Erlang's Exception Handling Revisited

Richard Carlsson
Department of Information Technology, Uppsala University, Sweden
richardc@csd.uu.se

Björn Gustavsson
Ericsson, Sweden
bjorn@erix.ericsson.se

Patrik Nyblom
Ericsson, Sweden
pan@erix.ericsson.se

ABSTRACT

This paper describes the new exception handling in the ERLANG programming language, to be introduced in the forthcoming Release 10 of the Erlang/OTP system. We give a comprehensive description of the behaviour of exceptions in modern-day ERLANG, present a theoretical model of the semantics of exceptions, and use this to derive the new `try`-construct.

1. INTRODUCTION

The exception handling in ERLANG is an area of the language that is not completely understood by most programmers. There are several details that are often overlooked, sometimes making the program sensitive to changes, or hiding the reasons for errors so that debugging becomes difficult. The existing `catch` mechanism is inadequate in many respects, since it has not evolved along with the actual behaviour of exceptions in ERLANG implementations. The addition of a new exception handling construct to replace the existing `catch` has long been discussed, but has not yet made its way into the language.

The purpose of this paper is twofold: first, to explain the realities of exceptions in ERLANG, and why the creation of a new exception-handling construct has been such a long and complicated process; second, to describe in detail the syntax and semantics of the finally accepted form of the `try`-construct, which is to be introduced in Erlang/OTP R10.

The layout of the rest of the paper is as follows: Section 2 describes in detail how exceptions in ERLANG actually work, and the shortcomings of the current `catch` operator. Section 3 explains how the new `try`-construct was derived, and why `try` in a functional language has different requirements than in an imperative language such as C++ or Java. In Section 4, we refine the exception model and give the full syntax and semantics of `try`-expressions, along with some concrete examples. Section 5 discusses related work, and Section 6 concludes.

Erlang Workshop '04 22/09/2004, Snowbird, Utah, USA

2. EXCEPTION HANDLING IN ERLANG

2.1 Exceptions as we know them

The exception handling in ERLANG, as described in [2], was designed to be simple and straightforward. An exception can occur when a built-in operation fails, such as an arithmetic operation, list operation, pattern matching, or case-clause selection, or when the user calls one of the built-in functions `exit(Term)` or `throw(Term)`.

The exception is then described by an ordinary ERLANG term (often called the *reason*), such as an atom `badarg` or `badarith`, or a tuple like `{badmatch, Value}` where `Value` is the data that could not be matched. To prevent exceptions from propagating out of an expression, the expression can be placed within a `catch` operator, as in

```
X = (catch f(Y))
```

(the parentheses are needed in this case because of the low precedence of `catch`), or more often

```
case catch f(Y) of
  ...
end
```

to immediately switch on the result. If the expression within the `catch` completes normally, the resulting value is returned as if the `catch` had not been there.

When an exception occurs, one of two things can happen: either the exception is not caught by the executing process, and in that case the process terminates, possibly propagating a signal to other processes; otherwise, the execution had entered (but not yet exited) one or more catch-expressions before the exception happened, and execution is resumed at the latest entered `catch`, unrolling the stack as necessary. The result of the `catch` expression then depends on how the exception was caused: if it occurred because of a call to `throw(Term)`, the result is exactly `Term`.[1] Otherwise, the result will have the form of a tuple `{'EXIT', Term}`. For example,

```
catch (1 + foo)
```

returns `{'EXIT', badarith}` (since `foo` is an atom), while

```
catch throw(foo)
```

returns the atom `foo`, the intention being that `throw` should be used for "nonlocal returns" (e.g., for escaping out of a deep recursion) and other user-level exceptions. The `exit` function, on the other hand, is intended for terminating the process, and behaves like a run-time failure, so that

[1]Much like `catch`/`throw` in Common Lisp.

```
catch exit(user_error)
```

returns {'EXIT', user_error}.

One of the consequences of this (explicitly described in [2] as a design decision) was that it became possible to "fake" a run-time failure or exit, by calling `throw({'EXIT', Term})`, or by simply returning the value {'EXIT', Term}. For example, in the following code:

```
R = catch (case X of
             1 -> 1 + foo;
             2 -> exit(badarith);
             3 -> throw({'EXIT', badarith});
             4 -> {'EXIT', badarith}
             5 -> throw(ok);
             6 -> ok
           end),
case R of
  {'EXIT', badarith} -> "1-4";
  ok -> "5-6"
end
```

the semantics of `catch` makes it impossible to tell whether the value of `R` (depending on `X`) is the result of catching a run-time failure or a call to `exit` or `throw`, or if the expression completed execution in a normal way. Usually, this is not a problem; for example, most ERLANG programmers would never use a tuple {'EXIT', Term} in normal code.

2.2 Where's the catch?

In some contexts, it becomes more important to know what has actually happened. For example, consider:

```
lookup(X, F, Default) ->
  case catch F(X) of
    {'EXIT', Reason} -> handle(Reason);
    not_found -> Default;
    Value -> Value
  end.
```

where `F` is bound to some function, which should either return a value depending on `X`, or call `throw(not_found)`.

Note that the possible returned values cannot include the atom `not_found`. To solve this in general, the return values would need a wrapper, such as {ok, Value}, to separate them from any thrown terms (assuming that {ok, ...} is never thrown, much like it is assumed that {'EXIT', ...} is not normally returned by any function). This limits the usefulness of `throw` somewhat, since it requires that the normal-case return values are marked, rather than the exceptional values, which is counterintuitive and bothersome.

An idiom used by a few knowledgeable ERLANG programmers to create almost-foolproof catches is the following:

```
lookup(X, F, Default) ->
  case catch {ok, F(X)} of
    {ok, Value} -> Value;
    {'EXIT', Reason} -> exit(Reason);
    not_found -> Default;
    Term -> throw(Term)
  end.
```

Since it is guaranteed that the creation of a tuple such as {ok, *Expr*} will never cause an exception if the subexpression *Expr* completes normally, we have a way of separating exceptions in `F(X)` from normal return values – as long as we trust that nobody calls `throw({ok, ...})` within `F(X)`. Furthermore, any caught exceptions that are not of interest at this point can simply passed to `throw` or `exit` again, hoping that some other catch will handle it.

This way of writing safer catches is however rarely seen in practice, since not many programmers know the trick, or bother enough to use it, since their catches mostly work anyway – at least until some other part of the code changes.

2.3 Current practice

The difficulty in properly separating exceptions from return values appears to be the main reason why although ERLANG has a catch/throw mechanism, it is still the case that in existing code, the predominant way of signalling the success or failure of a function is to make it return tagged tuples like {ok, Value} in case of success and {error, Reason} otherwise, forcing the caller to check the result and either extract the value or handle the error. This often leads to a clumsy programming style, in the many cases where errors are actually rare and it is even rarer that the caller wants to handle them. (If a significant part of all calls to a function tend to fail, the above can still be a good way of structuring the function interface, but typically, failure happens in only a very small fraction of all calls.)

In C programs [5], the code is often interspersed with many checks to see if function calls have returned valid results, even though there is seldom much that the programmer can do if this was not the case, except terminate the program. The lack of an exception mechanism makes the code less readable, more time-consuming to write, and more error prone since forgetting to check a value can be fatal. ERLANG programs suffer similar problems: even if the programmer cannot do anything constructive to handle an error, he must still remember whether a called function returns a naked value or {ok, Value}, and in the latter case must also decide what should happen if instead {error, Reason} is returned. The following idiom is often used:

```
{ok, Value} = f(X)
```

so that if the call succeeds, the relevant part of the result is bound to `Value`, and if the call instead returns {error, Reason}, it will cause a `badmatch` exception. The main drawback is that it points out the wrong cause of the problem, which was a failure within `f(X)`, and not in the pattern matching. Also, the wrapping convention remains a cause of irritation because one is forced to write awkward code like

```
{ok, Y} = f(X),
{ok, Z} = g(Y),
{ok, Value} = h(Z)
```

when it would have sufficed with

```
Value = h(g(f(X)))
```

if the functions had returned naked values and used exceptions to signal errors.

Sometimes, programmers attempt to handle the error case as follows:

```
case f(X) of
  {error, Reason} -> exit(Reason);
  {ok, Value} -> ...
end
```

but often, the error term `Reason` returned by the function is very particular to that function, and is not suitable for passing to `exit`, so that anyone who catches the resulting exception will only be confused since there is no longer any context available for interpreting the term. So even though the programmer simply wishes to pass on the problem to be handled by someone else, it really requires interpreting the error and creating a more comprehensible report. In fact, the `badmatch` solution above is to be preferred, because it will show precisely where the program gave up, rather than pass on a cryptic term with `exit`.

2.4 Processes and signals

Since ERLANG is a concurrent language, every program is executed by a *process* (similar to a thread), and many processes can be running concurrently in an ERLANG runtime system. A signalling system is used for informing processes about when other processes terminate. As for exceptions, an *exit signal* is described by a term, which if the process terminated normally (by returning from its initial function call) is the atom `normal`. If the process terminated because it called `exit(Term)` (and did not catch the exception), the exit term is exactly the value of `Term`; thus, a process can also terminate "normally" by calling `exit(normal)`, e.g. in order to easily exit from within a deep call chain. Similarly, if the process terminated because of a run-time failure that was not caught, the exit term is the same term that would be reported as `{'EXIT', Term}` in a `catch`-expression, as for instance `badarg` or `{badmatch, Value}`.

A special case is when a process terminates because it called `throw(Term)` and did not catch the exception. In this case, the exit term will be changed to `{nocatch, Term}`, to distinguish this case from other kinds of exits.

2.5 The Truth...

To simplify the above discussion (as many readers will doubtless have noticed), we have left out a few details about exceptions as they appear in modern ERLANG implementations. The presentation in the preceding sections follows the description of exceptions in "The ERLANG Book" [2] (Concurrent Programming in ERLANG, Second Ed., 1996).

The most apparent change since then is that when a run-time failure occurs (and is then either caught in a `catch` or causes the process to terminate), the term that describes the error will also include a symbolic representation of the topmost part of the call stack at the point where the error occurred. (This does not happen for calls to `exit` or `throw`.) The general format is `{Reason, Stack}`, where `Reason` is the normal error term as described in the previous sections. For example, calling `f(foo)` where:

```
f(X) -> "1" ++ g(X).
g(X) -> "2" ++ h(X).
h(X) -> X ++ ".".
```

will generate an exception with a descriptor term such as the following:

```
{badarg,[{erlang,'++',[foo,"."]},
         {foo,h,1},
         {foo,g,1},
         {foo,f,1}]}
```

Details in the stack representation may vary depending on implementation, cause of error, and call history.[2] (Also note that because of tail call optimization, many intermediate function calls cannot be reported, since there is by definition no trace left of them.)

If evaluation of ***`Expr`*** completed normally with result R
then
    the result of `catch` ***`Expr`*** is R,
else
    the evaluation threw an exception $\langle term, thrown \rangle$;
    if *thrown* is `true`
    then
        the result of `catch` ***`Expr`*** is *term*,
    else
        the result of `catch` ***`Expr`*** is `{'EXIT',` *term*`}`

Figure 1: Semantics of `catch` *`Expr`*

Thus, for example the call `f(0)` where

```
f(X) -> catch 1/X.
```

will actually return

```
{'EXIT', {badarith, [{foo,f,1}, ...]}}
```

in a modern system, rather than `{'EXIT', badarith}`. However, the following code:

```
catch exit(Term)
```

will still yield `{'EXIT', Term}`, without any symbolic stack trace, and similarly

```
catch throw(Term)
```

yields just `Term`, as before.

2.6 ...The Whole Truth...

Now, the observant reader may have noticed that although it would appear that an exception is fully determined by the "reason" term only, in fact at least one other component is necessary to completely describe an exception, namely, a flag that signals whether or not it was caused by `throw(Term)`. (This follows from the semantics of process termination and signals; cf. Section 2.4.)

Internally, an exception is then a pair $\langle term, thrown \rangle$, where *thrown* is either `true` or `false`, and *term* is the "reason" term. The semantics of `catch` ***`Expr`*** can now be described as shown in Figure 1. Note that it is the `catch` operator that decides (depending on the *thrown* flag) whether or not to wrap the reason term in `{'EXIT', ...}`.

2.7 ...And Nothing But The Truth

Something, however, is still missing. When `throw(Term)` is not caught, and causes the process to terminate (as described in Section 2.4), the exit term is no longer simply `{nocatch, Term}`, but rather `{{nocatch, Term}, [...]}`, with a symbolic stack trace just as for a run-time failure. This means that the stack trace cannot be added onto the "reason" term *until it is known what will happen to the exception*,

[2]The actual arguments to the last called function are not always included; only the arity of the function. The next-to-last call is often missing because its return address was never stored on the stack and could not be recovered.

If evaluation of *Expr* completed normally with result R
then
    the result of `catch` *Expr* is R,
else
    the evaluation threw exception $\langle term, thrown, trace \rangle$;
    if *thrown* is `true`
    then
        the result of `catch` *Expr* is *term*,
    else
        if *trace* is *null*
        then
            the result is {`'EXIT'`, *term*}
        else
            the result is {`'EXIT'`, {*term*, *trace*}}

Figure 2: Modified semantics of `catch` *Expr*

If evaluation of the initial call completed normally
then
    the exit term is the atom `normal`
else
    the evaluation threw exception $\langle term, thrown, trace \rangle$;
    if *thrown* is `true`
    then
        the exit term is {{`nocatch`, *term*}, *trace*}
    else
        if *trace* is *null*
        then
            the exit term is *term*
        else
            the exit term is {*term*, *trace*}

Figure 3: Semantics of process termination

since if it is caught in a `catch`, it *must not* include any stack trace.

As a consequence, we have to extend the full description of an exception to a triple $\langle term, thrown, trace \rangle$, where *trace* is either a symbolic stack trace or a special value *null*, so that *trace* is *null* if and only if the exception was caused by a call to `exit`.

The semantics of `catch` must also be modified as shown in Figure 2, so that in the case where the expression has thrown an exception, and *thrown* is `false`, we have a choice depending on the value of *trace*. The exit term for a terminating process is determined in a similar way, shown in Figure 3.

One last, not very well documented, detail is that when a process terminates due to an exception, and the exception was not caused by a call to `exit(Term)`, this event will be reported by the ERLANG runtime system to the *error logger* service. (In other words, as long as a process terminates normally, or through a call to `exit`, it is considered a normal event from the runtime system's point of view.) This shows once again that it is necessary to preserve the information about whether or not the exception was caused by `exit`, until it is known how the exception will be handled.

2.8 Love's Labour's Lost in Space

For the programmer, currently the only means of intercepting and inspecting an exception is the `catch` operator, but as we have seen, this will lose information which cannot be re-created. For example, as described in Section 2.2, the following code attempts to separate exceptions from normal execution, and transparently pass on all exceptions that do not concern it:

```
case catch {ok, ...} of
  {ok, Value} -> ...;
  {'EXIT', Reason} -> exit(Reason);
  not_found -> ...;
  Term -> throw(Term)
end
```

However, when `throw(Term)` is executed in the last clause, it will create a *new* exception $\langle term, thrown, trace \rangle$ having the same values for *term* and *thrown* as the caught exception, but with a different *trace*. This is observable if the exception causes the process to terminate, and since the original stack trace was lost, it will hide the real reason for the exception.

Furthermore, in the `exit(Reason)` case, the *trace* component of the new exception will be set to *null* by `exit`. Now note that if the caught exception had a non-*null* trace component, the `catch` will already have added that trace onto `Reason`, so in a sense, the term has been "finalized": if the new exception is caught in another `catch`, or causes the process to terminate, the term will look exactly the same as if it had never been intercepted by the above code. But there is one problem: since we used `exit` to pass on the exception, it will *not* be reported to the error logger if it causes the process to terminate – even if the original exception was caused by a run-time failure, which ought to be reported.

One built-in function that we have not mentioned so far, because it is not well known, and has mostly been used in some of the standard libraries, is `erlang:fault(Term)`, which is similar to `exit` but instead causes a run-time failure exception, i.e., such that *trace* $\neq$ *null* and *thrown* = `false`. We could then try to improve on the above code by splitting the {`'EXIT'`, `Reason`} case in two:

```
{'EXIT', {Reason, Stack}} when list(Stack) ->
    erlang:fault(Reason);
{'EXIT', Reason} ->
    exit(Reason);
```

which will preserve the error logging functionality, as long as we can trust that the first clause matches all run-time errors, and nothing else. But like in the `throw` case, we now lose the original stack trace when we call `erlang:fault(Reason)`. What if we tried `erlang:fault({Reason, Stack})` instead? Well, if the exception is caught again, it will then get the form {{`Reason, Stack1`}, `Stack2`}, and so on if the process is repeated. This preserves all the information, but could cause problems for code that expects to match on the `Reason` term and might not recognize the case when it is nested within more than one {`..., Stack`} wrapper.

Thus, with a fair amount of careful coding, we can now catch exceptions, examine them, and pass them on if they are not of interest, but still not without affecting their semantics in most cases – for `throw`, we lose the stack trace, and for run-time failures we modify the "reason" term. The method is also not foolproof – calls to `throw({ok, Value})`, `throw({'EXIT', Reason})`, or `exit({Term, [...]})` will all cause the wrong clause to be selected. Not to mention that the code is difficult to understand for anyone who is not very familiar with the intricacies of exceptions.

There really should be a better way.

```
try
  Expressions
catch
  Exception_1 -> Body_1;
  ...
  Exception_n -> Body_n
end
```

Figure 4: Basic form of `try`-expression

3. TRY AND TRY AGAIN

At least a few of the problems with `catch` have been widely known by ERLANG programmers, and for several years, there has been an ongoing discussion among both language developers and users about the addition of a new, more flexible and powerful construct for exception handling to the language. The following attempts to be a complete list of requirements for such a construct:

1. It should be possible to strictly separate normal completion of execution from the handling of exceptions.
2. It should be possible to safely distinguish exceptions caused by `throw` from other exceptions.
3. It should be possible to safely distinguish exceptions caused by `exit` from run-time failures.
4. The behaviour of the existing `catch` must not change; nor the behaviour when an exception causes process termination. Existing programs should work exactly as before.
5. It should be possible to use ordinary pattern matching to select which exceptions are to be handled. Exceptions which do not match any of the specified cases should automatically be re-thrown without changing their semantics.
6. It should be straightforward to rewrite all or most uses of `catch` to the new construct, so that there is no incentive for using `catch` in new code.
7. It should be simple to write code that guarantees the execution of "cleanup code" regardless of how the protected section exits.

3.1 A first try

A form of `try...catch...end` construct for ERLANG was first described in the tentative Standard Erlang Language Specification [3] (only available in a draft version) by Bartlund and Virding; however, this work was never completed. Their suggested construct (mainly inspired by C++ [10], Java [4], and Standard ML [7]) had the general form shown in Figure 4, where if evaluation of *Expressions* succeeded with result R, the result of the `try` would also be R; otherwise, the clauses would be matched in top-down order against either `{'THROW', Value}`, for an exception caused by a `throw`, or `{'EXIT', Reason}` for other exceptions. In the terminology of Section 2.6, it would make the *thrown* flag of the exception explicit (which the `catch` operator does not).

This would fulfill requirements 1 and 2 above, and partially requirements 5 and 4; in particular, the distinction between `exit` and run-time failures was not noted in [3]. In fact, it was during an early attempt by the first author of the present paper to implement the `try` construct, that many of the complications described in Section 2 were first uncovered. As it turned out, the *de facto* behaviour of exceptions in the Erlang/OTP implementation was no longer consistent with any existing description of the language. The only result at that time was that the inner workings of the exception handling were partially rewritten in preparation for future extension, but it was apparent that the `try` construct had to be redesigned before it could be added to the language.

3.2 Making a better try

Since then, several variations of the `try` construct have been considered, but all have been found lacking in various respects. The main problem has turned out to be the balance between simplicity and power of expression. For example, most of the time, the programmer who wishes to catch a `throw` will not be interested in viewing the stack trace, and should preferably not be forced to write a complicated pattern like `{'THROW', Term, Stack}` when the only important part is `Term`. (Also, it would be a waste of time to construct a symbolic trace which is immediately discarded by the catcher.) However, the stack trace should be available when needed.

Also, point 6 above was more of a problem than expected. As seen in some of our previous examples, the following is a common way of using `catch` in existing programs:

```
case catch f(X) of
  {'EXIT', Reason} -> handle(Reason);
  Pattern_1 -> Body_1;
  ...
  Pattern_n -> Body_n
end
```

i.e., which uses a single list of patterns for matching both in the case of success and in the case of catching an exception. As we have described, this makes it possible to mistake a returned result for an exception and vice versa. However, it is an extremely convenient idiom, because it is very often the case that regardless if the evaluation succeeds or throws an exception, a switch will be performed on the result in order to decide exactly how to proceed.

With the `try...catch...end` as suggested in [3], the same effect could only be achieved as follows:

```
R = try {ok, f(X)}
  catch
    Exception -> Exception
  end,
case R of
  {ok, Pattern_1}} -> Body_1;
  ...
  {ok, Pattern_n}} -> Body_n;
  {'THROW', Term} -> ...;
  {'EXIT', Reason} -> ...
end
```

using the trick from Section 2.2 to make sure that the result of normal evaluation is always tagged with `ok`; the main difference being that the `try` version cannot be fooled by e.g. calling `throw({ok, Value})`. So, although the above code is safe, it is quite inconvenient to have to write such a complicated expression for what could be so easily expressed using `catch`.

Analyzing the `try` construct in terms of continuations helps us understand what is going on here. (A continuation is simply "that code which will take the result R of the current expression and continue", and can thus be described as a function $c(R)$; for example, when a function call finishes, it conceptually passes the return value to its continuation, i.e., the return address.[3]) First of all, we have a main continuation c_0 which will receive the final result of evaluating the whole `try...catch...end` expression. Now consider the expression between `try...catch`: if its evaluation succeeds with result R, it will use the *same* continuation $c_0(R)$, i.e., R becomes the result of the whole `try`. On the other hand, if the expression throws an exception E, it will use *another* continuation $c_f(E)$, which we call the "fail-continuation".

The code in c_f is that which does the pattern matching on the exception E over the clauses between `catch...end`. If none of the patterns match, the exception will be re-thrown, and we are done. Otherwise, the first matching clause is selected, and its body is evaluated. (If this should throw a new exception, it will not be caught here since it is not within the `try...catch` section.) The resulting value is finally passed to c_0, and becomes the result of the `try`.

Now let's look at what the continuation $c_0(R)$ gets: its input R is either the result of evaluating the `try...catch` section (if that succeeded), or otherwise it is a value returned by one of the `catch...end` clauses. This is useful in typical situations where exceptions are handled by substituting a default value, as in:

```
Value = try lookup(Key, Table)
        catch
          not_found -> 0
        end
```

However, if we want to perform a *different* action in case the `try...catch` part succeeds, than if an exception occurs, we have no choice but to pass to c_0 not only the result, but also an indication of which path was actually taken. (This is what we did in the previous example, using `{ok, ...}` to tag the result upon success.) It is this limitation of the `try...catch...end` that forces us to go from control flow to a data representation and back to control flow again. It should be noted here that the exception handling in Standard ML [7] (and possibly other functional languages with exceptions) suffers from the same limitation.

A much more elegant solution would be if the programmer could specify code to be executed only in the success case, before control is transferred to the main continuation c_0. In addition to c_f, we therefore introduce a success-continuation c_s, so that if evaluation of the `try...catch` section succeeds with result R, the continuation used would be $c_s(R)$, rather than $c_0(R)$.

A practical syntax for expressing both the success case and the exception case in a single `try` is shown in Figure 5,[4] where the code in $c_s(R)$ does pattern matching on R over the clauses between `of...catch`. If a clause matches, its body is evaluated, and the result is passed to c_0, just like for an exception-handling clause. (If no clause should match, it is a run-time error, and will cause a `try_clause` exception.) Each clause may also have a guard, apart from the pattern, just like any `case`-clause; we have left this out for the sake of readability.

```
try Expressions of
  Pattern_1 -> Body_1;
  ...
  Pattern_n -> Body_n
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
end
```

Figure 5: General form of `try`-expression

Note that the old syntax from Figure 4, which leaves out the `of...` part, can still be allowed, by simply defining it as equivalent to

```
try Expressions of
  X -> X
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
end
```

(where `X` is a fresh variable), in effect making $c_s(R) = c_0(R)$, as before.

At this point, the reader might be wondering why the problem could not have been solved as follows, using the original form of `try`:

```
try
  case Expressions of
    Pattern_1 -> Body_1;
    ...
    Pattern_n -> Body_n
  end
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
end
```

The difference is that in this case, all the success actions *Body*$_i$, as well as the pattern matching, *are now within the protected part of the code*. Thus, the `catch...end` section will handle *all* exceptions that may occur – not only those within *Expressions*. If this was what we wanted, we might as well have moved the whole `case...end` to a new function `f` and simply written

```
try f(...)
catch
  ...
end
```

however, often we *do not* want the same exception handling for both the protected part and the success actions.

It is interesting to note that in the imperative languages which pioneered and made popular the `try...catch` paradigm, i.e., mainly Ada [6], C++ [10] and Java [4], there has never been any need for an explicit success-continuation syntax. The reason is simple: in these languages, the programmer can change the flow of control by use of "escapes"

[3] If it helps, just think of continuations as goto-labels.

[4] First suggested by Fredrik Linder, who also pointed out the weakness in the original `try`, in a discussion at the ERLANG Workshop in Firenze, Italy, 2001.

such as `return`, `goto`, `break`, and `continue`; for example, forcing an early return from within an exception-handling branch. In a functional language such as ERLANG, this is not an option.

4. PUTTING IT ALL TOGETHER

Now we are ready to specify the full syntax and semantics of the new `try` construct as it will appear in Release 10 of Erlang/OTP. We begin by revising the exception model from Section 2.

In Section 2.7, we came to the conclusion that exceptions had to be described by a triple $\langle term, thrown, trace \rangle$, and gave the semantics of `catch` and process termination in terms of this. In retrospect, we can refactor the representation to make it more intuitive and easier to work with.

4.1 Semantics of exceptions in Erlang

We define an ERLANG exception to be a triple

$$\langle class, term, trace \rangle$$

such that:

- *class* specifies the class of the exception: this must be either `error`, `exit`, or `throw`,
- *term* is any ERLANG term, used to describe the exception (also referred to as the "reason"),
- *trace* is a partial representation of the stack trace at the point when the exception occurred. The trace may also include the actual parameters passed in function calls; the details are implementation-dependent. There is no special *null* value for the *trace* component.

The different classes of exceptions can be raised as follows:

- When a run-time failure occurs (with reason *term*), or the program calls `erlang:fault(`*term*`)`, the raised exception will have the form $\langle$`error`, *term*, *trace*$\rangle$.
- When the program calls `exit(`*term*`)`, the exception will have the form $\langle$`exit`, *term*, *trace*$\rangle$.
- When the program calls `throw(`*term*`)`, the exception will have the form $\langle$`throw`, *term*, *trace*$\rangle$.

Let T be a function that creates a symbolic representation of a trace as an ERLANG term. The modified semantics of the `catch` operator is shown in Figure 6, and the semantics of process termination in Figure 7. Note that the behaviour remains functionally equivalent to what we described earlier in Figures 2 and 3.

4.2 Syntax and semantics of try-expressions

First, we note that the convention of using tagged tuples such as `{'EXIT', ...}` was introduced only as a means of distinguishing errors and exits from normal return values and thrown terms in the `catch` operator. In our new `try` construct, it is not necessary to stick to this convention. Instead, we will use a syntax which is both easier to read and requires less typing.

A `try`-expression has the general form from Figure 5, where *`Expressions`* and (for $i \in [1, n]$, and $j \in [1, m]$) all the *`Body`*$_i$, and *`Handler`*$_j$ are comma-separated sequences of one or more expressions, and the *`Pattern`*$_i$ are arbitrary

If evaluation of *`Expr`* completed normally with result R
then
  the result of `catch` *`Expr`* is R,
else
  the evaluation threw exception $\langle class, term, trace \rangle$;
  if *class* is `throw`
  then
    the result of `catch` *`Expr`* is *term*,
  else if *class* is `exit`
  then
    the result is {`'EXIT'`, *term*}
  else
    the result is {`'EXIT'`, {*term*, $T(trace)$}}

Figure 6: Final semantics of `catch` *`Expr`*

If evaluation of the initial call completed normally
then
  the exit term is the atom `normal`
else
  the evaluation threw exception $\langle class, term, trace \rangle$;
  if *class* is `exit`
  then
    the exit term is *term*
  else
    the event will be reported to the error logger;
    if *class* is `throw`
    then
      the exit term is {{`nocatch`, *term*}, $T(trace)$}
    else
      the exit term is {*term*, $T(trace)$}

Figure 7: Final semantics of process termination

patterns. As noted in Section 3.2, the `of...` part may be left out. Like in a `case`-expression, variable bindings made in *`Expressions`* are available within the `of...catch` section (but not between `catch...end`), and variables are exported from the clauses if and only if they are bound in all cases.

The exception patterns *`Exception`*$_j$ have the following general form:

```
Class:Reason
```

where *`Class`* is either a variable or a constant. If it is a constant, or a variable which is already bound, the value should be one of the atoms `error`, `exit`, or `throw`.

The *`Class`*`:` part of each exception pattern may be left out, and the pattern is then equivalent to

```
throw:Reason
```

The reason for this shorthand is that typically, only `throw` exceptions should be intercepted. `error` exceptions should in general be allowed to propagate upwards, usually terminating the process, so that unexpected run-time failures are not masked. Furthermore, an `exit` exception means that a decision was made to terminate the process, and a programmer should only override that decision if he knows what he is doing. Therefore, the default class is `throw`.

The semantics of `try` is shown in Figure 8. (As before, clause guards have been left out for the sake of readability, and for the same reason, we do not show the variable

If evaluation of *Expressions* completed normally
with result R,
then
the result of `try...end` is equivalent to that of

```
case R of
  Pattern_1 -> Body_1;
  ...
  Pattern_n -> Body_n
end
```

else
the evaluation threw exception $\langle class, term, trace \rangle$;
for each $Exception_j = Class_j{:}Reason_j$
let $Triple_j = \{Class_j, Reason_j, \_\}$
and for each $Exception_j = Reason_j$
let $Triple_j = \{\texttt{throw}, Reason_j, \_\}$;
the result of `try...end` is then equivalent to that of

```
case {class, term, trace} of
  Triple_1 -> Handler_1;
  ...
  Triple_m -> Handler_m;
  X -> rethrow(X)
end
```

where `X` is a fresh variable.

Figure 8: Semantics of `try`-expressions

scoping rules.) The *rethrow* operator is a built-in primitive which cannot be accessed directly by the programmer; its function is simply to raise the caught exception again, without losing any information. Note that the *trace* component of an exception is assumed to have a cheap, implementation-dependent representation which we do not want to expose to users.

To inspect the stack trace of an exception, a new built-in function `erlang:get_stacktrace()` is added, which returns $T(trace)$ where *trace* is the stack trace component of the last occurred exception of the current process, and T is the function used in the semantics of `catch` in Figure 6. If no exception has occurred so far in the process, an empty list is returned.

Finally, we could add a generic exception-raising function `erlang:raise(Class, Reason)`, where `Class` must be one of the atoms `error`, `exit`, and `throw`, and `Reason` may be any term. We will see one possible use of such a function in the following section.

4.3 Cleanup code

A very common programming pattern is to first prepare for a task (e.g. by allocating one or more resources), then execute the task, and finally do the necessary cleanup. It is then often important that the cleanup stage is executed independently of how the main task is completed, i.e., whether it completes normally or throws an exception. Figure 9 shows an example of how this could be done using `try`-expressions. However, the code has several weak points:

- The cleanup code, in this example the calls to `close`, must be repeated in each case. (We only show two cases, but in general there could be any number.)
- In the success case, we must bind the result to a new variable just to hold it temporarily while we perform the cleanup.

```
read_file(Filename) ->
  FileHandle = open(Filename, [read]),
  try
    read_opened_file(FileHandle)
  of
    Data ->
      close(FileHandle),
      Data
  catch
    Class:Reason ->
      close(FileHandle),
      erlang:raise(Class, Reason)
  end.
```

Figure 9: Allocate/use/cleanup example

```
try
  Expressions
after
  Cleanup
end
```

Figure 10: `try...after...end` expressions

- The generic failure-case, i.e., the code that catches any exception, performs the cleanup, and then re-throws the exception, is unnecessarily verbose.
- The explicit re-throw using `erlang:raise()` will not preserve the stack trace of the original exception.

All of the above problems can be solved by adding one more feature to our `try`-expressions. In Common Lisp [9], the special form `unwind-protect` is used to guarantee execution of cleanup-code; in Java [4], this is done by including a `finally` section in `try`-expressions. The same idea can be used in ERLANG. We start out by defining a new form of `try`-expression, shown in Figure 10,[5] with a well-defined meaning.

The semantics of `try...after...end` is shown in Figure 11. We can now easily allow the use of `after` directly together with `try...catch...end`, by defining the fully general syntax shown in Figure 12 as equivalent to

```
try
  try Expressions of
    Pattern_1 -> Body_1;
    ...
    Pattern_n -> Body_n
  catch
    Exception_1 -> Handler_1;
    ...
    Exception_m -> Handler_m
  end
after
  Cleanup
end
```

(where as before, the `of...` part may be left out), which guarantees that in all cases, the *Cleanup* code will be exe-

[5]The use of the `after` keyword for this purpose is not yet decided; possibly, a new keyword such as `finally` will be added instead.

If evaluation of *Expressions* completed normally
with result R,
then
    the result of `try...after...end` is equivalent
    to that of
        `begin`
            `X = R,`
            `Cleanup,`
            `X`
        `end`
    where `X` is a fresh variable,
else
    the evaluation threw exception $\langle class, term, trace \rangle$;
    the result is then equivalent to that of
        `begin`
            `Cleanup,`
            rethrow({*class*, *term*, *trace*})
        `end`

Figure 11: Semantics of `try...after...end`

```
try Expressions of
  Pattern_1 -> Body_1;
  ...
  Pattern_n -> Body_n
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
after
  Cleanup
end
```

Figure 12: Fully general try-expression

cuted, after evaluation of one of the $Body_i$ or $Handler_j$ has completed. This is expected to be the desired behaviour in most cases, since it gives the exception handling clauses a chance to act before any resources are deallocated by the cleanup code. It is also easy to manually nest `try`-expressions to get another evaluation order, e.g.:

```
try
  try
    Expressions
  after
    Cleanup
  end
of
  Pattern_1 -> Body_1;
  ...
  Pattern_n -> Body_n
catch
  Exception_1 -> Handler_1;
  ...
  Exception_m -> Handler_m
end
```

4.4 Examples

To demonstrate some uses of `try`-expressions, we begin by showing in Figure 13 how the `catch` operator may be

```
try Expr
catch
  throw:Term -> Term;
  exit:Term -> {'EXIT', Term};
  error:Term ->
    Trace = erlang:get_stacktrace(),
    {'EXIT', {Term, Trace}}
end
```

Figure 13: `catch` *Expr* implemented with `try`

```
open(Filename, ModeList) ->
  case file:open(Filename, ModeList) of
    {ok, FileHandle} ->
      FileHandle;
    {error, Reason} ->
      throw({file_error, Reason})
  end.
```

Figure 14: Wrapper for `file:open/2`.

implemented in a transparent way using `try`. The reader is invited to compare this to the semantics in Figure 6 and verify that they are equivalent.

Figure 14 shows a wrapper function for the standard library `file:open/2` function. The functions in the `file` module return {`ok, Value`} or {`error, Reason`}, while the wrapper always returns a naked value upon success, and otherwise throws {`file_error, Reason`}. The latter makes it easy to identify a file-handling exception even if it is not caught close to where it occurred. Note that if an exception is generated within `file:open/2`, we do not catch it, but allow it to be propagated exactly as it is.

Figure 15 demonstrates the use of `after` in a typical situation where the code allocates a resource, uses it, and afterwards does the necessary cleanup (cf. Figure 9). Note the two-stage application of `try`: First it is used to handle errors in `open` (see Figure 14). Since `FileHandle` is only defined if the call to `open` succeeds, only then need we (or can we) close the file. The next `try` makes sure that the cleanup is done regardless of how the code is exited. An important detail is that by keeping the allocation and the release of the resource close together in the code, no other part of the program needs to know that there is any cleanup to be done.

In an imperative language like Java, it is common to initially assign a null value to variables, and then let the protected section attempt to update them when it is allocating resources. The cleanup section can then test for each resource whether it needs to be released or not. In a functional language where variables cannot be updated, it is cleaner to handle each resource individually, and separate the allocation from the use-and-cleanup, as we did above.

5. RELATED WORK

The concept of user-defined exception handling seems to have originated in PL/I [1], and from there made its way (see e.g. [8]) in different forms into both Lisp [9] and Ada [6], as well as other languages. Ada in its turn was a direct influence on C++ [10], and thus indirectly on Java [4].

In the Object-Oriented languages C++ and Java, an exception is completely described by the thrown object. In

```
read_file(Filename) ->
  try open(Filename, [read]) of
    FileHandle ->
      try
        read_opened_file(FileHandle)
      after
        close(FileHandle)
      end
  catch
    {file_error, Reason} ->
      print_file_error(Reason),
      throw(io_error)
  end.
```

Figure 15: Allocate/use/cleanup with `try...after`

C++, any object can be thrown; in Java, only subclasses of `Throwable` may be thrown. For Java, this means that implementation-specific information like a stack trace may be easily stored in the object without exposing its internal representation. Since this cannot be done in ERLANG without adding a new primitive data type to the language, we have instead chosen to make the stack trace and any other debugging information part of the process state.

In Common Lisp [9], `catch` and `throw` are used for non-local return, and only indirectly for handling actual errors. Also, setting up a `catch` requires specifying a tag (any object, e.g. a symbol) for identifying the catch point, to be used later in the `throw`. To guarantee execution of cleanup-code regardless of how an expression is exited, the special form `unwind-protect` is used. The same effect is achieved in Java by including a `finally` section in `try`-expressions.

In Standard ML [7], the exception handling works much like in Ada, using the operators `raise` and `handle`. As in the originally suggested `try...catch...end` for ERLANG, there is no way of explicitly specifying a success-continuation.

6. CONCLUDING REMARKS

Exceptions in ERLANG has been a not very well understood area of the language, and the behaviour of the existing `catch` operator has for a long time been insufficient. We have given a detailed explanation of how exceptions in modern-day ERLANG actually work, and presented a new theoretical model for exceptions in the ERLANG programming language. Using this model, we have derived a general `try`-construct to allow easy and efficient exception handling, which will be introduced in the forthcoming Release 10 of Erlang/OTP.

7. ACKNOWLEDGMENTS

The authors would like to thank Robert Virding, Fredrik Linder, and Luke Gorrie for their comments and ideas.

8. REFERENCES

[1] American National Standards Institute, New York. *American National Standard: programming language PL/I*, 1979 (Rev 1998). ANSI Standard X3.53-1976.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[3] J. Barklund and R. Virding. Specification of the Standard Erlang Programming Language. Draft version 0.7, June 1999.

[4] J. Gosling, B. Joy, and G. Steele. *The Java™ Programming Language*. The Java Series. Addison-Wesley, 3rd edition, 2000.

[5] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1989.

[6] Military Standard. *Reference Manual for the Ada Programming Language*. United States Government Printing Office, 1983. ANSI/MIL-STD-1815A-1983.

[7] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[8] K. Pitman. Condition handling in the Lisp language family. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, number 2022 in LNCS. Springer-Verlag, 2001.

[9] G. L. Steele. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[10] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.