Session Summary: Object Orientation and Exception Handling for Ada

Chair: Thomas Wolf
Rapporteur: Tullio Vardanega

1 Motivations

The session started with a consensus view from the group about the powerful contribution potential that exceptions can offer to the construction of reliable systems.

Ada has rightly deserved the reputation of being prime language support for the development of systems with reliability characteristics. One would therefore expect that the language should take special care to address the needs of the reliable system community. Yet, the group noted, the design of the language enhancements added to Ada 95 held a narrow view of exceptions, which were seemingly regarded as simple-minded aid for rudimentary program debugging.

The few, timid enhancements made to exceptions in the language indeed had the merit of clarifying the notions of exception identity and exception occurrence. Nonetheless, the language revision apparently privileged the view of exceptions as a low-cost mechanism for fairly high-level error logging. As a regrettable result, exceptions are still not integrated with the typing model of the language.

A rapid scrutiny of the workshop submissions allowed the group to gather consensus on three major limitations with the current model of exceptions:

- the nature, the structure and the level of information that one can associate to the raising of exceptions are insufficient for the construction of less-than-rudimentary fault tolerance schemes;
- the lack of support for formal generic exceptions is unacceptably limiting because it excludes exceptions from the instantiation space of the language;
- the unability to organise exceptions in hierarchies deprives programmers of the means to opt between specialisation and generalisation of handling.

It was the opinion of the group that the resolution of these limitations calls for a better, in fact, a full integration of exceptions with the typing model of the language.

The group noted that the use of current language features would indeed permit to circumvent the first and the second problems above. To cope with the former, one should marshal the required data components in the message string associated to the exception occurrence. To cope with the latter, one should use a formal generic parameter of type Ada.Exceptions.Exception_Id and then use Ada.Exceptions.Raise_Exception to raise the corresponding exception in the instantiation. The group concurred, however, that both solutions appear to be needlessly contrived and low level. The group therefore considered them fully inadequate to general use and not in keeping with the high level nature of the language. These considerations reinforced the search by the group for a more general solution to the identified problems.

Three workshop submissions, those of Thomas Wolf, Tucker Taft and Pascal Leroy, outline independent extension proposals that specifically address those limitations. The group devoted a large proportion of the session to discussing the merit of each proposal.

2 Discussion

The three proposals fundamentally differ in the distance they place between the revised notion of exception type and the language standard view of tagged types.

At one extreme of the spectrum, Taft does not really elevate exceptions to the rank of true types, but it allows exceptions to have parameters and to be derived from other exceptions. The first feature would allow users to associate variable data, of elementary types only, to exception occurrences. (Taft observes that the need to preserve the ability to raise exceptions with just the exception identity would require exception parameters to have default values.) The second feature would allow programmers to create exception hierarchies. This proposal outlines no specific solution to the use of exception ‘types’ as formal generic parameters.

Raising an exception in Taft’s proposal corresponds to creating an exception occurrence with the provided parameters as its components. Handlers would be able to handle a specific exception or a ‘class’ of exceptions. Lexical order would matter in selecting the handler for a given exception occurrence.

Skeptical about the width of the user demand for a more powerful exception model, Taft also outlines an alternative

proposal that further restrains the implementation complexity incurred by the proposed enhancement and yet allows users to associate objects with the raising of exceptions. The group, however, felt that the alternative proposal failed to address the heart of the problem with exceptions not being types.

At the opposite extreme of the spectrum, Wolf considers exception types as full equivalent of tagged types.

Exception occurrences for Wolf would be instances (that is, objects) of the corresponding type.

The raise statement (as well as the Raise_Exception procedure of Ada.Exceptions) would become the primitive operation of the exception type at the root of the derivation hierarchy. Consequently, any derivation from the root exception type could in principle override the primitive raise procedure.

Similarly to Taft, Wolf allows exception handlers to be class wide and lexical order to determine the association between handlers and exception occurrences. In contrast with Taft, instead, Wolf does not contemplate any restrictions to the types of exception components, thereby opening up the door to the interaction between exception propagation and finalisation of controlled components.

Although making up for an attractive model, equalling exception types to tagged types exposes the declaration of exceptions in nested scopes to the same accessibility rule that applies to their proposed counterpart. ARM §3.9.1(3), in fact, disallows type derivations (and, thus, exception declarations in Wolf's model) at levels statically deeper than that of the corresponding parent type.

This notion would create a backward incompatibility with the current model, which poses no limitation to the declaration of exceptions in nested blocks. Wolf acknowledges the problem, but proposes no specific solution to it, in the hope of a possible revision of the accessibility rule, presently under the scrutiny of the ARG.

As middle ground between the above two extremes, Leroy proposes the explicit notion of exception type whose definition corresponds to defining an exception identity.

Leroy's exception types would be untagged, composite and limited and could be derived from by extension. No different from Taft's and Wolf's proposals, raising an exception for Leroy would create an exception occurrence. Consistently with the unabridged notion of exception type, however, Leroy allows exception occurrences to also be created by the declaration of exception objects. The limited nature of the type, however, would restrain the interest of that notion.

Similarly to Taft and Wolf, Leroy contemplates class-wide exception handlers. Contrary to the other two proponents, however, Leroy requires handlers in the same block to have non-overlapping types, so that lexical order would *not* determine the selection.

Leroy's proposal makes a point of *not* wanting exception types to be equalled with tagged types. The proposal provides a two-fold rationale for the decision. Firstly, Leroy seeks to avoid transferring onto exceptions the limitations that presently apply to tagged types. This approach has the distinct advantage of neatly avoiding the accessibility rule problem incurred by Wolf's proposal. Secondly, Leroy wishes to avoid imposing an unneeded burden on the existing language implementations whose exception support is other than that in use for tagged types.

Three issues, which are common to Leroy's and Wolf's proposals, were identified by the group as needing to be further investigated:

- exception types can be unconstrained, which would allow exception occurrences to have arbitrarily large size;
- components of exception types can be controlled, which would raise the issue of finalisation upon exception propagation;
- having exceptions are types for which access values can be created causes users to lose control over the static analysis of exception propagation.

The first issue above ultimately assigns the user the responsibility of the run-time overhead incurred on exception raising. The second issue configures as a normal case of interaction of finalisation with other language features. The third issue addresses a programming style concern, which could perhaps be addressed by dedicated guidance material.

3 Conclusions

Where Wolf attempted to give exceptions all of the object oriented flavour presently allowed by the language, Leroy limited his ambitions to addressing issues that coincided with the three problems identified by the session.

In spite of its obvious intellectual appeal, the Wolf proposal seems to go beyond the requirements identified by the group and to involve issues that would require a depth of analysis well beyond the scope of the workshop.

In this respect, the group felt that Leroy's proposal aimed higher than Taft and closer to the target of immediate interest to the workshop.

The session concluded with widespread consensus that the resolution of the limitations identified by the group should deserve the attention of the language revision process currently conducted by the ARG.

The group recommended that the refinement of a proposal built upon the rationale provided by Wolf and the approach outlined by Leroy should be pursued and the result should be submitted to the attention of the ARG as part of the current revision process.