Exceptions for Cecil

Keunwoo Lee (klee@cs)
University of Washington

August 15, 2000

Abstract

Exceptions are a convenient way of expressing error-handling, and we can add exceptions as a user-level library to Cecil with no additional mechanism. However, in the process of defining the exceptions library, several deficiencies in the Cecil language design came to light. In this document I present a simple exceptions library, as well as a few alternative implementations that were discarded because either they do not work with the current Cecil language, or they were unwieldy for the user.

1 The case for exceptions

Exceptions provide a mechanism for nonlocal handling of errors and other exceptional conditions. In the past, Cecil programmers idiomatically have passed a "handler" closure through all methods in the interval between the point of handler definition and the point at which the error might be raised. The chief advantages of this approach are as follows:

1. Error-handling execution paths are explicitly declared as parameters in the method declarations. In other words, one can understand all the ways a method might exit by looking at its parameter list; if the parameter list includes a closure, one assumes that the closure might execute a nonlocal return.
2. Closures are syntactically lightweight constructs with many uses, such as implementing Cecil's standard control structures. Nonlocal error handling can therefore be implemented with no additional mechanism (and hence no additional complexity) in the language.
3. There is no special protocol for determining whether a handler should resume at the point the error was raised or the point at which the handler closure was captured. The choice between a local or nonlocal return in the closure automatically specifies the desired action.

However, explicit exceptions also provide several benefits over the closure-passing style of error handling.

First of all, exceptions support flexible program evolution. Adding a new exception type, handler, or throw site simply requires that the programmer add an appropriate declaration. Adding a new handler closure requires changes to the profiles of all methods that intervene between the closure creation site and the error creation site.

Second, the declaration of exception types as explicit, named classes in a program aids both program comprehension and design. Exception class names serve as a form of documentation. More importantly, error conditions are often naturally organized in class heirarchies. For example, an I/O streams library might export a base class `io_exception`, which might be subclassed into e.g. `file_not_found` or `format_error` exceptions. Clients could then choose to differentiate I/O exceptions at the level of granularity most natural for their applications.

Third, some languages, such as C++ and Java, support *checked* exceptions. These languages allow programmers to declare an exception type as part of a method's profile, forcing a callers to catch or to rethrow the exception explicitly. Used properly, checked exceptions exploit the compiler's power to enforce error-handling policy throughout a system. Although checked exceptions slow program evolution[1], such

[1] In this respect, checked exceptions resemble the closure-passing style of error handling.

enforcement seems desirable from a software engineering standpoint. Furthermore, type systems with more flexible or expressive forms of automated exception checking may present viable research directions.

For of all these reasons, we would like to add exceptions to Cecil. However, we would prefer not to add a second type of (non-orthogonal) nonlocal return construct to the language. Fortunately, Cecil nonlocal returns provide a mechanism sufficiently powerful that we can add exceptions as a user-level library with no special hooks into the implementation or changes to the language design.

However, there are a few difficulties. I will first describe one possible (rather ugly) implementation, and then I will describe other, discarded designs.

2 Making exceptions work

2.1 Usage

In the current state of the exception library, the syntax for using exceptions is fairly intuitive. The user defines a "try/catch" block as follows:

```
try({
    -- ...
    throw( new_my_exception("hello") );
    -- ...
}, catch(my_exception, &(e) {
    -- ... handle exception
}));
```

The try/throw/catch functions behave as a C++, Java, etc. programmer might expect. The closure passed to `try()` executes. If an exception is thrown, the stack unwinds until a `catch()` taking the proper exception class is found, in which case the matching handler closure executes. If no matching `catch()` is found, the program exits with an error.

An exception class must do two things:

1. Inherit from the base class `exception`
2. Override the `class()` and `parent()` methods to return the exception class and the parent of the exception class, respectively.

The above example might use a class definition for `my_exception` as follows:

```
object my_exception isa exception;
    field user_data(@:my_exception):string;

    method new_my_exception(msg@:string)
        { concrete object isa my_exception { user_data := msg } }

    method class(@:my_exception)   { my_exception }
    method superclass(@:my_exception) { exception }
```

Of course, the `class()` and `superclass()` methods are intensely ugly. Every single exception descendant class must override these two methods to return the appropriate class object and superclass object. These methods are used internally by the exceptions library to distinguish classes in the exceptions heirarchy. They compensate for Cecil's lack of run-time type manipulation (e.g. C++'s `dynamic_cast`) or introspection.

Clearly, we would rather not require the user to remember to override these methods whenever (s)he defines an exception subclass. However, I considered a number of alternate schemes, and (of those schemes that work under the current language design) judged them all to be at least as ugly and usually more complicated. For example, I could have written `prim` code to do the equivalent of `dynamic_cast`; however, I wanted to avoid such hooks into the primitive implementation.

In any case, the burden on the user is relatively light (though aesthetically repulsive). As noted above, the try/throw/catch constructs behave as will be expected by programmers familiar with exceptions in other languages.

One final caveat: exception classes may not be template objects. The exception classes (e.g., `my_exception` in the example above) are used directly as parameters to methods, both by the client and the internal implementation. Therefore, exception classes must be manipulable runtime objects.

2.2 Implementation

In order to present this friendly face to the user, we must perform some minor acrobatics. Almost every piece of an exception package depends on every other, so explaining the code clearly presents something of a challenge. However, in general, there are 3 pieces required by any exceptions package:

A stack-unwinding mechanism. Any implementation of exceptions must have a list (implicit or explicit) of stack frames in which try/catch blocks are defined. The implementation must be able to return to an arbitrary stack frame when an exception is thrown.

An exception type system. What kind of values can be thrown as an exception? Languages differ widely in the design of their exception type system. Ada treats exceptions, which must be declared with the `exception` keyword, as a special kind of enumeration. C++ allows arbitrary types, including both primitive and composite types, to be used as exception types. Java exceptions are always classes that implement the `Throwable` interface.

The chief requirement of an exception type system is that the implementation must be able to "match" the exception class in a catch block with the class of a thrown exception. The rules for this matching determine the dispatching from a throw point to a handler.

Handler definition. The user must be able to define code that will be executed when an exception is thrown, and tie that code to the type of exception (s)he desires to catch. The handler will be rooted at a particular point in the code, indicating where execution resumes after handler completion[2].

We consider each roughly in turn, though there is considerable overlap.

2.2.1 Stack unwinding

The stack-unwinding mechanism is implemented with a thread-local stack of "try frames":

```
template object try_frame;
    field handler(@:try_frame):handler;
    field nlr(@:try_frame):&():bool;

    method new_try_frame(return_point@:&():bool, h@:handler) {
        concrete object isa try_frame
            { nlr := return_point, handler := h } }

template object thread;
    var field try_frames(@:thread)
        := concrete object isa stack[try_frame];
```

A try frame consists of a `handler` object (described in section 2.2.3), plus a nonlocal return closure which captures the scope in which the handler is defined. We use the `nlr` member to unwind the current thread's stack. Currently, since Cecil has no threads, obviously we define a single global thread as follows:

```
let var current_thread := concrete object isa thread;
method get_current_thread() { current_thread }
```

[2] I assume throughout this memo that exceptions do not have resume-at-throw semantics. An optional resume-at-throw mechanism, however, could be added with a fairly simple extension of the current scheme.

Try frames are pushed onto the stack as follows:

```
method try( cl@:&():void, h@handler ):bool
{
    push( get_current_thread().try_frames,
          new_try_frame( {^ false}, h ) );

    eval( cl );

    -- successful code termination (no non-local return)
    pop( get_current_thread().try_frames );
    true
}
```

This brief procedure is deceptively simple. The pieces fit together as follows:

1. First, we fetch the current thread's try stack.
2. Next, we capture a nonlocal return closure that simply returns false to the current continuation. If an exception is thrown, this closure will later be used to signal an abnormal return to the stack frame that invoked `try()`.
3. Third, we construct a try frame, consisting of the nonlocal return closure and the handler passed to `try()`. We push this frame onto the try stack.
4. Fourth, we execute the closure, which represents the body.
5. Finally, we know that the non-local return was not executed. Therefore, the closure exited normally, so we remove the handler from the try stack (it is no longer applicable) and return true.

We provide the return value, indicating whether the try returned normally or with an exception, for convenience.

With the steps defined above, we are prepared to handle exceptions. When the user throws an exception, the stack is unwound as follows:

```
method throw( thrown@throwable ):void {
    if ( not(catchable(thrown)), {
        print_line(thrown);
        error("Exiting...");
    });

    let var try_stack := get_current_thread().try_frames;
    loop({
        if (try_stack.is_empty, {
            print_line(thrown);
            error("Exiting...");
        }, {
            let var try_frame := pop( try_stack );

            if( match(try_frame.handler, thrown), {
                eval(try_frame.handler.closure, thrown);
                eval(try_frame.nlr);
            });
        });
    });
}
```

The purpose of the initial `if(not(catchable) ...)` test will be made clear shortly, in section 2.2.2. Leaving that aside for the moment, the central loop pops try frames off the stack repeatedly until it finds a handler whose exception type matches the thrown exception type. The definition of the `match()` predicate is the primary responsibility of the exception type system, also discussed in 2.2.2.

If a matching handler is found, we simply execute the handler's body and then return to the site where the handler was defined (previously stored as the try frame's `nlr` member). Once we have executed the nonlocal return, the stack unwinding component of exceptions is complete.

2.2.2 Exception type system

The exception library's type system is loosely modeled on Java's exception type system. The base class of all exceptions is `throwable`:

```
object throwable;
    method catchable(@:throwable):bool { false }
    method print(t@:throwable):void { print(t.to_string); }
    method to_string(@:throwable):void { "(uncatchable error)" }
```

As evidenced by the implementation of the `catchable()` method, not all "throwable" objects are "catchable". In this model, a fatal error can be signaled by defining a direct subclass of `throwable` that does not override the `catchable()` method. The exceptions library, as shown in section 2.2.1, will not attempt to handle such thrown objects.

The superclass of all catchable exceptions is `exception`:

```
object exception isa throwable;
    method catchable(@:exception):bool { true }

    method class(@:exception):throwable      { exception }
    method superclass(@:exception):throwable { throwable }

    method to_string(@:exception):void { "(generic exception)" }
```

As previously mentioned, user-defined exception classes must subclass `exception` redefine `class()` and `superclass()` appropriately.

The `class()` and `superclass()` methods are used by the exceptions library to match thrown exceptions against handler exceptions:

```
method match( may_handle@:handler, thrown@:throwable ):bool {
    let var thrown_class := class(thrown);
    let handle_class := may_handle.exception_class;

    while ({ not(thrown_class == throwable) }, {
        if ( thrown_class == handle_class, { ^ true });
        thrown_class := superclass(thrown_class);
    });

    false
}
```

This method essentially traverses the ancestor chain of the thrown exception, testing the identity of the class object against the class specified in the handler. The rest should be fairly obvious.

There are two serious drawbacks of this exception type system. First, as previously noted, the requirement on the user, who must redefine `class()` and `superclass()` for every exception class, is onerous.

Second, we cannot not properly support multiple inheritance of exception types. However, multiple inheritance of exception classes is rare enough that this does not seem like a major omission. In any case, it would be possible to fix this attribute of the class system with some effort, but it does not seem worthwhile (and would probably impose an even greater burden on the user).

2.2.3 Handler definition

The handler class's fields are defined as follows:

```
template object handler;
    field closure(@:handler):&():void;
    field exception_class(@:handler):exception;
```

The purpose of the fields has been detailed in sections 2.2.1 and 2.2.2.

The handler's constructor has an unusual name because of past usage:

```
    method catch( e@:exception, cl@:&(exception):void ):handler {
        concrete object isa handler
            { closure := cl, exception_class := e } }
```

Although one does not conventionally think of `catch()` as a constructor expression, its effect is naturally viewed as the construction of a "handler object".

Combined with the previously defined constructs, we obtain the desired try/catch block semantics and syntax, as introduced in section 2.1. The exception library is complete.

3 Failed designs and suggested fixes

One's first impulse upon seeing this exceptions library is to improve the implementation of exception dispatch. "Why can't we use dynamic dispatching to implement exception dispatch as well?" one asks.

The short answer is that Cecil methods are not properly scoped[3], and Cecil closures are not dynamically dispatched on their argument types. Therefore, we cannot capture a lexically scoped block of code that temporarily extends the inheritance graph.

The long answer involves an excursion into wishful thinking. How would we *like* exceptions to work, in an ideal world? Here is one possibility:

```
try({
   -- ...
   throw new_my_exception();
   -- ...
}, &(e@:my_exception)       { (-- ... --) }
   |&(e@:another_exception) { (-- ... --) }
   |&(e@:third_exception)   { (-- ... --) }
   isa handler
);
```

Here, I use the nonexistent operator `|&` (meant to be suggestive of both Cecil's `&` and ML's alternate argument declarator, `|`) to indicate that we are constructing a single closure that dynamically dispatches on a number of exception classes.

The `isa handler` indicates that this closure "inherits" from `handler`. Presumably, `handler` would be a named, globally defined closure; when invoked, it might return a value to the exceptions library indicating that the current exception cannot be caught by this try block:

```
    let handler := &(@:throwable) { cannot_catch };
```

It may be desirable, furthermore, to unify the closure and method mechanisms, so that the above becomes:

```
    method handler(@:throwable) { cannot_catch }
```

Having defined a few pretty constructs, we can now investigate why various approximations of this construct do not work in Cecil. First, let us attempt to implement handlers using lexically scoped methods, which, though not currently implemented, are at least in the language:

[3] Lexically scoped methods are in the language specification but currently not implemented.

```
method foo() {
    method handle(e@:my_exception)      { (-- ... --) }
    method handle(e@:another_exception) { (-- ... --) }

    try({
        bar();
    });
}

method bar() {
    throw(new_my_exception());
}
```

Assuming the `catch()` methods are in scope at the `throw()` invocation in bar[4], this is a fairly close approximation of the "ideal" handler syntax. However, we still run into a few problems.

The Cecil language specification 3.0 states that local methods extend, rather than override, the inheritance graph. This does not match the desired behavior of try/catch blocks. If a user defines a more specific exception handler in an outer scope, we do not want that handler to be invoked when a more general handler exists in an inner scope. We could use inheritance to get around this problem by specializing on a locally scoped object:

```
method foo() {
   object my_handler isa handler;
       method handle(h@:my_handler, e@:my_exception) { (-- ... --) }
   -- ...

   try({ bar(); }, my_handler);
}
```

This means requires that we additionally allow methods to specialize on local, runtime objects. To make this really convenient, we would like to be able to specialize on an object dynamically constructed with specialized methods, using something akin to a lambda expression or a Java anonymous class:

```
method foo() {
    try({ bar(); },
        object this isa handler with
             { method handle(h@:this, e@:my_exception)
                { (-- ... --) } }
    });
}
```

However, at this point we have obviously returned to the world of nonexistent language features, especially since lexically scoped methods are not implemented in the first place.

Methods, therefore, are unsuitable for the exception dispatching. Turning to closures, we find that closures simply do not dynamically dispatch at all, nor do closures have any inheritance mechanism. Therefore, any closure-based implementation of handlers must rely on class testing.

Since Cecil currently has no introspection or other mechanism whereby we can traverse the inheritance graph, any class-testing mechanism will rely on the user to override methods in exception subclasses. This leads inevitably to an implementation at least as ugly as our current one. A runtime type information mechanism akin to C++'s `dynamic_cast` would suffice to eliminate this requirement, but `dynamic_cast` is clearly an uglier language extension than dispatching closures or local method specialization.

[4] This would be true if a local method has pseudo-dynamic scope as well as lexical scope—that is, if a method extends the inheritance graph until its lexically enclosing scope returns. This interpretation clearly makes the implementation of nonlocal returns somewhat more complicated. An alternative interpretation would have local methods extending the inheritance graph only for message sends executed within the lexical scope in which the local methods are defined.

Taking a step back, we might ask: why do we even constrain the user to specialize a handler on the exception type? Why not simply allow the user to associate a predicate closure with the handler, such that the predicate returns true if the handler can handle the thrown exception? This is a valid point in the design space. It has the following virtues:

1. It increases the flexibility of the handler, since the predicate closure can execute arbitrary code.

2. It simplifies the definition of the exception library somewhat, since the class tests are eliminated. The user is free to define exception classes however (s)he wishes.

However, in practice I believe this solution would simply shift the responsibility of defining complex class identity tests to the exception user. Users would then define a large number of nonuniform, ad hoc mechanisms for disambiguating exceptions, some of which might not cooperate properly with one another. A user throwing an exception of type T would have no guarantee that only a handler capable of handling T would be invoked on it.

Hence, as implausible as it seems, I believe the exceptions library I have implemented is the closest approximation to sanity that can be achieved in the current Cecil language.

3.1 Finally?

Java exceptions include a "finally" construct, whereby a programmer may define extra code that will be executed regardless of any control flow, local or nonlocal, in the body of the try block. We would like to add this construct, for programmer convenience.

It would be simple to do so, except that Cecil also provides arbitrary nonlocal returns to programmers. The body of a try block appears, to the library, as an opaque closure. If a Cecil programmer executes his/her own nonlocal return within the try block body, the exceptions library loses control flow and hence cannot make any guarantees about execution. Providing "finally" in these circumstances would simply be deceptive.

Possible language design alterations that would fix this problem:

- Extend the `eval()` method to take another closure parameter, which would have "finally" semantics. This extra closure would always be executed by `eval()` after evaluating the primary closure, regardless of how the primary closure returned.

- Unify exceptions and nonlocal returns at the language level. If they were both implemented with a single unifying mechanism, a "finally"-like construct could be defined for this unifying mechanism. Such a construct would probably have to permit "finally" blocks to be attached to any method call or closure evaluation.

4 Extensions

The following are some possible extensions that could be added to the current library:

- Add a way to register a closure to execute when a non-catchable exception is thrown. This would serve two purposes. First, programmers could use this registration mechanism as the equivalent of C's `atexit()`. Second, programmers could be given a chance to perform some customized error logging or even last-ditch recovery.

- Add a protocol for specifying resume-at-throw semantics, rather than resume-at-try. The difficulty with adding such a protocol is that neither the try scope nor the throw scope really has enough information to determine whether the exception should resume-at-throw.