Exceptions in Erlang - Redux

Richard Carlsson
Uppsala University

Björn Gustavsson
Patrik Nyblom
Ericsson

What's the big deal?

- Exceptions in Erlang are simple, right?
- Raising exceptions:
  - `throw(Term)`
  - `exit(Term)`
  - run-time failures:

```
         foo = bar  => {badmatch,bar}
           1 + foo => badarith
no matching clause => case_clause
```

etc.

Evaluation and catches

- `catch <Expr>`
  - Evaluates `<Expr>`
  - If it “completes normally” with result R, the result of the whole thing is R.
  - Otherwise, the evaluation completed abnormally with an exception. The result then depends on the cause:

```
throw(T) => T
exit(T)  => {'EXIT', T}
1 + foo  => {'EXIT', badarith}
```

Purposes of exit and throw

- `throw(Term)` is for “nonlocal returns”, escaping from deep recursion.
- `exit(Term)` is for terminating the current process.
  - special case: `exit(normal)`
- Faking exits with throw:

  -
    ```
    catch (...throw({'EXIT',badarg})...)
       => {'EXIT',badarg}
    ```

What happened?

```
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
```

Sometimes, this is a problem

```
% XXX: We hope nobody inserts 'not_found'
% in the table!

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

...with a known workaround

```
% XXX: We hope nobody throws '{ok, Value}'
% from function F.

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

Throw/catch not widely used

- `catch` always catches every exception, so the programmer must write extra code to re-throw uninteresting ones.
- (Partly) because of these difficulties, throw/catch is not commonly used in Erlang for signalling/handling errors.
- Much more common: return either `{ok,Result}` or `{error,Reason}`.

I can't believe it's not C!

- This kind of code quickly gets tedious:

```
A = case a(...) of
       {ok, A1} -> A1;
       {error, R1} -> error_in_a(R1)
    end,
...
E = case e(...) of
       {ok, E1} -> E1;
       {error, R5} -> error_in_e(R5)
    end,
foo(A,B,C,D,E)
```

One good failure deserves another

- Often, you can't handle the error anyway, so you might as well just cause a badmatch!

```
{ok, A} = a(...),
...
{ok, E} = e(...),
foo(A,B,C,D,E)
```

- This can make the real cause of the error harder to find, but at least the code gets shorter...

Functional programming?

- Sometimes, these wrappers are extra annoying:

```
{ok, F} = f(...),
{ok, E} = e(F),
{ok, G} = g(G),
{result, G}
```

- In a better world, it could have been:

```
{result, g(e(f(...)))}
```

if errors were signalled through exceptions.

Don't do this

- A misguided attempt at error handling:

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

- The term `Reason` is often completely incomprehensible outside the context of the function f.
- Even `{ok,Value}=f(...)` might be better.

Processes and signals 101

- Erlang processes can be *linked.*
- When a process terminates, its linked processes will receive a signal containing the *exit term*.
- On normal termination (return from the process' initial function call), the exit term is the atom `'normal'`.
- On termination due to `exit(Term)`, the exit term is simply `Term`.

Processes and signals 101 (continued)

- On termination due to runtime errors the exit term is the corresponding error term (`badarg`, `badarith`, etc.).
- On termination due to `throw(Term)`, the exit term is `{nocatch,Term}`.
- Throws are supposed to be caught before they reach the top of the process' call stack; if not, it's considered an error.

Small white lies

- Everything said so far is according to “The Erlang Book” (Armstrong et al., 1996).
- Things have changed:
  - Symbolic stack traces
  - Error logger (system service)
    - “Abnormal” termination of any process is logged; “normal” termination is not.
    - Return from top-level call is normal.
    - `exit(Term)` counts as normal termination, regardless of `Term`.
    - `throw(Term)` causes abnormal termination.

Symbolic stack traces

- Example code:

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

- Evaluating `f(foo)` yields this error term:

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

- Does not happen for calls to `exit` or `throw`!

There is more to exceptions than meets the eye

- At least two pieces of information are needed to describe an exception:
  - The Erlang term which will be returned by a `catch`, or included in an exit signal.
  - A flag that shows whether or not the exception was caused by `throw(Term)`.
    - The term must be wrapped in `{nocatch,...}` if a `throw`-exception terminates the process.
    - The `{'EXIT',...}` wrapper cannot be added at the point of the exception. (And `exit` cannot be completely faked by `throw`.)
- Exception: <*term*, *thrown*>.

When is the stack trace added?

- If `throw(Term)` terminates the process, the exit term will be `{{nocatch,Term},[...]}`.
  - But if the exception is caught by `catch`, the result is only `Term`, without a stack trace.
- Cannot add stack trace before we know where the exception will end up!
- Exception: <*term*, *thrown*, *trace*>.
- The *trace* part is null if and only if the exception was caused by `exit`.

Semantics of `catch Expr`

- If evaluation of `Expr` completes normally with result R, the result of the catch is R
- otherwise, we got <*term*, *thrown*, *trace*>
  - if *thrown* is true, the result is just *term*
  - else, if *trace* is null, the result is `{'EXIT',`*term*`}`
  - otherwise, the result is `{'EXIT',` `{`*term*`,`*trace*`}}`

Semantics of process termination

- If evaluation of the initial call completes normally, the exit term is '`normal`'
- otherwise, we got <*term*, *thrown*, *trace*>
  - if *thrown* is true, the exit term is `{{nocatch,`*term*`},`*trace*`}`
  - else, if *trace* is null, the exit term is *term*
  - otherwise, the exit term is `{`*term*`,`*trace*`}`

Re-throwing kills information

- The catch operator catches all exceptions:

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

- `throw(Term)` will set a new stack trace, hiding where the first exception occurred.
- `exit(Reason)` changes logged errors into non-logged exits.

The function formerly known as `erlang:fault/1`

- Analogous to `exit(Term)` and `throw(Term)`.
- Raises the kind of exception caused by runtime errors such as `foo=bar` or `1+foo`.
- Mostly used in some standard library functions for raising runtime errors.
- Now also known as `erlang:error/1`.
- Use this to generate errors – not `exit/1`!
- (Does not solve the re-throwing problem, because it also sets a new stack trace.)

Interlude

Are you sufficiently confused?

“So tell me what you want, ...

- Strict separation between normal completion of evaluation and exceptions.
- Strict separation between `throw` and other exceptions, and also between `exit` and runtime errors (`error`/`fault`).
- No change at all to existing code – the old `catch` operator must work exactly as before.
- Use pattern matching to select which exceptions will be handled.

…what you really really want!”

- Automatic re-throw of unhandled exceptions as if they had not been caught at all.
- Easy to rewrite most uses of `catch` in existing code to the new construct.
- Simple to write code that guarantees execution “on the way out”, such as cleanup code for freeing allocated resources, no matter how we leave the protected code.

`try` according to Barklund

- Suggested in the “Standard Erlang Specification” draft by J. Barklund, ca 1998:

```
try
   Expressions
catch
   Pattern_1 -> Body_1
   ...
   Pattern_N -> Body_N
end
```

- Patterns matched against `{'EXIT',Term}` or `{'THROW',Term}`.
- Nonmatching exceptions are re-thrown.

Little did they know...

- The Standard draft did not recognize the difference between exits and runtime errors.
- The logging of errors was not mentioned.
- In fact, no existing description of the Erlang language was consistent with the de facto behaviour of exceptions in Erlang/OTP.
- Most of this was not realized until we tried to implement the suggested `try` construct.

`try` is easier said than done

- Need to separate three types of exceptions, rather than two.
- Must make the stack trace accessible somehow.
- Forcing users to switch on patterns like `{'THROW',{not_found,X},Stack}` will either
  - make them not use `try` unless they have to,
  - or, make them catch more than they should by writing patterns like {_,{not_found,X},_}
- Most of the time, users don't want to look at the stacktrace. (Mainly useful in top loops, etc.)

try this without a catch

- This kind of code is very common:

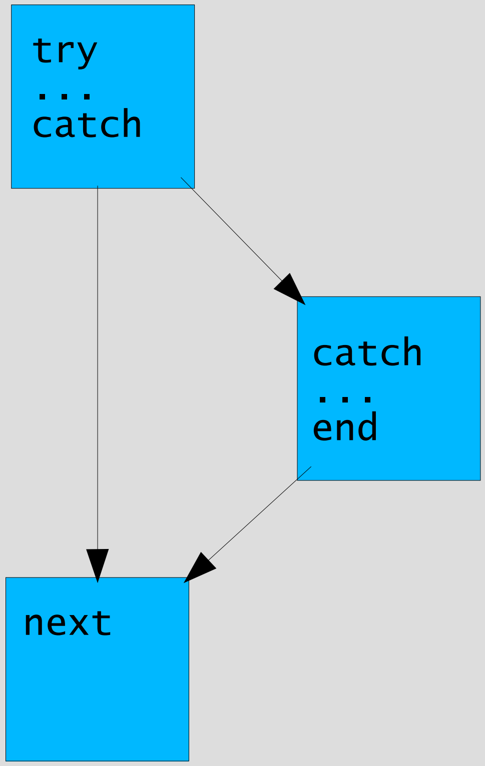
```
case catch f(X) of
   {'EXIT',Reason} -> handle(Reason);
   Pattern_1 -> Body_1;
      ...
   Pattern_N -> Body_N
end
```

- How can we replace this with an equivalent `try...catch...end`?

mix and match

```
% Can't be fooled by throw({ok,...}):
R = try
       {ok, f(X)}
    catch
       Exception -> Exception
    end,
case R of
   {ok,Pattern_1} -> Body_1;
      ...
   {ok,Pattern_N} -> Body_N;
   {'EXIT',Reason} -> handle(Reason);
   {'THROW',Term} -> throw(Term)
end
```

What's missing in this picture?

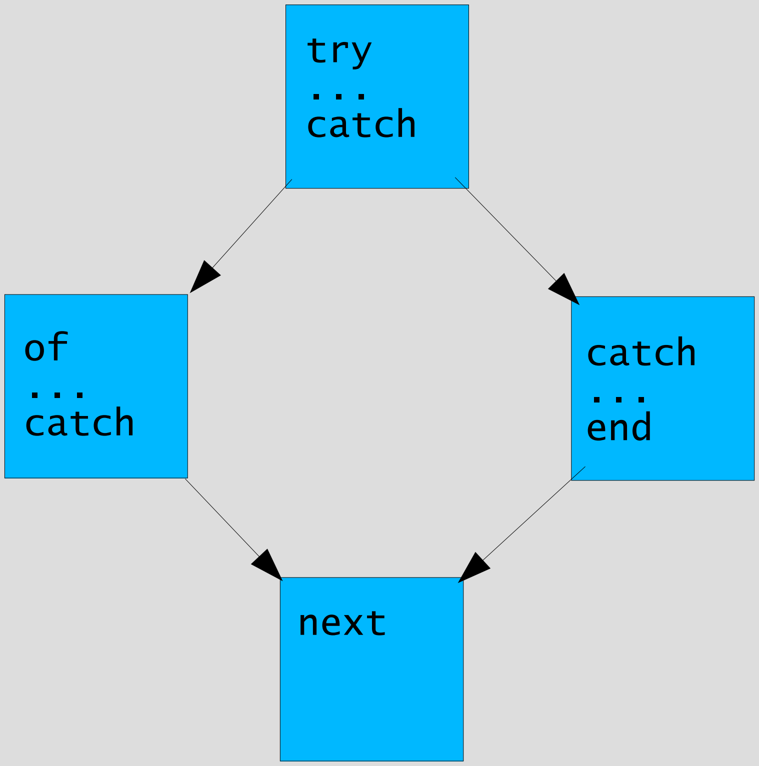


A better `try`

- Allow user to hook into the success case:

```
try
   Expressions
of
   Pattern_1 -> Body_1
   ...
   Pattern_N -> Body_N
catch
   Exception_1 -> Handler_1
   ...
   Exception_M -> Handler_M
end
```

Now we control all exits



A simple equivalence

We define

```
try
   Expressions
catch
   ...
end
```

to be syntactic sugar for

```
try Expressions of
   X -> X
catch
   ...
end
```

A bit of refactoring

- We generalize the *thrown* flag to *class*.
- Exception: <*class*, *term*, *trace*>.
  - exit(*term*) => <exit, *term*, *trace*>
  - throw(*term*) => <throw, *term*, *trace*>
  - error(*term*) => <error, *term*, *trace*>
- No null value for *trace* – it is always defined.
- Easy to rewrite the semantics for `catch` etc. to use this representation instead.

No more 'EXIT' or 'THROW'

- 'EXIT' was chosen for the old `catch` to be “different from typical runtime values”.
- Of course, there was never any guarantee.
- We don't need this for `try`!
- We introduce a new form of pattern for matching on exceptions:

```
Class:Term
```

where `Class` is an atom (`exit`, `error` or `throw`) or a variable (possibly bound).

The class can be left out

- If the `Class:` part is left out, the class defaults to `throw`.
  - Typically, programmers should not catch `exit` or `error` exceptions, unless they really know what they are doing!
  - Makes it more obvious when somebody actually tries to catch exits or errors.
  - Lazy programmers don't catch exceptions by mistake. (At least not as many.)
  - `try`/`throw` becomes a straightforward error handling mechanism for function calls.

Getting the stacktrace

- We have a new built-in function `erlang:get_stacktrace()`.
  - Returns the symbolic stack trace (a list of terms) of the latest occurred exception.
  - Yields an empty list if no exception has occurred so far.
  - No need to build the symbolic form of the stack trace (expensive) until it is required; when an exception occurs, a “quick-save” is made of the necessary data.

Example: catch as try

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

Cleaning up

- Very common pattern:
  - Allocate a resource
  - Do some stuff (if allocation succeeded)
  - Deallocate the resource
- Want to guarantee that the resource is always deallocated regardless of exit path.
- Possible with `try` as described, but verbose and clumsy.

Re-using old keywords

- We define a new form of `try`:

  ```
  try
     Expressions
  after
     Cleanup
  end
  ```

- Guarantees execution of `Cleanup`.
- Preserves the result of `Expressions` (both for normal completion and for exceptions).
- Exceptions in `Cleanup` take precedence.

The full Monty...

```
try
   Expressions
of
   Pattern_1 -> Body_1
   ...
   Pattern_N -> Body_N
catch
   Exception_1 -> Handler_1
   ...
   Exception_M -> Handler_M
after
   Cleanup
end
```

...is actually equivalent to

```
try
   try
      Expressions
   of
      Pattern_1 -> Body_1
      ...
   catch
      Exception_1 -> Handler_1
      ...
   end
after
   % Note that handlers run before cleanup.
   Cleanup
end
```

Rolling your own

- Easy to nest manually for other behaviour:

```
try
   try
     Expressions
   after
     Cleanup
   end
of
   ...
catch
   ...
end
```

- Allows tail calls in handlers.

A final example

```
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.
```

The End