Functional Programming in Scala

Chapter 4
Handling errors without exceptions

Jordan Moldow

Oct. 9, 2014

Handling errors without exceptions

- Pure functions are like mathematical functions: $f(x)$
  - Always returns the same single result
  - Produces no side effects in the outside world
- Throwing exceptions is a side effect, breaks referential transparency

Handling errors without exceptions

Key ideas:

- Use container type to expand codomain (range) of functions
- Return errors as values
- Use higher-order functions to
  - consolidate of error handling logic
  - preserve composability
  - "lift" normal functions to error handling functions

Handling errors without exceptions

The good and bad aspects of exceptions

Possible alternatives to exceptions

The `Option` data type

- Usage patterns for `Option` - the `Option` functor
- `Option` composition and lifting - the monad laws
- Wrapping exception-oriented APIs

The `Either` data type

Exercises

Throwing exceptions breaks referential transparency

```scala
def failingFn(i: Int): Int = {
  val y: Int = throw new Exception("fail!")
  try {
    val x = 42 + 5
    x + y
  }
  catch { case e: Exception => 43 }
}

scala> failingFn(12)
java.lang.Exception: fail!

def failingFn2(i: Int): Int = {
  try {
    val x = 42 + 5
    x + ((throw new Exception("fail!")): Int)
  }
  catch { case e: Exception => 43 }
}

scala> failingFn2(12)
res1: Int = 43
```

The bad aspects of exceptions

- Exceptions break the substitution model of reasoning
  - `throw new Exception("fail")` is context-dependent, taking on different meanings depending on which block it's in
- Exceptions can't be described in the type system
  - Does `f: Int => Int` always return? Might it fail? What exceptions might it throw? Who knows!
  - Java checked exceptions don't work with higher-order functions

The good aspects of exceptions

- Consolidate, centralize error-handling logic
- Error info (messages, stack traces, memory dumps)
- Exception subclasses
- Functions don't have to handle callee errors

Problem: Procedures aren't always total

- Total function: always has an output (like a mathematical function)
- Partial function: output undefined for some inputs
  - `mean: List[Double] => Double`
  - `sqrt: Double => Double`
  - (Not to be confused with partially applied functions)
- Pure functions must be total
- Need strategy for turning partial function into total function

Option 1 - Return bogus value in error case

- Return a sentinel value, or `NaN`, or `null`
- Can't attach extra information to errors
- Must manually check result at call sites / before uses of value
- No applicable in polymorphic code
- Requires special calling convention
- Not easy to compose
- Not easy to pass to higher-order functions

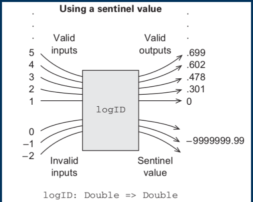
Option 2 - Return integer error codes

- Like assembly, C, Unix programs, etc.
- Not compatible with type system
- Plus all the bad things about Option 1
  - Especially bugs with not correctly error checking at call sites
  - `kill(fork())` bug - http://rachelbythebay.com/w/2014/08/19/fork/

Option 3 - Caller-provided default values

```scala
def mean(xs: IndexedSeq[Double], onEmpty: Double): Double =
  if (xs.isEmpty) onEmpty
  else xs.sum / xs.length
```

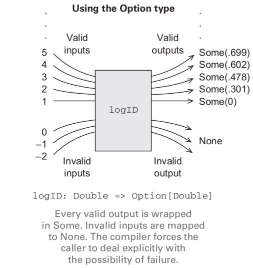
- Limited to passing / returning `Double`
- Parameter can only be used as a default value
- In error cases, can't branch or abort
- Immediate callers must decide default value



Mapping all invalid inputs to a special value of the same type as the valid outputs. Ambiguous, and compiler can't check that caller handles it correctly.

Option 4 - The `Option` data type

```scala
sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```




```scala
def mean(xs: Seq[Double]): Option[Double] =
  if (xs.isEmpty) None
  else Some(xs.sum / xs.length)
```

`mean` is now total

`None` is not `null`!

```scala
sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

- There is no such thing as a "generic" `None`
  - `None` $\notin$ `A`
  - Can't return `None` from function that returns an `A`
- For arbitrary polymorphic types `A`, `B`:
  - `None:Option[A]` $\neq$ `None:Option[B]`
  - `None:Option[A]` $\notin$ `Option[B]`
  - This isn't true if one type is a subclass of the other
- Type system prevents null pointer dereference
  - Because of use of case class, `op.get` won't compile unless `op` is statically guaranteed to be `Some(a)`
  - Unfortunately, this isn't true of Scala's builtin `Option`

`Option` as a container

```scala
sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

Think of `Option[A]` as a `List[A]` with length ≤ 1

- `None:Option[A]` $\approx$ `Nil:List[A]`
- `Some(a:A)` $\approx$ `List(a:A)`

Usage patterns for `Option`

```scala
sealed trait Option[+A] {
  // Apply f if the Option is not None.
  def map[B](f: A => B): Option[B]

  // The B >: A says that the B type parameter must be
  // a supertype of A.
  def getOrElse[B>:A](default: => B): B

  // Apply f, which may fail, to the Option if not None.
  def flatMap[B](f: A => Option[B]): Option[B]

  // `ob: => Option[B]` means don't evaluate ob unless needed.
  // The argument is non-strict / evaulated lazily
  // (just like if-else short-circuiting) - see chapter 5!
  def orElse[B>:A](ob: => Option[B]): Option[B]

  // Convert Some to None if the value doesn't satisfy f.
  def filter(f: A => Boolean): Option[A]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

Exercise 4.1

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B] = this match {
    case None => None
    case Some(a) => Some(f(a))
  }

  def getOrElse[B>:A](default: => B): B = this match {
    case None => default
    case Some(a) => a
  }

  def flatMap[B](f: A => Option[B]): Option[B] =
    map(f) getOrElse None

  def orElse[B>:A](ob: => Option[B]): Option[B] =
    map(Some(_)) getOrElse ob

  def filter(f: A => Boolean): Option[A] = {
    flatMap(a => if (f(a)) Some(a) else None)
  }
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

Usage scenarios for `Option`

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def getOrElse[B>:A](default: => B): B
  def flatMap[B](f: A => Option[B]): Option[B]
  def orElse[B>:A](ob: => Option[B]): Option[B]
  def filter(f: A => Boolean): Option[A]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

Usage patterns for `Option`

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def getOrElse[B>:A](default: => B): B
  def flatMap[B](f: A => Option[B]): Option[B]
  def orElse[B>:A](ob: => Option[B]): Option[B]
  def filter(f: A => Boolean): Option[A]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

1. Some initial computation `f: A => Option[B]` may fail
2. Apply further computations with `map`, `flatMap`
   - Subsequent computations only run when there is still a value
   - In error cases, `None` is carried through the computations
3. Optionally `filter` on predicates to generate error
4. Do error handling at end with `getOrElse` or `orElse`
   - `getOrElse` provides default value
   - `OrElse` provides new chain of computations to try

Language Comparison

Scala:

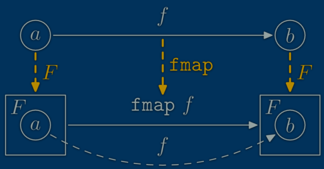
```
val dept: String =
  lookupByName("Joe").   // Impossible to forget None check.
  flatMap(_.dept).       // Type system does not allow you to.
  filter(_ != "Accounting").
  getOrElse("Default Dept")
```

Python:

```
employee = lookupByName("Joe")
# If you forget this line
if employee is not None:
  # this will raise AttributeError.
  department = employee.dept
  if (department is not None) and (department != "Accounting"):
    dept = department
```

The `Option` functor

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```



`g: A => B, f: B => C`

`f compose g = ((a:A) => f(g(a)))`

1. `map identity = identity`
2. `map (f compose g) = (map f) compose (map g)`

More in chapter 10!

`Option` composition - Exercise 4.3

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]


// Return None if any input is None.
// Otherwise, apply the function to the values.
def map2[A,B,C](a:Option[A],b:Option[B])(f:(A,B)=>C):Option[C]
def flatMap2[A,B,C](a:Option[A],b:Option[B])(f:(A,B)=>Option[C])
    :Option[C]
def map3[A,B,C,D](a:Option[A],b:Option[B],c:Option[C])
                  (f:(A,B,C)=>D):Option[D]
```

Option composition - Exercise 4.3

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]


// Return None if any input is None.
// Otherwise, apply the function to the values.
def map2[A,B,C](a:Option[A],b:Option[B])(f:(A,B)=>C):Option[C]
def flatMap2[A,B,C](a:Option[A],b:Option[B])(f:(A,B)=>Option[C])
    :Option[C]
def map3[A,B,C,D](a:Option[A],b:Option[B],c:Option[C])
                 (f:(A,B,C)=>D):Option[D]
```

```scala
map2(a,b)(f) = a flatMap (aa => b map (bb => f(aa, bb)))
flatMap2(a,b)(f) = a flatMap (aa => b flatMap (bb => f(aa, bb)))
map3(a,b,c)(f) = a flatMap (aa => b flatMap{bb =>
                        c map (cc => f(aa, bb, cc)}))
```

`Option` lifting

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]

def lift[A,B](f: A => B): Option[A] => Option[B] =
  _ map f
def flatLift[A,B](f: A => Option[B]): Option[A] => Option[B] =
  _ flatMap f
```

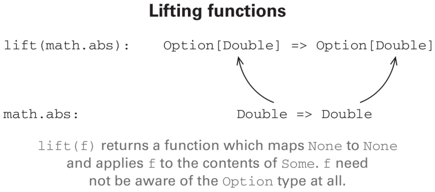
- Take ordinary functions, and lift them to functions on `Option`
- Can design API of `A => B`, `A => Option[B]` functions
- Don't need all your functions to accept `Option[A]`
- Can lift builtin Java, Scala functions

Option lifting

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]

def lift[A,B](f: A => B): Option[A] => Option[B] =
  _ map f
def flatLift[A,B](f: A => Option[B]): Option[A] => Option[B] =
  _ flatMap f

val absO: Option[Double] => Option[Double] = lift(math.abs)
```

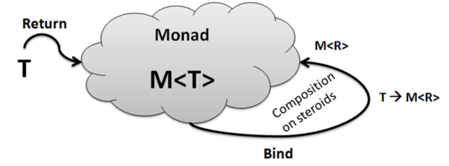


`lift(f)` returns a function which maps `None` to `None` and applies `f` to the contents of `Some`. `f` need not be aware of the `Option` type at all.

Option monad

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
  val bind = flatMap
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]

def unit[A](a: A): Option[A] =
  Some(a)
val return = unit
```



Option monad

```scala
sealed trait Option[+A] {
  def map[B](f: A => B): Option[B]
  def flatMap[B](f: A => Option[B]): Option[B]
  val bind = flatMap
}
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]

def unit[A](a: A): Option[A] =
  Some(a)
val return = unit
```

`aa: A, a: Option[A]`
`g: A => Option[B], f: B => Option[C]`
`f composeM g = ((aa:A) => g(aa) bind f)`

- `unit(aa) bind f == f(aa)`
- `a bind unit == a`
- `a bind (f composeM g) == (a bind f) bind g`

More in chapter 11!

Wrapping exception-oriented APIs

```scala
// We accept the A argument non-strictly,
// so we can catch any exceptions that
// occur while evaluating a and convert them to None.
def Try[A](a: => A): Option[A] =
  try Some(a)
  catch { case e: Exception => None }
```

The `Either` data type

```scala
sealed trait Either[+E, +A]
case class Left[+E](value: E) extends Either[E, Nothing]
case class Right[+A](value: A) extends Either[Nothing, A]

def mean(xs: IndexedSeq[Double]): Either[String, Double] =
  if (xs.isEmpty) Left("mean of empty list!")
  else Right(xs.sum / xs.length)

def Try[A](a: => A): Either[Exception, A] =
  try Right(a)
  catch { case e: Exception => Left(e) }

def safeDiv(x: Int, y: Int): Either[Exception, Int] =
  Try(x / y)
```

- Allows us to track error info
- *Disjoint union* of two types

The `Either` data type

```scala
sealed trait Either[+E, +A] {
  def map[B](f: A => B): Either[E, B]
  def flatMap[EE >: E, B](f: A => Either[EE, B]): Either[EE, B]
  def orElse[EE >: E,B >: A](b: => Either[EE, B]): Either[EE, B]
  val bind = flatMap
}
case class Left[+E](value: E) extends Either[E, Nothing]
case class Right[+A](value: A) extends Either[Nothing, A]

def unit[E, A](a: A): Either[E, A] =
  Right(a)
val return = unit
```

`Either[E, A]` is a functor / monad in its right type parameter

Exercises

- https://github.com/fpinscala/fpinscala/tree/master/exercises/src/main/scala/fpinscala/errorhandling
- https://github.com/fpinscala/fpinscala/tree/master/answerkey/errorhandling
- https://github.com/fpinscala/fpinscala/tree/master/answers/src/main/scala/fpinscala/errorhandling

Summary

Within type system, we can achieve the good aspects of exceptions:

- Consolidate, centralize error-handling logic
- Error info (messages, stack traces, memory dumps)
- Exception subclasses
- Functions don't have to handle callee errors

Key ideas:

- Use container type to expand codomain (range) of functions
- Return errors as values
- Use higher-order functions to
  - consolidate of error handling logic
  - preserve composability
  - "lift" normal functions to error handling functions

Conclusion

Questions?

Please post questions, exercise solutions, discussions on Jive

Next week: Chapter 5 - Strictness and laziness