Label algebras and exception handling [1]

Gilles Bernot[a] , Pascale Le Gall[a] , Marc Aiguier[a b]

[a] LaMI,
Laboratoire de Mathématiques et Informatique,
Université Evry - Val d'Essonne,
Bd. des Coquibus,
F-91025 Evry cedex,
France

[b] LRI,
UA CNRS 410,
Université Paris-Sud,
Bât. 490,
F-91405 Orsay cedex,
France

e-mail: {bernot,legall,aiguier}@univ-evry.fr

Abstract: We propose a new algebraic framework for exception handling which is powerful enough to cope with many exception handling features such as recovery, implicit propagation of exceptions, etc. This formalism treats all the exceptional cases; on the contrary, we show that within all the already existing frameworks, the case of bounded data structures with certain recoveries of exceptional values remained unsolved.
We justify the usefulness of "labelling" some terms in order to easily specify exceptions without inconsistency. Surprisingly, there are several cases where even if two terms have the same value, one of them is a suitable instance of a variable in a formula while the other one is not. The main idea underlying our new framework of *label algebras* is that the semantics of algebraic specifications can be deeply improved when the satisfaction relation is defined via assignments with range in *terms* instead of values. We give initiality results, which are useful for structured specifications, and a calculus for positive conditional label specifications, which is complete on ground formulas. *Exception algebras* and *exception specifications* are then defined as a direct application of label algebras. The usual inconsistency problems raised by exception handling are avoided by the possibility of labelling terms. We also sketch out how far the application domain of label algebras is more general than exception handling.

Key-words: algebraic specifications, exception handling, error handling, initial semantics, structured specifications, exception recovery, bounded data structures.

1 Introduction

For some kinds of software engineering projects (railways, aeronautics, hardware codesign), formal specifications methods are becoming of common use. In practice, specifiers do not ask for *universal* specification frameworks (e.g. a very general logic); they prefer specification languages *dedicated* to the problem under consideration. For each class of formal specification frameworks, a common challenge is to increase the capability of "tuning" syntax and semantics according to the needs of the specifiers. Among these needs, exception handling is a subject which, in practice, has been often neglected at the specification stage in software engineering. This results in incomplete specifications and various choices of "how to treat exceptional cases" are then often made at the programming stage. As usual when specifications are incomplete, this decreases the overall quality of the software: some exceptional cases are checked twice (e.g. in the calling module and in the called module), or even worse there are misunderstandings about how they should be treated, or still worse they are never

[1] Short and partial versions of this article can be found in [BL91] and in [BL93].

checked. Moreover, if the exceptional cases are not well specified, the corresponding bugs are very difficult to identify, as they do not cope with the standard verification and validation methods (e.g. proving or testing methods).

An important class of exceptional cases is related to "intrinsic" properties of the underlying abstract data structure: access to an empty data structure (e.g. top of an empty stack, or an element chosen in an empty set, etc.), or functions which are intrinsically not defined for certain values (e.g. "pop" for an empty stack, predecessor for 0 in natural numbers, or factorial for negative numbers, etc.). Another important class of exceptional cases relies on "dynamic" properties of the data structure (e.g. access to a non-initialized data, a non-initialized array cell, etc.). In addition, it is very important not to neglect certain limitations, due to the system itself or required by the specifier, mainly bounded data structures (e.g. arrays, intervals, etc.).

In this paper, a new framework for exception handling within algebraic specifications is proposed. Before defining what we call *exception algebras*, we will introduce a general framework, the *label algebras*, whose application domain is much more general than exception handling. The paper is organized as follows:

- In Section 2 we will point out two great usefulness of exception handling that are often neglected: *legibility* and *terseness*. Algorithms are considerably simplified when the programming language has exception handling features. By analogy, we will extract several requirements for formal specifications with exception handling in order to improve legibility and terseness. One of them is that "exception handling" does not only mean "error handling," it also means "rare case handling."

- In Section 3 we will enumerate the main difficulties raised by exception handling within the algebraic framework (often resulting in inconsistencies). The most difficult point is to simultaneously handle bounded data structures and certain recoveries of exceptional values. No previously existing framework is capable of solving this difficulty. The solution requires defining assignments on *terms* instead of values, and we will show the usefulness of "labelling" terms in order to easily specify exceptions.

- In Section 4 we will define the framework of label algebras. We will also sketch out how far this framework can be applied to several other classical subjects of abstract data types, such as partial functions, observability features, etc.

- The main results (e.g. initiality results, adjunction, the soundness of the associated calculus, or its completeness on ground formulas) will be established in Section 5.

- Exception signatures and exception algebras will be introduced in Section 6 as a particular case of label algebras, and the difference between *exception* and *error* will be rigorously defined.

- Exception specifications and their semantics will be defined in Section 7, and they are related to the semantics of label algebras (via a simple translation).

- Section 8 contains the fundamental results about exception algebras (directly deduced from the properties of label algebras). These results allow us to handle *structured* exception specifications.

- Section 9 provides a wide collection of simple examples of exception specifications. They illustrate many powerful aspects of exception specifications and show that all the mentioned classes of exceptional cases ("intrinsic" exceptions, "dynamic" exceptions, bounded data structures) can easily be specified. Lastly, a simple proof example illustrates our calculus.
- Recapitulation and perspectives can be found in Section 10.

We assume that the reader is familiar with algebraic specifications ([GTW78], [EM85], [Wir90], [GB84]) and with the elementary terminology of category theory ([BW90]).

2 Crucial aspects of exception handling

In this section, we will illustrate how exception handling usually improves legibility and terseness. We will also refer to the other classical desirable aspects.

2.1 Exception handling and programming languages

Let us consider a simple example of algorithm: a function which searches an element `e` in a list. Naive programmers often make the following mistake:

```
current := first ;
while ((current <> nil) and (current.value <> e)) do...
```

This only works if they are lucky with respect to the compiler ![1] Less naive programmers write:

```
current := first ; found := false ;
while ((current <> nil) and (not found))   do found := (current.value = e) ...
```

Similar solutions are not acceptable for specifications, because a specification has to be *abstract* and *legible*. Moreover the test `current <> nil` is done many times, while the end of the list is exceptional. Experienced programmers add a fictitious `last` cell at the end of a list (thus the empty list contains one cell); they write

```
last.value := e ; current := first ;
while (current.value <> e) do current := current.next ;
  ... ;
```

and the search fails if and only if `current=last` at the end. Of course, this solution is not abstract at all and the solution (as you may have guessed from the beginning of our story) is exception handling. The exception handler plays a role similar to the fictitious cell:

`when Illegal-pointer-access return` *... (the search has failed).*

The main algorithm is as simple as possible:

```
current := first ;
while (current.value <> e) do current := current.next ;
   ... ;
```

and the search for instance returns the place of `e` if the handler has not been called.

[1] more precisely if "`and`" is a lazy operator which evaluates first the left hand side argument.

In software engineering it is well known that the use of exception handling for situations which are not erroneous improves software quality by reducing the size of programs and improving legibility. Moreover, in languages such as CLU ([LG86]), this terseness does not harm the easiness to reason about the program.

Conclusion: *exception* handling is not only used for *error* handling; it is also a great tool for *legibility* and *terseness*.

- Legibility: the "rare cases" (e.g. "limit cases" as in bounded data structures) are extracted from the main text so that it becomes easily readable.
- Terseness: the exception handler, as well as the main text, goes straight to the point. Each statement has not to deal with the cases that it does not directly concern: the application domains are handled implicitly by the underlying semantics.

2.2 Exception handling and abstract specifications

For abstract specifications, legibility and terseness should be *a fortiori* a great usefulness of exception handling. We believe that a formal framework only capable of treating *error* handling is not fully satisfactory; specification and abstraction can take benefit of a full *exception* handling. From our point of view, an exception is not necessarily an error; it simply requires a special treatment which has to be clearly distinguished from the main properties. Thus, errors are only a particular case of exceptions.

Legibility can be improved as follows: in the text of a formal specification, the rare cases can be specified as "exceptions" apart from the normal axioms and accordingly the semantics has to *implicitly* restrict the scope of the normal axioms. When this partition is not available, it is often necessary to write complex axioms where additional conditions appear to restrict the scope of the axioms to normal (resp. exceptional) cases.

Terseness is rather a semantic issue: the specialized semantics for each part of the syntax (exceptional/normal properties) has to implicitly handle obvious general properties of exceptions. For instance, it is clear that errors should propagate by default (if a is erroneous, then $f(a)$ is also erroneous, except if it is recovered); such properties should not have to be explicitly specified.

Moreover, the following principles have been widely recognized to be crucial for abstract specifications with exception handling ([Gog78a], [GDLE84], [Bid84], [Ber86], [BBC86], [Sch91]):

- each exceptional (resp. erroneous) case should be declared with some exception name (resp. error message) which provides enough informations to treat it easily;
- all the relevant properties of exceptional state behaviours should be formally specified;
- the implicit exception propagation rule should nevertheless allow various recoveries of exceptional cases.

3 Algebraic specifications with exception handling

The main difficulty of exception handling for algebraic specifications is that all the "simple" semantics that we can imagine lead to inconsistencies. To illustrate this fact, let us try to

specify natural numbers with exception handling. Bounded natural numbers raise all the main difficulties of exception handling for algebraic specifications. We start with the simple "intrinsic" exception $pred(0)$, and we will add more and more sophisticated exceptional cases. Step by step, we will show that more and more sophisticated semantics are needed. At the end, we show that a legible and terse specification of bounded natural numbers with certain recoveries requires semantics based on terms instead of values.

3.1 Errors as constant operations

A simple idea would be to use the classical ADJ semantics [GTW78] [EM85], adding a new constant *error* of sort *Nat* and the axiom:

$$pred(0) = error$$

Of course, we have to face error propagation: what is the value of $succ(error)$? A natural idea is to add, for each operation f of the signature, axioms of the form:

$$f(\ldots error \ldots) = error$$

Unfortunately, the specification also contains the axiom:

$$x \times 0 = 0 \tag{1}$$

thus we get $error = 0$ (with $f = \times$, via the assignment $x = error$). We meet here the principle that "normal cases" should be distinguished from exceptional cases. The semantics of "normal axioms" should be *implicitly* of the form:

$$x \neq error \implies x \times 0 = 0$$

Notice that the existence of an initial algebra is not ensured in general (a negative atom appears in the axiom [WB80]). This fact has already been shown in [GTW78] where an explicit introduction of an Ok predicate is proposed.

3.2 Errors and Ok predicates

If the specification contains a boolean sort, we can define an Ok predicate which checks if a value is a normal value:

$$\begin{array}{l} Ok(error) = false \\ Ok(0) = true \\ Ok(succ(n)) = Ok(n) \\ Ok(pred(0)) = false \\ Ok(pred(succ(n))) = Ok(n) \\ \ldots \\ Ok(x \times y) = Ok(x) \ and \ Ok(y) \end{array}$$

If we want to express that an instance of the axiom (1) can be considered only if both members of the equation are first checked as normal values, then we write:

$$Ok(x \times 0) = true \ \wedge \ Ok(0) = true \implies x \times 0 = 0$$

and the existence of an initial algebra is ensured. Unfortunately, this approach does not succeed with respect to legibility and terseness, as already pointed out in [GTW78]: *"the resulting total specification (...) is unbelievably complicated."* It is also shown that the axioms defining Ok cannot be automatically generated without introducing inconsistencies ($true = false$); this is particularly obvious when recoveries are allowed. To be convinced, let the reader try to define the Ok predicate consistently when $succ(pred(0))$ is recovered... (See also [Gog78a].)

3.3 Errors and partial functions

Clearly, these difficulties result from the explicit introduction of an *erroneous value* in the signature. Moreover, the specification of the Ok predicate resembles the specification of definition domains. Thus, a simple idea could be to consider partial functions instead of total functions (e.g. $pred(0)$ being undefined), see for instance the pioneering work of [BW82] (many other references are relevant too). Unfortunately, specifying exceptions via partial functions is not powerful enough for a full exception handling. For instance exceptional cases can give rise to ulterior recoveries, especially for robust software: even if $f(x)$ is not defined, we can require for $g(f(x))$ to be defined (e.g. $succ(pred(0))$). More generally, we have often to specify properties concerning exceptional cases, even if they are not recovered. Consequently, exceptional cases should always keep some "semantic meaning," as we allow specific treatments of exceptional or erroneous values themselves. Partial functions do not offer this feature.

Nevertheless, if the specifier is not interested in recoveries and does not want to attach error messages to erroneous values, (s)he can use partial functions.

3.4 Error handling and subsorting

Since the work of Goguen in [Gog78b], the framework of order-sorted algebras has been widely advocating to be a solution for exception handling (see also [FGJM85][GM89]): the Ok-part of the sort *Nat* being a subsort *OkNat* of *Nat*.

For example, it is easy to declare that the sort *OkNat* is generated by 0 and *succ*, that *ErrNat* is the sort reduced to the singleton $\{error\}$, and that *Nat* is the union of *OkNat* and *ErrNat*. Then, we can restrict the scope of the axiom

$$x \times 0 = 0$$

to the sort *OkNat* and this prevents from the inconsistency described above.

Notice that type inference is required in order to determine the scope of an axiom. To be able to deduce that $pred(0)$ belongs to *ErrNat*, and that $pred(x)$ belongs to *OkNat* when x is a positive natural number, a sort *PosNat* is declared, which is equal to $succ(OkNat)$ (it is not difficult to prove that *PosNat* is a subsort of *OkNat*). Then, roughly speaking, the arity of *pred* is specified via overloading:

$$\begin{aligned} pred:&\ PosNat \rightarrow OkNat \\ pred:&\ \{0\} \rightarrow ErrNat \\ pred:&\ ErrNat \rightarrow ErrNat \end{aligned}$$

which implies, for instance, that *pred* can be shown as an operation from *OkNat* to *Nat*. Similarly, the Euclidean division can be specified with the arity

$$\begin{aligned} div &: OkNat \times PosNat \rightarrow OkNat \\ div &: Nat \times \{0\} \rightarrow ErrNat \\ div &: Nat \times ErrNat \rightarrow ErrNat \end{aligned}$$

and so on.

Unfortunately, things are not always so easy. This "subsorting approach" amounts to describe for *each* operation of the signature, the arguments that do not need exceptional treatments. It may be surprising but this approach is not terse at all. Let us specify the subtraction. The definition domain of the subtraction "$-$" is the set of all $(a, b) \in OkNat$ such that $a \geq b$. Contrarily to the division, this definition domain cannot be expressed as a Cartesian product of *Nat* subsorts. The solution is to define a new sort *Nat2* which is the Cartesian product $Nat \times Nat$ and to *explicitly* define the domain of "$-$" as a subsort $Dsub$ of *Nat2*. Even if we forget the large number of coercions required to type a simple expression (such as $(a - b) - pred(c)$), it remains that the specification of $Dsub$ will not be *terse*:

$$\begin{aligned} a \in OkNat &\implies (a, 0) \in Dsub \\ (a, b) \in Dsub &\implies (succ(a), succ(b)) \in Dsub \end{aligned}$$

The point is that these two typing axioms have to be compatible with the semantics of the subtraction; they require an effort from the specifier which is almost as difficult as the definition of the *Ok* predicate of [GTW78].

Notice moreover that the propagation of errors is actually *not* implicit, since the definition domain of each operation should be explicitly defined on all the elements of a sort.

The main advantage of the approaches based on subsorting is that the specification style fulfills the legibility criterion in general. Moreover, the names of the (erroneous) subsorts can be used to represent exception names (or error messages) in such a way that a precise error handling can be performed. The lack of terseness is the main disadvantage of these approaches because too many subsorts have to be explicitly defined in a specification (see also Section 3.6 where another strong limitation of subsorting is explained).

More precisely, the terseness criterion for exception handling is better fulfilled when the semantics are based on a declaration of the "*Ok-codomain*" of the operations rather than their "*Ok-domain*." The reason is simple: in general, all the operations of a data type share the same Ok-codomain, while each of them has its own Ok-domain. Let us consider the two following axioms defining the subtraction:

$$\begin{aligned} a - 0 &= a \\ succ(a) - succ(b) &= a - b \end{aligned}$$

If b is greater than a, it is clear that the expression $(a-b)$ is not reducible to a value of $OkNat$ according to these two axioms. We have in mind that it is sufficient to specify the subsort $OkNat$. Roughly speaking, if the axioms defining "$-$" allow us to find a result for $(a - b)$ in $OkNat$ then (a, b) implicitly belongs to $Dsub$, else $(a - b)$ is exceptional. For example:

$$\text{is the term } (succ(0) - succ(succ(0))) \text{ is a normal case?}$$

Our axioms only allow to deduce that this term is equal to $(0-succ(0))$ and it is impossible to combine the equalities up to a term belonging to $OkNat$ (of the form $succ^i(0)$). Consequently, we could automatically deduce that $(succ(0) - succ(succ(0)))$ is exceptional (at least in the initial algebra). On the contrary, $(succ(succ(0)) - succ(0))$ reduces to $succ(0)$ which belongs to $OkNat$. Thus, it is a normal case. It seems clear that an explicit specification of $Dsub$ is not required. The same remark applies to *pred*: the explicit specification of $PosNat$ is superfluous.

The first framework that took advantage of this idea (even if it was not explicitly analysed this way by the authors) is [GDLE84] where the Ok-part of a sort is described via "safe" operations.

3.5 Safe and unsafe operations

The simplest idea to describe the Ok-part of each sort is to distinguish a set of operations (subset of the signature) that generates the Ok-values. In [GDLE84][Gog87], the signature Σ is partitioned into "safe" and "unsafe" operations. For example, 0, *succ* and + are safe operations because, when applied to Ok-arguments, they always return Ok-results; on the contrary, *pred* and "$-$" are unsafe because 0 is Ok but $pred(0)$ is erroneous, and, for instance, 0 and $succ(0)$ are Ok but $0 - succ(0)$ is erroneous. The main advantage of this approach is that such a simple syntactic classification of functions describes the Ok and erroneous part of each sort; the Ok-values are those generated by the safe operations, all the other values are automatically erroneous. For example, the axioms defining the subtraction (previous subsection) are sufficient to automatically deduce its "Ok-domain." It is not difficult to prove that $(a - b)$ has an Ok-value (i.e. it is in the equivalence class of a term generated by 0, *succ* and +) if and only if a is greater or equal to b.

This way, we obtain a better terseness of specifications, but as shown in [GDLE84], this idea is not fully sufficient to solve the inconsistencies mentioned so far. Let us return to the axiom

$$x \times 0 = 0$$

and let us consider an instance of x which is an erroneous value (say *error*). We would still have that $error \times 0 = 0$. This does not induce an inconsistency because $error \times 0$ is not necessarily equal to *error* thanks to the refined error propagation principle of [GDLE84][Gog87]. It is automatically recovered (according to the "codomain driven" strategy). Of course, this implicit recovery is not necessarily wished by the specifier, and we have to provide a way of preventing it if necessary. This is the reason why the authors introduce a special type of variables (often denoted as "x_+") which can only serve for Ok-values. Then, the previous implicit recovery can be avoided by writing

$$x_+ \times 0 = 0$$

where the assignment $[x_+ \leftarrow error]$ is not allowed. (This special kind of variables is also used in [Bid84], but the proposed semantics is more complicated and gives less usable results, in particular because the initial algebra does not exist).

One of the main advantages of the framework of [GDLE84] [Gog87] is that, given a set of positive conditional axioms, a least congruence exists. Consequently an initial algebra exists, a left adjoint functor to the forgetful functor exists, and parameterization can be

easily defined. Structured specifications with error handling features can be easily studied in this framework.

Moreover the terseness criteria is satisfied, because the erroneous cases have not to be explicitly characterized. Legibility is also better achieved than with all the approaches mentioned above. However, in practice, the specifier has to be very careful in deciding when a "normal variable" (x) or an "Ok-variable" (x_+) should be used in an axiom. This is due to the fact that this approach does not offer a distinction between "normal axioms" and "exceptional axioms" (see Section 2.2 above). Legibility would be improved if such a distinction were provided.

An extension of this approach to order sorted algebras exists [Gog84]. All the mentioned advantages remain, while preserving the simplicity of the semantics.

As already pointed out in [Ber86] [BBC86], the main problem of this framework is that bounded data structures cannot be specified. The reason is simple: for bounded data structures almost all the operations are unsafe, except a few constants. For example, *succ* and + are not safe for bounded natural numbers ($succ(Maxint)$ is erroneous while $Maxint$ is Ok); consequently the Ok-part of the sort *Nat* would be reduced to 0 (at least in the initial algebra).

3.6 Bounded data structures and recovery axioms

The approaches mentioned above give solutions to the algebraic treatment of "intrinsic errors" (such as $pred(0)$), with implicit error propagation and possible recovery, but they are not able to treat the other kind of errors mentioned in Section 1, especially bounded data structures. Nevertheless, software engineering requires a careful treatment of these bounded data structures. If they are not taken into account at the specification level, then almost all the specified properties are actually false; and precisely, in practice, software requires a strong verification and validation effort near the bounds of the underlying data structures.

Let us sketch out a simple example to give an idea of the difficulties raised by bounded data structures for algebraic specifications, especially when recoveries are allowed. To specify bounded natural numbers it is indeed not too difficult to specify that all the values belonging to $[0 \ldots Maxint]$ are Ok-values [BBC86]; let us assume that this is done. We also have to specify that the operation *succ* raises an exception when applied to $Maxint$, e.g. $TooLarge$; let us assume that this is done too. When specifying the operation *pred*, we have the following axiom:

$$pred(succ(x)) = x \tag{2}$$

which is a "normal property" and, as such, should be understood with certain implicit preconditions such as "*if x and $succ(x)$ are Ok-values*." Assume now that we want to recover all $TooLarge$ values on $Maxint$. Then, we will necessarily have $succ(Maxint) = Maxint$.

Since these two values are equal, we have to choose: either both of them are erroneous values, or both of them are Ok-values. The first case is not acceptable because it does not cope with our intuition of "recovery". (Moreover, when considering the value $m = Maxint - 1$ we clearly need that $pred(Maxint) = m$, as a particular case of our "normal property" about *pred*; thus $succ(m) = Maxint$ must be considered as a normal value.) Unfortunately, since $succ(Maxint)$ is then a normal value, $x = Maxint$ is an acceptable assignment for our "normal

property" (2) and we get the following inconsistency:

$$m = pred(Maxint) = pred(succ(Maxint)) = Maxint$$

which propagates, and all values are equal to 0.

Remark 3.1 : A possible reaction to this inconsistency could be to say that *"the specifier should not have written such an inconsistent axiom; (s)he should have been careful and written something like*

$$\begin{array}{l} x \leq m = true \implies pred(succ(x)) = x \\ pred(Maxint) = m \end{array}$$

because (s)he knew that $succ(Maxint) = Maxint$*."* Our claim is that this way of thinking contradicts the terseness and legibility principles explained in Section 2. Exception handling should allow the specifier to say *"I declared* $succ(Maxint)$ *exceptional, consequently I should not have to worry about it when I write a normal property; the semantics should discard automatically the assignment* $succ(Maxint)$ *from the set of acceptable assignments."*

As a matter of fact, this example precisely reveals the difference that we make between "exception handling" and "error handling." The term $succ(Maxint)$ is not erroneous but it is exceptional; even if the term $succ(Maxint)$ is recovered on $Maxint$, the exception name *TooLarge* should *not* be propagated to $Maxint$.

This leads to the following idea: the term $succ^m(0)$ is[1] an acceptable assignment for the variable x in the equation (2) while the term $succ^{Maxint}(0)$ is not, even though $succ(succ^m(0))$ and $succ(succ^{Maxint}(0))$ have the same value. The term $succ^{Maxint}(0)$ (i.e. $succ(succ^m(0))$) is not exceptional while the term $succ^{Maxint+1}(0)$ (i.e. $succ(succ^{Maxint}(0))$) is exceptional. Thus, exception handling requires taking care of terms inside the algebras and good functional semantics for exception handling should allow such distinctions. This idea has been formalized in [Ber86][BBC86], where "Ok-terms" are declared instead of the safe operations of [GDLE84]. In this framework, the term $succ^{Maxint}(0)$ is "labelled" by *Ok* while the term $succ^{Maxint+1}(0)$ is not; and the acceptable assignments of a normal property (called "Ok-axiom") are implicitly restricted to Ok-terms only. This approach solves the inconsistencies generated by the recovery $succ^{Maxint+1}(0) = Maxint$. The declaration of Ok-terms looks like

$$\begin{array}{l} succ^{Maxint}(0) \in Ok \\ succ(n) \in Ok \ \Rightarrow \ n \in Ok \end{array}$$

Let us point out that subsorting (see Section 3.4 above) cannot be used to specify such bounded data structures with recoveries. The axiom (2) necessarily gives rise to a similar paradox because sorts are attached to values. Two terms having the same value share the same subsorts; consequently $succ^{Maxint}(0)$ and $succ^{Maxint+1}(0)$ cannot be distinguished.

Another idea of [Ber86] [BBC86] is that several exceptional cases can require the same kind of treatment, while keeping distinct values; they are grouped under common exception names. In this framework, exception names are predicates on values. For example, the value of $succ(Maxint)$ belongs to $TooLarge$ and this can be specified by $succ(Maxint) \in TooLarge$.

[1] $succ^i(0)$ is an abbreviation for $succ(succ(\ldots(0)\ ))$ where *succ* appears i times.

We showed that the special label Ok, which concerns normal cases, cannot be carried by values. The following example shows that exception names have also to be carried by terms, not values.

Example 3.2 : Let us assume that every value of the form $succ^i(Maxint)$ $(i \geq 1)$ is attached to the name *TooLarge*. Let us assume that we want to recover every *TooLarge* value on *Maxint*. A possible way of expressing this recovery is to say *"if the operation succ raises the exception TooLarge, then do not perform it."* It is formally specified as:

$$succ(n) \in TooLarge \quad \Rightarrow \quad succ(n) = n \tag{3}$$

When the exception name *TooLarge* is carried by *values*, the term $succ(Maxint)$ being equal to the term *Maxint*, both of them belong to *TooLarge*. For $m = Maxint - 1$, we get the following inconsistency:

$$Maxint = succ(m) = m$$

because $Maxint = succ(m)$ belongs to *TooLarge*, thus axiom (3) applies. This inconsistency propagates and all values are equal to 0.

Remark 3.3 : Not all readers will accept this idea of recovery from exceptions within specifications. However, one should not forget that such semantics of exception handling are usual and well founded in programming languages (e.g. CLU [LG86]). It would be a pity if specification languages had semantics with a weaker expressive power. Moreover, it is the only way to specify recoveries of exceptions *after* they have been declared.

Consequently, in the framework of [Ber86] [BBC86], it was not possible to specify this kind of recovery. This was the case for all existing algebraic frameworks for exception handling, because exception names (if provided) were always carried by values.

Nevertheless, the solution is simple: even if $succ(Maxint)$ is recovered on *Maxint*, the exception name *TooLarge* does not propagate to *Maxint*. Exception names do not go through recoveries. As a consequence, exception names should be treated in a similar way as the label Ok; they concern terms, not values.

3.7 Other extensions with multityping

Roughly speaking, exception handling requires a special "typing" of *terms*. We shall call *labels* these special "types". From this point of view, the label algebras defined below are an extension of more standard algebraic approaches with "multityping" such as order sorted algebras [Gog78b][FGJM85]. It is why we give a brief overview of several other approaches also based on "multityping".

Unified algebras in [Mos89], Equational typed algebras in [MSS89] or G-algebras in [Meg90] allow to explicitly mention sorts (type names) within the axioms. For such extensions of the notion of sorts, we can imagine to take benefit of an explicit manipulation of the type names in order to manipulate them as exception names. For example in [MSS89], the signature does not contain a set of sorts, but formulas admit an additional binary predicate ":". An atom of the form $t : t'$ means that t is of type t'. In order to represent the *TooLarge* exception name, one could imagine to introduce a constant operation *TooLarge* (with arity 0); then, one could write, as in Example 3.2:

$$succ^{Maxint+1}(0) : TooLarge$$
$$succ(n) : TooLarge \Rightarrow succ(n) = n$$

Unfortunately, in [MSS89] as well as in [Mos89] and [Meg90], types pass through equalities (i.e. two terms having the same value share the same types). Consequently, they lead to the same inconsistencies as ordinary subsorts described in Sections 3.4 and 3.6.

From another point of view, [Mos89] and [MSS89] allow to treat sorts exactly as ordinary terms: one can consider operations taking sorts as arguments (in particular one can introduce exception names with arguments). Our framework of label algebras will not allow such facilities.

Few logical frameworks allow to distinguish two terms having the same value. They mainly have been introduced to solve some type inference problems within "simple" order sorted algebras. The main weakness of the first approaches of subsorting ([Gog78b][FGJM85]) was that typing was implicit in the formulas. The smallest type of a term was consequently very difficult to determine (when it exists). Worse: it was sometimes undecidable. For theorem proving purposes, several recent works about "constraints" have been developed (e.g. [CD91] but many other references are relevant too).

In these frameworks with constraints, the considered formulas are of the form:

$$\varphi \Rightarrow \psi$$

where φ belongs to a logic L_1 and ψ to a logic L_2; L_1 and L_2 sharing a sub-signature Σ_0. The formula φ is said to be "the constraint." The formula ψ is then considered "under the constraint φ." The point is that φ is assumed decidable.

Roughly speaking, from an algebraic point of view, the semantics rely on a Σ_0-morphism $\mu : D \to A$ where D is a domain such that every formula of L_1 is decidable and A is any domain for L_2. The formula $\varphi \Rightarrow \psi$ is satisfied if and only if at each time φ is valid in D, ψ is valid in A for every corresponding substitution that factors through μ.

Closer to our motivations, this approach can be applied to:

- $D = T_\Sigma$ with $L_1 = \{t : s \mid t \in T_\Sigma,\ s \text{ is a type name}\}$
- A is a Σ-algebra with L_2 allowing equalities or positive conditional equalities.

provided that we have a complete, static type inference on terms. It is then possible to type *terms* in $D = T_\Sigma$ independently of their *value* in A. This allows to write axioms such as:

$$succ(n) : Toolarge \Rightarrow succ(n) = n$$

without the inconsistencies mentioned above.

Unfortunately, the point is that in all these approaches, it is impossible to write axioms of the reversed form

$$\psi \Rightarrow \varphi$$

with $\psi \in L_2$ and $\varphi \in L_1$ (properties of D cannot be consequences of properties of A).

This limitation forbids a credible and complete treatment of exception handling. For example a specifier is not allowed to write :

$$heigt(X) \geq Maxheigt = true \Rightarrow push(x, X) : Overflow$$

where, of course, the equality $heigt(X) \geq Maxheigt = true$ has to be checked in A while the typing $push(x, X) : Overflow$ is checked in $D = T_\Sigma$.

The framework proposed in [Poi87] can be considered as a particular case of this approach, where typing is decided via a proposed set of inference rules. Accordingly, as pointed out in [Poi87], this does not allow to treat bounded data structures.

4 Label algebras

All these considerations have been our main motivation to develop the new framework of *label algebras*. The rest of this paper is devoted to define and study label specifications, label algebras and their applications.

4.1 About values, terms and labels

Usually, algebras are (heterogeneous) sets of values [GTW78][EM85]. A signature is usually a couple $\Sigma = <S, F>$ where S is a finite set of sorts (or type names) and F is a finite set of operation names with arity in S; the objects (algebras) of the category $Alg(\Sigma)$ are heterogeneous sets, A, partitioned as $A = \{A_s\}_{s \in S}$, and with, for each operation name "$f : s_1 \dots s_n \rightarrow s$" in Σ $(0 \leq n)$, a total function $f_A : A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s$; the morphisms of $Alg(\Sigma)$ (Σ-morphisms) being obviously the sort preserving, operation preserving applications.

As a consequence of our approach, labelled terms are also considered as "first class citizen objects." Given an algebra A, the satisfaction of a property is defined using terms (instead of the usual definition which only involves values). A simple idea could be to consider both A and T_Σ (the ground term algebra over the signature Σ) when defining the satisfaction relation. Unfortunately, such an approach does not allow satisfactory treatments of the non finitely generated algebras, i.e. algebras such that the initial Σ-morphism from T_Σ to A is not surjective. How is one to deal with both terms and non reachable values ? The algebra $T_\Sigma(A)$ allows us to consider both terms and non reachable values, let us remember its definition.

Given a heterogeneous "set of variables" $V = \{V_s\}_{s \in S}$, the *free Σ-term algebra with variables in V* is the least Σ-algebra $T_\Sigma(V)$ (with respect to the preorder induced by the Σ-morphisms) such that $V \subseteq T_\Sigma(V)$.

Since V is not necessarily finite or countable, we can consider in particular $T_\Sigma(A)$ for every algebra A. An element of $T_\Sigma(A)$ is a Σ-term such that each leaf can contain either a constant of the signature, or a value of A. For example, if $A = \mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ is the algebra of all integers over the signature $<\{Nat\}, \{zero :\rightarrow Nat; succ\_, pred\_ : Nat \rightarrow Nat\}>$, then $succ(succ(zero))$, $succ(succ(0))$, $succ(1)$, etc. are *distinct* elements of $T_\Sigma(\mathbb{Z})$, even though they have the same value when evaluated in $\mathbb{Z}$.

The main technical point underlying our framework is to systematically use $T_\Sigma(A)$ directly inside the *label algebras* in order to have a refined treatment of labelling. For example, $Maxint$ and $succ(Maxint)$, are distinct elements of $T_\Sigma(A)$ (only $succ(Maxint)$ being exceptional, labelled by $TooLarge$). This allows us to have a very precise definition of the satisfaction relation, using assignments with range in $T_\Sigma(A)$ instead of A.

Intuitively, a term represents the "history" of a value; it is a "sequence of calculations" which results in a value. Of course, several histories can provide the same value. This is the reason why labelling is more powerful than typing: it allows to "diagnose" the history in

order to apply a specific treatment or not. Nevertheless, we can relate each term to its final value via the canonical evaluation morphism:

$$eval_A : T_\Sigma(A) \longrightarrow A$$

deduced from the Σ-algebra structure of A:

$$\forall a \in A,\ eval_A(a) = a \quad \text{(remember that } A \subseteq T_\Sigma(A)\text{)}$$
$$\forall f \in \Sigma,\ \forall t_1 \ldots t_n \in T_\Sigma(A),\ eval_A(f(t_1, \ldots, t_n)) = f_A(eval_A(t_1), \ldots, eval_A(t_n))$$

Of course, *in the end*, the satisfaction of an equality is checked on values; thus, $eval_A$ is a crucial tool for defining the satisfaction relation on equational atoms. However, the considered assignments can be precisely restricted to certain kinds of terms/histories *before* checking equalities on values (via conditional axioms), and this is the reason why all the inconsistencies mentioned above can be solved via label algebras.

We shall use the following simplified notations:

Notation 4.1 : Given a Σ-algebra A, $T_\Sigma(A)$ will be denoted by $\overline{A}$. Moreover, let $\mu : A \to B$ be a Σ-morphism, $\overline{\mu} : \overline{A} \to \overline{B}$ denotes the unique canonical Σ-morphism which extends μ to the corresponding free algebras. Let us note that: $\mu \circ eval_A = eval_B \circ \overline{\mu}$.

4.2 Basic definitions

Definition 4.2 : *A label signature* is a triple $\Sigma L = <S, F, L>$ where $\Sigma = <S, F>$ is a (usual) signature and L is a (finite) set of *labels*.

Definition 4.3 : Given a label signature $\Sigma L = <\Sigma, L>$, a ΣL-algebra $\mathcal{A}$ is a couple $(A, \{l_A\}_{l \in L})$ where:

- A is a Σ-algebra,
- $\{l_A\}_{l \in L}$ is a L-indexed family such that, for each l in L, l_A is a subset of $\overline{A}$.

Notice that there are no conditions about the subsets l_A: they can intersect several sorts, they are not necessarily disjoint and their union $(\bigcup_{l \in L} l_A)$ does not necessarily cover $\overline{A}$.

Example 4.4 : Let $<\{Nat\}, \{zero :\to Nat, succ\_ : Nat \to Nat\}, \{TooLarge\}>$ be a label signature for the natural numbers. An example of label algebra $\mathcal{A} = <A, \{TooLarge\}>$ can be defined on this signature as follows:

- A is the interval $[0..Maxint]$ of $\mathbb{N}$ with $zero_A = 0$, $succ_A(i) = i + 1$ for i in $[0..Maxint[$ and $succ_A(Maxint) = Maxint$.
  Then $\bar{A} = \{succ^i(a) \mid i \in \mathbb{N}, a \in A \cup \{zero\}\}$.
- $TooLarge_A = \{succ^i(a) \mid i \in N, a \in A \cup \{zero\}, i + eval_A(a) = Maxint + 1\}$ (It would have been also possible to choose $i + eval_A(a) \geq Maxint + 1$).

Notice that in this example, we illustrate the fact that two terms having the same value may not be labelled in the same manner. Thus, the terms $succ(Maxint)$ and $Maxint$ have the same value, but $succ(Maxint)$ is labelled by $TooLarge$ while $Maxint$ is not.

The label $TooLarge$ serves to mark the terms which are an overstepping of the bound $Maxint$.

Definition 4.5 : Let $\mathcal{A} = (A, \{l_A\}_{l \in L})$ and $\mathcal{B} = (B, \{l_B\}_{l \in L})$ be two ΣL-algebras, a ΣL-morphism $h : \mathcal{A} \to \mathcal{B}$ is a Σ-morphism from A to B such that $\overline{h} : \overline{A} \to \overline{B}$ preserves the labels: $\forall l \in L,\ \overline{h}(l_A) \subseteq l_B$.

When there is no ambiguity about the signature under consideration, ΣL-algebras and ΣL-morphisms will be called *label algebras* and *label morphisms*, or even algebras and morphisms. Given a label signature ΣL, label algebras and label morphisms (with the usual composition) clearly form a category:

Definition 4.6 : The category of all ΣL-algebras is denoted by $Alg_{Lbl}(\Sigma L)$.

Definitions 4.7 : Let $\Sigma L = <\Sigma, L>$ be a label signature. Let $\mathcal{A} = (A, \{l_A\}_{l \in L})$ be a ΣL-algebra.

- A ΣL-*relation* with labels (or *label relation*) on $\mathcal{A}$ is a couple $\mathcal{R} = (R, \{l_R\}_{l \in L})$ where R is a binary relation on A compatible with the sorts[1] and $\{l_R\}_{l \in L}$ is a family of subsets of $\overline{A}$.
- A ΣL-congruence (or *label congruence*) is a ΣL-relation $\Theta = (\equiv_\Theta, \{l_\Theta\}_{l \in L})$ such that $\equiv_\Theta$ is a usual Σ-congruence on A and $l_A \subseteq l_\Theta$ for each l in L.

Proposition 4.8 : Let $\mathcal{A} = (A, \{l_A\}_{l \in L})$ be a ΣL-algebra and let $\Theta = (\equiv_\Theta, \{l_\Theta\}_{l \in L})$ be a ΣL-congruence. Let $A/_\Theta$ be the usual quotient Σ-algebra of A by the Σ-congruence $\equiv_\Theta$ and $q : A \to A/_\Theta$ the corresponding quotient Σ-morphism. Let $\{l_{A/_\Theta}\}_{l \in L}$ be defined by $l_{A/_\Theta} = \overline{q}(l_\Theta)$ for each l in L.
The couple $(A/_\Theta, \{l_{A/_\Theta}\}_{l \in L})$ is a ΣL-algebra, denoted by $\mathcal{A}/_\Theta$, and q is a label morphism. This label algebra is called the *quotient algebra* of $\mathcal{A}$ by Θ.

(The proof is immediate)

Notations 4.9 : Let ΣL be a label signature.

- Given a set of variables V, $\mathcal{T}_{\Sigma L}(V)$ is the ΣL-algebra such that the underlying Σ-algebra is the term algebra $T_\Sigma(V)$ and for each l in L, $l_{T_\Sigma(V)}$ is empty.
- $\mathcal{T}_{\Sigma L}$ is defined by $\mathcal{T}_{\Sigma L} = \mathcal{T}_{\Sigma L}(\emptyset)$ and is called the ground term ΣL-algebra.
- $\mathcal{T}riv$ is the trivial ΣL-algebra defined by the underlying Σ-algebra $Triv$ which contains only one element in $Triv_s$ for each s in S, and for each l in L, $l_{\mathcal{T}riv} = \overline{Triv}$.

The ΣL-algebra $\mathcal{T}_{\Sigma L}$ (resp. $\mathcal{T}riv$) is clearly initial (resp. terminal) in $Alg_{Lbl}(\Sigma L)$. Moreover, as usual, a ΣL-algebra $\mathcal{A}$ is called *finitely generated* if and only if the initial ΣL-morphism from $\mathcal{T}_{\Sigma L}$ to $\mathcal{A}$ is an epimorphism. It is clear from the definitions that $\mathcal{A}$ is finitely generated if and only if the underlying morphism from T_Σ to A is surjective, which means that every value of A is reachable by a ground term.

Definitions 4.10 : The full subcategory of $Alg_{Lbl}(\Sigma L)$ containing all the finitely generated algebras is denoted by $Gen_{Lbl}(\Sigma L)$. Moreover, the signature ΣL is said *sensible* if and only if $\mathcal{T}riv$ belongs to $Gen_{Lbl}(\Sigma L)$.

[1] $R \subseteq \bigcup_{s \in S} A_s \times A_s$ or equivalently R is a family of disjoint binary relations R_s for $s \in S$ with $R_s \subseteq A_s \times A_s$.

The category $Gen_{Lbl}(\Sigma L)$ has the same initial object as $Alg_{Lbl}(\Sigma L)$, and if ΣL is sensible (i.e. if there exists at least one ground term of each sort) then it has the same terminal object too.

Not surprisingly, a "label specification" will be defined by a (label) signature and a set of well formed formulae (axioms):

Definition 4.11 : Given a label signature ΣL, a ΣL-*sentence* (or *axiom*) is a well formed formula built on:

- *equational atoms* of the form $(u = v)$ where u and v are Σ-terms with variables, u and v belonging to the same sort,
- *labelling atoms* of the form $(w \epsilon l)$ where w is a Σ-term with variables and l is a label belonging to L,
- *connectives* belonging to $\{\neg, \wedge, \vee, \Rightarrow\}$ and *quantifiers* belonging to $\{\forall, \exists\}$.

(Every variable is implicitly universally quantified.)

The predicate " ϵ " should be read "*is labelled by*".

Definition 4.12 : A *label specification* is a pair $SP = <\Sigma L, Ax>$ where ΣL is a label signature and Ax is a set of ΣL-sentences.

The *satisfaction relation* is the main definition of this section. It is of first importance to remark that we consider assignments with range in $\overline{A} = T_\Sigma(A)$ (terms) instead of A (values):

Definition 4.13 : Let $\mathcal{A} = (A, \{l_A\}_{l \in L})$ be a ΣL-algebra.

- Let u and v be two terms of the same sort in $T_\Sigma(V)$. Let $\sigma : V \to \overline{A}$ be an assignment (V covering all the free variables of u and v). $\mathcal{A}$ satisfies $(u = v)$ with respect to σ (i.e. $\mathcal{A} \models_\sigma (u = v)$) means that $eval_A(\sigma(u)) = eval_A(\sigma(v))$ [$eval_A$ being the canonical evaluation morphism from $\overline{A}$ to A and the symbol "=" being the set-theoretic equality in the carrier of A].
- Let $w \in T_\Sigma(V)$, $l \in L$ and let $\sigma : V \to \overline{A}$ be an assignment (V covering all the free variables of w). $\mathcal{A}$ satisfies $(w \epsilon l)$ with respect to σ (i.e. $\mathcal{A} \models_\sigma (w \epsilon l)$) means that $\sigma(w) \in l_A$ [the symbol "$\in$" being the set-theoretic membership].
- $\mathcal{A}$ satisfies a ΣL-sentence φ (i.e. $\mathcal{A} \models \varphi$) if and only if for all assignments $\sigma : V \to \overline{A}$ (V covering all the free variables of φ), $\mathcal{A} \models_\sigma \varphi$.
- Given two ΣL-sentences φ_1 and φ_2, $\mathcal{A}$ satisfies $(\varphi_1 \wedge \varphi_2)$ if and only if $\mathcal{A}$ satisfies φ_1 and $\mathcal{A}$ satisfies φ_2.
- Given a ΣL-sentence φ, $\mathcal{A}$ satisfies $(\neg\varphi)$ if and only if for all assignments $\sigma : V \to \overline{A}$ (V covering all the free variables of φ), $\mathcal{A}$ does not satisfies φ with respect to σ.
- Given a ΣL-sentence φ, $\mathcal{A}$ satisfies $(\forall x, \varphi)$ if and only if $\mathcal{A}$ satisfies φ.
- Given a ΣL-sentence φ, $\mathcal{A}$ satisfies $(\exists x, \varphi)$ if and only if there exists a term $t \in \overline{A}$ such that $\mathcal{A}$ satisfies φ with respect to all assignments $\sigma : V \to \overline{A}$ such that $\sigma(x) = t$ (V covering all the free variables of φ).
- Similar rules apply for axioms of the form $(\varphi_1 \vee \varphi_2)$ or $(\varphi_1 \Rightarrow \varphi_2)$ where φ_i are ΣL-sentences.

A label algebra satisfies a label specification if and only if it satisfies all its axioms.

Example 4.14 : Let us consider the label signature of natural numbers given in Example 4.4. We can consider the following axiom (mentioned in Example 3.2):

$$succ(n) \,\epsilon\, TooLarge \;\Rightarrow succ(n) = n$$

The label algebra $\mathcal{A}$ defined in Example 4.4 satisfies this axiom. In this algebra, the term $succ^{Maxint+1}(0)$ is labelled by *TooLarge* while the term $succ^{Maxint}(0)$ is not. Thus the assignment $[n \leftarrow succ^{Maxint}(0)]$ satisfies the premises while $[n \leftarrow succ^{Maxint-1}(0)]$ does not, even though $succ(n)$ gets the same value in both cases.

Given a label specification *SP*, the full subcategory of $Alg_{Lbl}(\Sigma L)$ containing all the algebras satisfying *SP* is denoted by $Alg_{Lbl}(SP)$. (A similar notation holds for Gen_{Lbl}.)

Notice that $Alg_{Lbl}(SP)$ or $Gen_{Lbl}(SP)$ can be empty categories (for example when *SP* contains φ and $\neg\varphi$). Provided that the axioms of *SP* never contain the connective "$\neg$", $Alg_{Lbl}(SP)$ has the same terminal object as $Alg_{Lbl}(\Sigma L)$: $\mathcal{T}riv$. However, as usual, initiality results can be easily obtained only for positive conditional specifications [WB80]. These results are provided in Section 5.

Definition 4.15 : A ΣL-sentence is called *positive conditional* if and only if it is of the form:

$$a_1 \wedge \ldots \wedge a_n \;\Rightarrow\; a$$

where the a_i and a are (positive) atoms (if $n = 0$ then the sentence is reduced to a). A specification is called *positive conditional* if and only if all its axioms are positive conditional sentences.

4.3 Some applications of label algebras

Although we have introduced the theory of label algebras as a general framework for the purpose of exception handling, it can also be used for many other purposes. We have mentioned so far that labels can be used to represent exception names. More generally, labels provide a great tool to express several other features already developed in the field of (first order) algebraic specifications. In this section, we outline some possible applications of the framework of label algebras.

We have mentioned in Section 3.6 that the framework of label algebras can be shown as an extension of more standard algebraic approaches based on "multityping." More precisely, we can *specify multityping* by means of label specifications. The difference between a label and a type is that labels are carried by terms (in $\overline{A}$) while type names are carried by values (in A). However, a label l can easily play the role of a type name: it is sufficient to saturate each fiber of $eval_A : \overline{A} \to A$ which contains a term labelled by l. This is easily specified by a ΣL-sentence of the form:

$$x \,\epsilon\, l \;\wedge\; x = y \;\Longrightarrow\; y \,\epsilon\, l$$

where x and y are variables. For every model A satisfying such axioms for every l belonging to L, two terms u and v of $\overline{A}$ having equal values in A are necessarily labelled by the same labels, thus labels can play the role of types. Notice that we should write one axiom of

this form for each sort belonging to S because the variables x and y are typed with respect to S in our framework. Nevertheless, insofar as we intend to simulate types by labels, S should be a singleton. Thus, the "typing" of terms, as well as variables, becomes explicit in the precondition of each axiom. This approach leads to consider typing as "membership constraint." (see section 3.7).

An advantage of such an approach is that additional properties about types, according to the needs of the considered application, can be easily specified within the same framework. For example, let us consider a property such as $s \leq s'$ between two sorts in the framework of *order sorted algebras* [FGJM85]. It can be specified within the framework of label specifications:

$$x \epsilon s \implies x \epsilon s'$$

where s and s' are labels which simulate the corresponding (sub)sorts.

In the same way, it is possible to specify *dependent types* such as binary search tree (the specifications of natural numbers and booleans are supposed already written):

$S =$ $\{All\}$
$\Sigma =$ $\{empty :\rightarrow All; node : All\ All\ All \rightarrow All; root, max, min : All \rightarrow All\}$
$L =$ $\{Bool, Nat, Notdefined, Bst, Sta, Gta\}$
(Bst for Binary Search Tree; Sta for $Smaller\text{-}Than\text{-}All$ and Gta for $Greater\text{-}Than\text{-}All$)

with the following axioms, under initial semantics:

$empty \epsilon Bst$
$max(empty) \epsilon Sta$
$min(empty) \epsilon Gta$
$x \epsilon Sta \wedge n \epsilon Nat \implies x \leq n = true$
$x \epsilon Gta \wedge n \epsilon Nat \implies n \leq x = true$
$a \epsilon Bst \wedge b \epsilon Bst \wedge n \epsilon Nat \wedge max(a) \leq n = true \wedge n \leq min(b) = true \implies node(a, n, b) \epsilon Bst$
$root(empty) \epsilon Notdefined$
$node(a, n, b) \epsilon Bst \implies root\ (node(a, n, b)) = n$

Algebraic specifications with *partial functions* can also be represented by label specifications. Algebraic specifications for partial algebras often rely on an additional predicate D which is used to specify the definition domain of each operation of the signature ([BW82] and others). Thus, atoms are either equalities, or of the form $D(t)$, where t is a term with variables. It is of course not difficult to translate $D(t)$ to $(t \epsilon IsDefined)$; we simply have to specify the propagation of the definition domains with respect to any operation f of the signature:

$$f(x_1, \ldots, x_n) \epsilon IsDefined \implies x_1 \epsilon IsDefined \wedge \ldots \wedge x_n \epsilon IsDefined$$

Then, the label $IsDefined$ can be used in the preconditions of the axioms defining the partial operations in such a way that every label algebra $\mathcal{A}$ satisfying the resulting label specification has the property that $eval_A(IsDefined_A)$ is a subset of A that behaves like a partial algebra satisfying the original specification (see also [AC91]).

In the same way, labels can be used to give a refined semantics of the predefined *predicates of specification languages*. For example in the language PLUSS [Bid89] [Gau92], an expression of the form "t `is defined when` *something*" can be specified by the following label axiom:

$$something \implies t \epsilon IsDefined$$

More generally, labels are simply unary predicates on terms; thus, they can at least be used as *predicates* on values without any exception handling connotation. The advantage of such predicates is that their semantics is not defined via a hidden boolean sort: using booleans to define predicates is often unsatisfactory because it assumes that the specification is consistent with respect to boolean values. An example is given below:

$0 \epsilon \mathit{Even}$
$n \epsilon \mathit{Even} \ \Rightarrow \ \mathit{succ}(n) \epsilon \mathit{Odd}$
$n \epsilon \mathit{Odd} \ \Rightarrow \ \mathit{succ}(n) \epsilon \mathit{Even}$
$\mathit{exp}(n, 0) = \mathit{succ}(0)$
$\mathit{succ}(m) \epsilon \mathit{Odd} \ \Rightarrow \ \mathit{exp}(n, \mathit{succ}(m)) = \mathit{exp}(n, m) \times n$
$m \in \mathit{Even} \ \Rightarrow \ \mathit{exp}(n, m) = \mathit{exp}(n \times n \ , \ m \ \mathit{div} \ \mathit{succ}(\mathit{succ}(0)) \ )$

Another possible application of the framework of label algebras is the one of algebraic specifications with *observability* issues. A crucial aspect of observational specifications is that "what is observable" has to be carefully specified. It is often very difficult to prove that two values are observationally equal (while it is sufficient to exhibit two observations which distinguish them to prove that they are distinct). In [Hen89], R. Hennicker uses a predicate *Obs* to characterize the observable values. This powerful framework leads to legible specifications and it provides some theorem proving methods. Of course, the predicate *Obs* can be represented by a label. Moreover, it has been shown in [BB91] that there are some specifications which are inconsistent when observability is carried by values. It is shown that these inconsistencies can be avoided when observability is expressed with respect to a subset $\Sigma \mathit{Obs}$ of the signature Σ (leading consequently to a subset of the *terms* instead of *values*). The framework of [BB91] introduces two distinct notions that induce a hierarchy in the definition of observability. The terms that only contain operations belonging to $\Sigma \mathit{Obs}$ are said to "allow observability" (the other ones can never be observed). Then, a term "allowing observability" really becomes "observable" only if it belongs to an observable sort. It is not difficult to specify the observational hierarchy defined in [BB91] by using two distinct labels denoted *AllowsObs* and *Obs*. For each operation f allowing observability (i.e. belonging to the considered subset $\Sigma \mathit{Obs}$ of the signature), it is sufficient to consider the following label axiom:

$$x_1 \epsilon \mathit{AllowsObs} \ \wedge \ldots \wedge \ x_n \epsilon \mathit{AllowsObs} \implies f(x_1, \ldots, x_n) \epsilon \mathit{AllowsObs}$$

The fact that a term allowing observability becomes observable if and only if it belongs to an observable sort s can easily be specified by the label axiom (one axiom for each observable sort):

$$x \epsilon \mathit{AllowsObs} \implies x \epsilon \mathit{Obs}$$

where x is a variable of sort s. Hopefully, the advantages of Hennicker's approach are preserved, since they mainly rely on the explicit specification of the predicate *Obs*.

Summing up, the framework of label algebras is clearly not directly usable by a "working specifier." All the possible applications mentioned above require some generic label axioms which are *implicit*. These axioms should be considered as modifiers of the semantics, in order to preserve *legibility* and *terseness* of the specifications. Thus, the framework of label algebras provides us with "low level" algebraic specifications. When an algebraic specification *SP* is written according to some special semantics (e.g. observational specifications or exception algebras), it has to be "compiled" (translated) to a label specification $Tr(\mathit{SP})$.

5 Fundamental results

5.1 Initiality results

This section deals with initiality results for positive conditional label specifications. We show that the classical results of [GTW78] can be extended to the framework of label algebras. The important results of this section are mainly the theorems 5.1, 5.4 and 5.12. The other results of this section, and all the proofs, can be skipped in a first reading.

We will first prove the following fundamental technical result.

Theorem 5.1 : Let *SP* be a positive conditional ΣL-specification. Let $\mathcal{X} = (X, \{l_X\}_{l \in L})$ be a ΣL-algebra. Let $\mathcal{R} = (R, \{l_R\}_{l \in L})$ be a label relation over $\mathcal{X}$. There is a least *SP*-algebra $\mathcal{Y}$ (according to the preorder relation induced by the label morphisms) such that:

1. there exists a label morphism $h_Y : \mathcal{X} \to \mathcal{Y}$;
2. $(\mathcal{Y}, h_Y)$ is compatible with $\mathcal{R}$ (i.e. $\forall x, y \in X,\ x\ R\ y \implies h_Y(x) = h_Y(y)$ and $\forall t \in T_\Sigma(X), x \in l_X \implies \overline{h_Y}(t) \in l_Y$).

Proof : Let F be the family of all $(\mathcal{Z}\ ,\ h_Z : \mathcal{X} \to \mathcal{Z})$, where $\mathcal{Z}$ is a *SP*-algebra and h_Z is a label morphism such that $(\mathcal{Z}, h_Z)$ satisfies the conditions (1) and (2) of the theorem. F is not empty because $\mathcal{T}riv$ (with the unique trivial morphism from $\mathcal{X}$ to $\mathcal{T}riv$) clearly belongs to F. Thus, we can consider the ΣL-congruence $\Theta_F = (\equiv_F, \{l_F\}_{l \in L})$ defined as follows:

- $\forall x, y \in X,\ (\ x \equiv_F y \ \Leftrightarrow\ (\forall (\mathcal{Z}, h_Z) \in F,\ h_Z(x) = h_Z(y))\ )$
- $\forall l \in L,\ \forall x \in \overline{X},\ (\ x \in l_F \ \Leftrightarrow\ (\forall (\mathcal{Z}, h_Z) \in F,\ \overline{h_Z}(x) \in l_Z)\ )$

Let us remark that $\equiv_F$ is clearly a Σ-congruence (the compatibility with the operations of Σ results from the one of all the h_Z such that $(\mathcal{Z}, h_Z)$ belongs to F). Let us also remark that, for the same reason, l_F contains l_X for each l. Thus, Θ_F is a ΣL-congruence on $\mathcal{X}$. Let $\mathcal{Y}$ be the quotient algebra $\mathcal{X}/_{\Theta_F}$ and h_Y the corresponding quotient label morphism[1].

Moreover, for every $\mathcal{Z}$ such that $(\mathcal{Z}, h_Z)$ is in F, there exists a ΣL-morphism μ_Z from $\mathcal{Y}$ to $\mathcal{Z}$: it is defined by $\forall x \in X,\ \mu_Z(h_Y(x)) = h_Z(x)$ (μ_Z exists by definition of h_Y, and we have $\mu_Z \circ h_Y = h_Z$). Consequently, if $(\mathcal{Y}, h_Y)$ belongs to F then it is its smallest element and the theorem is proved. It is trivial from the definition of $(\mathcal{Y}, h_Y)$ that it satisfies the conditions (1) and (2) of the theorem. Thus it is sufficient to prove that $\mathcal{Y}$ satisfies *SP*. It is the purpose of the next lemma. ♢

Lemma 5.2 : $\mathcal{Y}$ (as defined in the proof of Theorem 5.1) satisfies each axiom of *SP*.

Proof : Let $(\ a_1 \wedge \ldots \wedge a_n \ \Rightarrow\ a\ )$ be an axiom of *SP* (a_i and a being positive atoms). Let $\sigma : V \to \overline{Y}$ be any assignment covering all the variables of the axiom. By definition of $\mathcal{Y}$, we have:

$$(\forall i = 1..n,\ \mathcal{Y} \models \sigma(a_i)) \iff (\forall (\mathcal{Z}, h_Z) \in F,\ (\forall i = 1..n,\ (\mathcal{Z} \models \overline{\mu_Z}(\sigma(a_i)))))$$

[1] Θ_F is indeed the kernel of h_Y

Since all $\mathcal{Z}$ such that $(\mathcal{Z}, h_Z) \in F$ satisfy SP, it comes:

$$(\forall i = 1..n,\ \mathcal{Y} \models \sigma(a_i)) \implies (\forall \mathcal{Z} \in F,\ \mathcal{Z} \models \overline{\mu_Z}(\sigma(a)))$$

By definition of $\mathcal{Y}$, we get:

$$(\forall i = 1..n,\ \mathcal{Y} \models \sigma(a_i)) \implies \mathcal{Y} \models \sigma(a)$$

and we obtain that $\mathcal{Y}$ satisfies the axiom under consideration. It proves the lemma, and concludes the proof of Theorem 5.1. ♢

The following lemma shows a universal property of $\mathcal{Y}$.

Lemma 5.3 : With the notations of Theorem 5.1, for every SP-algebra $\mathcal{Z}$ satisfying conditions (1) and (2), there exists a unique morphism $\mu_Z : \mathcal{Y} \to \mathcal{Z}$ such that $\mu_Z \circ h_Y = h_Z$.

Proof : Existence: already been proved; unicity: from the surjectivity of h_Y. ♢

Theorem 5.4 : Let SP be a positive conditional label specification.

The categories $Alg_{Lbl}(SP)$ and $Gen_{Lbl}(SP)$ have an initial object, denoted $\mathcal{T}_{SP}$. Moreover, $\mathcal{T}riv$ is final in $Alg_{Lbl}(SP)$ (and in $Gen_{Lbl}(SP)$ if the signature is sensible).

Proof : The assertion about $\mathcal{T}riv$ is trivial. The label algebra $\mathcal{T}_{SP}$ is obtained by applying Theorem 5.1 with $\mathcal{X} = \mathcal{T}_{\Sigma L}$ and $\mathcal{T}_{SP} = \mathcal{Y}$, R being the empty binary relation. ♢

The purpose of the remainder of this subsection is to study *structured* positive conditional label specifications. We define the *forgetful functor* U_μ associated with a structured specification and the *synthesis functor* F_μ; and we prove that F_μ is left adjoint to U_μ.

Definition 5.5 : Let ΣL_1 and ΣL_2 be two label signatures. Let $\mu : \Sigma L_1 \to \Sigma L_2$ be a signature morphism.[2] The forgetful functor $U_\mu : Alg_{Lbl}(\Sigma L_2) \to Alg_{Lbl}(\Sigma L_1)$ is defined as follows:

- for each ΣL_2-algebra $\mathcal{A}$, $U_\mu(\mathcal{A})$ is the ΣL_1-algebra $\mathcal{B}$ defined by:
  $\forall s \in S_1,\ B_s = A_{\mu(s)}$; $\forall l \in L_1,\ l_B = \mu(l)_A \cap \overline{B}$; and $\forall f \in \Sigma_1,\ f_B = \mu(f)_A$;
- for each ΣL_2-morphism $\eta : \mathcal{A} \to \mathcal{A}'$, $U_\mu(\eta) : U_\mu(\mathcal{A}) \to U_\mu(\mathcal{A}')$ is the ΣL_1-morphism $U_\mu(\eta)$ defined by all the restrictions of η of the form:
  $U_\mu(\eta)_s = \eta_{\mu(s)} : A_{\mu(s)} \to A'_{\mu(s)}$.

$\overline{U_\mu(\eta)}$ clearly preserves the labels of L_1; thus $U_\mu(\eta)$ is actually a ΣL_1-morphism.

In the sequel, we only consider the signature morphisms which correspond to signature inclusions. The corresponding forgetful functor is then denoted by U. It is not difficult to show that all our results still apply for arbitrary morphisms μ (the injectivity of μ is never used). This simplification allows us to ignore the syntactic transformation induced by μ; it considerably clarifies the sequel of this section.

[2] Signature morphisms are defined in an obvious way: $S_1 \to S_2$, $\Sigma_1 \to \Sigma_2$ and $L_1 \to L_2$

Theorem 5.6 : Let SP_1 and SP_2 be two label specifications such that $SP_1 \subseteq SP_2$. Let U be the forgetful functor from $Alg_{Lbl}(\Sigma L_2)$ to $Alg_{Lbl}(\Sigma L_1)$. The restriction of U to $Alg_{Lbl}(SP_2)$ can be co-restricted to $Alg_{Lbl}(SP_1)$.

More generally, given two signatures $\Sigma L_1 \subseteq \Sigma L_2$, for all ΣL_2-algebras $\mathcal{A}$ and for all ΣL_1-sentences φ we have:

$$\mathcal{A} \models \varphi \implies U(\mathcal{A}) \models \varphi$$

Proof : Let V be the set of variables of φ. We have to prove:

$$(\forall \sigma : V \to \overline{A},\ \mathcal{A} \models \sigma(\varphi)\ ) \implies (\forall \sigma : V \to \overline{U(A)},\ U(\mathcal{A}) \models \sigma(\varphi)\ )$$

Since $\overline{U(A)}$ is included in $\overline{A}$ and the labels are preserved, this implication is trivial. $\diamondsuit$

Remark 5.7 : Theorem 5.6 never requires for the sentence φ to be positive conditional. In particular SP_1 and SP_2 are not necessarily positive conditional specifications.

Let us remark that the reverse implication of Theorem 5.6 is not valid in general, as shown in the following example. Consequently, the so-called "satisfaction condition" does not hold for label algebras; the framework of label algebras is not an institution (see [GB84]), at least with the natural definitions of signature morphisms and sentence translations.

Example 5.8 : Let ΣL_1 be the label signature defined by

$$S_1 = \{\ thesort\ \},\ F_1 = \{\ c1 : \to thesort\ \} \text{ and } L_1 = \{\ thelabel\ \}$$

Let ΣL_2 be the label signature defined by

$$S_2 = S_1,\ F_2 = \{\ c1 : \to thesort\ ,\ c2 : \to thesort\ \} \text{ and } L_2 = L_1.$$

We clearly have $\Sigma L_1 \subset \Sigma L_2$.

Let $\mathcal{A}$ be the ΣL_2-algebra defined by $A = \{a = c1_A = c2_A\}$ (A is a singleton) and $thelabel_A = \{\ a\ ,\ c1\ \}$ (let us remind that $T_{\Sigma_2}(A) = \{\ a\ ,\ c1\ ,\ c2\ \}$).

The ΣL_1-algebra $U(\mathcal{A})$ is then characterized by

$$U(A) = \{a = c1_{U(A)}\} \text{ and } thelabel_{U(A)} = \{\ a\ ,\ c1\ \}$$

thus, $thelabel_{U(A)} = T_{\Sigma_1}(U(A))$.

Consequently, $U(\mathcal{A})$ satisfies the ΣL_1-sentence "$x\ \epsilon\ thelabel$" while $\mathcal{A}$ does not (as $c2$ does not belong to $thelabel_A$).

The following technical notation defines a free algebra which will be useful to define the synthesis functor.

Notation 5.9 : Let ΣL_1 and ΣL_2 be two label signatures such that $\Sigma L_1 \subseteq \Sigma L_2$. Let $\mathcal{A}$ be a ΣL_1-algebra. Let $T_{\Sigma_2}(A)$ be the usual free Σ_2-algebra with variables in A and let $\mathcal{T}_{\Sigma_2}(\mathcal{A})$ be the corresponding ΣL_2-algebra with empty label sets. The ΣL_2-relation with labels $\mathcal{R}_A = (R_A, \{l_R\}_{l \in L})$ is defined as follows:

1. $\forall t, t' \in T_{\Sigma_2}(A),\ (t\ R_A\ t') \Leftrightarrow (t \in T_{\Sigma_1}(A)) \wedge (t' \in T_{\Sigma_1}(A)) \wedge (eval_A(t) = eval_A(t'))$
2. Let $\alpha : A \rightarrow T_{\Sigma_2}(A)$ be the inclusion of A into $T_{\Sigma_2}(A)$.
   Let $\overline{\alpha} : T_{\Sigma_1}(A) \rightarrow T_{\Sigma_1}(T_{\Sigma_2}(A))$ be the canonical Σ_1-morphism which extends α.
   Let $\iota : T_{\Sigma_1}(T_{\Sigma_2}(A)) \rightarrow T_{\Sigma_2}(T_{\Sigma_2}(A))$ be the canonical inclusion deduced from the inclusion $\Sigma_1 \subseteq \Sigma_2$.
   Let (finally) $i : T_{\Sigma_1}(A) \rightarrow T_{\Sigma_2}(T_{\Sigma_2}(A))$ be the composition of $\overline{\alpha}$ and ι.
   For every label $l \in L_1$, the set l_R is the subset of $T_{\Sigma_2}(T_{\Sigma_2}(A))$ defined by $l_R = i(l_A)$.
3. For every label $l \in (L_2 - L_1)$, $l_{T_{\Sigma_2}(A)}$ is empty.

$\mathcal{T}_{\Sigma_2}(\_)$ is sometimes called the free functor w.r.t Σ_2 over Σ_1.

Definition 5.10 : Let SP_1 and SP_2 be two positive conditional label specifications such that $SP_1 \subseteq SP_2$. Let $\mathcal{A}$ be a SP_1-algebra and let us consider the label relation $\mathcal{R}_A$ defined in Notation 5.9. By definition, $F(\mathcal{A})$ is the least SP_2-algebra such that:

1. there exists a morphism $h_A : \mathcal{T}_{\Sigma_2}(\mathcal{A}) \rightarrow F(\mathcal{A})$;
2. $(F(\mathcal{A}), h_A)$ is compatible with $\mathcal{R}_A$.

($F(\mathcal{A})$ exists, from Theorem 5.1.)

Theorem 5.11 : With the notations of Definition 5.10, for each SP_1-morphism $\nu : \mathcal{A} \rightarrow \mathcal{A}'$, let the SP_2-morphism $F(\nu)$ be defined as follows:

- let $\overline{\nu}$ be the canonical ΣL_2-morphism from $\mathcal{T}_{\Sigma_2}(\mathcal{A})$ to $\mathcal{T}_{\Sigma_2}(\mathcal{A}')$ deduced from ν. Let $h = h_{A'} \circ \overline{\nu}$ (from $\mathcal{T}_{\Sigma_2}(\mathcal{A})$ to $F(\mathcal{A}')$).
- $(F(\mathcal{A}'), h)$ satisfies the conditions (1) and (2) with respect to $\mathcal{A}$. Consequently there exists a unique morphism $\mu_{F(\mathcal{A}')} : F(\mathcal{A}) \rightarrow F(\mathcal{A}')$ such that $h = \mu_{F(\mathcal{A}')} \circ h_A$ (cf. Lemma 5.3).
- By definition, $F(\nu) = \mu_{F(\mathcal{A}')}$.

Then, F is a functor from $Alg_{Lbl}(SP_1)$ to $Alg_{Lbl}(SP_2)$.

Proof : We have to show that $F(\nu' \circ \nu) = F(\nu') \circ F(\nu)$ for all SP_1-morphisms $\nu' : \mathcal{A}' \rightarrow \mathcal{A}''$ and $\nu : \mathcal{A} \rightarrow \mathcal{A}'$. This directly results from $\overline{\nu' \circ \nu} = \overline{\nu'} \circ \overline{\nu}$ and from the unicity of the morphism $\mu_F(\mathcal{A}'') : F(\mathcal{A}) \rightarrow F(\mathcal{A}'')$, which is by definition equal to $F(\nu' \circ \nu)$. $\diamondsuit$

Theorem 5.12 : Let SP_1 and SP_2 be two positive conditional label specifications such that $SP_1 \subseteq SP_2$.

The *synthesis functor* $F : Alg_{Lbl}(SP_1) \rightarrow Alg_{Lbl}(SP_2)$ is a left adjoint[3] for the forgetful functor $U : Alg_{Lbl}(SP_2) \rightarrow Alg_{Lbl}(SP_1)$.

Proof : Let $\mathcal{A}$ be a SP_1-algebra. Let $\alpha : A \rightarrow T_{\Sigma_2}(A)$ be the inclusion of A into $T_{\Sigma_2}(A)$. Let $I_A : A \rightarrow F(\mathcal{A})$ denote the composition of α and h_A: $I_A = h_A \circ \alpha$. Let us remark that I_A can be co-restricted to $U(F(\mathcal{A}))$, as A only contains values of sort belonging to S_1. Since h_A

[3]Following a classical terminology in mathematics, notice that the synthesis functor is not a free functor as it includes some quotients (w.r.t. the axioms of SP_2).

is compatible with the relation $\mathcal{R}_A$ (as defined in Notation 5.9), I_A is compatible with the operations of Σ_1 and with the labels of L_1. Consequently I_A is a ΣL_1-morphism from $\mathcal{A}$ to $U(F(\mathcal{A}))$.

From the Yoneda lemma [BW90], it results that it suffices to prove that $(F(\mathcal{A}), I_A)$ is a universal arrow to the forgetful functor U. This means that for all SP_2-algebras $\mathcal{B}$ and all SP_1-morphism $\eta : \mathcal{A} \to U(\mathcal{B})$, there exists a unique SP_2-morphism $\eta' : F(\mathcal{A}) \to \mathcal{B}$ such that $\eta = U(\eta') \circ I_A$.

Let us first remark that there exists a unique ΣL_2-morphism $h_B : \mathcal{T}_{\Sigma_2}(\mathcal{A}) \to \mathcal{B}$ which extends η. Moreover, $(\mathcal{B}, h_B)$ satisfies the conditions (1) and (2) of Definition 5.10. From Lemma 5.3, there exists a unique ΣL_2-morphism η' from $F(\mathcal{A})$ to $\mathcal{B}$ such that $h_B = \eta' \circ h_A$.

It comes $h_B \circ \alpha = \eta' \circ h_A \circ \alpha$. Since h_B is an extension of η, and $h_A \circ \alpha$ an extension of I_A, this equality contains our result: $\eta = U(\eta') \circ I_A$. Moreover, any other morphism ρ satisfying $\eta = U(\rho) \circ I_A$ is then such that $\rho \circ h_A$ is an extension of η. But h_B is the unique extension of η, thus $\rho \circ h_A = h_B$. Finally, the unicity of η' (i.e. $\eta' = \rho$) results from Lemma 5.3. $\diamondsuit$

Remark 5.13 : (For experienced readers...) We showed in this subsection that the framework of label algebras does not form an institution [GB84], even if restricted to positive conditional sentences (cf. Example 5.8). However, it forms a pre-institution with the "rps" property [SS91]. We also proved in [LeG93] that the framework of positive conditional label algebras forms a specification frame which has free constructions [EBO91][EBCO91]. In this paper, we imposed an unnecessary restriction: renaming and non-injective signature morphisms have not been dealt with. We have been motivated by a pedagogical approach. We believe that some of our technical definitions (in particular Notation 5.9) would have been much harder to understand if the signature morphisms had been explicit.

Let us point out that the specification frame of label algebras has *not* amalgamations (as defined in [EBCO91]). The reason *a priori* is that we have shown in Section 4.3 that observational semantics can be modeled within label algebras, and [EBCO91] has proved that observational semantics have not amalgamations in general. It is the same for extensions (at least if we do not restrict the definition of morphisms).

5.2 The label calculus

We show in this section that the label calculus presented below is sound. Moreover we prove that it is complete w.r.t. positive conditional ground formulas.

Definition 5.14 : Given a label signature $\Sigma L = <S, F, L>$ and a heterogeneous set of variables V, the *label calculus* is defined by the following set of inference rules, where Ax denotes a set of positive conditional axioms, a and b denote atoms, Γ denotes a finite associative and commutative conjunction[4] of atoms, t, t_i, u_j and v_j denote Σ-terms with variables, $\rho : V \to T_\Sigma(V)$ denotes a substitution and $f : s_1 \ldots s_n \to s$ denotes any operation of F, u_j and v_j being of sort s_j.

[4]More precisely, the preconditions of label axioms are considered as finite sets of atoms, the symbol $\wedge$ being the insertion in those sets. This exempt the user from explicitly managing associativity and commutativity rules for the conjunction in the inference steps.

Axiom introduction:
if $(\Gamma \Rightarrow a)$ is an axiom of Ax **then** $Ax \vdash (\Gamma \Rightarrow a)$

Tautology:
$Ax \vdash (a \Rightarrow a)$

Monotonicity:
if $Ax \vdash (\Gamma \Rightarrow a)$ **then** $Ax \vdash (\Gamma \wedge b \Rightarrow a)$

Modus Ponens:
if $Ax \vdash (\Gamma \wedge b \Rightarrow a)$ **and** $Ax \vdash (\Gamma \Rightarrow b)$ **then** $Ax \vdash (\Gamma \Rightarrow a)$

Reflexivity:
$Ax \vdash t = t$

Symmetry:
if $Ax \vdash (\Gamma \Rightarrow t_1 = t_2)$ **then** $Ax \vdash (\Gamma \Rightarrow t_2 = t_1)$

Transitivity:
if $Ax \vdash (\Gamma \Rightarrow t_1 = t_2)$ **and** $Ax \vdash (\Gamma \Rightarrow t_2 = t_3)$ **then** $Ax \vdash (\Gamma \Rightarrow t_1 = t_3)$

Replacement:
if, $\forall\, j = [1..n],\ Ax \vdash (\Gamma \Rightarrow u_j = v_j)$ **then** $Ax \vdash (\ \Gamma \Rightarrow f(u_1..u_n) = f(v_1..v_n)\ )$

Substitution:
if $Ax \vdash (\Gamma \Rightarrow a)$ **then** $Ax \vdash (\rho(\Gamma) \Rightarrow \rho(a))$

We recognize classical rules of equational reasoning (taking into account positive conditional formulas) except the Leibniz law (replacement equal by equal). More precisely, for some properties $\mathcal{P}$, the deduction rule $\frac{\mathcal{P}(x)\ ,\ x=y}{\mathcal{P}(y)}$ would not be sound with respect to the semantics of label algebras. We saw that the Leibniz law has not to be satisfied with respect to the label algebra semantics (cf. the algebra $\mathcal{A}$ of Example 6.6 in the Section 6.2 below). There is no rule which specifically concerns labelling, except the rule **Substitution** that constructs new label atoms. On the contrary all other algebraic approaches require specific rules to ensure the Leibniz law (e.g. Equational Typed Logic [MSS89]).

Theorem 5.15 : *(Soundness of the label calculus)*

Let Ax be a set of positive conditional label axioms. Let φ be any positive conditional formula. If the underlying signature is sensible, then we have:

$$[\ Ax \vdash \varphi\ ] \quad \Longrightarrow \quad [\ \forall\, \mathcal{A} \in Alg_{Lbl}(Ax)\,,\ \mathcal{A} \models \varphi\ ]$$

Proof sketch : (The signature has to be sensible because, else, the **Transitivity** rule would not be sound.) We prove the soundness by induction on the proof length. Let us assume that the last rule applied is **Substitution**.

Let $\Gamma \Rightarrow a$ be a formula and $\rho : V \to T_\Sigma(V)$ be a substitution such that $(\rho(\Gamma) \Rightarrow \rho(a))$ is the formula φ. The induction hypothesis is: $\forall \mathcal{A} \in Alg(Ax),\ \mathcal{A} \models (\Gamma \Rightarrow a)$. By definition of the satisfaction relation, it means: $\forall \sigma : V \to \overline{A},\ \mathcal{A} \models (\sigma(\Gamma) \Rightarrow \sigma(a))$. In particular $\forall \sigma' : V \to \overline{A},\ \mathcal{A} \models (\ \sigma'(\rho(\Gamma)) \Rightarrow \sigma'(\rho(a))\ )$ (via $\sigma' \circ \rho = \sigma$). Therefore, by definition of the satisfaction relation, we get: $\mathcal{A} \models (\rho(\Gamma) \Rightarrow \rho(a))$.

We have already pointed out that **Substitution** is the most specific rule to treat labelling; we will not treat the other rules in this article (they behave in a similar way). $\diamondsuit$

In order to prove the completeness of the label calculus for positive conditional ground formulas we follow a proof similar to the Birkhoff's one [Bir35] (also similar to the one of [MSS89]).

Lemma 5.16 : Let $\Sigma = <S, F>$ be a signature. Let A be a S-set such that A_s is never empty. Let V be a S-set of variables. For any substitution $\mu : V \to T_\Sigma(A)$ there exist two substitutions $\mu_1 : V \to T_\Sigma(V)$ and $\mu_2 : V \to A$ such that $\mu = \overline{\mu_2} \circ \mu_1$.
Moreover, they can be chosen in such a way that μ_2 is injective on the variables occuring in all the terms in the image of μ_1. Consequently there exists a map $\mu_3 : A \to V$ such that $\mu_1 = \overline{\mu_3} \circ \mu$.
In addition, if A is a Σ-algebra then the substitution $\mu_2 \circ \mu_1 : V \to A$ is equal to $eval_A \circ \overline{\mu_2} \circ \mu_1$.

(the proof is trivial)

Definition 5.17 : With the notations of Definition 5.14, we note $\Theta_\Gamma = (\equiv_\Gamma, \{l_\Gamma\}_{l \in L})$ the ΣL-relation defined on $\mathcal{T}_{\Sigma L}$ as follows.

- For all terms u and v of $T_\Sigma(V)$, $u \equiv_\Gamma v$ if and only if

  $$\exists u', v' \in T_\Sigma(V)\,,\ \exists \eta : V \to T_\Sigma \text{ s.t. } u = \eta(u'),\ \ v = \eta(v') \text{ and } Ax \vdash (\Gamma \Rightarrow u' = v')$$

- For any term w of $T_\Sigma(T_\Sigma)$, $w \in l_\Gamma$ if and only if

  $$\exists\, w' \in T_\Sigma(V)\,,\ \exists \eta : V \to T_\Sigma(T_\Sigma) \text{ s.t. } w = \eta(w') \text{ and } Ax \vdash (\Gamma \Rightarrow w' \,\epsilon\, l)$$

By notation abuse, we will write:

$$\Theta_\Gamma = \{\ \eta(a) \mid \eta : V \to T_\Sigma(T_\Sigma),\ Ax \vdash (\Gamma \Rightarrow a)\ \}$$

with the convention that η stands for $eval_{T_\Sigma} \circ \eta$ if a is an equational atom.

Notations 5.18 : From **Reflexivity, Symmetry, Transitivity** and **Replacement** the relation $\equiv_\Gamma$ is a usual congruence and, the label sets being empty in $\mathcal{T}_{\Sigma L}$ (Notations 4.9), Θ_Γ is a label congruence. Let $\mathcal{T}_\Gamma = \mathcal{T}_{\Sigma L}/_{\Theta_\Gamma}$ denote the corresponding quotient algebra and $q : \mathcal{T}_{\Sigma L} \to \mathcal{T}_\Gamma$ the quotient morphism.

We also denote $\iota : V \to T_\Sigma(V)$ the inclusion of V in $T_\Sigma(V)$.

Remark 5.19 : For every substitution $\sigma : V \to T_\Sigma(T_\Gamma)$, there exists a substitution ρ from V to $T_\Sigma(T_\Sigma)$ such that $\sigma = \overline{q} \circ \rho$.

Lemma 5.20 : For any conjunction of atoms Γ and for any atom a of Γ, we have $\mathcal{T}_\Gamma \models a$.

Proof : From **Tautology** and **Monotonicity** we get: $Ax \vdash (\Gamma \Rightarrow a)$. Thus, by definition of Θ_Γ for all $\eta : V \to T_\Sigma(T_\Sigma)\,,\ \eta(a) \in \Theta_\Gamma$. Consequently, for all η, we have: $\mathcal{T}_\Gamma \models \overline{q}(\eta(a))$. From Remark 5.19 we deduce that for all σ, we have: $\mathcal{T}_\Gamma \models \sigma(a)$. ♢

Lemma 5.21 : For any ground conjunction of atoms Γ and for any ground atom a, if $\mathcal{T}_\Gamma \models a$ then $Ax \vdash (\Gamma \Rightarrow a)$.

Proof : $\mathcal{T}_\Gamma \models a$ implies that for any $\sigma : V \to T_\Sigma(T_\Gamma)$, $\mathcal{T}_\Gamma$ satisfies $\sigma(a)$. In particular it satisfies $\overline{q}(\overline{\iota}(a))$. Since a does not contain variables, $\overline{q}$ is injective on $\overline{\iota}(a)$, so we have $\overline{\iota}(a) = \eta(a')$ with $Ax \vdash (\Gamma \Rightarrow a')$. Moreover, $a = eval_{T_\Sigma}(\overline{\iota}(a)) = eval_{T_\Sigma}(\eta(a'))$. From **Substitution**, $Ax \vdash (eval_{T_\Sigma}(\eta(\Gamma)) \Rightarrow eval_{T_\Sigma}(\eta(a')))$. Since Γ is a conjunction of ground atoms, we get $Ax \vdash (\Gamma \Rightarrow eval_{T_\Sigma}(\eta(a')))$. ♢

Lemma 5.22 : For any ground conjunction of atoms Γ and for any ground atom a, if $\mathcal{T}_\Gamma \models (\Gamma \Rightarrow a)$ then $Ax \vdash (\Gamma \Rightarrow a)$.

(Trivially results from Lemmas 5.20 and 5.21).

Lemma 5.23 : For any ground conjunction of atoms Γ and for any atom a, if $\mathcal{T}_\Gamma$ satisfies $\sigma(a)$ with $\sigma = \overline{\sigma_2} \circ \sigma_1$ as in Lemma 5.16, then $Ax \vdash (\Gamma \Rightarrow \sigma_1(a))$.

Proof : $\mathcal{T}_\Gamma$ satisfies $\sigma(a)$ means that there exist an atom a' and a substitution $\eta : V \to T_\Sigma(T_\Sigma)$ such that $\overline{\sigma_2}(\sigma_1(a)) = \overline{q}(\eta(a'))$ and $Ax \vdash (\Gamma \Rightarrow a')$. Let σ_3 as in Lemma 5.16 and let $\mu = \overline{\sigma_3} \circ \overline{q} \circ \eta$, we have $\mu(a') = \sigma_1(a)$. From **Substitution** and from the fact that Γ does not contain variables, it comes $Ax \vdash (\Gamma \Rightarrow \mu(a'))$ i.e. $Ax \vdash (\Gamma \Rightarrow \sigma_1(a))$. ♢

Lemma 5.24 : For any ground conjunction of atoms Γ, $\mathcal{T}_\Gamma \models Ax$.

Proof : Let $(\Gamma' \Rightarrow a') \in Ax$ and $\sigma : V \to T_\Sigma(T_\Gamma)$. Let us assume that $\mathcal{T}_\Gamma$ satisfies $\sigma(\Gamma')$. Let $\sigma = \overline{\sigma_2} \circ \sigma_1$ as in Lemma 5.16.

From **Axiom introduction**: $Ax \vdash (\Gamma' \Rightarrow a')$.

From **Substitution**: $Ax \vdash (\sigma_1(\Gamma') \Rightarrow \sigma_1(a'))$.

From **Monotonicity**: $Ax \vdash (\Gamma \wedge \sigma_1(\Gamma') \Rightarrow \sigma_1(a'))$.

From Lemma 5.23 and **Modus Ponens**: $Ax \vdash (\Gamma \Rightarrow \sigma_1(a'))$, which means that for any $\eta : V \to T_\Sigma(T_\Sigma)$, $\mathcal{T}_\Gamma$ satisfies $\eta(\sigma_1(a'))$. The conclusion comes by choosing $\eta = \overline{\sigma_2}$. ♢

Theorem 5.25 : *(Completeness of the label calculus on ground formulas)*

Let Ax be a set of label axioms. Let φ be any positive conditional ground formula. If the underlying signature is sensible, then we have:

$$[\, \forall\, \mathcal{A} \in Alg_{Lbl}(Ax)\,,\ \mathcal{A} \models \varphi\,] \quad \Longrightarrow \quad [\, Ax \vdash \varphi\,]$$

Proof : (Let us notice that the signature must be sensible in order to justify the use of Lemma 5.16 in all the previous proofs.) Let us assume that:

$$\forall\, \mathcal{A} \in Alg_{Lbl}(Ax)\,,\ \mathcal{A} \models \varphi$$

From Lemma 5.24 $\mathcal{T}_\Gamma \in Alg_{Lbl}(Ax)$, thus $\mathcal{T}_\Gamma \models \varphi$ and Lemma 5.22 allows to conclude. ♢

Our calculus is only a slight generalization of the standard many-sorted Horn-clauses calculus, thus it seems at least not surprising that it works when ground formulas are examined. Unfortunately (against all the hopes of the authors...) Maura Cerioli proved that our label calculus is not complete and it seems that big troubles arise if we look for a calculus that is complete for elementary formulas with variables.

Example 5.26 : *(due to Maura Cerioli)*

Let us consider the label signature defined by: $S = \{s_1, s_2\}$; the set of operations consists of one constant $c :\rightarrow s_1$ and a ternary function $f : s_1 \times s_1 \times s_1 \rightarrow s_2$; and a unique label l. (The signature is sensible.) Consider the following set of axioms Ax:

$x = c$
$f(x, x, y)\ \epsilon\ l$
$f(x, y, x)\ \epsilon\ l$
$f(y, x, x)\ \epsilon\ l$

Let $\mathcal{A}$ be an algebra which satisfies Ax. Its carrier of sort s_1 is a singleton because of the first axiom. Hence the carrier of $T_\Sigma(A)$ is reduced to 2 elements of sort s_1: a_A and a. Therefore, for every evaluation $\sigma : V \rightarrow T_\Sigma(A)$, at least two among $\sigma(x)$, $\sigma(y)$, $\sigma(z)$ coincide and hence, because of the last three axioms, $f(x, y, z)\ \epsilon\ l$ holds for all algebras satisfying Ax. Unfortunately, the label calculus does not allow to deduce this formula with variables.

Such formulas get a strange implicit validity. It seems rather "illogical" that a formula holds essentially because there is not a sufficient number of elements to distinguish the variables, but that this lack is not carried out by equality (as it happens for Equational Typed Logic [MSS89] for example).

6 Exception algebras

The framework of exception algebras presented below is a specialization of the one of label algebras, where the labels are used for exception handling purposes. For convenience, we keep the same terminology "exception algebra" as in the framework of [Ber86], [BBC86], but it is not the same underlying formalism. Our approach is much simpler and more appropriate (cf. Example 3.2 in Section 3.5).

6.1 Introduction: label algebras and exception algebras

As already explained, the normal cases and the exceptional ones will be distinguished without any ambiguity in an exception algebra. Thus a particular label will be distinguished to characterize the normal cases. As in almost all the frameworks about algebraic specifications with exception handling, it will be named Ok. Moreover, exception names and error messages shall be represented by labels (of course, distinct from Ok). This allows us to take exception names into account in the (label) axioms; thus, an extremely wide spectrum of exception handling and error recovery cases can be specified. Intuitively, when $t \in l_A$ in an exception algebra $\mathcal{A}$ for $l \neq Ok$, it will mean that the calculation defined by t leads to the exception name l; if $l = Ok$, it will mean that the calculation defined by t is a "normal" calculation (i.e. it does not need an exceptional treatment and the calculation is successful). Most of the time, if $t \in Ok_A$ then all its subterms are labeled by Ok and lead to Ok-values.[1]

As shown in Section 3, when specifying a data structure with exception handling features, the specifier has first to declare the desired Ok-domain. For instance the interval $[0 \ldots Maxint]$ can be declared as follows:

[1] except for certain data structures such as intervals which do not contain 0, see Section 9.2.

$$succ^{Maxint}(0) \in Ok$$
$$succ(n) \in Ok \implies n \in Ok$$

(where $succ^{Maxint}(0)$ stands for $succ(succ(\ldots(succ(0))..))$, the operation $succ$ being applied $Maxint$ times.) Let us assume that the specification contains the following "normal axiom:"

$$pred(succ(n)) = n$$

Then, for example, the term $pred(succ(0))$ should also belong to the Ok-domain because its calculation does not require any exceptional treatment and leads to the Ok-term 0 via the previous normal axiom. We showed in Section 3 that the terseness criteria is not fulfilled when we explicitly describe all the normal terms in an exhaustive manner. Thus, labelling by Ok should preferably be implicitly propagated through the axioms kept for normal cases. These axioms will be called Ok-axioms, and this implicit propagation rule will be an important component of their semantics, as described in Section 7.2. Consequently, the semantics of exception specifications will be more elaborated than the semantics of label specifications, as label algebras have no implicit aspects.

Another important implicit aspect required by exception handling is the so-called "common future" property. Let us assume that $\mathcal{A}$ is an algebra such that a term u has the same value than a term v (i.e. $eval_A(u) = eval_A(v)$; e.g. u can be an exceptional term recovered on the Ok-term v). We clearly need that for every operation f of the signature, $f(\ldots, u, \ldots)$ behaves exactly as $f(\ldots, v, \ldots)$ does. This means that $f(\ldots, u, \ldots)$ and $f(\ldots, v, \ldots)$ have the same value and raise the same exception names. For example, let $\mathcal{A}$ represent the natural numbers bounded by $Maxint$; the terms $succ^i(0)$ with $0 \leq i \leq Maxint$ being labelled by Ok. Let us assume that $succ^{Maxint+1}(0)$ is recovered on $succ^{Maxint}(0)$. Intuitively, once this recovery is done, we want that everything happens as if $succ^{Maxint+1}(0)$ were never raised; this is the very meaning of the word recovery. The recovery should simply work as a systematic replacement of $succ^{Maxint+1}(0)$ by $succ^{Maxint}(0)$. The same succession of operations applied to $succ^{Maxint}(0)$ or to $succ^{Maxint+1}(0)$ should always give the same results; it should return the same value and raise exactly the same exception names. For example if $succ^{Maxint+1}(0)$ is labelled by $TooLarge$ then the term $t = succ^{Maxint+2}(0)$ should also be labelled by $TooLarge$, as $succ^{Maxint+1}(0) = t[succ^{Maxint+1}(0) \leftarrow succ^{Maxint}(0)]$.

Notice that, in a label algebra $\mathcal{A}$, $eval_A(u) = eval_A(v)$ implies that $eval_A(f(\ldots, u, \ldots))$ is equal to $eval_A(f(\ldots, v, \ldots))$, but it does not imply that the terms $f(\ldots, u, \ldots)$ and $f(\ldots, v, \ldots)$ have the same labels. The common future property means more generally that, for every term t containing u as strict subterm, the term $t[u \leftarrow v]$ is labelled by the same exception labels than t. This property is called "common future" and will be an important implicit aspect of the semantics of exception algebras.

6.2 Exception signatures

Definition 6.1 : An *exception signature* ΣExc is a label signature $< S, F, L >$ such that Ok, Exc and Err do not belong to L. The elements of L are called *exception labels*.

The labels Ok, Exc and Err are not allowed as exception labels because they will be used to characterize the Ok-terms, exceptional terms and erroneous terms respectively.

Example 6.2 : $NatExc = < \{Nat\}, \{0, succ\_, pred\_\}, \{Negative, TooLarge\} >$ is a possible exception signature for an exception specification of bounded natural numbers.

As motivated in Section 6.1, exception algebras over the signature ΣExc cannot be directly defined as label algebras over the same signature. We add the label Ok and we also have to add the "common future" property.

Notation 6.3 : Let $\Sigma Exc = \ < S, F, L >$ be an exception signature. In the sequel of this paper, $\widetilde{L}$ will denote $L \cup \{Ok\}$. Moreover $\Sigma\widetilde{L}$ will be the label signature $< S, F, \widetilde{L} >$ (deduced from ΣExc).

Definition 6.4 : *$\mathcal{A}$ satisfies the common future property for l* if and only if for all operations $f : s_1 \dots s_n \to s$ of the signature with $n > 0$, and all terms $u_1 \dots u_n$ and $v_1 \dots v_n$ of $\overline{A}$ (of sort $s_1 \dots s_n$ respectively), we have:

$$(\bigwedge_{i=1}^{n} eval_A(u_i) = eval_A(v_i)) \ \wedge \ f(u_1, \dots, u_n) \in l_A \implies f(v_1, \dots, v_n) \in l_A$$

Definition 6.5 : An *exception algebra* over the exception signature ΣExc is a label algebra $\mathcal{A}$ over the signature $\Sigma\widetilde{L}$ that satisfies the common future property for every l in L.

This definition calls for some comments:

- Our definition of common future property is strictly less restrictive than:

  $$\forall t, t' \in \overline{A}\ , \ \ eval_A(t) = eval_A(t') \implies (t \in l_A \Leftrightarrow t' \in l_A)$$

  If we consider this last property, everything happens exactly as if labelling were attached to values. Consequently our semantics would be equivalent to the ones of [BBC86] and we have shown in Example 3.2 that this is not suitable. Precisely, the common future property is a weaker constraint than the labelling of values, and it ensures a significant difference; for instance Example 3.2 gets the suitable initial model (see also Section 9).

- Remark that the label Ok is not concerned with the common future property. Otherwise, if $succ(Maxint)$ is recovered on $Maxint$, we would have that $pred(succ(Maxint))$ is labelled by Ok ($pred(Maxint)$ being labelled by Ok). Clearly, even if the term $pred(succ(Maxint))$ is recovered, it remains exceptional because an exceptional treatment has been required in its history. The axiom $pred(succ(x)) = x$ being a normal axiom, if $pred(succ(Maxint))$ was not considered as exceptional then the assignment $x = Maxint$ would be an acceptable assignment and we would have the inconsistency $pred(Maxint) = Maxint$ (see also Section 7.2).

- Notice that the common future property implies that the labelling of a term t by an exception label mainly relies on the heading symbol of t. More precisely, for every operation f of the signature, if we have $eval_A(u_i) = eval_A(v_i)$ for all i in $\{1, \dots, n\}$, then $f(u_1, \dots, u_n)$ and $f(v_1, \dots, v_n)$ carry the same exception labels. Consequently, for every operation f, we can inventory the set of labels that can be raised by f. It represents the classical exception declarations of programming languages with exception handling, such as CLU or ADA (see for instance [LG86]), where a given function can raise only a subset of the exception names, which are declared in the heading of the function.

Example 6.6 : According to the exception signature $NatExc$ defined above, we can consider for example the exception algebra $\mathcal{A} = (A, \{l_A\}_{l \in \widetilde{L}})$ defined by:

- $A = \{\ldots, -2, -1, 0, 1, 2, \ldots, Maxint\}$
  the operations $succ_A$ and $pred_A$ are defined as usual on integers with the restriction $succ_A(Maxint) = Maxint$.
- $Negative_A =$

$$\left\{ \begin{array}{cccccc} \ldots, & pred(pred(0)), & pred(0), & succ(pred(0)), & & \ldots, \\ \ldots, & -2, & -1, & succ(-1), & succ(succ(-1)), & \\ \ldots, & succ(-3), & succ(-2), & succ(succ(-2)), & & \ldots, \end{array} \right\}$$

  $Negative_A$ contains here at the same time negative values and terms. All these terms have a negative value by classical evaluation in the set of integers or else have at least a subterm which would have a negative value by evaluation.
- $TooLarge_A = \{succ^{Maxint+1}(0), succ(Maxint), succ(succ(Maxint)), \ldots\}$
- $Ok_A = \left\{ \begin{array}{cccccc} \ldots, & succ(0), & succ(1), & \ldots, & \ldots, & \\ 0, & 1, & 2, & 3, & \ldots, & Maxint \\ pred(1), & pred(2), & pred(3), & \ldots, & \ldots, & pred(Maxint) \end{array} \right\}$

Definition 6.7 : Let $\mathcal{A}$ and $\mathcal{B}$ be two exception algebras with respect to the exception signature ΣExc. An exception morphism $\mu : \mathcal{A} \to \mathcal{B}$ is a $\Sigma\widetilde{L}$-morphism from $\mathcal{A}$ to $\mathcal{B}$.

We accept ΣExc-signature, ΣExc-algebra, ΣExc-morphism as additional notations respectively for exception signature, exception algebra and exception morphism with respect to the exception signature ΣExc.

Definition 6.8 : Given an exception signature ΣExc, the category of all ΣExc-algebras, and ΣExc-morphisms, is denoted by $Alg_{Exc}(\Sigma Exc)$.

Definition 6.9 : $Gen_{Exc}(\Sigma Exc)$ is the full subcategory of $Alg_{Exc}(\Sigma Exc)$ containing all the finitely generated algebras.

Theorem 6.10 : Let ΣExc be an exception signature. Let $Fut_{\Sigma,L}$ be the positive conditional label specification which contains all the $\Sigma\widetilde{L}$-axioms of the form:

$$x_1 = y_1 \ \wedge \ldots \wedge \ x_n = y_n \ \wedge \ f(x_1, \ldots, x_n) \ \epsilon \ l \ \implies \ f(y_1, \ldots, y_n) \ \epsilon \ l$$

where f is any non-constant operation of Σ (i.e. $n > 0$), x_i and y_i are variables of sorts given by the arity of f, and l is any exception label of L.

The label specification $Fut_{\Sigma,L}$ specifies the ΣExc-algebras, i.e. $Alg_{Exc}(\Sigma Exc)$ is equal to $Alg_{Lbl}(Fut_{\Sigma,L})$.

Proof : From Definition 4.13 and Definition 6.4, a $\Sigma\widetilde{L}$-algebra $\mathcal{A}$ satisfies $Fut_{\Sigma,L}$ if and only if it satisfies the common future property. Thus, $Alg_{Exc}(\Sigma Exc) = Alg_{Lbl}(Fut_{\Sigma,L})$. $\diamondsuit$

If there are p non-constant operations in Σ and q exception labels in L, then $Fut_{\Sigma,L}$ contains $p \times q$ axioms.

Theorem 6.11 : $Alg_{Exc}(\Sigma Exc)$ has an initial object denoted $\mathcal{T}_{\Sigma Exc}$.

Proof : $Alg_{Exc}(\Sigma Exc)$ is included in $Alg_{Lbl}(\Sigma\widetilde{L})$ and the initial object of $Alg_{Lbl}(\Sigma\widetilde{L})$, $\mathcal{T}_{\Sigma\widetilde{L}}$, satisfies the common future property. Thus $\mathcal{T}_{\Sigma Exc} = \mathcal{T}_{\Sigma\widetilde{L}}$ is initial in $Alg_{Exc}(\Sigma Exc)$. ♢

6.3 Exceptions and errors

By analogy with programming languages, a term is exceptional if and only if it raises an exception name in its history.

Definition 6.12 : Let $\mathcal{A}$ be a ΣExc-algebra. The set of *exceptional terms* according to $\mathcal{A}$ is the least subset of $\overline{A}$, denoted by Exc_A, such that:

1. for all labels $l \in L$, $l_A \subseteq Exc_A$;
2. for all non-constant operations $(f : s_1 \dots s_n \to s)$ of the signature $(n > 0)$, for all terms $t_1, \dots, t_n$ (according to the arity of f), if at least one of the t_i belongs to Exc_A then $f(t_1, \dots, t_n)$ belongs to Exc_A.

Notice that, of course, $l = Ok$ is not taken into account in this definition.

Let us remark that an exceptional term is not necessarily erroneous because it can be recovered on an Ok-value. Nevertheless it remains exceptional because its recovery is an exception handling.

Definition 6.13 : Let $\mathcal{A}$ be a ΣExc-algebra; the set of *Ok-values* of A is $A_{Ok} = eval_A(Ok_A)$.

Notice that Ok_A is a set of terms (subset of $\overline{A}$) while A_{Ok} is a set of values (subset of A). Then, erroneous terms can easily be defined:

Definition 6.14 : Let $\mathcal{A}$ be a ΣExc-algebra. The set of *erroneous terms* according to $\mathcal{A}$ is the least subset of $\overline{A}$, denoted by Err_A, such that:

1. for all labels $l \in L$, for all terms $t \in l_A$, if $eval_A(t) \notin A_{Ok}$ then $t \in Err_A$; i.e.:

$$[(\bigcup_{l \in L} l_A) - eval_A^{-1}(A_{Ok})] \subseteq Err_A$$

2. for all non-constant operations $(f : s_1 \dots s_n \to s)$ of the signature $(n > 0)$, for all terms $t_1, \dots, t_n$ (according to the arity of f), if at least one of the t_i belongs to Err_A and if $eval_A(f(t_1, \dots, t_n)) \notin A_{Ok}$ then $f(t_1, \dots, t_n)$ belongs to Err_A .

Moreover, the set of erroneous values of A is by definition $A_{Err} = eval_A(Err_A)$.

These definitions call for some comments.

1. By construction of A_{Err}, we have $A_{Ok} \cap A_{Err} = \emptyset$. However, $A_{Ok} \cup A_{Err}$ does not necessarily cover A. Intuitively, in this case, it represents *partial functions*. For example, let us assume that $\mathcal{A}$ is an algebra representing natural numbers such that the terms $n \; div \; 0$ are not labelled, but do not get an Ok-value. Then it means that the operation div does not raise an explicit exception, and the division by 0 is undefined.

2. Every erroneous term is exceptional ($Err_A \subseteq Exc_A$), but the converse is false because an exception can be recovered. However, let us remark that Err_A is not equal to $(Exc_A - Ok_A)$, and also that it is not equal to $(Exc_A - eval_A^{-1}(A_{Ok}))$. More precisely we have:

$$Err_A \subseteq (Exc_A - eval_A^{-1}(A_{Ok})) \subseteq (Exc_A - Ok_A)$$

but none of the reverse inclusions is ensured. For example, let $\mathcal{A}$ be an algebra representing bounded natural numbers where $succ(Maxint) = Maxint$, the terms n *div* 0 being not labelled (as in the previous example). Then the term ($succ(Maxint)$ *div* 0) is exceptional (because $succ(Maxint)$ is exceptional), it is not recovered, but it is not erroneous ($succ(Maxint)$ is not erroneous, since it is recovered). It is simply equal to ($Maxint$ *div* 0), and undefined.

3. Notice that the definitions of Err_A, Exc_A, A_{Ok} and A_{Err} are independent of any specification; they are intrinsically defined from the exception algebra $\mathcal{A}$.

4. Of course, we can consider that Exc and Err are new labels, and automatically build a label algebra over the label signature $< S, \Sigma, L \cup \{Ok, Exc, Err\} >$ from any exception algebra. However, we should be aware that *exception morphisms do not preserve the label Err*, because exception morphisms can add recoveries (see Example 6.16 below). It is not difficult to show that they preserve the label Exc, i.e. $\overline{\mu}(Exc_A) \subseteq Exc_B$.

Definition 6.15 : An exception algebra $\mathcal{A}$ is called *total* if $A = A_{Ok} \cup A_{Err}$.

Example 6.16 : In the exception algebra $\mathcal{A}$ described in Example 6.6 of the previous section, we have for instance:

- $pred(0)$ and all the terms that contain $pred(0)$ as subterm are exceptional because $pred(0)$ belongs to $Negative_A$;
- $succ^{Maxint+1}(0)$ is recovered since it is exceptional and its value is equal to the value of the Ok-term $succ^{Maxint}(0)$;
- $pred(0)$ is an erroneous term since it belongs to $Negative_A$ without belonging to $eval_A^{-1}(A_{Ok})$;
- -1 is an erroneous value since it is the result of the evaluation of the erroneous term $pred(0)$;
- If we consider an algebra $\mathcal{B}$ that additionally recovers $pred(0)$ on 0, there exists an exception morphism from $\mathcal{A}$ to $\mathcal{B}$, which is the quotient morphism, but it does not preserve the label Err, as $pred(0)$ does not belong to Err_B.

7 Exception specifications

As shown in Section 2, it is preferable to separate the axioms concerning exceptional cases from the ones concerning normal cases in order to preserve legibility and terseness of specifications. The axioms of an exception specification will be separated in two parts.

- The first part, called $GenAx$, is mainly devoted to exception handling. It has three main purposes:

1. We have shown in the sections 3.4 and 3.5 that it is first necessary to characterize the Ok-domains of the underlying data structures. They will be specified in $GenAx$ by positive conditional axioms with a conclusion of the form $t \in Ok$, whose meaning is that t will be a normal term. Thus, these axioms will be used as starting point to generate the set of Ok-terms.
2. It is also necessary to attach exception names to the exceptional cases, in order to facilitate the specification of specialized exception handling. They will be specified in $GenAx$ by positive conditional axioms with a conclusion of the form $t \in l$ where l belongs to L, whose meaning is that the heading function of the term t raises the exception name l.
3. The third purpose of $GenAx$ is to handle the exceptional cases, in particular to specify recoveries, according to the previous labelling of terms. They will have a conclusion of the form $u = v$.

As the axioms of $GenAx$ concern all the terms, exceptional or not, the satisfaction of such axioms does not require some particular mechanism; it will simply be the same as for label axioms. It is the reason why the three purposes mentioned above are grouped under the name "generalized axioms" (they have common semantics); however, for a concrete syntax devoted to exception specifications, it could be preferable to distinguish these three purposes.

- The second part, called $OkAx$, is entirely devoted to the normal cases, and will only concern terms labelled by Ok. As extensively shown in Section 3, the semantics of $OkAx$ will be carefully restricted to Ok-assignments only, in order to avoid inconsistencies.

  We will define a special semantics for Ok-axioms that will both specify equalities between Ok-terms and carefully propagate labelling by Ok through these equalities (following the motivation given in Section 6.1).

An exception specification *SPEC* is defined as a triple $<\Sigma Exc, GenAx, OkAx>$ where ΣExc is an exception signature, $GenAx$ a set of generalized axioms (defined in Section 7.1 below) and $OkAx$ a set of Ok-axioms (defined in Section 7.2 below).

7.1 Generalized axioms

Definition 7.1 : Let ΣExc be an exception signature. A set of *generalized axioms* with respect to the exception signature ΣExc is a set $GenAx$ of positive conditional label axioms with respect to the label signature $\Sigma\widetilde{L}$.

Definition 7.2 : Given an exception signature ΣExc, an exception algebra $\mathcal{A}$ satisfies a generalized axiom "α" ($\mathcal{A} \models \alpha$) if and only if the underlying label algebra of $\mathcal{A}$ satisfies it, regarded as a label axiom.

Given a set $GenAx$ of generalized axioms, $\mathcal{A}$ satisfies $GenAx$ if and only if $\mathcal{A}$ satisfies all the axioms of $GenAx$.

Example 7.3 : Let $NatExc = <\{Nat\}, \{0, succ\_, pred\_\}, \{TooLarge, Negative\}>$ be the exception signature given in Example 6.2. An example of $GenAx$ for a specification of natural numbers bounded by *Maxint* is given by:

$succ^{Maxint}(0) \epsilon Ok$
$succ(n) \epsilon Ok \Rightarrow n \epsilon Ok$
$succ^{Maxint+1}(0) \epsilon TooLarge$
$pred(0) \epsilon Negative$
$succ^{Maxint+1}(0) = succ^{Maxint}(0)$

The two first axioms specify the Ok domain of Nat. In most examples, they define recursively the set of "normal forms" which belong to the intended Ok domain. It is not necessary to declare *all* the Ok-terms (the label Ok will automatically be propagated to terms such as $pred(succ(0))$ via the Ok-axioms, as described in Section 7.2). Even if it is generally easier to specify the Ok domain this way, it is not mandatory. We never require for the axioms of an exception specification to be canonical term rewriting systems, and a fortiori, we never require to actually specify normal forms. It is only desirable to declare at least one term for each intended Ok-value.

The third and fourth axioms declare exception names. Their meaning is that the operation $succ$ (resp. $pred$) raises the exception $TooLarge$ (resp. $Negative$) when applied to $Maxint$ (resp. 0).

The last axiom recovers $succ^{Maxint+1}(0)$ on $succ^{Maxint}(0)$. Let us note that the generalized axiom $succ^{Maxint+1}(0) \epsilon TooLarge$ is then not directly necessary, but it could have been useful if we replaced the last axiom by:

$$n \epsilon TooLarge \implies n = succ^{Maxint}(0)$$

which can make the specification more easily understandable, or by

$$succ(n) \epsilon TooLarge \implies succ(n) = n$$

which is consistent now, because the label $TooLarge$ is not propagated to $succ^{Maxint}(0)$ (see Example 3.2). On the contrary, since $pred(0)$ is not recovered, it is necessary to label it in order to write a "self complete" specification (defined in Section 8.1).

Let us note that we operate in a total framework; however this does not force to always define a recovery condition. For example, this specification does not imply for $pred(0)$ to be equal to an Ok-term; consequently, in the initial model, it denotes an exceptional term since it is labelled by $Negative$. More precisely, as soon as $pred(0)$ is labelled by $Negative$ and not recovered, it becomes erroneous. This fact can be understood as an error exit.

Theorem 7.4 : Let ΣExc be an exception signature and let $GenAx$ be a set of generalized axioms. The class of exception algebras which satisfies $GenAx$ is equal to $Alg_{Lbl}(Fut_{\Sigma,L} \cup GenAx)$. ($Fut_{\Sigma,L}$ is defined in Theorem 6.10.)

(The proof is immediate).

7.2 Ok-axioms

Definition 7.5 : Let ΣExc be an exception signature. A set of *Ok-axioms* with respect to the exception signature ΣExc is a set $OkAx$ of positive conditional $\Sigma\widetilde{L}$-axioms with a conclusion of the form: $v = w$. Thus, an Ok-axiom is of the form

$$u_1 \epsilon l_1 \ \wedge \ldots \wedge \ u_m \epsilon l_m \ \wedge \ v_1 = w_1 \ \wedge \ldots \wedge \ v_n = w_n \implies v = w$$

where the u_i, v_j, w_j, v and w are Σ-terms with variables and the l_i are labels of $\widetilde{L}$. (m or n may be equal to 0.)

Definition 7.6 : Let ΣExc be an exception signature. An exception algebra $\mathcal{A}$ satisfies an Ok-axiom of the form:

$$u_1 \in l_1 \ \wedge \ldots \wedge \ u_m \in l_m \ \wedge \ v_1 = w_1 \ \wedge \ldots \wedge \ v_n = w_n \implies v = w$$

if and only if for all assignments σ with range in $\overline{A}$ (covering all the variables of the axiom) which satisfy the precondition, i.e.

$$(\bigwedge_{i=1}^{m} \sigma(u_i) \in l_{iA}) \ \wedge \ (\bigwedge_{j=1}^{n} eval_A(\sigma(v_j)) = eval_A(\sigma(w_j)))$$

the two following properties hold:

1. *Ok propagation*: if at least one of the terms $\sigma(v)$ or $\sigma(w)$ belongs to Ok_A and the other one is of the form $f(t_1, \ldots, t_p)$ with all the t_i belonging to Ok_A (p may be equal to 0), then both $\sigma(v)$ and $\sigma(w)$ belong to Ok_A.
2. *Ok equality*: if $\sigma(v)$ and $\sigma(w)$ belong to Ok_A then $eval_A(\sigma(v)) = eval_A(\sigma(w))$.

$\mathcal{A}$ satisfies $OkAx$ if and only if $\mathcal{A}$ satisfies all the Ok-axioms of $OkAx$.

The semantics of $OkAx$ call for some comments.

1. The first property of the definition models a propagation of the Ok label: a term can be labelled by Ok through an Ok-axiom only if all its *direct strict subterms* (i.e. all the arguments of the heading function) are already Ok. This rule allows us to carefully propagate the label Ok. It corresponds to an *innermost evaluation* which avoids inconsistencies: a recovered exceptional term cannot be treated by the Ok-axioms. Intuitively, an innermost evaluation models an implicit propagation of exceptions: if t is not an Ok-term then $f(\ldots, t, \ldots)$ cannot be turned into an Ok-term via the Ok-axioms (recoveries are handled by generalized axioms because they are exceptional treatments). Thus, $t \in Ok_A$ does not mean "t has an Ok value;" it means "t does not require an exceptional treatment in its history."

   Let us note that this propagation starts from the Ok-terms declared in $GenAx$.

2. The second property specifies the equalities that hold for the normal cases. Two terms are required to have the same evaluation according to an Ok-axiom only if they are both labelled by Ok.

Example 7.7 : Let $NatExc = \ <\{Nat\}, \{0, succ\_, pred\_\}, \{TooLarge, Negative\}>$ and let $GenAx$ be given as in Example 7.3. The set $OkAx$ of Ok-axioms has only to specify the operation *pred* in all normal cases. It can be given by the single following axiom:

$$pred(succ(n)) = n$$

Notice that $OkAx$ is actually terse and legible, compared to the approaches described in Section 3. Moreover the inconsistency raised by the recovery

$$succ^{Maxint+1}(0) = succ^{Maxint}(0)$$

does not occur any more; the instance

$$pred(succ^{Maxint}(0)) = pred(succ^{Maxint+1}(0)) = succ^{Maxint}(0)$$

is no longer an instance of the Ok-axiom because the term $succ^{Maxint+1}(0)$ [therefore the term $pred(succ^{Maxint+1}(0))$] is not required to be an Ok-term in our framework (even though $eval_A(succ^{Maxint+1}(0))$ is equal to $eval_A(succ^{Maxint}(0))$). Thus, $pred(succ^{Maxint+1}(0)) = succ^{Maxint}(0)$ is *not* a consequence of $OkAx$. This is a good example of our restricted propagation of the label Ok through the Ok-axioms; it shows how the semantics of Ok-axioms models an implicit propagation of exceptions.

Definition 7.8 : An *exception specification* is a triple $<\Sigma Exc, GenAx, OkAx>$ where ΣExc is an exception signature, $GenAx$ is a set of generalized axioms and $OkAx$ is a set of Ok-axioms.

Let $SPEC = <\Sigma Exc, GenAx, OkAx>$. A ΣExc-algebra $\mathcal{A}$ satisfies $SPEC$ if and only if it satisfies $GenAx$ and $OkAx$, as sets of generalized axioms and Ok-axioms respectively.

We denote by $Alg_{Exc}(SPEC)$ the full subcategory of $Alg_{Exc}(\Sigma Exc)$ containing all the algebras satisfying $SPEC$ ($SPEC$-algebras for short). $Gen_{Exc}(SPEC)$ is the full subcategory of $Alg_{Exc}(SPEC)$ containing the finitely generated $SPEC$-algebras.

Note that the semantics of an axiom with a conclusion of the form $v = w$ vary whether it is considered as a generalized axiom or as an Ok-axiom, according to the principle that the specification of normal cases and exceptional cases are distinguished, with different implicit semantics.

Remark 7.9 : Let $BoundedNat$ be the exception specification given in the example above. The exception algebra $\mathcal{A}$ described in Example 6.6 satisfies $BoundedNat$.

Remark 7.10 : As a consequence of the semantics of $OkAx$, non-strict operations or lazy evaluation cannot be specified by means of Ok-axioms because of the innermost evaluation principle. However, the specification of non-strict operations or lazy evaluation may be specified by using generalized axioms. More precisely, this kind of specification intrinsically relies on exception handling because it concerns all the terms, even if they contain exceptional or erroneous subterms. Consequently, such operations have to be specified via generalized axioms. For example, to specify a non-strict $if\_then\_else\_$ operation, the two following axioms have to be put into $GenAx$:

$$if\ true\ then\ u\ else\ v = u$$

$$if\ false\ then\ u\ else\ v = v$$

Being considered as generalized axioms, they concern all the terms $\sigma(u)$ and $\sigma(v)$, including the non-Ok ones. Even if $\sigma(v)$ is erroneous, the term $\sigma(if\ true\ then\ u\ else\ v)$ can result in an Ok-value, as soon as $\sigma(u)$ has an Ok-value. This particular case of using recovery exactly coincides with the usual notion of lazy evaluation.

Let us note that the first "naive" (but terse) algorithm given in Section 2.1 works correctly provided that the operation $\_and\_$ is a lazy operator which evaluates first the left hand side argument. This lazy evaluation can be specified in $GenAx$ as follows:

$$false\ and\ v = false$$

the other usual axiom ($true\ and\ v = v$) can be left in $OkAx$ for instance.

Lemma 7.11 : Let ΣExc be an exception signature. Let α be an Ok-axiom. There is a set of $\Sigma\widetilde{L}$-axioms, denoted $Tr(\alpha)$, such that for every ΣExc-algebra $\mathcal{A}$, $\mathcal{A}$ satisfies the Ok-axiom α if and only if the underlying $\Sigma\widetilde{L}$-algebra of $\mathcal{A}$ satisfies $Tr(\alpha)$, regarded as a set of label axioms. The transformation $Tr : \alpha \rightarrow Tr(\alpha)$ only depends on the signature $\Sigma\widetilde{L}$.

Proof : By definition, the Ok-axiom α is of the form

$$P \implies v = w$$

where P is the precondition of α (P may be empty). Three cases can occur, depending on whether the terms v and w are reduced to a variable or not.

1. If v and w are not reduced to variables, then $v = f(v_1, \ldots, v_p)$ and $w = g(w_1, \ldots, w_q)$ where f and g belong to the signature and v_i and w_j are terms with variables (p or q may be equal to 0). Let $Tr(\alpha)$ be the following set of $\Sigma\widetilde{L}$-axioms:

$$P \wedge (\bigwedge_{i=1}^{p} v_i \in Ok) \wedge w \in Ok \implies v \in Ok$$
$$P \wedge v \in Ok \wedge (\bigwedge_{j=1}^{q} w_j \in Ok) \implies w \in Ok$$
$$P \wedge v \in Ok \wedge w \in Ok \implies v = w$$

   From Definition 4.13, the underlying label algebra of an exception algebra $\mathcal{A}$ satisfies the two first label axioms if and only if it satisfies the Ok propagation of Definition 7.6; moreover, $\mathcal{A}$ satisfies the last label axiom if and only if it satisfies the Ok equality of Definition 7.6. Consequently, $\mathcal{A}$ satisfies α as an Ok-axiom if and only if it satisfies these three axioms as label axioms.

2. If exactly one of the terms v or w is reduced to a variable, say v, then w is is of the form $g(w_1, \ldots, w_q)$. Let n be the cardinal of the signature (the number of operations belonging to ΣExc). For each operation f of ΣExc, let σ_f be the assignment defined by $\sigma_f(v) = f(z_1, \ldots, z_p)$ where z_i are fresh variables[1] and $\sigma_f(x) = x$ for all the other variables x appearing in α. Let, finally, $Tr(\alpha)$ be the set containing the following $n+2$ label axioms:

$$\sigma_f(P) \wedge (\bigwedge_{i=1}^{p} z_i \in Ok) \wedge \sigma_f(w) \in Ok \implies \sigma_f(v) \in Ok$$

   (for all f in the signature) and

$$P \wedge v \in Ok \wedge (\bigwedge_{j=1}^{q} w_j \in Ok) \implies w \in Ok$$
$$P \wedge v \in Ok \wedge w \in Ok \implies v = w$$

   Let us note that for each assignment σ with range in $\overline{A}$, either $\sigma(v)$ is a constant element of A, or there exists an assignment γ and a σ_f such that $\sigma = \gamma \circ \sigma_f$. Consequently,

[1] according to the arity of f; p may be equal to 0.

for the same reason as before, the underlying label algebra of an exception algebra $\mathcal{A}$ satisfies the $(n+1)$ first axioms if and only if it satisfies the Ok propagation; moreover, it satisfies the last axiom if and only if it satisfies the Ok equality of Definition 7.6. Consequently, $\mathcal{A}$ satisfies α as an Ok-axiom if and only if it satisfies these $(n+2)$ axioms as label axioms.

3. If v and w are two distinct variables (if they are equal $Tr(\alpha) = \emptyset$ is sufficient), let $Tr(\alpha)$ be the set containing the following $(2n+1)$ label axioms:

$$\sigma_f(P) \ \wedge \ (\bigwedge_{i=1}^{p} z_i \ \epsilon \ Ok) \ \wedge \ w \ \epsilon \ Ok \ \implies \ \sigma_f(v) \ \epsilon \ Ok$$

(for all f in the signature) and

$$\tau_g(P) \ \wedge \ v \ \epsilon \ Ok \ \wedge \ (\bigwedge_{j=1}^{q} z_j \ \epsilon \ Ok) \ \implies \ \tau_g(w) \ \epsilon \ Ok$$

(for all g in the signature, the τ_g being defined with respect to w in a similar manner as the σ_f have been defined with respect to v) and

$$P \ \wedge \ v \ \epsilon \ Ok \ \wedge \ w \ \epsilon \ Ok \ \implies \ v = w$$

For the same reasons as before, an exception algebra $\mathcal{A}$ satisfies the Ok-axiom α if and only if it satisfies these $(2n+1)$ label axioms.

This proves the lemma. ♢

Theorem 7.12 : Let $SPEC = \ <\Sigma Exc, GenAx, OkAx>$ be an exception specification.
Let $Tr(SPEC)$ be the label specification defined by the label signature $\Sigma\widetilde{L}$ and the set of label axioms containing: all the axioms of $Fut_{\Sigma,L}$ (defined in Theorem 6.10), $GenAx$ and all the $Tr(\alpha)$ for $\alpha \in OkAx$ (defined in Lemma 7.11 above).
We have $Alg_{Exc}(SPEC) = Alg_{Lbl}(Tr(SPEC))$. $Tr(SPEC)$ is called the *translation* of the exception specification $SPEC$ into a label specification.

Proof : Directly results from Theorem 7.4 and Lemma 7.11. ♢

Let us note that $Tr(SPEC)$ contains only positive conditional axioms. Section 8.1 uses $Tr(SPEC)$ to obtain initiality results for exception algebras. An example of translation can be found in Section 9.3.

8 Main results and structured exception specifications

8.1 Fundamental results

The translations proved in Section 7 above ensure that all the initiality results obtained for label algebras hold for exception algebras.

Theorem 8.1 : Let *SPEC* be an exception specification.

$Alg_{Exc}(SPEC)$ and $Gen_{Exc}(SPEC)$ have an initial object, denoted $\mathcal{T}_{SPEC}$.

Moreover, $\mathcal{T}riv$ is final in $Alg_{Exc}(SPEC)$ (and in $Gen_{Exc}(SPEC)$ if the signature is sensible).

Proof : $Tr(SPEC)$ being a positive conditional label specification, it results from: $Alg_{Exc}(SPEC) = Alg_{Lbl}(Tr(SPEC))$ and $Gen_{Exc}(SPEC) = Gen_{Lbl}(Tr(SPEC))$. ♢

Definition 8.2 : An exception specification *SPEC* is called *self complete* if and only if the algebra $\mathcal{T}_{SPEC}$ is total (see Definition 6.15).

Definition 8.3 : Let ΣExc_1 and ΣExc_2 be exception signatures such that $\Sigma Exc_1 \subseteq \Sigma Exc_2$ (i.e. $S_1 \subseteq S_2$, $\Sigma_1 \subseteq \Sigma_2$ and $L_1 \subseteq L_2$).
The forgetful functor $U : Alg_{Exc}(\Sigma Exc_2) \to Alg_{Exc}(\Sigma Exc_1)$ is defined as the forgetful functor on the underlying label algebras.

This definition is sensible because for every exception algebra $\mathcal{A}$, the label algebra $U(\mathcal{A})$ is an exception algebra (i.e. satisfies the common future property). Indeed, from Theorem 6.10, U can also be shown as the forgetful functor from $Alg_{Lbl}(Fut_{\Sigma_2,L_2})$ to $Alg_{Lbl}(Fut_{\Sigma_1,L_1})$, the Fut_{Σ_i,L_i} being defined as in Theorem 6.10.

Theorem 8.4 : Let $SPEC_1$ and $SPEC_2$ be exception specifications with $SPEC_1 \subseteq SPEC_2$. Let U be the forgetful functor from $Alg_{Lbl}(\Sigma Exc_2)$ to $Alg_{Lbl}(\Sigma Exc_1)$. The restriction of U to $Alg_{Lbl}(SPEC_2)$ can be co-restricted to $Alg_{Lbl}(SPEC_1)$.

More generally, given two exception signatures $\Sigma Exc_1 \subseteq \Sigma Exc_2$, for every ΣExc_2-algebra $\mathcal{A}$ and for every ΣExc_1-axiom φ (Ok or generalized) we have:

$$\mathcal{A} \models \varphi \implies U(\mathcal{A}) \models \varphi$$

Proof : If φ is a generalized axiom, it directly results from Theorem 5.6. If φ is an Ok-axiom, it results from Theorem 5.6 and from the fact that $\Sigma Exc_1 \subseteq \Sigma Exc_2$ implies $Tr_{\Sigma Exc_1}(\varphi) \subseteq Tr_{\Sigma Exc_2}(\varphi)$. ♢

Definition 8.5 : Let $SPEC_1$ and $SPEC_2$ be exception specifications with $SPEC_1 \subseteq SPEC_2$. The *synthesis functor* $F : Alg_{Exc}(SPEC_1) \to Alg_{Exc}(SPEC_2)$ is by definition the synthesis functor $F : Alg_{Lbl}(Tr(SPEC_1)) \to Alg_{Lbl}(Tr(SPEC_2))$ defined on label algebras in Theorem 5.11.

Theorem 8.6 : Let $SPEC_1$ and $SPEC_2$ be exception specifications with $SPEC_1 \subseteq SPEC_2$. The synthesis functor $F : Alg_{Exc}(SPEC_1) \to Alg_{Exc}(SPEC_2)$ is a left adjoint for the forgetful functor $U : Alg_{Exc}(SPEC_2) \to Alg_{Exc}(SPEC_1)$.

Proof : Trivially results from $Alg_{Exc}(SPEC_i) = Alg_{Lbl}(Tr(SPEC_i))$, from the definitions of U and F that coincide with the ones on label algebras, and from Theorem 5.12. ♢

8.2 Structured exception specifications

The initiality results given in Section 8.1 can be used to define *hierarchical consistency* and *sufficient completeness* for structured exception specifications in a similar way as usual.

Definition 8.7 : Let $SPEC_1$ and $SPEC_2$ be two exception specifications. The specification $SPEC_2$ is an *enrichment* of the specification $SPEC_1$ if and only if $SPEC_1 \subseteq SPEC_2$ (i.e. $S_1 \subseteq S_2$, $\Sigma_1 \subseteq \Sigma_2$, etc.). In this case, $SPEC_1$ is often called the *predefined* specification and $\Delta SPEC = SPEC_2 - SPEC_1$ is often called the *presentation of interest*.

Example 8.8 : $Stack$ can be specified as an enrichment of $BoundedNat$ ($BoundedNat$ is specified in Example 6.6).

$$
\begin{array}{lcl}
\Delta S & : & Stack \\
\\
\Delta\Sigma & : & empty : \rightarrow Stack \\
 & & push\_\_ : Stack\ Nat \rightarrow Stack \\
 & & pop\_ : Stack \rightarrow Stack \\
 & & top\_ : Stack \rightarrow Nat \\
\\
\Delta L & : & Underflow \\
 & & BadAccess \\
\\
\Delta GenAx & : & empty \in Ok \\
 & & p \in Ok \ \wedge \ n \in Ok \implies push(p,n) \in Ok \\
 & & pop(empty) \in Underflow \\
 & & top(empty) \in BadAccess \\
\\
\Delta OkAx & : & top(push(p,n)) = n \\
 & & pop(push(p,n)) = p \\
\\
 & Where & : \quad n : Nat \ ; \ p : Stack
\end{array}
$$

This presentation is rather simple with respect to exception handling; no recovery is specified. Nevertheless, the most important thing is that the two Ok-axioms of $Stack$ only concern assignments σ such that both $\sigma(p)$ and $\sigma(n)$ are Ok-terms. In particular, the exceptional terms of the predefined specification $BoundedNat$ are automatically excluded from the semantics of the Ok-axioms. For example, the term $top(push(empty, pred(0)))$ does not result to $pred(0)$ (via the first Ok-axiom of $Stack$) because it is not an Ok-term.

Let us remember that hierarchical consistency means that the presentation does not introduce new equalities over the predefined values (the so-called "no collapse" property). Without exception handling, hierarchical consistency is expressed by means of the unit of adjunction with respect to F and U: it should be injective on the initial object T_{SPEC_1}. With exception handling, the labels should also be taken into account. Hierarchical consistency has to forbid the existence of new labelling of predefined terms by predefined labels. For example, $Stack$ should not imply that $pred(0)$ becomes labelled by $TooLarge$ if this is not a consequence

of $BoundedNat$. A similar definition of hierarchical consistency has already been given in [Ber86], [BBC86].

Definition 8.9 : Let $SPEC_1$ and $SPEC_2$ be exception specifications with $SPEC_1 \subseteq SPEC_2$. The associated enrichment is *hierarchically consistent* if and only if:

1. the unit of adjunction $I_{\mathcal{T}_{SPEC_1}}$ is injective;
2. $\forall l \in \widetilde{L_1},\ \ \forall t \in T_{\Sigma_1}(T_{SPEC_1}),\ \ \overline{I_{\mathcal{T}_{SPEC_1}}}(t) \in l_{U(T_{SPEC_2})} \implies t \in l_{T_{SPEC_1}}$.

Remarks 8.10 :

1. From Theorem 8.6, for all $SPEC_1$-algebra $\mathcal{A}$, $Hom_{SPEC_1}(\mathcal{A}, U(F(\mathcal{A})))$ is canonically isomorphic to $Hom_{SPEC_2}(F(\mathcal{A}), F(\mathcal{A}))$ (see [BW90]). The *unit of adjunction* is the exception morphism from $\mathcal{A}$ to $U(F(\mathcal{A}))$ associated with the identity of $Hom_{SPEC_2}(F(\mathcal{A}), F(\mathcal{A}))$. This morphism is the morphism I_A defined in the proof of Theorem 5.12. If $\mathcal{A}$ is the initial exception algebra $\mathcal{T}_{SPEC_1}$, then $F(\mathcal{A}) = \mathcal{T}_{SPEC_2}$ because left adjoint functors preserve initial objects. Thus, the unit of adjunction $I_{\mathcal{T}_{SPEC_1}}$ is indeed the initial morphism from $\mathcal{T}_{SPEC_1}$ to $U(\mathcal{T}_{SPEC_2})$.
2. The reverse implication of the second property is always satisfied since $I_{\mathcal{T}_{SPEC_1}}$ is an exception morphism ($\overline{I_{\mathcal{T}_{SPEC_1}}}$ preserves the labels).

The presentation $Stack$ given in Example 8.8 is hierarchically consistent: no collapses are added in the sort Nat and no predefined labelling is added since the axioms only concern the new labels.

Let us recall that sufficient completeness means that the presentation does not add new values in the predefined sorts (the so-called "no junk" property). Without exception handling, sufficient completeness means that the unit of adjunction $I_{\mathcal{T}_{SPEC_1}}$ is surjective. With exception handling, such a definition is not suitable. For example, the term $top(empty)$ is of predefined sort Nat but its value does not belong to the ones defined by $BoundedNat$. The value of $top(empty)$ is a new erroneous value which has been introduced by the stack data structure, and there is no reason to take such errors into account when specifying $BoundedNat$. The only important point is that the presentation $Stack$ allows us to deduce that $top(empty)$ is erroneous (as it is labelled by $BadAccess$ and not recovered). It is logical that, when specifying natural numbers, we do not foresee all the possible erroneous values introduced by all the possible ulterior enrichments. Thus, we have to accept new values of predefined sorts, provided that they are erroneous. A similar definition of sufficient completeness has already been given in [Ber86] [BBC86].

Definition 8.11 : Let $SPEC_1$ and $SPEC_2$ be exception specifications with $SPEC_1 \subseteq SPEC_2$. The associated enrichment is *sufficiently complete* if and only if

$$U(T_{SPEC_2} - T_{SPEC_2,Err}) \ \subseteq \ I_{\mathcal{T}_{SPEC_1}}(T_{SPEC_1})$$

(In the notation "$U(T_{SPEC_2} - T_{SPEC_2,Err})$," U should not be understood as the forgetful functor defined on exception algebras because $(T_{SPEC_2} - T_{SPEC_2,Err})$ is not an algebra. Here, U should be understood as the underlying forgetful functor simply defined on heterogeneous sets.)

The presentation *Stack* given in Example 8.8 is sufficiently complete: all the new values added to the sort Nat are erroneous, as they are obtained by evaluation of terms containing $top(empty)$ or $pop(empty)$ as subterms (which are labelled by $BadAccess$ or $Underflow$ respectively).

In this section, we have more or less restricted our study to the initial approach. In particular, our definitions about "structured specifications" deal with the initial models only. This approach is not always satisfactory for specifying realistic software, where a "loose" approach is often suitable. It has been shown in [Ber87] that it is not suitable to define hierarchical consistency or sufficient completeness on all the models of $Alg_{Exc}(SPEC)$ or $Gen_{Exc}(SPEC)$. More elaborated modular semantics should be used. However, even if we are only interested in loose semantics, initiality results are often very convenient to avoid trivial models. For example, in [Bid89], the so-called "basic specifications" (the ones that do not import other specifications) must have an initial model. We believe that our definitions of hierarchical consistency and sufficient completeness provide a good starting point to define what "protecting an imported data structure" means in a loose approach. We have shown in particular that erroneous junk is allowed; thus, a naive definition of "conservative extensions" is not convenient for exception or error handling. This fact should have an important impact on the general definitions of the semantics of modularity. Let us point out that all the existing frameworks for modularity ([AW86], [Bid89], [EBO91] and others), which follow a more or less "institution independent" viewpoint, are *not* suitable for a framework with exception or error handling because they all consider such a naive definition of conservative property (consequently, they do not allow erroneous junk such as $top(empty)$ for the stack module).

9 Some examples

Section 9.1 gives several possible exception specifications of natural numbers. In order to give some insights into the semantics of exception specifications, the impact of the axioms on the initial algebra is described carefully.
Section 9.2 gives several exception specifications of classical data structures. These specifications are not difficult to understand (as already mentioned, the specification of bounded natural numbers raises all the difficulties of exception handling for algebraic specifications); they mainly give an overview of how exception specifications look like.
Section 9.3 gives an example of proof according to the label calculus, from a specification of natural numbers.

9.1 Several versions of natural numbers

Let us first specify natural numbers without bound. The specification $Nat1$ given below, over the signature $NatExc = <\{Nat\}, \{0, succ\_, pred\_\}, \{Negative\}>$, is rather similar to the specifications of natural numbers that can be done in the framework of [GDLE84]: the first two generalized axioms mean that 0 and *succ* are "safe operations."

$$\begin{array}{lll} GenAx & : & 0 \in Ok \\ & & n \in Ok \implies succ(n) \in Ok \end{array}$$

$$OkAx \quad : pred(succ(n)) = n$$

$$Where \ : \quad n \ : Nat$$

- The first two generalized axioms imply that the terms $succ^i(0)$ with $0 \leq i$ are Ok-terms.
- $pred(succ(n)) = n$ is an Ok-axiom; therefore it only applies to Ok-terms, in particular $n = pred(0)$ is not an acceptable assignment (to obtain the same result in the framework of [GDLE84] it is necessary to explicitly replace n by n_+, see Section 3.5).
- $pred(0)$ is not an Ok-term because the propagation of the label Ok through the Ok-axioms never matches the term $pred(0)$. More generally, the Ok-terms are the terms t such that every subterm of t contains at least as many occurrences of $succ$ as occurrences of $pred$.
- The Ok-axiom implies that each Ok-term t has the same value that the term of the form $succ^i(0)$, where i is the difference between the number of occurrences of $succ$ and $pred$ in t.
- Notice that $pred(0)$ is not labelled by an exception name. Thus, in our framework, the specification $Nat1$ is not self complete. (The label $Negative$ is not used in $GenAx1$.)

All the examples $Nat2$ to $Nat7$ given in this section contain $Nat1$, and have the same signature $NatExc$. Moreover, $Nat2$ to $Nat7$ will not contain any additional Ok-axiom, nor additional generalized axiom with a conclusion of the form "$u \in Ok$". Consequently, the set of Ok-terms will not change; $pred(0)$, even if it is recovered, will always remain exceptional. $Nat2$ to $Nat7$ show how the idea of labelling terms can be used in order to reach a very precise specification of fine exception handling features.

As $Nat1$ is not self complete, let us consider $Nat2$, which contains $Nat1$, such that $GenAx2$ contains $GenAx1$ and the following additional generalized axiom:

$$pred(0) \in Negative$$

Since $pred(0)$ is labelled by $Negative$ and not recovered, it is erroneous. $Nat2$ is self complete.

Let us remark that $Nat2$ labels only one term in the initial model: $pred(0)$. There is no explicit propagation of the error raised by $pred$ on 0. All terms containing $pred(0)$ as strict subterm are not labelled. For high quality software, it is sometimes convenient to require exception handlers that explicitly forward the exception names through the operations (if they do not recover them). Let us consider $Nat3$, which contains $Nat2$, such that $GenAx3$ contains the following two additional generalized axioms:

$$\begin{array}{rcl} n \in Negative & \Longrightarrow & succ(n) \in Negative \\ n \in Negative & \Longrightarrow & pred(n) \in Negative \end{array}$$

The specification $Nat3$ clearly fulfills this requirement; all erroneous terms are labelled by $Negative$, modeling the innermost error that has been raised in the term.

Let us note that, even if all the erroneous terms are labelled by $Negative$, they remain distinct. If we want to have a single erroneous value, as in the approach proposed in [GTW78], it is sufficient to consider $Nat4$, which contains $Nat3$, such that $GenAx4$ contains the following additional generalized axiom:

$$n \in Negative \implies n = pred(0)$$

$Nat4$ is an example where the generalized axioms are not only used for recovery purposes. Here, the additional axiom is used to collapse all the exceptional terms on a single erroneous value.

Instead of labelling the exceptional terms in order to reach self completeness (as in examples $Nat2$ to $Nat4$), it is possible to directly recover the non Ok-terms, without labelling them. Let us consider $Nat5$, which contains $Nat1$, such that $GenAx5$ contains $GenAx1$ and the following additional generalized axiom:

$$pred(0) = 0$$

This axiom recovers all the exceptional ground terms. More precisely, $pred(0)$ is recovered on 0, $succ(pred(pred(0)))$ is recovered on $succ(0)$ and so on. Let us note that $Nat5$ remains nevertheless consistent because the terms $pred(0)$, $succ(pred(pred(0)))$, etc. are not Ok-terms, even if they are recovered; thus they are not acceptable assignments of the Ok-axioms. The initial model of $Nat5$ actually behaves on the Ok-part as natural numbers do.

In $Nat5$, $pred(0)$ is silently recovered, the intermediate exceptional state is not signaled (i.e. it is not labelled by an exception name). Even if it is recovered, this exception can be signaled by specifying $Nat6$, which contains $Nat5$, such that $GenAx6$ contains $GenAx5$ and the following additional generalized axiom:

$$pred(0) \; \epsilon \; Negative$$

Intuitively, we can consider that the label $Negative$ plays the role of a warning message.

Many other exception handling examples can be provided. For example the exception specification $Nat7$ over $NatExc = \; <\{Nat\}, \{0, succ\_, pred\_\}, \{Negative\}>$ differs from the two previous recovery cases:

$$\begin{array}{lll} GenAx & : & 0 \; \epsilon \; Ok \\ & & n \; \epsilon \; Ok \implies succ(n) \; \epsilon \; Ok \\ & & pred(0) \; \epsilon \; Negative \\ & & n \; \epsilon \; Negative \implies pred(n) \; \epsilon \; Negative \\ & & n \; \epsilon \; Negative \wedge n = m \implies m \; \epsilon \; Negative \\ & & succ(pred(n)) = n \end{array}$$

$$OkAx \quad : \quad pred(succ(n)) = n$$

$Nat7$ does not directly describe an explicit recovery of a particular exceptional term (as the axiom $pred(0) = 0$ does). It describes a general property, $succ(pred(n)) = n$, that concerns exceptional cases as well as normal cases. Several instances are recoveries, provided that the exceptional term under consideration contains more occurrences of $succ$ than of $pred$. For instance, $succ(succ(pred(0)))$ is recovered on the Ok-term $succ(0)$.

We sum up below the different features of $Nat1$ to $Nat7$.

	$Nat1$	$Nat2$	$Nat3$	$Nat4$	$Nat5$	$Nat6$	$Nat7$
recovery	No	No	No	No	Yes	Yes	Yes
labelling	No	Yes	Yes	Yes	No	Yes	Yes
explicit propagation of labelling	No	No	Yes	Yes	No	No	Yes
property on exceptional terms	No	No	No	Yes	No	No	Yes

9.2 Other examples

We give an example of an exception specification of binary trees over natural numbers.

ΔS : $Tree$

$\Delta\Sigma$: $empty :\rightarrow Tree$
$node\_\ \_\ \_ : Tree\ Nat\ Tree\ \rightarrow\ Tree$
$root\_ : Tree \rightarrow Nat$
$left\_ : Tree \rightarrow Tree$
$right\_ : Tree \rightarrow Tree$

ΔL : $Void$

$\Delta GenAx$: $empty \in Ok$
$t1 \in Ok\ \wedge\ t2 \in Ok\ \wedge n \in Ok\ \Longrightarrow\ node(t1, n, t2) \in Ok$
$root(empty) \in Void$
$left(empty) \in Void$
$right(empty) \in Void$
$t \in Void\ \Longrightarrow\ t = empty$

$\Delta OkAx$: $left(node(t1, n, t2)) = t1$
$right(node(t1, n, t2)) = t2$
$root(node(t1, n, t2)) = n$

$Where\ :\quad n\ :\ Nat\ ;\ \ t, t1, t2\ :\ Tree$

$BinTree$ is a presentation of interest, whose predefined specification can be any specification of natural numbers (e.g. $Nat2$ to $Nat7$). Let us note that the exception label $Void$ intersects two different sorts ($Tree$ and Nat): for example the terms $left(empty)$ and $root(empty)$ are labelled by $Void$. In this example, we have chosen to recover every exceptional binary tree labelled by $Void$ on the empty tree. Let us note that the variable t in the last generalized axiom is of sort $Tree$; thus, the exceptional term $root(empty)$, of sort Nat, is of course not collapsed with the empty tree because it is not an acceptable assignment of t.

We give an example of exception specification of queues (FIFO) over the natural numbers.

ΔS : $Queue$

$\Delta\Sigma$: $empty\ :\ \rightarrow Queue$
$add\_\ \_\ :\ Nat\ Queue \rightarrow Queue$
$remove\_\ :\ Queue \rightarrow Queue$
$first\_\ :\ Queue \rightarrow Nat$

ΔL : $Underflow\ ,\ ErrorQueue\ ,\ ErrorFirst$

$\Delta GenAx$: $empty \in Ok$
$q \in Ok\ \wedge\ n \in Ok\ \Longrightarrow\ add(n, q) \in Ok$
$remove(empty) \in Underflow$

$first(empty) \in ErrorFirst$
$q \in Underflow \implies remove(q) \in ErrorQueue$
$q \in Underflow \ \wedge \ n \in Ok \implies add(n, q) = add(n, empty)$

$\Delta OkAx$: $remove(add(n, empty)) = empty$
$remove(add(n, add(m, q))) = add(n, (remove(add(m, q))))$
$first(add(n, empty)) = n$
$first(add(n, add(m, q))) = first(add(m, q))$

$Where$: n, m : Nat ; q : $Queue$

Let us note that we have specified two kind of exceptional queues. On the one hand, if only one *remove* is applied to the empty queue (leading to $Underflow$), then it remains possible to *add* an Ok natural number to it; the resulting queue is recovered to the queue containing this natural number. (Notice that, however, a queue labelled by $Underflow$ is not directly recovered to the empty queue.) On the other hand, if another *remove* is applied to such an erroneous queue, or if an erroneous natural number is added to a queue, then there are no specified recoveries; only one exceptional application of *remove* is allowed for recovering *add*. This exception specification gives a good example where partial functions are not powerful enough to describe the same data structure; $remove(empty)$ is not defined, but $add(n, remove(empty))$ is defined (see Section 3.3).

We give an example of bounded stacks over the natural numbers. It is rather similar to the bounded natural number example studied so far in the paper.

ΔS : $Stack$

$\Delta\Sigma$: $empty$: $\rightarrow Stack$
$push\_\_$: $Nat\ Stack \rightarrow Stack$
$pop\_$: $Stack \rightarrow Stack$
$top\_$: $Stack \rightarrow Nat$

ΔL : $Underflow$, $Overflow$, $ErrorTop$

$\Delta GenAx$: $\bigwedge_{i=1..Max} x_i \in Ok \implies push(x_1, push(x_2, \ldots, push(x_{Max}, empty))) \in Ok$
$push(x, X) \in Ok \implies X \in Ok$
$pop(empty) \in Underflow$
$top(empty) \in ErrorTop$
$push(x_1, push(x_2, \ldots, push(x_{Max+1}, empty))) \in Overflow$
$push(x, X) \in Overflow \implies push(x, X) = X$

$\Delta OkAx$: $pop(push(x, X)) = X$
$top(push(x, X)) = x$

$Where$: $x_1, \ldots, x_{Max+1}, x$: Nat ; X : $Stack$

The last generalized axiom means "if the operation *push* raises $Overflow$ then do not perform it." Let us remember that this specification is consistent within our framework; it would lead

to a trivial initial algebra in all the other existing frameworks because it requires for the exception label $Overflow$ to be carried by terms, not by values (see Example 3.2).

In practice, it is unpleasant to deal with terms such as $push(x_1, \ldots, push(x_{Max}, empty))$. We would prefer to specify the height of a stack and use it to characterize the Ok-terms. This leads to the following specification:

ΔS : $Stack$

$\Delta\Sigma$: $empty\ :\ \rightarrow Stack$
$push\_\ \_\ :\ Nat\ Stack \rightarrow Stack$
$height\_\ :\ Stack \rightarrow Nat$
$pop\_\ :\ Stack \rightarrow Stack$
$top\_\ :\ Stack \rightarrow Nat$

ΔL : $Underflow\ ,\ Overflow\ ,\ ErrorTop$

$\Delta GenAx$: $empty \in Ok$
$X \in Ok\ \wedge\ height(X) < Max = true\ \wedge\ x \in Ok\ \Longrightarrow\ push(x, X) \in Ok$
$pop(empty) \in Underflow$
$top(empty) \in ErrorTop$
$height(X) = Max\ \Longrightarrow\ push(x, X) \in Overflow$
$push(x, X) \in Overflow\ \Longrightarrow\ push(x, X) = X$

$\Delta OkAx$: $height(empty) = 0$
$height(push(x, X)) = succ(height(X))$
$pop(push(x, X)) = X$
$top(push(x, X)) = x$

$Where\ :\quad x\ :\ Nat\ ;\ \ X\ :\ Stack$

Notice that the before last generalized axiom *cannot* be replaced by

$$Max < height(X) = true\ \Longrightarrow\ X \in Overflow$$

because the operation $height$ is only defined on Ok-terms (as $height$ is defined in $OkAx$). Moreover, putting the axioms defining $height$ in $GenAx$ without adding any precondition is not hierarchically consistent because it would lead to $succ(Max) = Max$, according to the last recovery axiom. Similarly, the two first generalized axioms *cannot* be replaced by

$$height(X) \leq Max = true\ \Longrightarrow\ X \in Ok$$

because it does not imply that the Ok-stacks only contain Ok natural numbers; moreover, in this case, there would be no Ok-term of sort stack at all in the initial algebra if $height$ remains defined in $OkAx$.

An example of exception specification of intervals is given below, where the interval $[3, 8]$ is specified (with $\Sigma Exc = <\{Interv\}, \{0, succ\_, pred\_\}, \{TooLow, TooLarge\}>$).

$GenAx$: $succ^8(0) \in Ok$
$succ^4(n) \in Ok\ \Longrightarrow\ succ^3(n) \in Ok$

$succ^2(0) \epsilon TooLow$
$succ(n) \epsilon TooLow \implies n \epsilon TooLow$
$pred(succ^3(0)) \epsilon TooLow$
$succ^9(0) \epsilon TooLarge$

$OkAx$: $pred(succ(n)) = n$

$Where$: $n : Interv$

The only interesting point of this example is to illustrate the fact that a subterm of an Ok-term is not necessarily an Ok-term ($succ^3(0)$ is Ok while $succ^2(0)$ is not).

Our last example belongs to the "dynamic" class of exceptional cases. We give a specification of bounded arrays, where a new array is not supposed initialized. The ranges of an array are of sort $Index$, that can be any sort such that the boolean operations "$<$" "$\leq$" and "eq" are provided; usually it is required for "$\leq$" to define a total order; natural numbers can be used for example. The elements stored in the array belong to the sort $Elem$, which can be any sort.

ΔS : $Array$

$\Delta\Sigma$:
$create\_\_ : Index\ Index \rightarrow Array$
$store\_\_\_ : Elem\ Array\ Index \rightarrow Array$
$fetch\_\_ : Array\ Index \rightarrow Elem$
$lower\_ : Array \rightarrow Index$
$upper\_ : Array \rightarrow Index$

ΔL : $BadRange$, $OutOfRange$, $NonInitialized$

$\Delta GenAx$:
$low \epsilon Ok \wedge up \epsilon Ok \wedge low \leq up = true \implies create(low, up) \epsilon Ok$
$\left\{ \begin{array}{l} a \epsilon Ok \wedge ind \epsilon Ok \wedge x \epsilon Ok \\ \quad \wedge lower(a) \leq ind = true \\ \quad \wedge ind \leq upper(a) = true \end{array} \right\} \Rightarrow store(x, a, ind) \epsilon Ok$
$low \leq up = false \implies create(low, up) \epsilon BadRange$
$ind < lower(a) = true \implies store(x, a, ind) \epsilon OutOfRange$
$upper(a) < ind = true \implies store(x, a, ind) \epsilon OutOfRange$
$ind < lower(a) = true \implies fetch(a, ind) \epsilon OutOfRange$
$upper(a) < ind = true \implies fetch(a, ind) \epsilon OutOfRange$
$lower(a) \leq ind = true \wedge ind \leq upper(a) = true \implies$
$fetch(create(low, up), ind)) \epsilon NonInitialized$
$eq(ind1, ind2) = false \wedge fetch(a, ind1) \epsilon NonInitialized \implies$
$fetch(store(x, a, ind2), ind1) \epsilon NonInitialized$

$\Delta OkAx$:
$lower(create(low, up)) = low$
$upper(create(low, up)) = up$
$lower(store(x, a, ind)) = lower(a)$
$upper(store(x, a, ind)) = upper(a)$
$store(x, store(y, a, ind), ind) = store(x, a, ind)$

$$eq(ind1, ind2) = false \implies$$
$$store(x, store(y, a, ind1), ind2) = store(y, store(x, a, ind2), ind1)$$
$$fetch(store(x, a, ind), ind) = x$$

$$Where : \quad low,\ up,\ ind,\ ind1,\ ind2 : Index\ ;\ \ x,\ y : Elem\ ; a : Array$$

The term $create(low, up)$ creates a new array of range $[low, up]$. The operations *lower* and *upper* retrieve the acceptable range of an array. Notice that the last generalized axiom is useful, even if it seems redundant with the three last Ok-axioms, because the Ok-axioms only concern the Ok-terms, while the purpose of the last generalized axiom is to label erroneous terms. Another possibility would be to remove the last generalized axiom and to move the three last Ok-axioms into $GenAx$ (then, they would apply to all terms, including the exceptional ones).

9.3 An example of proof using the label calculus

In section 9.1, we presented an example of exception specification called $Nat7$. Let us prove for example, from this specification, the sentence

$$pred(pred(succ(0))) \in Negative$$

using the rules of label calculus (Definition 5.14, Section 5.2). As the label calculus is devoted to label algebras, one cannot use directly $Nat7$ since it is an exception specification and not a label specification. However $Nat7$ can be translated into an equivalent label specification by the Definition 6.4 and Lemma 7.11 of the section 7.2. $Tr(Nat7)$ is the set of 12 following axioms:

1. $0 \in Ok$
2. $n \in Ok \Rightarrow succ(n) \in Ok$
3. $pred(0) \in Negative$
4. $n \in Negative \Rightarrow pred(n) \in Negative$
5. $n \in Negative \wedge n = m \Rightarrow m \in Negative$
6. $succ(pred(n)) = n$
7. $n \in Ok \wedge succ(n) \in Ok \Rightarrow pred(succ(n)) \in Ok$
8. $n \in Ok \wedge pred(succ(succ(n))) \in Ok \Rightarrow succ(n) \in Ok$
9. $n \in Ok \wedge pred(succ(pred(n))) \in Ok \Rightarrow pred(n) \in Ok$
10. $n \in Ok \wedge pred(succ(n)) \in Ok \Rightarrow pred(succ(n)) = n$
11. $n = m \wedge succ(n) \in Negative \Rightarrow succ(m) \in Negative$
12. $n = m \wedge pred(n) \in Negative \Rightarrow pred(m) \in Negative$

Axioms 7, 8, 9 and 10 come from the translation described in Lemma 7.11 of the Ok-axiom $pred(succ(n)) = n$; they specify the propagation of the label Ok and the equality between Ok-terms. The axioms 11 and 12 result from Definition 6.4; they specify the common future property.

From the label calculus and the specification above, one have following inference steps:

$[a] \quad 0 \in Ok$

[from axiom 1]

[b] $0 \in Ok \Rightarrow succ(0) \in Ok$
[from axiom 2 and **Substitution** with $n \leftarrow 0$]

[c] $succ(0) \in Ok$
[from **Modus Ponens** applied to [a] and [b]]

[d] $0 \in Ok \wedge succ(0) \in Ok \Rightarrow pred(succ(0)) \in Ok$
[from axiom 7 and **Substitution** with $n \leftarrow 0$]

[e] $pred(succ(0)) \in Ok$
[from **Modus Ponens** applied to [a], [c] and [d]]

[f] $0 \in Ok \wedge pred(succ(0)) \in Ok \Rightarrow pred(succ(0)) = 0$
[from axiom 10 and **Substitution** with $n \leftarrow 0$]

[g] $pred(succ(0)) = 0$
[from **Modus Ponens** applied to [a], [e] and [f]]

[h] $pred(0) \in Negative$
[from axiom 3]

[i] $pred(0) \in Negative \wedge 0 = pred(succ(0)) \Rightarrow pred(pred(succ(0))) \in Negative$
[from axiom 12 and **Substitution** with $n \leftarrow 0$ *and* $m \leftarrow pred(succ(0))$]

[j] $pred(pred(succ(0))) \in Negative$
[from **Modus Ponens** applied to [g] [h] and [i]]

10 Conclusion

We have introduced a distinction between *exception handling* and *error handling* for algebraic specifications. According to our terminology, exception handling is more powerful because some cases can take benefit of exceptional treatments without being erroneous. This improves legibility and terseness of specifications. We have shown that exception handling requires a refined notion of the satisfaction relation for algebraic specifications. The scope of an axiom should be restricted to carefully chosen patterns, because a satisfaction relation based on assignments with range in *values* often raises inconsistencies. A more elaborated notion of assignment is considered: assignment with range in *terms*. This allows us to restrict the scope of an axiom to certain suitable patterns, and solves the inconsistencies raised by exception handling.

We have also shown that exception names, or error messages, are better carried by terms, and that they are advantageously represented by *labels*. Labels do not go through equational atoms; thus, two terms having the same value do not necessarily carry the same labels. We have first defined the framework of *label algebras*, that defines suitable semantics for labels. The scope of the label axioms is carefully delimited by labels which serve as special marks on terms.

Then, we have proposed a new algebraic framework for exception handling, based on label algebras, which is powerful enough to cope with all suitable exception handling features such as implicit propagation of exceptions, possible recoveries, declaration of exception names, etc. As shown in Section 9, all the exceptional cases can easily be specified ("intrinsic" exceptions of an abstract data type, "dynamic" exceptional cases and bounded data structures). This approach solves all the inconsistencies raised by all the existing frameworks (see Section 3) and succeeds with respect to *legibility* and *terseness* of specifications, that are two crucial criteria

for formal specifications with exception handling. More precisely, legibility and terseness are obtained because two different kinds of axioms have been distinguished, with distinct implicit semantics: the generalized axioms treat the exceptional cases, and the Ok-axioms only treat the normal cases.

The usual inconsistencies raised by exception handling for algebraic specifications are solved in our framework because we carefully define the difference between *exception* and *error*. An error is an exception which has not been recovered. Even if an exceptional term has been recovered, it remains exceptional because an exceptional treatment has been required in its "history."

Although we have introduced the theory of label algebras as a general framework for exception handling purposes, the application domain of label algebras seems much more general than exception handling. Labels provide a great tool to express several other features developed in the field of (first order) algebraic specifications. We have outlined in Section 4.3 how label algebras can be used to specify several more standard algebraic approaches such as order sorted algebras [Gog78b], partial functions [BW82] or observability issues [Hen89][BB91]. However, all the specific applications of label algebras require certain *implicit* label axioms in order to preserve legibility and terseness. Thus, the framework of label algebras provides us with "low level" algebraic specifications: in a generic way, the specific semantic aspects of a given approach (e.g. subsorting or exception handling) are specified by a well chosen set of label axioms.

When restricting our approach to positive conditional label formulas, we retrieve the classical results of the standard positive conditional approach of [GTW78] such as the existence of initial algebras and the existence of left adjoint functors for structured specifications. We also proposed a sound calculus, the label calculus, which is complete with respect to positive conditional ground formulas.

Although we have studied "structured" exception specifications in Section 8.2, we have not studied "modular constraints" according to elaborated modular semantics such as the ones of [AW86], [Bid89] or [EBO91]. Nevertheless, we have shown in Section 5 that the framework of label algebras restricted to positive conditional axioms, and consequently the one of exception algebras, form a specification frame which has free constructions [EBO91][EBCO91].[1] These results provide us with a first basis to study more elaborated notions of modularity for label specifications. However, modularity should be studied according to the specific application under consideration (behavioural specifications, exception specifications, etc.): for instance, we showed in Section 8.2 that the definition of "sufficient completeness" for exception specifications allows erroneous junk. We also pointed out in Section 8.2 that the existing frameworks for modularity do not cope with exception handling because they do not allow erroneous junk. Consequently, the definition of a suitable modular approach capable of treating algebraic frameworks with exception handling remain an open question.

Several other extensions of the framework of label algebras will probably give promising results. Intuitively, labels are unary predicates on terms. In order to facilitate certain applications of label algebras, we intend to generalize labels to "labels with multiple arguments." Higher order label specifications may also be dealt with in future work, as well as a complete label calculus compatible with these generalizations.

Last, but not least, let us mention that bounded data structures play a crucial role in the theory of *testing* because test data sets should contain many elementary tests near the

[1] but it does not form a liberal institution [GB84].

bounds. In [LeG93], exception algebras are used to extend to exception handling the theory of test data selection from algebraic specifications described in [BGM91], [BGLM93].

Acknowledgments: We would like to thank Pierre Dauchy and Anne Deo-Blanchard for a careful reading of the draft version of this paper. We would also like to thank Pippo Scollo for judicious remarks on [BL91]. Moreover, we are indebted to Maura Cerioli for a very nice counter example to the completeness of the label calculus with respect to positive conditional formulas with variables. This work has been partially supported by GDR Programmation and EEC Working Group COMPASS.

References

[AC91] Astesiano E., Cerioli M. : "*Non-strict don't care algebras and specifications*", TAPSOFT CAAP, Brighton U.K., April 1991, Springer-Verlag LNCS 493, p.121-142.

[Aig92] Aiguier, M. : "*Un calcul pour les algèbres étiquetées*", LRI, Université de Paris-Sud, 1992, Technical Report.

[AW86] Astesiano E., Wirsing M. : "*An introduction to ASL*", Proc. of the IFIP WG2.1 Working Conference on Program Specifications and Transformations, 1986.

[BBC86] 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, n.1, pp.13-45, Elsevier Science Pub. B.V., North-Holland, Nov 1986.

[BB91] Bernot G., Bidoit M. : "*Proving the correctness of algebraically specified software: modularity and observability issues*", Proc. of AMAST-2, Second Conference of Algebraic Methodology and Software Technology, Iowa City, Iowa, USA, May 1991.

[Ber86] 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, Feb. 1986.

[Ber87] Bernot G. : "*Good functors ... are those preserving philosophy !*", Proc. Summer Conference on Category Theory and Computer Science, September 1987, Springer-Verlag LNCS 283, pp.182-195

[BGM91] Bernot G., Gaudel M.C., Marre B. : "*Software testing based on formal specifications: a theory and a tool*", Software Engineering Journal, Vol.6, No.6, p.387-405, November 1991.

[Bid84] Bidoit M. : "*Algebraic specification of exception handling by means of declarations and equations*", Proc. 11th ICALP, Springer-Verlag LNCS 172, July 1984.

[Bid89] Bidoit M. : "*Pluss, un langage pour le développement de spécifications algébriques modulaires*", Thèse d'état, Université of Paris-Sud, 1989.

[Bir35] Birkhoff G. : "*On the structure of abstract algebras*", Proc. of the Cambridge Philosophical Soc. 31, pp.433-454, 1935.

[BGLM93] Bernot G., Gaudel M.C., Le Gall P., Marre B. : Proc. of the 2nd Int. Conf. on Achieving Quality in Software (AQuIS'93), Venice, Italy, 18-20 Oct. 1993.

[BL91] Bernot, G. and Le Gall, P. : "*Label algebras : a systematic use of terms*", 8th Workshop on Specification of Abstract Data Types joint with the 3rd COMPASS Workshop, Dourdan Springer-Verlag LNCS 655, pp.144-163, 1991.

[BL93] Bernot, G. and Le Gall, P. : "*Exception handling and term labelling*", Proc. of TAPSOFT'93 (FASE), Orsay, LNCS 668, pp.421-436, Springer-Verlag, 1993.

[BW82] Broy M., Wirsing M. : "*Partial abstract data types*", Acta Informatica, Vol.18-1, 1982.

[BW90] Barr M., Wells C. : "*Category Theory for Computer Science*", Prentice Hall, 1990.

[CD91] Comon, H. and Delor, C. : "*Equational formulas with membership constraints*", LRI, Université de Paris-Sud, Tech. report n.650, 1991. To appear in Information and Computation.

[EBCO91] Ehrig H., Baldamus M., Cornelius F., Orejas F. : "*Theory of algebraic module specification including behavioural semantics, constraints and aspects of generalized morphisms*", Proc. of AMAST-2, Second Conference of Algebraic Methodology and Software Technology, Iowa City, Iowa, USA, May 1991.

[EBO91] Ehrig H., Baldamus M., Orejas F. : "*New concepts for amalgamation and extension in the framework of specification logics*", Research report Bericht-No 91/05, Technische Universitat Berlin, May 1991.

[EM85] Ehrig H., Mahr B. : "*Fundamentals of Algebraic Specification 1. Equations and initial semantics*", EATCS Monographs on T.C.S., Vol.6, Springer-Verlag, 1985.

[FGJM85] Futatsugi K., Goguen J., Jouannaud J-P., Meseguer J. : "*Principles of OBJ2*", Proc. 12th ACM Symp. on Principle of Programming Languages, New Orleans, January 1985.

[GB84] Goguen, J.A. and Burstall, R.M. : "*Introducing Institutions*", Proc. of the Workshop on Logics of Programming, Springer-Verlag LNCS 164, pp.221-256, 1984.

[Gau92] Gaudel M.C. : "*Structuring and modularizing algebraic specifications: the PLUSS specification language, evolution and perspectives*", Proc. of the 9th Symp. on Theoretical Aspects of Computer Science (STACS'92), Cachan, LNCS 577, pp.3-18, Feb. 1992.

[GDLE84] Gogolla M., Drosten K., Lipeck U., Ehrich H.D. : "*Algebraic and operational semantics of specifications allowing exceptions and errors*", T.C.S. 34, 1984, pp.289-313.

[GM89] Goguen J.A., Meseguer J. : "*Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations*", Technical Report SRI-CSL-89-10, SRI, July 1989.

[Gog78a] Goguen J.A. : "*Abstract errors for abstract data types*", Formal Description of Programming Concepts, E.J. NEUHOLD Ed., North Holland, pp.491-522, 1978.

[Gog78b] Goguen J.A. : "*Order sorted algebras: exceptions and error sorts, coercion and overloading operators*", UCLA, Semantics Theory of Computation Report n.14, Dec. 1978.

[Gog84] Gogolla M. : "*Partially ordered sorts in algebraic specifications*", Proc. 9th Colloquium on Trees in Algebra and Programming (CAAP), Bordeaux, Bruno Courcelle (Ed.), Cambridge University Press, Cambridge (1984), pp.139-153.

[Gog87] Gogolla M. : "*On parametric algebraic specifications with clean error handling*", Proc. Joint Conf. on Theory and Practice of Software Development, Pisa (1987), Springer-Verlag LNCS 249, pp.81-95.

[GTW78] Goguen J.A., Thatcher J.W., Wagner E.G. : "*An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*", Current Trends in Programming Methodology, ed. R.T. Yeh, Printice-Hall, Vol.IV, pp.80-149, 1978.

[Hen89] Hennicker R. : "*Implementation of Parameterized Observational Specifications*", TapSoft, Barcelona, LNCS 351, vol.1, pp.290-305, 1989.

[LG86] Liskov B., Guttag J. : "*Abstraction and specification in program development*", The MIT Press and McGraw-Hill Book Company, 1986.

[LeG93] Le Gall P. : "*Les algèbres étiquetées : une sémantique pour les spécifications algébriques fondées sur une utilisation systématique des termes. Application au test de logiciel avec traitement d'exceptions*", PhD Thesis n.2555, Université de Paris-Sud 1993.

[Meg90] Megrelis, A. : "*Algèbre galactique - Un procédé de calcul formel, relatif aux semi-fonctions, à l'inclusion et à l'égalité*", PhD Thesis, Université de Nancy I, 1990.

[Mos89] Mosses, P. : "*Unified algebras and Institutions*", Proc. of IEEE LICS'89, Fourth Annual Symposium on Logic in Computer Science, Asilomar, California, 1989.

[MSS89] Manca, V. and Salibra, A. and Scollo, G. : "*Equational Type Logic*", Conference on Algebraic Methodology and Software Technology, Iowa City, TCS 77, pp.131-159, 1989.

[Poi87] Poigne, A. : "*Partial algebras, Subsorting, and dependent types*", 5th Workshop on Specification of Abstract Data Types, Gullane, Springer-Verlag, LNCS 332, 208-234, 1987.

[Sch91] Schobbens P.Y. : "*Clean algebraic exceptions with implicit propagation*", Proc. of AMAST-2, Second Conference of Algebraic Methodology and Software Technology, Iowa City, Iowa, USA, May 1991.

[SS91] Salibra, A. and Scollo, G. : "*A soft stairway to institutions*", 8th Workshop on Specification of Abstract Data Types, Dourdan, 1991.

[WB80] Wirsing M. and Broy M. : "*Abstract data types as lattices of finitely generated models*", Proc. of the 9th Int. Symposium on Mathematical Foundations of Computer Science (MFCS), Rydzyna, Poland, Sept. 1980.

[Wir90] Wirsing M. : "*Algebraic specifications*", HandBook of Theoretical Computer Science, North Holland, 675-788, 1990.