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. From this, it is impossible to pass exception 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 Ada.Exceptions package. However, this new mechanism is still only intended for error reporting purposes, and does not provide a general communication framework.

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). We would be able to parameterize exceptions, much like in Java, and even parameterize them with non-String data.

```
type root_exception_type is abstract tagged limited private;

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

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

```
exception
  Ada95_Exception;
  User_Exception;
```

This situation is very unfortunate because other

supported a more flexible exception paradigm in Ada.

import static javax.swing.JFrame.

Exceptions as Types

The purpose of this paper is to propose a new
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

the 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.

A root exception type may be declared as:

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

A derived exception type may then be declared as:

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:

exception
Message : String;

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 declared by a

new exception declaration:

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

If we declare an exception type, as above, the

exception type is defined:

type
Exception
is
efault-exception;
new Error-exception

end exception;

With this analogy, the exception identity

for an exception type may be declared by a new

default exception type, whose ancestor is an exception type.

except for compatibility, an exception declared by a

new exception type may only be passed to a

class-wide exception type that is a subclass of the

ancestor class.

There is no such thing as an usual exception type.

When compiling, the compiler issues a warning:

a new exception type declared by a new

exception type is implicitly a class-wide exception type.

The warning is misleading, however, as the

exception type is not a class-wide type, and cannot

be used as a parameter type. The warning is due to

the fact that the compiler does not understand

exception types.

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 declared by a

new exception declaration:

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

If we declare an exception type, as above, the

exception type is defined:

type
Exception
is
efault-exception;
new Error-exception

end exception;

With this analogy, the exception identity

for an exception type may be declared by a new

default exception type, whose ancestor is an exception type.

except for compatibility, an exception declared by a

new exception type may only be passed to a

class-wide exception type that is a subclass of the

ancestor class.

There is no such thing as an usual exception type.

When compiling, the compiler issues a warning:

a new exception type declared by a new

exception type is implicitly a class-wide exception type.

The warning is misleading, however, as the

exception type is not a class-wide type, and cannot

be used as a parameter type. The warning is due to

the fact that the compiler does not understand

exception types.

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 declared by a

new exception declaration:

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

If we declare an exception type, as above, the

exception type is defined:

type
Exception
is
efault-exception;
new Error-exception

end exception;

With this analogy, the exception identity

for an exception type may be declared by a new

default exception type, whose ancestor is an exception type.

except for compatibility, an exception declared by a

new exception type may only be passed to a

class-wide exception type that is a subclass of the

ancestor class.

There is no such thing as an usual exception type.

When compiling, the compiler issues a warning:

a new exception type declared by a new

exception type is implicitly a class-wide exception type.

The warning is misleading, however, as the

exception type is not a class-wide type, and cannot

be used as a parameter type. The warning is due to

the fact that the compiler does not understand

exception types.

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 declared by a

new exception declaration:

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

If we declare an exception type, as above, the

exception type is defined:

type
Exception
is
efault-exception;
new Error-exception

end exception;

With this analogy, the exception identity

for an exception type may be declared by a new

default exception type, whose ancestor is an exception type.

except for compatibility, an exception declared by a

new exception type may only be passed to a

class-wide exception type that is a subclass of the

ancestor class.

There is no such thing as an usual exception type.

When compiling, the compiler issues a warning:

a new exception type declared by a new

exception type is implicitly a class-wide exception type.

The warning is misleading, however, as the

exception type is not a class-wide type, and cannot

be used as a parameter type. The warning is due to

the fact that the compiler does not understand

exception types.

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 declared by a

new exception declaration:

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

If we declare an exception type, as above, the

exception type is defined:

type
Exception
is
efault-exception;
new Error-exception

end exception;

With this analogy, the exception identity

for an exception type may be declared by a new

default exception type, whose ancestor is an exception type.

except for compatibility, an exception declared by a

new exception type may only be passed to a

class-wide exception type that is a subclass of the

ancestor class.

There is no such thing as an usual exception type.

When compiling, the compiler issues a warning:

a new exception type declared by a new

exception type is implicitly a class-wide exception type.

The warning is misleading, however, as the

exception type is not a class-wide type, and cannot

be used as a parameter type. The warning is due to

the fact that the compiler does not understand

exception types.
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:

```ada
raise Ada95_Exception'Class (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:

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

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

```ada
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:

```ada
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 convenient view of the exception occurrence.

```
when a User Exception =>
when other Exception =>
```

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

### Implementation Issues

When implementing a new language extension, there are several important points to consider:

- **Syntax:** Language syntax should be carefully designed to avoid ambiguity.
- **Static Semantics:** Exception types must be handled carefully in the compiler to ensure type checking is correct.
- **Dynamic Semantics:** Runtime behavior must be carefully designed to handle exception occurrences in a consistent manner.

### Implementation

- When implementing exception handling, it is important to ensure that exception types are properly handled to maintain the integrity of the program.
- Exception types should be designed to be easily extendable, allowing for future enhancements.
- Runtime checks should be performed to ensure that exception types are handled correctly and that the program state is maintained.

### Conclusion

In conclusion, the implementation of exception handling requires careful consideration of syntax, type checking, and runtime behavior. By designing exception types to be flexible and extendable, we can ensure that the program remains robust and maintainable over time.
It might be possible to limit the dynamic complexity of exceptions 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 Ada 95 is a thorny one. One might be tempted to declare the entire package Ada.Exceptions obsolete, 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 Occurrence, and there is only one type for all the exception declarations. With the proposed model, code which handles exceptions gets an occurrence of some exception type, and all the exception types are different. 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 lightly. It does seem however that 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 the benefits of modern exception capabilities offset the overall cost of such a large language change.