ is initial in Gen($\mathbf{SPEC_1}$). They also prove that consistency can always be reduced to a hierarchical consistency property.

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

1) for all t and t' in T_{Σ_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 $SEM_{\mathbf{IMPL}}$ is a monomorphism
4) $SEM_{\mathbf{IMPL}}$ is an initial $\mathbf{SPEC_1}$-algebra
5) the initial morphism from $T_{\mathbf{SPEC_1}}$ to $U_{\Sigma_1}(T_{\mathbf{ID}})$ is a monomorphism
6) $\mathbf{ID}+\mathbf{EQ}+..+\mathbf{SPEC_0}$ is hierarchically consistent over $\mathbf{SPEC_1}$

Proof :
[1 $\Longleftrightarrow$ 2] results from the fact that $SEM_{\mathbf{IMPL}}$ is equal to the part of $T_{\mathbf{EQ}}$ concerning the signature $(\mathbf{S_1}, \Sigma_1)$.
[2 $\Longleftrightarrow$ 3] results from the fact that $T_{\mathbf{SPEC_1}}$ is finitely generated over Σ_1. Notice that the initial morphism $T_{\mathbf{SPEC_1}} \rightarrow SEM_{\mathbf{IMPL}}$ exists, from Theorem 3.
[3 $\Longleftrightarrow$ 4] results from the fact that $SEM_{\mathbf{IMPL}}$ is finitely generated over Σ_1.
[3 $\Longleftrightarrow$ 5] results from $SEM_{\mathbf{IMPL}} = U_{\Sigma_1}(T_{\mathbf{EQ}})$, and from $T_{\mathbf{EQ}} = T_{\mathbf{ID}}$ (Theorem 3).
[5 $\Longleftrightarrow$ 6] is clear since the initial morphism $T_{\mathbf{SPEC_1}} \rightarrow U_{\Sigma_1}(T_{\mathbf{ID}})$ is the adjunction unit associated with the presentation $\mathbf{ID}+..+\mathbf{SPEC_0}$ over $\mathbf{SPEC_1}$. □

For the same reasons as Theorem 3, Theorem 4 facilitates the consistency proofs, since they can be handled using rewriting techniques or structural induction.

Examples 7 :
The only axioms that may destroy the consistency of $\mathbf{ID}+..+\mathbf{SPEC_0}$ over $\mathbf{SPEC_1}$ are the axioms of sort in $\mathbf{S_1}$.
In the *STACK* by *ARRAY* example, these axioms are:

$$\begin{array}{rcl} \overline{\rho_{STACK}}(X) = \overline{\rho_{STACK}}(Y) & \Longrightarrow & X = Y \\ \overline{\rho_{NAT}}(m) = \overline{\rho_{NAT}}(n) & \Longrightarrow & m = n \end{array}$$

These axioms imply to prove that two descriptive terms represented by the same constructive value (in $T_{\mathbf{EQ}}$) are equal (in $T_{\mathbf{SPEC_1}}$). Thus, we must consider each axiom of $\mathbf{A_{OP}} \cup \mathbf{A_{EQ}}$, and prove that it does not create inconsistencies. Let us consider, for instance, the axiom:

$$\overline{push}(< n >_{NAT}, < t, i >_{STACK}) \quad = \quad < t[i] := n, succ(i) >_{STACK} .$$

Since we work on the stack *values* (not on the stack ground terms), we can handle our proofs with respect to the normal forms of *STACK*. It is possible to prove, by structural induction, that $< t, i >_{STACK}$ represents the stack $push(t[i-1], push(\cdots, push(t[0], empty)..))$. Thus, our proof is clear, as $\overline{push}(\overline{\rho_{NAT}}(n), \overline{\rho_{STACK}}(X))$ represents *push(n,X)*. Other axioms are handled in a similar manner using the normal forms.
In the *SET* by *STRING* example, axioms whose sort belongs to $\mathbf{S_1}$ are:

$$\begin{array}{rcl} \overline{\rho_{SET}}(X) = \overline{\rho_{SET}}(Y) & \Longrightarrow & X = Y \\ \overline{\rho_{NAT}}(n) = \overline{\rho_{NAT}}(m) & \Longrightarrow & n = m \\ \overline{\rho_{BOOL}}(a) = \overline{\rho_{BOOL}}(b) & \Longrightarrow & a = b \end{array}$$

These axioms imply to show that two descriptive terms represented by the same constructive value (in $T_{\mathbf{EQ}}$), are equal (in $T_{\mathbf{SPEC_1}}$). Thus, we must consider each axiom of $\mathbf{A_H} \cup \mathbf{A_{OP}} \cup \mathbf{A_{EQ}}$, and prove that it does not create inconsistencies. The consistency of $\mathbf{A_H}$ is not difficult to prove. Before proving the consistency of the other axioms, we first prove the following "lemma:" if the string s represents the set X then

$$x \in X \quad = \quad occurs(x, s)$$

(this results from the last axiom of $\mathbf{A_{OP}}$ in Example 3 and from the axiom $< n >_{NAT} = \overline{\rho_{NAT}}(n)$ of $\mathbf{A_{REP}}$, since NAT is the common specification between SET and $STRING$).
Then, similar to the use of normal forms in the $STACK$ example, we remark that $\emptyset$ is represented by λ (first axiom of $\mathbf{A_{OP}}$); and if s represents X then *ins(n,X)* is represented by *add(n,s)* each time $n \in X$ is false. (This results from the second axiom of $\mathbf{A_{OP}}$ and from our "lemma.")
Next, the consistency of all axioms of **IMPL** is straightforward. For example, from the following axioms

$$\begin{array}{rcl} occurs(x,s) = False & \Longrightarrow & \overline{ins}(< x >_{NAT}, < s >_{SET}) = < add(x,s) >_{SET} \\ occurs(x,s) = True & \Longrightarrow & \overline{ins}(< x >_{NAT}, < s >_{SET}) = < s >_{SET} \end{array}$$

we get:

$$\begin{array}{rcl} x \in X = False & \Longrightarrow & ins(x,X) = ins(x,X) \\ x \in X = True & \Longrightarrow & ins(x,X) = X \end{array}$$

which do not create inconsistencies.
From the equality representation

$$< add(x, add(y,s)) >_{SET} = < add(y, add(x,s)) >_{SET}$$

we get:

$$ins(x,ins(y,X)) = ins(y,ins(x,X))$$

which does not create inconsistencies.

Remark 4 :
Theorems 3 and 4 reduce the most difficult correctness proofs to *hierarchical consistency* criteria, which mainly leads to theorem proving methods. It is well known that, in many cases, hierarchical consistency is difficult to check. Nevertheless, these results focalize the implementation correctness problem to this well known abstract data type problem. Moreover, hierarchical consistency is considerably more usable than purely semantical criteria, such as the existence of a morphism. Also, it should be noted that the semantical reasonings introduced in Example 7 can be replaced by more systematic (but less concise) methods based on rewriting theory.

Now, we are able to define the acceptability of an abstract implementation:

Definition 6 :
IMPL is *acceptable* if and only if it is op-complete, data protected, valid and consistent.

7.5 CORRECT IMPLEMENTATIONS

When defining *acceptability* of abstract implementation so far, we were only interested in the implemented data structure (initially described by $\mathbf{SPEC_1}$). Most existing abstract implementation formalisms do not add other conditions for correctness. However, acceptability do not care about the interactions between the implementation and other specifications (used by, or using, the implementation). In particular, acceptable implementations may alter already implemented (resident) specifications.

Definition 7 *(protection of the resident specification)* **:**
An implementation **IMPL** *protects the resident data structure* if and only if $\mathbf{EQ}+..+\mathbf{SPEC_0}$ is persistent over the resident specification $\mathbf{SPEC_0}$.

Protection of the resident data structure implies that the semantical result $SEM_{\mathbf{IMPL}}$ is finitely generated over $\Sigma_\mathbf{1}$:

Proposition :
Let **IMPL** be any abstract implementation. If **IMPL** protects the resident data structure then **IMPL** is data protected (definition 3).

Proof :
In the proof of Theorem 2, we showed that it suffices to prove that $\mathbf{EQ}+..+\mathbf{SPEC_0}$ is sufficiently complete over **SP**. By hypothesis, $\mathbf{SPEC_0}$ contains **SP**, and $\mathbf{SPEC_0}$ is sufficiently complete over **SP**. Thus, the sufficient completeness of $\mathbf{EQ}+..+\mathbf{SPEC_0}$ over $\mathbf{SPEC_0}$ gives the conclusion. □

As already mentioned in the beginning of section 7, our acceptability criteria are related to an initial semantics. In particular, the validity of **IMPL** signifies that the initial algebra $T_\mathbf{EQ}$ validates the $\mathbf{SPEC_1}$ axioms (Theorem 3). It implies that all *finitely generated* $(\mathbf{EQ}+..+\mathbf{SPEC_0})$-algebras validate $\mathbf{A_1}$ but it does not implies that all $(\mathbf{EQ}+..+\mathbf{SPEC_0})$-algebras validate $\mathbf{A_1}$ (see Example 8 below).

Definition 8 *(Full validity)* **:**
An implementation is *fully valide* if and only if all $(\mathbf{EQ}+..+\mathbf{SPEC_0})$-algebras validate $\mathbf{A_1}$.

Notice that, in practice, this exactly means that the *equality representation* is powerful enough; in such a way that validity can be proved by equational reasoning, without using structural induction. In particular, all implementations following the [EKMP 82] semantics (i.e. $\mathbf{A_{EQ}} = \mathbf{A_1}$ in our framework, cf. Section 6) are trivialy fully valide.
Similarly to Theorem 3, an implementation is fully valide if and only if for each $(\mathbf{EQ}+..+\mathbf{SPEC_0})$-algebra A the adjunction morphism from A to $F_\mathbf{ID}(A)$ is injective (i.e. if and only if **ID** is *strongly hierarchically consistent* over $\mathbf{EQ}+..+\mathbf{SPEC_0}$).

Definition 9 *(Correct implementations)* **:**
An abstract implementation is *correct* if and only if it is op-complete, it protects the resident data structure, it is fully valide and it is consistent.

Of course, "correct" implies "acceptable."

8 REUSE OF ABSTRACT IMPLEMENTATIONS

8.1 IMPLEMENTATIONS AND ENRICHMENTS

Let $\mathbf{SPEC_1}$ be a specification implemented via **IMPL**. Let **PRES** be a presentation over $\mathbf{SPEC_1}$. We have shown (Section 2.3) that every proof concerning **PRES** is done with respect to $\mathbf{SPEC_1}$, but not with respect to the specification of **IMPL**. The constructive implementation of $\mathbf{PRES}+\mathbf{SPEC_1}$ is not specified by $\mathbf{PRES}+\mathbf{SPEC_1}$, it is specified by $\mathbf{PRES}+\mathbf{EQ}+..+\mathbf{SPEC_0}$, where $\mathbf{EQ}+..+\mathbf{SPEC_0}$ is the whole specification of the implementation of $\mathbf{SPEC_1}$. The following theorem proves that everything is going well whenever the presentation **PRES** is persistent over $\mathbf{SPEC_1}$: the "user view" of the constructive specification $\mathbf{PRES}+\mathbf{EQ}+..\mathbf{SPEC_0}$ is isomorphic to the descriptive data structure specified by $\mathbf{PRES}+\mathbf{SPEC_1}$. This result is entirely due to our intermediate constructive sorts.

Theorem 5 :
If **IMPL** is a correct abstract implementation of $\mathbf{SPEC_1}$, then for all persistent presentations **PRES** over $\mathbf{SPEC_1}$, we have:

$$U_{\Sigma_1+\Sigma_{\mathbf{PRES}}}(T_{\mathbf{EQ}+\mathbf{PRES}}) \quad = \quad T_{\mathbf{SPEC_1}+\mathbf{PRES}}$$

This theorem proves that the presentation **PRES**, together with the abstract implementation of $\mathbf{SPEC_1}$, always provides the user with the expected results.

Before proving Theorem 5, we recall the following lemma (proved in [Ber 86] with positive conditional axioms).

Lemma :
If $\mathbf{P_a}$ and $\mathbf{P_b}$ are two *persistent* presentations over a specification **Spec**, with disjoint signatures, then $\mathbf{P_b}$ is still a persistent presentation over ($\mathbf{P_a}$+**Spec**).

Proof of Theorem 5 :
We remark that correctness of **IMPL** ensure that $\mathbf{ID}+..+\mathbf{SPEC_0}$ is persistent over $\mathbf{SPEC_1}$ (Theorem 4 and data protection). Thus, we deduce from the previous lemma that **PRES** + $\mathbf{ID}$ +..+ $\mathbf{SPEC_0}$ is persistent over $\mathbf{SPEC_1}+\mathbf{PRES}$: $U_{\Sigma_1+\Sigma_{\mathbf{PRES}}}(T_{\mathbf{ID}+\mathbf{PRES}})$ is isomorphic to $T_{\mathbf{SPEC_1}+\mathbf{PRES}}$. Consequently, it suffices to prove that $T_{\mathbf{ID}+\mathbf{PRES}}$ is isomorphic to $T_{\mathbf{EQ}+\mathbf{PRES}}$. This results from the fact that all ($\mathbf{EQ}+..+\mathbf{SPEC_0}$)-algebras validate $\mathbf{A_1}$ (full validity). □

Here is an example of *acceptable* implementation which is not *correct*, and an example of persistent presentation which does not cope with this implementation.

Example 8 :
Let $\mathbf{SPEC_0}$ defined by $\mathbf{S_0}$=*NAT*, $\Sigma_\mathbf{0}$=*0,succ_* with usual arities, and $\mathbf{A_0} = \emptyset$.
Let $\mathbf{SPEC_1}$ defined by $\mathbf{S_1}$=*UNAT*, $\Sigma_\mathbf{1}$ = *zero:*→*UNAT* , *next:UNAT*→*UNAT* , and $\mathbf{A_1}$ containing the following axiom:

$$next(x)=next(y) \Longrightarrow x=y$$

Of course the initial algebras are isomorphic (to **N**), thus *UNAT* can be implemented by *NAT* with $\mathbf{A_H} = \mathbf{A_{EQ}} = \emptyset$. The implementation is clearly acceptable; however it is not correct (not fully valide).
Let **PRES** simply adding a constant τ to the signature, and the axiom: $next(\tau) = next(zero)$. **PRES** is persistent over $\mathbf{SPEC_1}$=*UNAT* (τ must be equal to *zero*) but it is not persistent over the implementation of $\mathbf{SPEC_1}$ (τ is not equal to *zero* in $T_{\mathbf{PRES}+\mathbf{EQ}+..+\mathbf{NAT}}$: the implementation does not ensure the injectivity of *next*). $T_{\mathbf{PRES}+\mathbf{EQ}+..+\mathbf{NAT}}$ is a non finitely generated ($\mathbf{EQ}+..+\mathbf{NAT}$)-algebra which does not validate the axiom of $\mathbf{A_1}$.

8.2 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 implementation. But all our correctness proofs are done with respect to the descriptive specification $\mathbf{SPEC_0}$, not with respect to the specification of the implementation of $\mathbf{SPEC_0}$. We prove in this section that the composition of two correct implementations always yields 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 number of operations being implemented by the lowest level implementation; moreover, this implies that all the lower level implementations must be redefined every time a new implementation is added.
The following theorem proves that the user view, obtained after 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 $\mathbf{IMPL}(1,2)$ obtained from the specification of $\mathbf{IMPL_2}$ by substituting the specification of $\mathbf{IMPL_1}$ for $\mathbf{SPEC_1}$.

$$\mathbf{IMPL}(1,2) = \mathbf{SPEC_0} + (\mathbf{SORTimpl_1} + \mathbf{H_1} + \cdots + \mathbf{EQ_1}) + (\mathbf{SORTimpl_2} + \mathbf{H_2} + \cdots + \mathbf{EQ_2})$$

If $\mathbf{IMPL_1}$ and $\mathbf{IMPL_2}$ are both correct, then we have:

$$U_{\Sigma_2}(T_{\mathbf{IMPL}(1,2)}) \quad = \quad T_{\mathbf{SPEC_2}}$$

Proof :
Since $\mathbf{IMPL_2}$ is correct, $(\mathbf{SORTimpl_2} + \cdots + \mathbf{EQ_2})$ is persistent over $\mathbf{SPEC_1}$. Thus, Theorem 5 (with $\mathbf{PRES}$=$\mathbf{SORTimpl_2} + \cdots + \mathbf{EQ_2}$) proves that $U_{\Sigma(\mathbf{SPEC_1}+..+\mathbf{EQ_2})}(T_{\mathbf{IMPL}(1,2)}) = T_{\mathbf{EQ_2}}$. In particular, $U_{\Sigma_2}(T_{\mathbf{IMPL}(1,2)}) = U_{\Sigma_2}(T_{\mathbf{EQ_2}}) = SEM_{IMPL_2}$. Moreover, the correctness of $\mathbf{IMPL_2}$ implies that $SEM_{IMPL_2} = T_{\mathbf{SPEC_2}}$, which ends our proof. □

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

9 CONCLUSION

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

- Abstract implementation is done by means of intermediate *constructive* values, which are distinct from the *descriptive* values to be implemented.
- These constructive sorts are synthesized by means of *synthesis operations* which extend the classical notion of abstraction operations.
  The correspondence between the descriptive sorts/operations to be implemented and the constructive sorts/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 facilitate the specifications.

- All correctness proof criteria for abstract implementation are "simple" ones (sufficient completeness, hierarchical consistency or fair presentations). This feature provides the specifier with "classical" methods such as *term rewriting* methods, *structural induction* methods or *syntactical* criteria.
- Abstract implementations are compatible with the notion of *enrichment*.
- The composition of several correct implementations always yields 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 semantical level of our abstract implementation is built with "simple" functors. Consequently, it is not difficult to extend this formalism, for instance to abstract data types with *exception handling* [BBC 86, Ber 86], or to *parameterization* since parameterization mainly relies on synthesis functors and pushouts (see [ADJ 80]).

ACKNOWLEDGEMENTS :

It is a pleasure to express gratitude to Michel Bidoit, Christine Choppy and Marie-Claude Gaudel for interesting suggestions, stimulating discussions and careful reading of previous versions of this paper.
This work was partially supported by ESPRIT Project METEOR, at LRI (Orsay, France).

10 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, October 1976).

[ADJ 78]
Goguen J.A., Thatcher J.W., Wagner E.G. : "*Abstract data types as initial algebras and the correctness of data representation*", Current Trends in Programming Methodology 4, (Yeh R. Ed), Prentice-Hall, 1978, p. 80-149.

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

[BBC 86]
Bernot G., Bidoit M., Choppy C. : "*Abstract data types with exception handling : an initial approach based on a distinction between exceptions and errors*", Theoretical Computer Science, Vol. 46, num. 1, p. 13-45, November 1986.

[BCFG 86]
Bougé L., Choquet N., Fribourg L., Gaudel M-C. : "*Test sets generation from algebraic specifications using logic programming*", Journal of Systems and Software, vol.6, num.4, November 1986.

[Ber 86]
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*", PhD thesis, University of Paris-Sud, Orsay, February 1986.

[Ber 87]
Bernot G. : "*Good functors ... are those preserving philosophy !*", Proc. Summer Conference on Category Theory and Computer Science, September 1987, Springer-Verlag LNCS 283, pages 182-195.

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

[Bou 82]
Bougé L. : "*Modélisation de la notion de tests de programme*", PhD thesis, University of Paris VI, October 1982.

[EKMP 82]
Ehrig H., Kreowski H., Mahr B., Padawitz P. : "*Algebraic implementation of abstract data types*", Theoretical Computer Science 20, 1982, pages 209-263.

[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.

[Gau 86]
Gaudel M-C. : "*Automation of testing in software development*", Position paper, IFIP 86, panel on Automation of software development, Dublin (Ireland), September 1986.

[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).

[GM 88]
Gaudel M-C., Moineau T. : "*A theory of software reusability*", to appear in Proc. ESOP 88, Nancy, Mars 1988.

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

[Hoa 72]
Hoare C.A.R. : "*Proofs of correctness of data representation*", Acta Informatica 1, n. 1, 1972, p. 271-281.

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

[San 87]
Sannella D. : "*Implementations revisited*", 5th Workshop on Specification of abstract data types, Edinburgh, September 1987, abstracts in LFCS report 87-41.

[Sch 87]
Schoett O. : "*Data abstraction and the Correctness of Modular Programming*", PhD thesis,

University of Edinburgh, 1987.

[SW 82]
Sannella D., Wirsing M. : "*Implementation of parameterized specifications*", Report CSR-103-82, Department of Computer Science, University of Edinburgh.

[SW 83]
Sannella D., Wirsing M. : "*A kernel language for algebraic specification and implementation*", Proc. Intl. Conf. on Foundations of computation Theory, Springer-Verlag, LNCS 158, 1983.