Exceptions as Types

Pascal Leroy
Senior Staff Software Engineer
Rational Software Corp.
pleroy@rational.com
8 June 2001

Purpose

The purpose of this paper is to propose a new language feature, dubbed *exception types*, to support a more flexible exception paradigm in Ada. This mechanism attempts to retain compatibility with Ada 95, while providing a general communication framework similar to that which exists in C++ or Java.

Ada 83 had a very basic exception mechanism. It was not possible to pass information when raising an exception, and exceptions were not integrated with the overall type structure of the language, making it impossible to pass exception-related information to subprograms, or to store it in data structures, etc.

Ada 95 added slightly more powerful capabilities in this area, with the facilities provided by the predefined package Ada.Exceptions. However, the new mechanism is still only intended for very basic error detection and logging, and doesn't provide a general-purpose communication mechanism. It has the merit of clarifying the notions of exception identity and exception occurrence, but exceptions are still not integrated with the typing model like protected objects or tasks are.

It is in fact possible to use the Message associated with an exception occurrence to pass (relatively small) objects when raising an exception, but this has to be done by marshalling the object into the string. While adequate for some purposes, this approach is unnecessarily contrived.

This situation is very unfortunate, because other modern programming languages like C++ or Java, which are in other ways inferior to Ada, have a much more powerful exception mechanism.

An Object-Oriented Analogy

Assume for a moment that we wanted to represent exceptions using object-oriented inheritance, much like what is done in Java (bear in mind that this is only an analogy, not the real mechanism proposed by this paper). One way to do it would be to posit an anonymous type from which all exceptions are derived:

```
type root_exception_type is
   abstract tagged
   limited private;
```

From this type, exceptions would be declared as type extensions. So any good old Ada 95 exception like:

```
Ada95_Exception : exception;
```

would be effectively declared by:

```
type Ada95_Exception is
   new root_exception_type
   with record
      Message : String (…);
   end record;
```

Note that the component Message reflects the fact that every exception occurrence in Ada 95 may be parameterized by a string.

Following the same model, we would be able to declare exception parameterized by non-String data:

```
type User_Exception is
   new root_exception_type
   with record
      C1 : T1;
      C2 : T2;
   end record;
```

or extend existing exceptions to associate more data with an exception occurrence:

```
type Extended_Exception is
   new User_Exception
   with record
      C3 : T3;
      C4 : T4;
   end record;
```

With this analogy, the exception identity corresponds roughly to the tag of the type, and an exception occurrence is more-or-less an object of type *root_exception_type*'Class.

There are a number of reasons why we don't want to use tagged types to represent exceptions, though. First, some of the rules related to tagged types would cause inacceptable limitations; for instance, it would not be possible to declare an exception in a nested scope, because that would violate the accessibility rule of RM95 3.9.1(3). Second, existing implementations might use for exceptions a representation totally different from that of tagged types; we don't want to force these implementations to change. Finally, many of the capabilities of tagged types, like dispatching calls or membership tests, are irrelevant in this context.

A Proposal

Still, the above analogy has the merit of showing a number of underlying principles:

- An exception declares a type, which is characterized by its exception identity.
- From that type, exception occurrences may be created, typically by a raise statement.

I will now use these principles to present a proposal for extending the exception model of Ada 95.

Exception Types

A new class of types, called *exception types*, is added to the language. Exception types are untagged, composite, limited types (much like task types). Note that this has the consequence that exception types are return-by-reference types, which simplifies implementation.

There is no such thing as a formal exception type, so an exception type may only be passed to a generic formal limited private type or a formal derived type whose ancestor is an exception type.

A *root exception type* may be declared by a new form of type declaration:

```
type User_Exception is
   exception
      C1 : T1;
      C2 : T2;
   end exception;
```

A *derived exception type* may then be declared by extending the above type:

```
type Extended_Exception is
   new User_Exception
   with exception
      C3 : T3;
      C4 : T4;
   end exception;
```

(The syntax here is only a suggestion, and could be reworked if need be; I just tried to make it similar to tagged types but still sufficiently distinct that it doesn't become confusing.)

A conventional exception declaration like:

```
Ada95_Exception : exception;
```

is then taken, for compatibility, to declare a root exception type with a single component Message of type String:

```
type Ada95_Exception
   (Length : Natural) is
   exception
      Message : String
                (1 .. Length);
   end exception;
```

An exception type may be used to complete a limited private type.

Even though exception types are not tagged, the attribute ‘Class is defined for them, and designates a class of types as described in RM95 3.2(2). A class-wide exception type is especially useful as a parameter type, and in an exception handler, as it makes it possible to operate on any exception in an entire hierarchy.

Raising An Exception

Raising an exception creates an exception occurrence (it is also possible to create exception occurrences by declaring objects of exception types, but this is not too interesting as exception types are limited).

For compatibility, an exception declared by a conventional exception declaration may be raised by a conventional raise statement:

```
raise Ada95_Exception;
```

On the other hand, an exception declared by an exception type declaration must be raised by a new form of raise statement, which specifies values for all the components of the exception occurrence:

```
raise Ada95_Exception'
        (Length => 3,
         Message => "foo");
```

This is the only context where an aggregate for an exception type is legal (aggregates may not normally be of a limited type). Note that extension aggregates are not usable in this context because exception types are not tagged.

Exception Handling

An exception handler like:

```
when E :
     User_Exception => …
```

catches any exception occurrence of type User_Exception. Similarly an exception handler like:

```
when E :
     User_Exception'Class => …
```

catches any exception occurrence of an exception type covered by User_Exception'Class.

All the handlers in a given block must have non-overlapping exception types. For example, the following is illegal:

```
exception
   when User_Exception =>
   when Extended_Exception =>
```

This rule is different from what Java does, but it seems more consistent with other rules of Ada (e.g., for case statements or conventional exception handlers). It ensures that which alternative is selected doesn't depend on the lexical order of the handlers. Inside a handler, the choice parameter provides a constant view of the exception occurrence being handled. Component selection can be used to extract information from the occurrence:

```
exception
   when E : User_Exception =>
      if Func (E.T1) =
         Func (E.T2) then …
```

An others choice may be used to cover all the exception types not covered by other handlers.

Implementation Issues

At this point, it is worth giving some thought the implementation complexity of this proposal. While it is hard to foresee the implementation impact of language changes, it is clear that three areas have to be considered:

- Syntax. Assuming that the new syntax doesn't make the grammar hopelessly ambiguous, this part should not be problematic.
- Static semantics. Exception types are a new concept, and presumably there would be a number of legality and static semantics rules associated with them. However, one area where they have very little impact is generics, because we are not adding new formal types. In general exception types are very similar to protected or task types, so one would expect that they could be handled similarly by compilers.
- Dynamic semantics. I can see three potentially problematic areas here:
    - Exception types can be unconstrained types, and therefore the occurrences may have an arbitrarily large size. In Ada 95, the Message string could be limited to 200 characters, so this might put an additional burden on implementations. On the other hand, this is somewhat mitigated by the fact that exception occurrences are limited, so they cannot be resized at the drop of a hat.
    - Exception types may have controlled subcomponents. From a language design standpoint, this requires to define precisely when occurrences are finalized. From an implementation standpoint, this is another interaction between finalization and the rest of the language.
    - Class-wide types appear problematic. Existing implementation typically allocate a unique identifier to each exception declaration, and use these identifiers when raising/handling an exception. They are not prepared to deal with hierarchical relationships between these identifiers. It remains to be seen if/how such relationships could be represented efficiently without undue impact on implementations.

It might be possible to limit the dynamic semantics complexity by imposing restrictions on the contents of exception types: it seems quite unreasonable to have tasks or protected objects in an exception type. But in terms of language design, it is hard to see how this can be done without violating the contract model of private types.

Compatibility Issues

The question of compatibility with the facilities currently provided by Ada 95 is a thorny one. One might be tempted to declare the entire package Ada.Exceptions obsolescent, and say that the proposed capabilities would effectively replace it. Some of the operations in that package would still be worthwhile for exception types (e.g., Exception_Identity or Exception_Name) but they could be provided by language-defined attributes applicable to exception types.

But then there is the issue of type Exception_Occurrence. When existing Ada 95 code handles an exception, it gets an occurrence of type Exception_Occurrence, and there is only one type Exception_Occurrence for all the exception declarations. With the proposed model, code which handles an exception gets an occurrence of some exception type, and all the exception types are different types. It appears hard to reconcile the two views.

Conclusion

Undoubtedly, if this proposal is deemed worth pursuing, there is still a lot of language design to be done to sort out the interactions with other language features and the compatibility issues, among others.

Adding a new class of types to the language should not be done light-heartedly. It seems however the best avenue if we want to provide powerful exception capabilities while minimizing incompatibilities and impact on implementation. However, it must be acknowledged that the above proposal would be very costly to implement, and could destabilize compilers; therefore, we must ask ourselves whether we think the benefits of modern exception capabilities offset the overall cost of such a large language change.