



Exceptional Monads

Principles of Functional Programming

Martin Odersky

Exceptions

Exceptions in Scala are defined similarly as in Java.

An exception class is any subclass of `java.lang.Throwable`, which has itself subclasses `java.lang.Exception` and `java.lang.Error`. Values of exception classes can be thrown.

```
class BadInput(msg: String) extends Exception(msg)
throw BadInput("missing data")
```

A thrown exception terminates computation, if it is not handled with a `try/catch`.

Handling Exceptions with try/catch

A try/catch expression consists of a *body* and one or more *handlers*.

Example:

```
def validatedInput(): String =  
  try getInput()  
  catch  
    case BadInput(msg) => println(msg); validatedInput()  
    case ex: Exception => println("fatal error; aborting"); throw ex
```


try/catch Expressions

An exception is caught by the closest enclosing catch handler that matches its type.

This can be formalized with a variant of the substitution model. Roughly:

```
try e[throw ex] catch case x: Exc => handler
-->
[x := ex]handler
```

Here, `ex: Exc` and `e` is some arbitrary “*evaluation context*”

- ▶ that throw `ex` as next instruction to execute and
- ▶ that does not contain a more deeply nested handler that matches `ex`.

Critique of try/catch

Exceptions are a low-overhead way for handling abnormal conditions.

But there have also some shortcomings.

- ▶ They don't show up in the types of functions that throw them. (in Scala, in Java they do show up in throws clauses but that has its own set of downsides).
- ▶ They don't work in parallel computations where we want to communicate an exception from one thread to another.

So in some situations it makes sense to see an exception as a normal function result value, instead of something special.

This idea is implemented in the `scala.util.Try` type.

Handling Exceptions with the Try Type

Try resembles Option, but instead of Some/None there is a Success case with a value and a Failure case that contains an exception:

```
abstract class Try[+T]  
case class Success[+T](x: T) extends Try[T]  
case class Failure(ex: Exception) extends Try[Nothing]
```

A primary use of Try is as a means of passing between threads and processes results of computations that can fail with an exception.

Creating a Try

You can wrap up an arbitrary computation in a Try.

```
Try(expr)    // gives Success(someValue) or Failure(someException)
```

Here's an implementation of Try.apply:

```
import scala.util.control.NonFatal

object Try:
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch case NonFatal(ex) => Failure(ex)
```


Creating a Try

You can wrap up an arbitrary computation in a Try.

```
Try(expr)    // gives Success(someValue) or Failure(someException)
```

Here's an implementation of Try.apply:

```
import scala.util.control.NonFatal

object Try:
  def apply[T](expr: => T): Try[T] =
    try Success(expr)
    catch case NonFatal(ex) => Failure(ex)
```

Here, NonFatal matches all exceptions that allow to continue the program.

Composing Try

Just like with Option, Try-valued computations can be composed in for-expressions.

```
for
  x <- computeX
  y <- computeY
yield f(x, y)
```

If computeX and computeY succeed with results Success(x) and Success(y), this will return Success(f(x, y)).

If either computation fails with an exception ex, this will return Failure(ex).

Definition of flatMap and map on Try

```
extension [T](xt: Try[T]):  
  def flatMap[U](f: T => Try[U]): Try[U] = xt match  
    case Success(x) => try f(x) catch case NonFatal(ex) => Failure(ex)  
    case fail: Failure => fail  
  
  def map[U](f: T => U): Try[U] = xt match  
    case Success(x) => Try(f(x))  
    case fail: Failure => fail
```

So, for a Try value t,

```
t.map(f) == t.flatMap(x => Try(f(x)))  
         == t.flatMap(f andThen Try)
```


Exercise

It looks like Try might be a monad, with `unit = Try`.

Is it?

- ☐ Yes
- ☐ No, the associative law fails
- ☐ No, the left unit law fails
- ☐ No, the right unit law fails
- ☐ No, two or more monad laws fail.

Solution

It turns out the left unit law fails.

```
Try(expr).flatMap(f) != f(expr)
```

Indeed the left-hand side will never throw a non-fatal exception whereas the right-hand side will throw any exception thrown by `expr` or `f`.

Hence, `Try` trades one monad law for another law which is more useful in this context:

An expression composed from 'Try', 'map', 'flatMap' will never throw a non-fatal exception.

Call this the “bullet-proof” principle.

Conclusion

We have seen that for-expressions are useful not only for collections.

Many other types also define `map`, `flatMap`, and `withFilter` operations and with them for-expressions.

Examples: `Generator`, `Option`, `Try`.

Many of the types defining `flatMap` are monads.

(If they also define `withFilter`, they are called “monads with zero”).

The three monad laws give useful guidance in the design of library APIs.