Zero-Overhead Exception Handling Using Metaprogramming

Markus Hof, Hanspeter Mössenböck, Peter Pirkelbauer
Johannes Kepler University Linz, A-4040 Linz
{hof, moessenboeck}@ssw.uni-linz.ac.at

We present a novel approach to exception handling which is based on metaprogramming. Our mechanism does not require language support, imposes no run time overhead to error-free programs, and is easy to implement. Exception handlers are implemented as ordinary procedures. When an exception occurs, the corresponding handler is searched dynamically using the type of the exception as a search criterion. Our implementation was done in the Oberon System but it could be ported to most other systems that support metaprogramming.

1. Motivation

Exception handling is the ability to separate the reaction to a program failure (i.e., to an exception) from the place where the failure occured. This has two advantages:

- It keeps algorithms free of error handling code.
- It allows a programmer to implement the reaction to different occurrences of the same exception in a single place.

Exception handling was suggested in the seventies ([Good75]) and is part of many modern languages such as *Java* [ArGo96] , C++ [Stro86], Eiffel [Meye92], Modula-3 [Nels91] or Smalltalk [GoRo83]. The exception handling facilities of these languages differ from each other in the syntactical notation they use, in the way how they allow a program to continue after an exception, how they check that all possible exceptions will be handled, and how exception handling is implemented.

Besides supporting exception handling in a programming language it can also be supported by library functions (e.g., [Mill88]). This has the advantage of keeping the language small although it might not be as readable as with language support.

In this paper we present a novel approach to exception handling which is based on metaprogramming. Our mechanism does not require language support, imposes no run time overhead to error free programs, and is easy to implement. We implemented exception handling for the Oberon System [WiGu92] but it could have been done for most other systems that support metaprogramming.

The rest of the paper is organized as follows: In Section 2 we give an overview of existing exception handling notations and implementations. In Section 3 we shortly explain the metaprogramming facilities of Oberon, which are then used in Section 4 to introduce our approach to exception handling. In Section 5 we finally compare our technique with mechanisms used in other languages.

2. Common Exception Handling Mechanisms

Exception handling is part of many modern programming languages. In this section we give an overview of the notations that are used in these languages and sketch some common implementation techniques. In Section 4 we introduce our own notation and implementation.

General principles. Exception handling is built around three concepts: a *block* of statements that is protected against exceptions; one or more *exception handlers* that are specified for the protected block; and a mechanism to *raise exceptions*. If an exception is raised during the excution of the protected block, the corresponding handler is executed. Some exceptions can also be raised by the system because of illegal operations (e.g. division by zero). After the exception was handled the program can be continued in one of three ways:

- *Terminate semantics*: The protected block is terminated and the program continues with the first statement after the block.
- *Resume semantics*: The execution of the protected block is resumed after the point where the exception was raised.
- *Retry semantics*: The protected block is re-executed from the beginning (after the exception handler has repaired some conditions which caused the block to fail in the first try).

Many languages support only terminate semantics. For a discussion of the pros and contras of the various semantics see [Stro94].

C++ and Java. The programmer can protect a statement sequence against exceptions by enclosing them in a *try* statement (Fig. 1). If the execution of such a statement sequence raises an exception by means of a *throw* statement an appropriate handler takes control. Handlers are appended to the *try* statement as *catch* clauses. After the execution of the *catch* clause the program continues with the first statement after the *try* statement. Exceptions are objects which are thrown (i.e. raised) in a *throw* statement and caught in a *catch* statement. The *catch* clause specifies the class of the exception object that is to be caught. Both C++ and Java support only terminate semantics. For details see [ElSt90] and [ArGo96].

```
try {
   … some calculations …
   if (…) throw Overflow();
   Foo();  // may also throw exceptions
}
catch (Overflow& ovfl) {… handle overflow …}
catch (Underflow& ufl) {… handle underflow…}
```

Fig. 1 Exception handling notation in C++ and Java

Eiffel. The Eiffel exception handling mechanism is essentially based on the principle of contracts. Classes and methods establish contracts with their clients by specifying preconditions, postconditions and class invariants. If one of these conditions fail, an exception is raised which can be handled in a *rescue* clause of some currently executing method (Fig. 2). In addition, the user may also raise user-defined exceptions. Exceptions are denoted by integer codes, and the *rescue* clause has to analyze these codes with an *if* statement. A *rescue* clause usually repairs the failure (if possible) and then retries the method. If no *retry* is specified the method fails, propagating the exception to the *rescue* clause of its caller. Thus Eiffel supports just retry semantics. For details see [Meye92].

```
Foo (…) is
   require … precondition …
   do
      Foo1();  -- may raise exceptions
      Foo2();
   ensure … postcondition …
   rescue
      if exception = … then
         … repair the failure …
         retry;
      end
   end;
```

Fig. 2 Exception handling notation in Eiffel

Smalltalk. Smalltalk provides the most powerful mechanism. Exception handling is applied to blocks; exceptions are represented by classes. An exception is raised by sending a *signal* message to an exception class. The corresponding handler is itself specified as a block (Fig. 3). After the exception has been handled, the program can terminate the block that raised the exception, retry it or resume its execution after the point where the exception was raised. The user may specify an *ensure* block which is always executed after the protected block, no matter if an exception occurred or not. For details see [GoRo83].

```
[… some action … MyException signal. … some action…]
on: MyException do: [:theException | … handle exception … theException return]
on: MyException2 do: [:theException | … handle exception … theException resume]
on: MyException3 do: [:theException | … handle exception … theException retry]
ensure: [… local cleanup …]
```

Fig. 3 Exception handling notation in Smalltalk

Implementations. Current implementations are either based on C's *setjmp/longjmp* mechanism or on tables generated by the compiler. We will shortly sketch both variants. A more extensive description can be found in [KöSt90].

Setjmp/longjmp. The function *setjmp*(*s*) saves the current machine state in a buffer *s* and returns 0. The function *longjmp*(*s*) restores the machine state from the buffer *s* (including the program counter). As a result, the execution will continue in the *setjmp* routine where this state was saved. After a *longjmp*, however, *setjmp* will return 1. This makes the following implementation of

```
try {… block …} catch {… handler …}
```

possible:

```
IF setjmp(s) = 0 THEN
   Push(s); … block … Pop(s)
ELSE (*execution continues here after a longjmp*)
   … handler …
END
```

If an exception is raised in *block*, the following code is executed:

```
Pop(s);
longjmp(s);
```

If the handler cannot handle the exception, it re-raises it, so that a *longjmp* to the previous *setjmp* is executed, and so on. This implementation is straightforward but it leads to some run-time overhead (*setjmp*, *Push*, *Pop*) even in the case of error-free programs.

Range tables. The compiler generates a table in which the range of every *try* block (start address, end address) as well as the addresses of the corresponding handlers are recorded. If an exception occurs, the current program counter is looked up in the table (if it does not fall into any of the ranges, the program counter of the caller is tried). When the appropriate range is found, the stack is unwound and the corresponding handler is called. This imposes no run-time overhead in the case of error-free programs but requires the compiler and loader to set up the table.

3. Metaprogramming

Our exception handling technique makes use of *metaprogramming*, so we will shortly explain this term and show how it can be used in the Oberon System.

Metaprogramming means the ability to treat programs as data, for example to get information about the names and the structure of their variables, types and procedures. If a program can acquire information also about itself, this is called *reflection*.

Metaprogramming and reflection were pioneered by *Lisp* [McCa60] and *Smalltalk* [GoRo83]. Today this feature is available in many modern languages such as *Java* [ArGo96], *CLOS* [Atta89] or *Beta* [MMN93]. An implementation for *Oberon* is described in [Temp94] and [StMö96].

In Oberon, information about programs is organized in sequences and can be accessed with iterators, which are called *Riders*. A Rider can enumerate the procedures, types or global variables of a module as well as the activation frames of the currently active procedures on the stack. When a Rider is positioned on an element of such a sequence (e.g., on a variable), its fields contain information about this element, for example its name and its data type. If an element is itself structured (e.g., a record variable), one can "zoom in" and enumerate the elements of the inner structure.

The following example shows how to iterate over the currently active procedures and print their names as well as the names of the modules in which the procedures are declared.

```
VAR r: Ref.Rider;

Ref.OpenStack(NIL, r);              (*a rider is placed on the topmost frame of the procedure stack*)
WHILE r.mode # Ref.End DO
   Out.String(r.name); Out.Ln;      (*print the name of the corresponding procedure*)
   Out.String(r.mod); Out.Ln;       (*print the name of the module declaring this procedure*)
   r.Next                           (*proceed to the next frame*)
END
```

A second example shows how to iterate over the procedures of a module "M" and look for a procedure that has a reference parameter of type "T" as its first parameter. The list of parameters is obtained by zooming into the procedure.

```
VAR r, r1: Ref.Rider; type: Types.Type;

Ref.OpenProcs("M", r);                (*r is placed on the first procedure of module M*)
WHILE r.mode # Ref.End DO
   r.Zoom(r1);                        (*r1 is placed on the first parameter of the procedure*)
   IF r1.mode = Ref.VarPar THEN       (*if it is a reference parameter*)
      type := r1.Type();              (*get its type*)
      IF type.name = "T" THEN …found…; RETURN END;
   END;
   r.Next                             (*proceed to the next procedure*)
END
```

4. Exception Handling with Metaprogramming

In this section we introduce an exception handling technique based on metaprogramming. It needs no special language constructs and does not require compiler support. Error free programs are not slowed down. Overhead occurs only in the case of exceptions. We describe our implementation for the Oberon system, but the same technique could also be applied in any other system that supports metaprogramming.

Exception objects. Exceptions are objects of an exception class, which is a subclass of *Exception* (Fig. 4). There are *system exceptions* and *user exceptions*. System exceptions (e.g., division by zero) are triggered automatically while user exceptions are raised by the user program using the library call *Exceptions.Raise(exception)*.

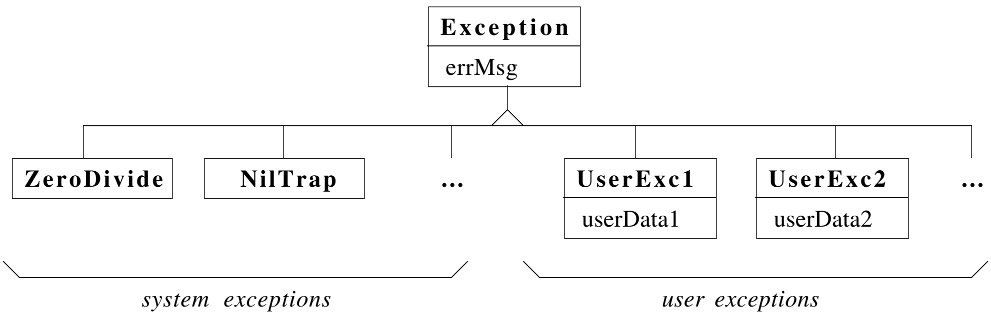

Fig. 4 Hierarchy of exception classes

Exception handlers. Exceptions are *caught* (i.e. handled) by an *exception handler* which is an ordinary procedure *H* with the following characteristics:

- *H* is declared local to some currently executing procedure *P*, i.e., to one with an activation frame on the procedure stack.
- *H* has a single reference parameter of type *E*, which is the type of the exception to be caught or a superclass thereof.
- Both *H* and *P* have the same return type or none.

The following example shows a procedure *Foo* that calls a procedure *Read* in order to read from a file. If the end of the file is reached, *Read* raises an exception of type *EofException* (a subclass of *Exception*), which is caught by the handler *HandleEof* in *Foo*:

```
PROCEDURE Foo (): INTEGER;
   VAR f: File; ch: CHAR;

   PROCEDURE HandleEof (VAR eof: EofException): INTEGER;  (*the handler*)
   BEGIN
      Close(f);
      RETURN 1 (*error code for eof*)
   END H;

BEGIN (*Foo*)
   Open(f, "…");
   REPEAT Read(f, ch); … UNTIL …;
   Close(f);
   RETURN 0 (*no error*)
END Foo;

PROCEDURE Read (f: File; VAR ch: CHAR);
   VAR eof: EofException
BEGIN
   IF …end of file … THEN Exceptions.Raise(eof) ELSE … END
END Read;
```

Raising an exception (e.g. *Exceptions.Raise(eof)*) leads to a call of the appropriate handler (e.g. *HandleEof*). When the handler returns, execution continues after the call of the procedure to which the handler is local. In the above example control returns to the caller of *Foo*.

A procedure like *Foo* may contain multiple handlers for different kinds of exceptions (i.e., multiple local procedures with parameters of different exception types). If a handler for

EofException is not found in *Foo*, the search continues in the caller of *Foo*, then in its caller and so on, until a matching handler is found or the topmost procedure in the call chain is reached. If no matching handler is found, a standard error message is produced and the program terminates.

A handler may again raise an exception, in particular, it may re-raise the same exception that it is currently handling. In this case the search for a new handler starts in the caller of the procedure that contains the current handler (in the above example, the search starts in the caller of *Foo*).

Note that a handler has access to the local variables of the enclosing procedure. *HandleEof* can, for example, close the file *f* declared in *Foo* .

Exception handling semantics. The above example makes use of *terminate semantics*: After the handler was executed, the containing procedure *P* is terminated and the program continues after the call of *P*. Instead, we can also make use of *resume semantics*. That means, that the handler returns to the point where the exception was raised, and execution continues with the instruction after the *Raise*. Resume semantics can be requested for user-raised exceptions with the library call *Exceptions.Resume* like in the following example:

```
PROCEDURE Handler (VAR e: SomeException);
BEGIN
  IF … the failure can be repaired … THEN
    …Repair it…;
    Exceptions.Resume  (*return using resume semantics*)
  END;
  (*return using terminate semantics*)
END Handler;
```

Implementation. When an exception occurs, the system has to look for an appropriate handler. It uses metaprogramming facilities to search the procedure stack for a procedure with a local procedure that can be used as a handler. The following pseudo code shows an outline of this algorithm.

```
PROCEDURE Raise (VAR e: Exception);
  E := dynamic type of e;
  FOR all frames on the stack in reverse order DO
    P := procedure that created this frame;
    FOR all local procedures H of P DO
      IF (H has a single reference parameter of type E or a supertype of E)
      & (H has the same return type as P) THEN
        Execute H;
        Return to the caller of P or to the point where the exception was raised,
        depending on the chosen exception handling semantics
      END
    END
  END;
  Terminate the program with a standard error message
END Raise;
```

The metaprogramming facilities necessary for this implementation are described in the examples of Section 3. *Raise* uses a rider to iterate over all stack frames. For every frame it gets the procedure *P* and the module *M* to which the frame belongs. It then uses another rider to iterate over all procedures of module *M* looking for a procedure *P.H* (a procedure *H* local to procedure *P*) that has the required characteristics.

Assume that we found the handler *H*. What remains to be shown is how the handler is called and how it returns depending on the chosen exception handling semantics. When an exception is raised, the procedure stack looks as in Fig. 5.

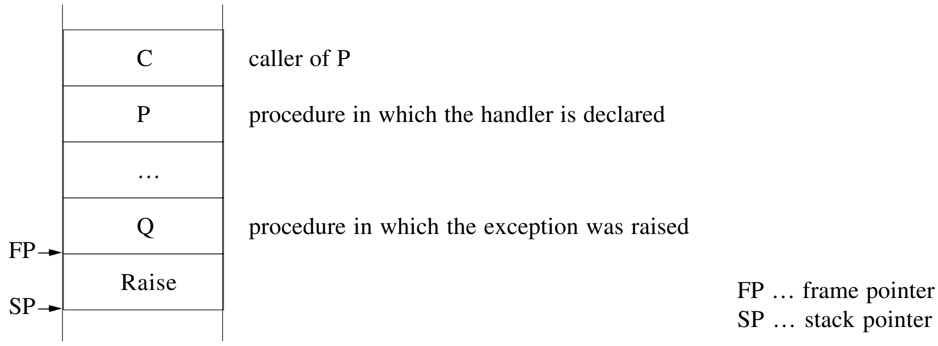

Fig. 5 Stack of procedure frames at the time when an exception occurs in procedure Q

For system exceptions, which are triggered automatically, the system will generate a call to the procedure *Raise* as if they were user exceptions. This will lead to the same picture as in Fig. 5. In both cases *Raise* will search for the handler *H* and its enclosing procedure *P* as described above and then execute the following code:

```
Push e;                              (*exception parameter*)
Push frame pointer of P;             (*static link*)
Call H;
(*this point is only reached under terminate semantics*)
FP := beginning of C's frame;
SP := end of C's frame;
PC := return address of P            (*return to the caller of P*)
```

When the handler *H* is executing, the stack looks as in Fig. 6a. The static link of *H* allows accessing the local variables of *P*. When H returns, *Raise* modifies the registers FP (frame pointer), SP (stack pointer) and PC (program counter) so that the stack is cleaned up and execution continues after the call of *P* (Fig. 6b).

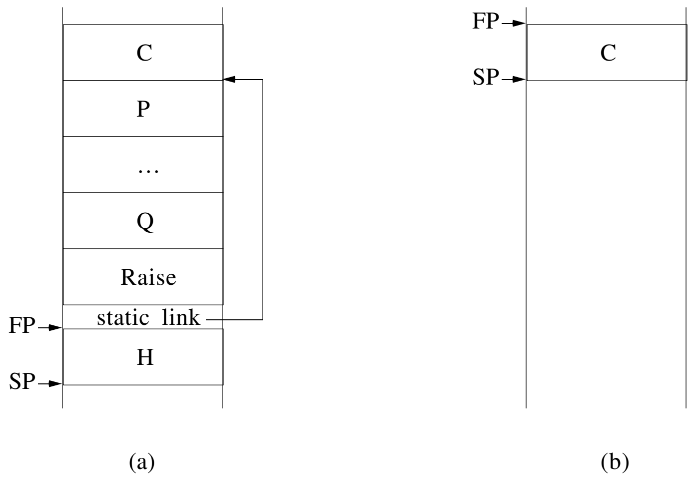

Fig. 6 Procedure stack: (a) during the execution of the handler *H* and (b) after the control was transferred back to the caller of *P*

If *H* calls *Resume* (which is possible only for user-raised exceptions), this procedure returns control in the following way:

```
PROCEDURE Resume;
   FP := beginning of Q's frame;     (*Q is the procedure that raised the exception*)
   SP := end of Q's frame;
   PC := return address of Raise     (*return to Q*)
END Resume;
```

If an exception handler *H* raises an exception itself the same mechanism starts again: The end of the stack will contain another frame of *Raise* as well as a frame of some new handler *H1*. Note, that the search for *H1* must start with the frame of *C* (the caller of *P* in Fig. 5) in order to avoid cycles. Therefore, every invocation of *Raise* must remember where to continue the search if an exception is raised in the handler. This information is stored in some global exception context before the handler is invoked. Since in a multi-threaded environment *Raise* must be reentrant, the exception context is part of the thread's environment.

5. Comparison

In the following section we compare our exception handling notation and implementation with the other techniques described in Section 2. A summary is shown in Table 1.

Table 1 Exception handling mechanisms in various programming languages

	C++ / Java	**Eiffel**	**Smalltalk**	**Oberon**
Notation	language based	language based	library based	library based
Exceptions	objects	numbers	objects	objects
Granularity	block level	procedure level	block level	procedure level
Resumption semantics	terminate	retry	terminate resume retry	terminate resume (retry)

Notation. Our exception handling technique does not require special language support. Protected blocks are procedure bodies, handlers are local procedures with special parameters, and exceptions are raised with a library call. The main advantage of this approach is that we did not have to change an existing language (in our case Oberon). Our technique can be combined with any language as long as the environment supports metaprogramming.

A possible disadvantage of our notation is that exception handling does not stand out as clearly as with special keywords for protected blocks and handlers. It is also not easy to check statically which exceptions a procedure may raise, since we do not require the unhandled exceptions to be specified in the signature of a procedure, as it is the case for example in Java.

In our approach, as in most other implementations, exceptions are objects. This has the advantage that exceptions can be subclassed to carry arbitrary information from the exception point to the handler. In Eiffel this is not possible because exceptions are just numbers.

Granularity. We support exception handling on the procedure level and not on the block level. Due to our experience this is sufficient. If a finer granularity is needed, any statement sequence can be turned into a procedure.

Program resumption. Our implementation currently supports terminate semantics and resume semantics. In principle we could also support retry semantics but this was not implemented so far because it was not considered necessary. Java and C++ support only terminate semantics and Eiffel supports just retry semantics. Only Smalltalk is as flexible as our approach, supporting all three variants of program resumption.

Efficiency. Our implementation does not impose any run time overhead on error free programs. Only in the case of an exception, the system has to search for a handler, which takes about 1 ms in a typical case (measured on a Power Macintosh with 66 MHz). In contrast to that, the *setjmp/longjmp* mechanism described in Section 2 leads to run time costs for every protected block, even if no exception is raised. The *range table* technique (also described in Section 2) does not slow down error free programs, but it requires storage overhead (the range tables) and compiler support, because the tables have to be generated by the compiler. This is not the case with our implementation. The meta-information that we need is already there in the Oberon system so that it does not impose any additional overhead.

Heap cleanup. If blocks are terminated due to an exception, there may be data on the heap for which the deallocate statements or destructor invocations were skipped. This is not a problem in most of the mentioned languages (including Oberon) since they rely on automatic garbage collection. In C++, however, proper deallocation of objects is an issue and complicates exception handling considerably [KöSt90].

5. Conclusions

We suggested a zero-overhead exception handling technique based on metaprogramming. It was implemented without extending the programming language or the compiler. Exception free programs are not slowed down at all. A slight run time penalty has to be paid only if an exception occurs. Our implementation does not need special data structures at run time, except for meta information about programs, which is available anyway in many languages (e.g. in Java or Smalltalk). Our technique supports various program resumption semantics and allows the programmer to declare custom exception classes with exception-specific information. We implemented our technique for Oberon but it could have been done for any other language that supports metaprogramming.

We found that our technique is powerful, efficient and easy to implement. It can be the method of choice if one does not want to change the programming language or the compiler.

Refereces

[ArGo96] Arnold K., Gosling J.: The Java Programming Language. Addison Wesley 1996.

[Atta89] Attardi G. et al.: Metalevel Programming in CLOS. Proc. European Conference on Object-Oriented Programming (ECOOP'89). Cambridge University Press, 1989.

[ElSt90] Ellis M.A., Stroustrup B.: The Annotated C++ Reference Manual. Addison-Wesley 1990.

[Good75] Goodenough J. B.: Exception Handling: Issues and a Proposed Notation. Communications of the ACM, vol.18, no.12, December 1975.

[GoRo83] Goldberg A., Robson D.: Smalltalk-80, the Language and its Implementation. Addison Wesley 1983.

[McCa60] McCarthy J.: Recursive Functions of Symbolic Expressions and their Computation by a Machine. Communications of the ACM, vol.3, no.4, 1960.

[KöSt90] König A., Stroustrup B.: Exception Handling for C++ (revised), Proc. USENIX C++ Conference, 149-176, (1990)

[Meye92] Meyer B.: Eiffel — The Language. Prentice Hall 1992.

[Mill88] Miller W. M.: Exception Handling without Language Extensions. Proc. USENIX C++ Conference. Denver CO., October 1988.

[MMN93] Lehrmann-Madsen O., Moller-Pedersen B., Nygaard K.: Object-Oriented Programming in the BETA Programming Language. Addison Wesley 1993.

[Nels91] Nelson G. (ed.): Systems Programming with Modula-3. Addison-Wesley 1991.

[StMö96] Steindl C., Mössenböck H.: Metaprogramming Facilities in Oberon for Windows and Power Macintosh. Technical Report 4, Institute of Computer Science, University of Linz, 1996.

[Stro86] Stroustrup B.: The C++Programming Language. Addison Wesley 1986.

[Stro94] Stroustrup B.: The Design and Evolution of C++. Addison Wesley 1994.

[Temp94] Templ J.: Metaprogramming in Oberon. Dissertation, ETH Zurich, 1994.

[WiGu92] Wirth N., Gutknecht J.: Project Oberon—TheDesign of an Operating System and Compiler. Addison-Wesley 1992.