“Functional Programming in Scala”
Chapter 4
Handling errors without exception(s)

Nathaniel Osgood

Important Topics

- Referential transparency (context-free expressions)
- Functional means of handling errors
- Problems with exceptions
- Problems with sentinel values/distinguished return codes
- Type-safe return values (via Option, Either, Try)

Problems with exceptions

- Lack of referential transparency
- High performance cost
- Difficulty in reasoning about control flow
- Lack of type safety
- Difficulty of composing into higher order functions
- Cruft/Bloat: Verbosity around handling

Problems with sentinel values/distinguished return codes

- Lake of (enforced) type safety (& enforced awareness)
- Difficulty in using in higher order functions
- Difficult to apply to type parameterized code
- Lack of canonical values
- Cruft/Bloat: Verbosity around handling

Option: Encoding Possible Absence of a Value

- Handling of possibly absent values that is
  - Type-safe
  - Uniform with other collection (e.g., can map, flatMap, filter)
- Related points
  - Has Some(a)/None option
  - Offers compose operations (e.g., via flatmap)
  - isEmpty/ifDefined
  - getOrElse: Get value or another value
  - orElse: Perform some (otherwise unevaluated) option only if empty
  - Filter
  - None is ignored by sum
- Common uses: Return value of calculation, output of filter, value from data structure
- Using flatmap, map define long “pipelines” of operations, which naturally handle and just pass through failure at any stage
- This is a key monad (a “contained” value that wraps values with metadata)

Option Examples 1:

- Try(scala.io.StdIn.readLine().toDouble).toOption
- Seq("2.13", "4.3", "", "as").flatMap(str => Try(str.toDouble).toOption).sum
- Seq.fill(1000)(nextInt(100)).zip(Seq.fill(1000)(nextInt(100))).flatMap(pair => Try(pair._1 / pair._2).toOption).sum
- v.filter(_.length > 2).map(_.length).reduceLeftOption((a,b) => a+b)

Using Options to detect problems, which are then filtered out

```
import scala.io._

val collURLs = Seq("http://www.usask.ca", "http://www.cnn.com", "http://www.shrf.ca", "http://www.fakefoobar.com")

val collPageOptionStrings = collURLs.map((str:String) => Try(Source.fromURL(str).mkString).toOption)

// total up the sizes

collPageOptionStrings.filter(_.isDefined).map(_.get).map(_.size).sum
```

Comprehensions via “for” loops

- For “comprehensions” consist of using a for loop to construct a collection
  - These are reminiscent of the comprehensions seen in Python
- Because for loops (like other Scala statements) have ***values***, can create such collections very readily, e.g.,

  values = for (i <- m to n; j <- p to q [if guardcondition]) yield expr
- For collections that include **Option** values, for values that are empty (i.e., that are None) nothing is performed for that value
- Note that the types of functionality possible with such comprehensions can sometimes be handled in with constructs such as map & filter
- **These play key role in making monadic operations transparent**, as they desugar to calls to flatMap, map, and withFilter
  - Key point: we can write the operations in the loop with respect to the unwrapped values
- These define partial function; only values over which this is defined are used
- Often we write looping factors in a { } block with separate lines (**; not needed**)

“for” Loops Basics

Can contain **multiple generators**

```
for (int i <- m to n; j <- p to q [if guardcondition]) { body }
```

Example Guarded condition:

```
for (int i <- m to n; j <- p to q if i != j) { body }
```

With destructuring (extraction):

```
for (PersonCaseClass(age,income,name) <- population)
    println(s"$name has income $income")
```

Intermediate variables: for (int i <- m to n; j <- p to q; prod = i*j; prod > 0) { body }

This is very commonly used with **range** expressions such as

m to n

m until n (goes up to n – 1)

m until (n, k) (goes up to n – 1, in steps of k)

For Expression [Comprehensions]/loop: Anatomy

- Generators, e.g.,
  - for (p <- population)
  - for (nbhd <- getNeighborhoods())
  - for (x <- expr1; y <- expr2)
  - for (x <- expr1; y <- expr2(x))

Key: These variables on the left hand side actually refer to The values *within* the containers to the right – not to the dontainers themselves. These are automatically extracted for us, without needing to write an enclosing function.

Unless constrained, this gives combinatorial semantics, like nested loops: Iterate over all pairs of contained values of x & y

- Embedded definitions of values
  - for (nbhd <- getNeighborhoods(); **meanIncome = nbhd.meanIncome**)
- Filters
  - for (nbhd <- getNeighborhoods() if nbhd.meanIncome **<= povertyLine**)
- For **for expressions**, these are followed by a yield expression e.g.,

  yield nbhd.population
- Such for expressions translate into pipelines of calls to
  - **for expressions**: flatMap (all but innermost)/map (innermost)/filterWith
  - **for loop ("statement"):** foreach/filterWith

Central Points

- For expressions allow us to focus on the code operating on the **elements of a container**
  - The operations on the container itself are taken care of in the background
- For expressions are especially suited for dealing with **Monads** (TBC)
- For expressions are compiled into (and are interchangeable with) calls to map, flatMap, filter(With)
  - To understand the constraints and error messages associated with for expressions, it is useful to understand how they are compiled into such calls

Desugaring for expressions [comprehensions] to pipelined operators

- for {
  x1 <- expr1
  x2 <- expr2
  x3 <- expr3
  …
  xFinalLHS <- exprFinalRHS

} yield exprYield **desugars into**

expr1.**flatMap**(x1 => expr2.**flatMap**(x2 => expr3.**flatMap**(x3 => … exprFinalRHS.**map**(xFinalLHS => exprYield)))))

- An "if" clause following an **exprN** above desugars into a corresponding "withFilter" call on that **exprN** (with condition in Boolean-returning function)
- for { x1 <- expr1 if cond(x1); x2 <- expr2 } yields exprYield => expr1 withFilter (x1 => cond(x1)).**flatMap**(x1 => expr2.**map**(x2 => exprYield)

Try the following…

```
import scala.io.StdIn;
for {
        v1 <- Try(readLine().toInt).toOption
        v2 <- Try(readLine().toInt).toOption
        v3 <- Try(readLine().toInt).toOption
        v4 <- Try(readLine().toInt).toOption
        z = v1+v2+v3+v4
} yield z
```

```
import scala.util.Random._
case class Person(age:Int, yearlyIncome:Option[Double], name: String);
// Create members of the population, recognizing that some have no income
val population = Vector.fill(1000)(Person(nextInt(100), if (nextBoolean) Some(nextDouble * 30000.0) else None, nextString(3)))
// case where abort computation if result is not available for any subpiece
for {
        ageLowerThreshold <- Try(scala.io.StdIn.readLine().toInt).toOption
        ageUpperThreshold <- Try(scala.io.StdIn.readLine().toInt).toOption
        totalIncome = population.filter(person => (person.age >= ageLowerThreshold && person.age <= ageUpperThreshold)).flatMap(_.yearlyIncome).sum
} yield totalIncome
```

Another example

```
def tryUserQuotient() =
{
   for {
   a <- Try(scala.io.StdIn.readLine().toInt).toOption
   b <- Try(scala.io.StdIn.readLine().toInt).toOption
   quotient <- Try(a/b).toOption
   } yield quotient
}
```

Note that this performs integer division for legal arguments and quotients

Example 1: Incrementally working an Option[T]

```
def findSmokingInPageAtURL(strURL: String) =
    for {
    url <- Some(strURL)
    strPageContents <- Try(Source.fromURL(url).mkString).toOption
    matchingString <- """[sS]moking""".r.findFirstIn(strPageContents)
    } yield matchingString

findSmokingInPageAtURL("https://en.wikipedia.org/wiki/Truth_Initiative")
```

Comparison

```
import scala.io._
val collURLs = Seq("http://www.usask.ca", "http://www.cnn.com", "http://www.shrf.ca", "http://www.fakefoobar.com")
```

Option 1:

```
val collPageOptionStrings = collURLs.map((str:String) => Try(Source.fromURL(str).mkString).toOption)
// to total up the sizes
collPageOptionStrings.filter(_.isDefined).map(_.get).map(_.size).sum
```

Option 2

```
(for (url <- collURLs; pageString <- Try(Source.fromURL(url).mkString).toOption) yield (pageString.size)).sum
```

Additional Topics

- Lifting
  - Bringing a function (e.g., A => B) into Option[A] => Option[B]
  - For comprehension as providing “lift” functionality with minimal cruft when dealing with “Option” (and other monads)
- map vs. flatMap
  - map: value in container
- Other monads for error handling
  - “Either” data type
  - “Try” data type

Some Ideas for Projects

- Within Spark, mining social media & analyzing with machine learning algs
  - Twitter
  - Facebook
  - Google search API
  - Examples: Influenza symptoms, vaccination safety/effectiveness, foodborne Illness, suicide references or occurrences
- Findings from cross linking sensor-data with browsed page contents
- Findings from cross linking sensor-data with EMA responses
  - e.g., probability of being answered, timing of answering
  - Behavioural correlates to answers
- Findings from cross linking sensor-data with weather & EMA responses
- Repurposing existing systems as transformers for Spark
  - AnyLogic
  - Code to support HMM analysis from “Scala for Machine Learning” (available via library)