Exception Handling in a Strongly Typed Object Oriented Language

Noemi Rodriguez
Roberto Ierusalimschy
José Lucas Rangel

Departamento de Informática — PUC-Rio
Rua M. S. Vicente 255, Gávea
22453-900 Rio de Janeiro, RJ

noemi, roberto, rangel@inf.puc-rio.br

Abstract

This paper describes a proposal for combining exception handling and static type checking in an object oriented language. Basic issues for exception handling mechanisms, such as termination versus resumption and the association between signaler and handler are discussed. The sub-typing rules of a statically checked OOL are extended in order to take exceptions into account. Finally, some examples are presented to illustrate the proposed mechanism.

Keywords

object oriented languages, static type checking, exception handling

1 Introduction

Exception Handling Mechanisms (EHMs) have been widely recognized as useful in Object Oriented Languages (OOLs), and have been introduced in languages such as C++ [ES90], Eiffel [Mey88], and Sather [SOM93]. Nevertheless, the use of EHMs is still controversial, and there is no unanimous opinion on the subject: some OOLs use no EHM at all, and each of the languages that include exception handling does it in a different way.

This paper describes the EHM of the programming language School. School [RIR93] is an OOL featuring separate hierarchies for specifications and implementations, structural sub-typing, and generic types with restrictions. The main goal of School is to provide the same (or higher) degree of reuse and flexibility attained in other OOLs while guaranteeing the absence of execution errors due to type problems.

In School, a *type* corresponds to a partial specification, mainly stating the names, parameters and results types for the operations valid on all objects it describes. Exception signaling is viewed as one form of returning from a method, with the exception viewed as the result returned in this case. The result type of a method can thus be considered the union of the "normal" result type and of the type of the exceptions possibly signaled by this method. The reason for this approach is that the exclusion of the exceptional result from the type specification would defeat our goal of complete static checking, since in this case an operation would have the possibility of finishing without returning the stated result type.

Adding exceptions to specifications brings in a new problem, namely, their integration into the sub-typing rules. The EHM we propose for School in this paper deals with this problem; we define rules for inheritance which permit the inclusion of exceptions while still guaranteeing the absence of dynamic type errors.

The proposed rules are quite intuitive; exceptions are viewed as objects and are required to follow the subtype patterns defined by the types in which they are signaled, in much the same way as parameters to methods. At first glance, this approach could seem to have the consequence of forcing all subtypes of a given type T to signal at least the exceptions signaled in T. This would be 0, since the refinement of a method may eliminate its exceptions. Our proposal handles this situation adequately, as we will see.

2 Related Work

In this section we briefly discuss some proposals for exception handling, attempting to establish their relation to our work.

Exception handling is viewed in two main ways. Some authors, such as [LS79] and [YB85] consider it to be an important linguistic facility, to be used by programmers in the treatment of either errors or simply unusual conditions. This can be contrasted with the paradigm presented in the Ada rationale, where exceptions were introduced for dealing with fatal errors, making the provision of "graceful degradation" the main role of the EHM.

In the object oriented world, C++ and Sather are the main representatives of the first approach, where exception handling is viewed as a general programming facility. In C++, an exception can be signaled at any point in the code by *throwing* an object. When a *throw* statement is executed, a corresponding *catch* statement is sought along the dynamic chain. A *catch* statement matches a *throw* statement if it specifies a class which is the same or a superclass of the thrown object. A similar exception handling mechanism is defined for Sather ([SOM93]).

One point in these proposals which seems to present a problem is the fact that they introduce a possible way of determining the exact type of an object at runtime. For instance, if a C++ class C has as descendants classes C_1, C_2, and C_3, a member function may receive as a parameter a polymorphic object of class C and *throw* it, and with handlers defined in its caller for classes C_1, C_2, and C_3, the execution path will correspond to a test of the class of the thrown object. The use of dynamic tests for establishing the real type of a polymorphic variable is a controversial issue; the case against it is argued, for instance, in [Mey88].

Another controversial point in the C++ and Sather EHMs is the uncontrolled search of handlers along the dynamic chain, as will be discussed in section 4.1.

The programming language Eiffel [Mey88] also includes exception handling. However, differently from what we have here, the approach used in that proposal is that an exception corresponds to a failure, or to the "inability of a routine to maintain its part in a contract".

[Car92] presents another proposal for exception handling in OOLs, which presumes the definition of different *levels* of handling. When an exception e is signaled by a method M, there may, in the first place, exist a local handler for e in M itself. In this case, this code is executed and execution proceeds normally. Handlers may also be defined at the class level. If a local handler is not defined for e, and the class C which contains M contains a (defined or inherited) handler for e, M will be terminated, this handler will be executed and control will be returned to the caller of M with no indication that an exception has occurred. Finally, if class C does not contain a handler for e, e will be propagated to the caller of M. In this case, the search for a handler will follow the same steps described above.

The proposal described above has as a goal the conciliation of object oriented features with efficiency. However, from the point of view of strong typing, it poses some problems. In Tool [SPA92], the base language used in that proposal, exception handlers are not defined in class interfaces, thus making it difficult for a programmer who is using a class to understand the sequence of flow in the case of signaling. Besides, after the execution of a handler, control is directly returned to the caller of the signaling method; to guarantee static checking, the handler must then return a value of the same type as the declared returned value of the signaling method, but this cannot possibly be enforced by the compiler in the presence of automatic propagation. Finally, it seems to us that an exception which the class itself knows how to handle is not really an exception, being maybe more adequately programmed as a normal call to a local method. The gain in efficiency resulting from the fact that control does not return to the signaling method before returning to its caller could be achieved by introducing tail call elimination in the compiler.

In non object oriented languages, several proposals for exception handling exist, as described in [Goo75], [Lev77], [LS79], [YB85], and [I^+79]. These differ mainly in the goal of the exception handling mechanism, in the choice of paradigms for association of signalers to handlers and also between termination and resumption after handling has taken place. These issues will be discussed in section 4.1.

Black [Bla82] argues the case against exception handling mechanisms. He discusses how conventional programming language mechanisms may be used in many typical exception handling examples. To describe the possibility of exceptional termination of a routine, Black proposes the use of unions in return type declarations. In this aspect, the approach used in School is similar to Black's. However, in his proposal, no semantic distinction is made between normal and exceptional cases, making it mandatory that the programmer explicitly test the result type after each routine call. As discussed in [Rod90], the fact that a mechanism may be simulated by other features of a language is not sufficient reason for excluding it from the language; otherwise, assembly would be in common use to this day.

3 School

School is a programming language designed to keep the basic semantics of object-oriented languages, mainly Smalltalk, while offering a secure type system. All values in School are objects. Variables do not contain objects, but references to them, and both assignment and parameter passing manipulate references. All communication between objects is based on late-binding, and the binding depends solely on the receiver.

The rule that has guided the development of School's type system has been to avoid any construction that can cause a "message not understood" error at run-time; obeying this rule, we have tried to make the language as flexible as possible. In order to achieve higher flexibility, separate hierarchies for types and classes (specifications and implementations), structural subtyping, and constrained genericity were introduced in School. As a result, we obtained a terse language that can model most type facilities presented by other OOLs.

The importance of separate hierarchies in an OOL is now widely recognized [Val]. The main idea underlying this concept is the understanding of types as specifications, and classes as implementations. The type of an object is its external appearance, that is, its interface to the outside world. In School, a type declaration states all operations available in objects of that type, and the types of parameters and eventual results of each operation. For example, a point could be declared as:

```
type Point is
  proc x -> Int;
  proc y -> Int;
  proc moveBy (p : Point);
end Point;
```

On the other hand, the class of an object dictates its internal shape, that is, its structure and the code to handle it. A possible School implementation for points is shown below. Class declaration headings may include the external object type (*Point*, in the case of objects of the class *PointCart*).

```
class PointCart : Point is
  var xPos,yPos : Int ;    -- instance variables
  constructor newC (xi, yi: Int) is
    xPos := xi; yPos := yi;
  end;
  proc x () -> Int is Result := xPos end;
  proc y () -> Int is Result := yPos end;
  proc moveBy (p : Point) is
    xPos := xPos + p.x;
    yPos := yPos + p.y;
  end;
end PointCart;
```

Note that the external type *Point* of objects of class *PointCart* is declared after the class name. In general, a type may have several implementations. In declarations of variables, parameters, etc., only the type is given; the class must be specified only at the time of the creation of an object. As stated above, School adopts referential semantics, so that variables contain only references to objects; until an object is created for a variable, it refers to the *nil* object. The *nil* object is the sole value of type *Nil*, which is defined as a subtype of any type possibly declared in School. A consequence of this definition is that no type can be a subtype of *Nil*.

New objects are created through calls to *constructors*, which explicitly refer to the class name, with the "!" syntax, as in

p := PointCart!newC(10,20)

A call to a constructor allocates space for the created object, after which the constructor code is executed, initializing the object. Constructors are not part of type specifications. One of the reasons for this is that the creation of an object is dependent on its implementation, and so it does not make sense to specify the needed information (initialization parameters) in a type description. Besides, some classes may need several different constructors, which would make the specification rather awkward.

To say that a type A is a *subtype* of a type B means that A is compatible, from an external point of view, with B. In other words, an object of type A can be used wherever an object of type B is expected. Notice that A and B do not need to have similar implementations. On the other hand, to say that a class A is a *subclass* of a class B means that A inherits methods and variables from B. As they do not need to have compatible interfaces, A can freely modify the inherited features. Using the classification proposed in [WZ88], sub-typing must have behavior compatibility, or at least signature compatibility, while sub-classing is free to adopt cancel compatibility. Therefore, there is no compromise between the flexibility of cancel compatibility and the security of strong typing.

The independence between sub-typing and sub-classing allows a programming language to adopt *structural type compatibility*. That means that a type is a subtype of another one as a consequence of their compatibility, and not the other way round. The problem which remains is how to assess compatibility.

Following our main rule, stated above, we want to allow a type A to be a subtype of B as long as there is no possibility of errors when using an A object in the place of B. "Message not understood" errors (no such method in an object) can be avoided by defining that a type A is a subtype of B ($A \prec B$) if and only if, for each method X in B, with signature $P_{B_1} \times \ldots \times P_{B_n} \rightarrow R_{B_1} \times \ldots \times R_{B_m}$, there is a method X in A, with signature $P_{A_1} \times \ldots \times P_{A_n} \rightarrow R_{A_1} \times \ldots \times R_{A_m}$, where for all $i \leq m$, $R_{A_i} \prec R_{B_i}$ and, for all $i \leq n$, $P_{B_i} \prec P_{A_i}$. If this condition is satisfied, we say that the signature of X in A is a subsignature of the signature of X in B. The apparent inversion in the last condition is known as the "contra-variance rule", and is needed to assure correctness [CW85]. A formal definition of sub-typing is given in [Ier92]. In that paper, it is shown that this definition is not only sufficient but also necessary to ensure the absence of run-time errors. In any language accepting A in place of B with $A \not\prec B$, it is possible to write a routine that generates a run-time error.

As an example of sub-typing, consider the declarations:

```
type Point is
  proc x -> Int;
  proc y -> Int;
  proc moveBy (p : Point);
end Point;

type Point1 is                    type Point2 is
  proc x () -> Int;                 proc x () -> Int;
  proc y () -> Int;                 proc y () -> Int;
  proc moveBy (p : Point1);         proc moveBy (p : Point);
  proc copy () -> Point1;           proc copy () -> Point2;
end Point1;                       end Point2;
```

We have: `Point2` $\prec$ `Point`, but not: `Point1` $\prec$ `Point` (incompatible parameter for method `moveBy`), or `Point` $\prec$ `Point1` (missing method `copy`).

The main advantage of structural compatibility is its flexibility. However, this same flexibility is sometimes considered a drawback. The argument is that a type can be a subtype of another one by mistake. This problem can be avoided with the use of properties [AvdL90]. A property is just a name attached to a type, intended to represent a characteristic of the type. For instance, a type `Stack` could have `LIFO` as one property. A type A is then considered subtype of B only if, besides the above condition, A includes all properties of B. Because properties can be easily simulated through dummy functions, having the property name and no parameters or results, School does not include this facility.

While the type hierarchy is automatically deduced from the types (structural compatibility), the class hierarchy is built on explicit declarations (nominal compatibility). When a class is declared as *including* other classes it inherits, by default, all variables and methods from them.

The collection of headers of all procedures of a class (locally defined or inherited from included classes) compose the *internal type* of a class. If a type is given in the class definition, this type is the *external type* of the class, and it must be a super-type of the internal type. If a type is not given, the external type of a class is its internal type. If a class includes another one, its internal type must be a subtype of the internal type of the included class.

In order to keep independent hierarchies, the external type of a class has no relationship with the external types of included classes.

4 Exception Handling

Before describing a specific proposal for Exception Handling, we discuss in Section 4.1 some basic issues this proposal must address. Section 4.2 presents exception handling in School.

4.1 Basic Principles for Exception Handling

A first issue to be considered is the choice between the termination and resumption paradigms. In the *termination model*, the routine where an exception is signaled is interrupted at the point of signaling and its current activation is terminated. This is the model used in C++, CLU, and Ada. In the *resumption model*, execution of the signaling routine is resumed after the handler has finished its job. The idea is that the handler may "clear" the exceptional condition, allowing the signaling routine to proceed with its work. This model is supported, along with termination, in [Lev77] and [YB85].

The advantages of supporting resumption are basically connected to the programming flexibility which may be gained. In what follows, we argue that this flexibility comes at a very high cost, explaining our choice for termination.

Our first argument against resumption is based on the reasoning presented in [Cri79]. If a program unit M calls another unit N, and N signals an exception, resumption implies that the handler in M should, at this point, reestablish some assertion which has been negated (so as to allow the execution of an operation which has been diagnosed as invalid). This condition may be a precondition or any other invariant in the body of N. If it is a precondition of N, then it seems to make more sense for M to call N again with the correct parameters. If it is not, it is some intermediate condition in the implementation of N. But in this case, in order to reestablish it, or in fact even to understand it, it will be necessary for M to have knowledge about invariants in N which might include variables and values not apparent from the outside, which means knowledge about N's internal state. This is in disagreement with the principles of abstraction and modularity.

Another argument against resumption is that, after the handler has executed, it will be necessary for the signaling routine to once more test the offending condition, since it is possible that the handler has not been able to correct it. In this case, a stronger exception will have to be signaled, or a loop inserted to guarantee a valid final state. This implies greater complexity in the programming of a signaling routine.

It has also been pointed out ([Bla82]) that the cases where it is really possible to call a handler and continue execution without any undue interaction between different levels of abstraction can be very well modeled by the use of procedure parameters. This is specially natural in the context of object orientation, as will be discussed in section 5.

The discussion above has led us to the conclusion that the expressive power gained by introducing resumption in an EHM does not compensate the increased programming complexity and lack of programming modularity which come as consequences. Resumption is thus not included in the EHM proposed for School. Therefore, the signaling of any exception always implies in the termination of the signaling method.

Another important issue in the definition of an EHM is the rule for association between signaler and handler. Some proposals, such as [Ada83] and [Str92], allow a search along the dynamic chain until a handler for the signaled exception is found. This requires a default handler at the outermost level, so that a handler is always found. Others, such as [LS79] and [YB85], require that a handler be present in any routine which executes a call to a potentially signaling routine.

We consider this second option to be more appropriate. Again we adopt arguments based on principles of modularity, presented in [YB85] and [Cri79]. The implementation of each routine in a program must be based on calls to other routines whose specification (parameters, results, and signaled exceptions) is known but whose implementation should be irrelevant to the new routine. Thus, it cannot be expected that a routine define adequate handlers for exceptions signaled by routines not directly called by it.

Another relevant point is that dynamic search for a handler must necessarily allow for the possibility of no handler being found. This situation typically results in an execution error. Since we consider the exception signaling to be part of type definitions, this would be an execution error due to typing, which is exactly the kind of errors we are trying to eliminate.

Therefore, in School, every exception signaled by a routine must be handled in the caller of this routine, using the handler associated to the innermost block enclosing the call. Since the declaration of possibly signaled exceptions is part of the declaration of methods, the compiler can easily check that all calls are appropriately associated to handlers.

Note that the handler may simply pass the signal up the dynamic chain, if the same exception is declared in the calling routine. In such cases, the implementation may guarantee the "direct jump" to the effective handler, not wasting the time to go through each step in the calling chain; conceptually, however, this is not important.

4.2 Exceptions and Sub-typing

In School, exceptions are signaled by a *signal* statement; when this statement is executed, the current routine is terminated. Signatures of routines must list the exceptions possibly generated by them, as in CLU. Handlers for exceptions are defined in *except when* statements, which can be bound to any structured or simple statement, using the syntax:

```
except when <exceptionType> do <handler> end
```

As in C++, a handler matches an exception if it specifies any super-type of this exception. Any call to a routine must lie in the scope of exception handlers for exceptions possibly signaled by the routine. So, *signaling* in School is similar to *throwing* in C++, with the imposition that the exception be necessarily caught in the routine which made the call to the signaling routine.

In order to statically ensure the above condition, exceptions must now be taken into consideration when comparing routine signatures. We will consider in the following discussion that each method signals at most one exception. We will see later that in fact only one exception type is necessary.

We extend the definition in the previous section, saying now that a type A is a subtype of B ($A \prec B$) if and only if, for each method X in B, with signature

$$P_{B_1} \times \ldots \times P_{B_n} \rightarrow R_{B_1} \times \ldots \times R_{B_m} \text{ signals } E_b,$$

there is a method X in A, with signature

$$P_{A_1} \times \ldots \times P_{A_n} \rightarrow R_{A_1} \times \ldots \times R_{A_m} \text{ signals } E_a,$$

where for all $i \leq m$, $R_{A_i} \prec R_{B_i}$, for all $i \leq n$, $P_{B_i} \prec P_{A_i}$, and $E_a \prec E_b$.

To see why the exception signaled by a method X in a type A must indeed be a subtype of the exception signaled by method X in any super-type of A to guarantee static checking, consider the following example.

```
type B is
   proc m() signals E;
end B;
...
   var b: B;
   ...
   b.m() except when E: ...
...
```

The compiler is able to check that the call `b.m()` is in the scope of a handler for an exception of type E. Consider now that at runtime the value of variable b is an object of type A, a subtype of B. If method X in A signals an exception of type I, and if I is not a subtype of E, we may incur in an execution error, for there is no handler for this type of exception.

The restriction above is also intuitive, since it seems natural that subtypes define more specific descriptions of the values their methods may return than their super-types.

Methods in School are limited to signaling at most one type of exception. This is not a severe limitation, because such type may be in fact a super-type of several different types of exceptions signaled by the method, or, in particular, a type implemented by different classes of exceptions, as will be discussed in the examples.

From the discussion above, it may seem that, when a programmer defines a type T meant to be a subtype of a type S, each method in type T is required to signal at least the exception signaled by the corresponding method in S. This requirement would be rather inconvenient. As mentioned in the introduction, it is quite natural for a refinement of a method m to deal better with exceptional cases, and possibly eliminate exceptions signaled in m. This is dealt with in School in the following way. A method declaration containing no signaled exceptions is treated by the compiler as syntactic sugar for the same declaration with the signaling of an exception of type *Nil*. Since*Nil* is a subtype of any type, a method signaling no exceptions creates no clash whatsoever with the subtype hierarchy.

Unhandled Exceptions

In this proposal we repeatedly stress the need to avoid the possibility of unhandled exceptions since these can lead to unexpected execution errors. However, it may be considered too hard on the programmer to be forced to always define handlers for all exceptions possibly signaled, even when he is sure that signaling will not occur. Most EH proposals offer some facility for dealing with this situation. One example of such a facility is automatic propagation, discussed in section 4.1. CLU offers a limited form of automatic propagation. If a signaled exception does not have a corresponding handler in the calling routine, this routine terminates signaling the predefined exception *failure*. Although this makes automatic propagation take on a much more controlled form, it still allows programs to terminate due to execution errors (unhandled exception) which could have been foreseen by the compiler.

Another solution offered in several languages is to adopt a predefined name which encompasses all exceptions. Examples of this are the names *others* in Ada and CLU and ... in C++. A handler associated to this name will match the occurrence of any exception, making it easy for the programmer to define a "default" handler in a module or routine.

This facility is offered in School by requiring that all possibly signaled exceptions be subtypes of a predefined type *Exception*, defined as:

```
type Exception is
  proc GetMsg -> String;
end Exception;
```

Since all exceptions must be subtypes of *Exception*, a default handler can be defined using this type. Thus, to propagate exceptions in any block, one may associate to the block a handler like:

```
except when (e:Exception) do
  signal e;
end
```

5 Examples

In this section some programming examples are used to illustrate the proposed EHM. In order to make the analysis of the application of the proposed mechanism more interesting, the examples in this section have been selected from the existing Exception Handling literature. This allows for comparison with other proposed solutions, and avoids the pitfall of specially chosen "well-behaved" examples.

The first example is drawn from [LS79], and is also discussed in [YB85]. In this example, *sum_stream* is a procedure which reads decimal numbers from a character stream and returns their sum. Input must contain a sequence of number fields separated by blank spaces. Each number field must consist of a nonempty sequence of digits, optionally preceded by a single minus sign. *sum_stream* signals *overflow* if the sum of numbers or an intermediate sum is out of the implemented range of integers. *Unrepresentable_integer* is signaled if the stream contains an individual number that is out of the implemented range of integers. *Bad_format* is signaled if the stream contains a field that is not an integer.

The implementation of *sum_stream* presented in [LS79] consists of a simple loop which accumulates the sum, using a procedure *get_number* to remove the next integer from the stream. *get_number* signals *end_of_file* if the stream contains no more fields, in which case *sum_stream* will return the accumulated sum. *get_number* also signals *bad_format*, if an illegal input field

```
type badFormat is
  proc GetMsg() -> String;
  proc DefaultResult() -> Int;
  proc OffendingString() -> String;
  proc EndOfFile() -> Boolean;
end badFormat;
```

```
class unrepInt: badFormat is
var
  msg: String;
  ...
constructor NewUnrep (off: String) is
  msg := cString!New(off);
end NewUnrep;

proc DefaultResult() -> Int is
  Result := MAXINT;
end DefaultResult;
...
end unrepInt;
```

Figure 1: Declaration of exceptions used by *sum_stream*

in encountered, and *unrepresentable_integer*, if the number is too large for the machine representation; these exceptions are passed upward by *sum_stream*. Finally, a handler for *overflow* in *sum_stream* catches exceptions signaled by the integer adding routine.

This example is quite naturally translated into School. The main interest of this implementation is in dealing with methods which signal different exceptions using a mechanisms which allows only one type of exception to be signaled. Figure 1 shows the specification of an exception type, *badFormat*, which is used to describe the different exceptions signaled by *get_number*. The same figure presents the definition of one of the classes which implements this type. The other classes would be similar and are not shown due to lack of space.

Different classes *invChar*, *unrepInt*, and *endOfFile* may implement the same type *badFormat*, representing, respectively, an invalid character, a sequence of valid characters which compose an illegal integer, and end of file.

Figure 2 shows the implementation of methods *getNumber* and *sumStream*. The implementation of *sumStream* shows how, on return from getNumber, a calling method may use the different possible results by calling the specified methods for *badFormat*. A small simplification is introduced. The same exception, *unrepInt*, is signaled in the case of one individual number in the input stream being too large and in the case of overflow in the sum of the numbers.

Note the use of the notation *<name>: <exceptionType>* to declare the exception which is being handled.

The important point in the example is that, since signaling corresponds to returning an "exception object", this object may carry with it all necessary information. In non-object oriented languages, different exceptions must be signaled to convey information about the exceptional event necessary to its handling. In an object oriented framework, objects carry information with themselves; thus, one type is sufficient for the description of different exceptional events.

The next example is drawn from [CG92], which explores the idea of *data oriented* exceptions. This idea was also present in Levin's work, and allows for the association of handlers to data instead of blocks of code. [CG92] proposes an extension to Ada. In this extension, any new data type must declare the exceptions possibly signaled by its operations, and the declaration of variables must associate handler routines to each of these exceptions.

In an object oriented language this can be easily modeled: the definition of a data type may include handling routines. Different implementations of this type may code this handling in different ways. Figure 3 shows the coding in School of one of the examples in [CG92].

```
proc getNumber() -> Int
              signals badFormat
is
  f : String := getField(input);
    except when (e: badFormat) do
      signal e
    end;
  Result := f.s2i();
    except when (e:badFormat) do
      signal e
    end;
end getNumber;
```

```
proc sum_stream() -> Int
              signals badFormat
...
while true loop
    s := s + getNumber();
end;
  except when (e: badFormat) do
    if e.EndOfFile() then
      Result := s
    else
      Err.Write(e.GetMsg());
      signal e
  end;
end;
...
end sum_stream;
```

Figure 2: Implementation of *getNumber* and *sum_stream*

In this example, different instances of type *Stack* must deal differently with overflow. Type *tStack* specifies that every stack must have a handler for overflow. Classes *Stack1* and *Stack2* implement different handlers. The code contains a method *M* which receives a stack object as a parameter and executes a call to method *Push*. The handler for exception *overflow*, signaled by *Push*, simply calls the specific handler for this object, which will depend of its class.

6 Final Remarks

This work is part of a project where the goal is to study the meaning of types in object oriented languages and to extend this meaning in order to allow the type system to check different interesting properties in a program. The programming language School, as described in [RIR93], was developed as a first phase towards this goal. Its main contribution was to show that it is possible to abolish all run time type errors from an object oriented language while still retaining programming flexibility. The central idea used for this is the concept of a type as a specification of an object's behavior.

The work presented in this paper is an extension of this first phase. We believe that the addition of exception handling to a language enhances programming flexibility. However, to maintain the concept of a type as a specification of an object's behavior, it has been necessary to describe the possibility of a method terminating through signaling in the type declaration, ie, in the method's header. This in turn implied in studying the role of exception signaling in the subtype rules. The EHM described for School deals with these issues, integrating exceptions and sub-typing in order to guarantee static checking.

The School project has been investigating the role of type systems in ensuring other properties besides the absence of execution errors. Specifically, a solution to the problem of statically determining whether a function is side-effect free has been proposed. This solution, which relies completely on the type system, is described in [RI94].

We are now working on an extension to School for distributed programming. The goal again is to investigate the role of the type system in the provision of a programming facility, in this case distribution.

```
type tStack{T} is
  proc Push(T) signals overflow;
  ...
  proc OverFlowHandler();
end tStac;

class Stack1{T}: tStack{T} is
  ...
  proc OverFlowHandler is
    Expand(growthRate)
  end OverFlowHandler;
end Stack1;

class Stack2{T}: tStack{T} is
  ...
  proc OverFlowHandler is
    Retain(90);
  end OverFlowHandler;
end Stack2;

...
  proc M(s: tStack{Int}) is
    ...
    s.Push(i)
      except when overflow do
        s.OverFlowHandler()
      end;
    ...
  end M
```

Figure 3: implementation of *data oriented* exceptions in School

References

[Ada83] ANSI. *Ada Programming Language*, 1983. ANSI/MIL-STD 1815A.

[AvdL90] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. *Sigplan Notices*, 25(10), 1990. OOPSLA/ECOOP'90 Proceedings.

[Bla82] Andrew Black. *Exception Handling: the Case Against.* PhD thesis, University of Oxford, 1982.

[Car92] Sergio Carvalho. Exception handling in object oriented languages: A Proposal. Technical report, Departamento de Informática, PUC-Rio, 1992.

[CG92] Q. Cui and J. Gannon. Data-oriented exception handling. *IEEE Trans. on Software Engineering*, 18(5), May 1992.

[Cri79] Flaviu Cristian. *Le Traitement des Exceptions dans les Programmes Modulaires.* PhD thesis, Université Scientifique et Médicale de Grenoble, 1979.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Goo75] J. Goodenough. Exeption handling: issues and a proposed notation. *Comm. ACM*, 18(12), December 1975.

[I⁺79] Jean Ichbiach et al. Rationale for the design of the Ada programming language. *SIGPLAN Notices*, 14(6), June 1979.

[Ier92] Roberto Ierusalimschy. A denotational approach for type-checking in object oriented languages. Technical Report 12/92, Departamento de Informatica, PUC-Rio, Rio de Janeiro, 22453, 1992.

[Lev77] Roy Levin. *Program Structures for Exceptional condition Handling.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, jun 1977.

[LS79] Barbara Liskov and Alan Snyder. Exception handling in CLU. *IEEE Trans. on Software Engineering*, SE-5(6), November 1979.

[Mey88] Bertrand Meyer. *Object Oriented Software Construction.* Prentice-Hall, 1988.

[RI94] N. Rodriguez and R. Ierusalimschy. Side effect free functions in object-oriented languages. In *XXI SEMISH*, 1994.

[RIR93] N. Rodriguez, R. Ierusalimschy, and J. L. Rangel. Types in School. *Sigplan Notices*, 28(8), 1993.

[Rod90] Noemi Rodriguez. Tratamento de exceções. Technical Report 10/90, Departamento de Informatica, PUC-Rio, Rio de Janeiro, 22453, 1990.

[SOM93] C. Szypersky, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. Technical Report TR 93-064, ICSI, Berkeley, CA, 1993.

[SPA92] SPA, Sistemas, Planejamento e Análise. *TOOL for Windows: the object oriented language*, 1992.

[Str92] B. Stroustrup. *The C++ Programming Language*. Prentice-Hall, 1992.

[Val] The interoperable objects revolution. R. Valdés (ed). Special Issue Dr. Dobb's, winter 94/95.

[WZ88] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP'88 Proceedings*, pages 55–77, 1988. LNCS 322.

[YB85] Shaula Yemini and Daniel Berry. A modular verifiable exception handling mechanism. *ACM Trans. on Programming Languages and Systems*, 7(2), April 1985.