CSci 555: Functional Programming
Functional Programming in Scala
Handling Errors without Exceptions

H. Conrad Cunningham

7 March 2019

Contents

4 Handling Errors without Exceptions **2**

- 4.1 Introduction . . . 2
- 4.2 Aside: On Null References . . . 2
- 4.3 An `Option` Algebraic Data Type . . . 3
  - 4.3.1 Method chaining in Scala . . . 3
  - 4.3.2 Type variance issues . . . 4
  - 4.3.3 Parameter-passing modes . . . 5
  - 4.3.4 Implementing the `Option` methods . . . 6
  - 4.3.5 Using `Option` for statistical `mean` and `variance` . . . 8
  - 4.3.6 Using `Option` in the labelled digraph . . . 9
  - 4.3.7 Lifting . . . 10
  - 4.3.8 For comprehensions . . . 11
  - 4.3.9 Translating (desugaring) for-comprehensions . . . 11
  - 4.3.10 Adding for-comprehensions to data types . . . 13
- 4.4 An `Either` Algebraic Data Type . . . 14
- 4.5 Standard Library . . . 15
- 4.6 Summary . . . 16
- 4.7 Source Code for Chapter . . . 16
- 4.8 Exercises . . . 16
- 4.9 Acknowledgements . . . 16
- 4.10 References . . . 17
- 4.11 Terms and Concepts . . . 17

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848

University, MS 38677
(662) 915-5358

Note: This set of notes accompanies my lectures on Chapter 4 of the book *Functional Programming in Scala* [Chiusano 2015] (i.e. the Red Book).

Prerequisites: In this set of notes, I assume the reader is familiar with the programming concepts and Scala features covered in my *Notes on Scala for Java Programmers* [Cunningham 2019a], *Recursion Styles, Correctness, and Efficiency (Scala Version)* [Cunningham 2019b], *Type System Concepts* [Cunningham 2019c], and *Functional Data Structures* [Cunningham 2019d] (i.e. Chapter 3 of the Red Book [Chuisano 2015]).

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of March 2019 is a recent version of Firefox from Mozilla.

4 Handling Errors without Exceptions

4.1 Introduction

A benefit of Java or Scala exception handling (i.e. using `try-catch` blocks) is that it consolidates error handling to well-defined places in the code.

However, code that throws exceptions typically exhibits two problems for functional programming.

1. It is not *referentially transparent*. It has side effects. Its meaning is thus dependent upon the context in which it is executed.
2. It is not *type safe*. Exceptions may cause effects that are of a different type than the return value of the function.

The key idea from Chapter 4 *is to use ordinary data values to represent failures and exceptions* in programs. This preserves referential transparency and type safety while also preserving the benefit of exception-handling mechanisms, that is, the consolidation of error-handling logic.

To do this, we introduce the `Option` and `Either` algebraic data types. These are standard types in the Scala library, but for pedagogical purposes Chapter 4 introduces its own definition that is similar to the standard one.

This set of notes also introduces Scala features that we have not previously discussed extensively: type variance, call-by-name parameter passing, and for-comprehensions.

4.2 Aside: On Null References

What should a function do when it is required to return an object but no suitable object exists?

One common approach is to return a *null reference*. British computing scientist Tony Hoare, who introduced the null reference into the Algol type system in the mid-1960s, calls that his "billion dollar mistake" because it "has led to innumerable errors, vulnerabilities, and system crashes" [Hoare 2009].

The software patterns community documented a safer approach to this situation by identifying the *Null Object* design pattern [Woolf 1998][Wikipedia 2019]. This well-known object-oriented pattern seeks to "encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior" [SourceMaking 2019]. That is, the object must be of the correct type. It must be possible to apply all operations on that type to the object. But the operations should have a neutral behavior, with no side effects. The null object should actively do nothing!

The functional programming community introduced *option types* (e.g. Haskell's Maybe and Either types) as an approach to the same general problem. Scala's Option and Either correspond to Haskell's Maybe and Either types. Rust's Option and Swift's Optional are similar to Haskell's Maybe.

The concept of *nullable types* [Wikipedia 2019], such as Java's Optional, Python's None, and C#'s ? type annotations, are closely related to option types.

4.3 An Option Algebraic Data Type

We define a Scala algebraic data type Option using sealed trait Option with case class Some to wrap a value and case object None to denote an empty value. We specify the operations on the data type using the *method-chaining* style.

```
import scala.{Option => _, Either => _, _} // hide standard
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]
```

The import statement above hides the standard definitions of Option and Either from the Scala standard library. The versions developed in this chapter are similar, but add useful features above what is available in the standard library.

Before we move on to the definition of these methods, let's review the concept of method chaining and then consider issues raised in the signatures of the getOrElse and orElse methods: one related to the generic parameters and the other to the default parameters.

4.3.1 Method chaining in Scala

Consider function map in the Option trait. It is implemented in this chapter as a method on the classes that extend the Option trait.

Method map takes two arguments and returns a result object. Its implicit "left" argument is the receiver object for the method call, an Option object represented internally by the variable this. Its explicit "right" argument is a function. The result returned is itself an Option object.

Suppose obj is an Option[A] object and f is an A => B function. In standard object-oriented style, we can issue the call obj.map(f) to operate on object obj with method `map` and argument `f`. Scala allows us to apply such a method in an *infix operator* style as follows:

```
obj map f
```

Note that `map` takes an `Option[A]` as its left operand (i.e. argument) and returns an `Option[B]` object as its result. Thus we can *chain* such method calls as follows (where functions `p` and `g` have the appropriate types):

```
obj.map(f).filter(p).map(g)
```

In Scala's infix operator style, this would be:

```
(obj map f) filter p)) map g
```

But these operators associate to the left, so can leave off the parentheses and write the above as follows

```
obj map f filter p map g
```

or perhaps more readably as:

```
obj map(f) filter(p) map(g)
```

Note that this last way of writing the chain suggests the *data flow* nature of this computation. The data originates with the source `obj`, which is then transformed by a `map(f)` operator, then by a `filter(p)` operator, and then by a `map(g)` operator to give the final result.

Now let's consider an issue raised in the signatures of the `getOrElse` and `orElse` methods related to the generic parameter.

4.3.2 Type variance issues

As we have discussed previously, the `+A` annotation in the `Option[+A]` definition declares that parameter `A` is *covariant*. That is, if `S` is a subtype of `T`, then `Option[S]` is a subtype of `Option[T]`. Also remember that `Nothing` is a subtype of all other types.

For example, suppose type `Beagle` is a subtype of type `Dog`, which, in turn, is a subtype of type `Mammal`. Then, because of the covariant definition, `Option[Beagle]` is a subtype of `Option[Dog]`, which is a subtype of `Option[Mammal]`. This is intuitively what we expect.

However, in `getOrElse` and `orElse`, we use type constraint `B >: A`. This means that `B` must be equal to `A` or a supertype of `A`. We also define value parameter of these functions to have type `Option[B]` instead of `Option[A]`.

Why must we have this constraint?

See the chapter notes for the *Functional Programming in Scala* book [Chiusano 2015] for more detail on this complicated issue. We sketch the argument below.

In some sense, the `+A` annotation declares that, in all contexts, it is safe to cast this type `A` to some supertype of `A`. The Scala compiler does not allow us to use this annotation unless we can cast all members of a type safely.

Suppose we declare `orElse` (incorrectly!) as follows:

```
trait Option[+A] {
    def orElse(o: Option[A]): Option[A]
    ...
}
```

We have a problem because this declaration of `orElse` only allows us to cast `A` to a subtype of `A`.

Why?

As with any function, `orElse` can only be safely passed a subtype of its declared input type. That is, a function of type `Dog => R` can be passed an object of subtype like `Beagle` or of type `Dog` itself, but it cannot be passed a supertype object of `Dog` such as `Mammal`.

As we saw in the notes on Chapter 3 [Cunningham 2019d], Scala functions are contravariant in their input types.

But `orElse` has a parameter type of `Option[A]`. Because of the covariance of `A` (declared in the trait), this parameter only allows subtypes of `A`—not supertypes as required by the covariance.

So, we have a contradiction.

For the incorrect signature of `orElse`, the Scala compiler generates an error message such as "Covariant type `A` occurs in contravariant position."

We can get around this error by using a more complicated signature that does not mention `A` (declared in the trait) in any of the function arguments, such as:

```
trait Option[+A] {
    def orElse[B >: A](o: Option[B]): Option[B]
    ...
}
```

Now consider the second new feature appearing in the signatures of `getOrElse` and `orElse`—the `=> B` and `=> Option[B]` types for the parameters.

4.3.3 Parameter-passing modes

Scala's primary parameter-passing mode is *call by value* as in Java and C. That is, the program evaluates the caller's argument and the resulting value is bound to the corresponding formal parameter within the called function.

If the argument's value is of a primitive type, then the value itself is passed. If the value is an object, then the address of (i.e. reference to) the object is passed.

A call-by-value parameter is called *strict* because the called function always requires that parameter's value before it can execute. The corresponding argument must be evaluated *eagerly* before transferring control to the called function.

Scala also has *call-by-name* parameter passing. Consider the `default: => B` feature in the declaration

```
def getOrElse[B >: A](default: => B): B
```

The type notation `=> B` means the calling program leaves the argument of type `B` unevaluated. That is, the calling program wraps the argument expression in an parameterless function and passes the function to the called method. This automatically generated parameterless function is sometimes called a *thunk*.

Every reference to the corresponding parameter causes the thunk to be evaluated. If the method does not access the corresponding parameter during some execution, then the parameter is never evaluated.

As with all higher-order arguments in Scala, a thunk is passed as a *closure*. In addition to the function, the closure captures any *free variables* occurring in the expression—that is, the variables defined in the caller's environment but not within the expression itself.

Note: The closure actually captures the variable itself, not its value. So if the free variable is either a reference to a `var` or to a mutable data structure, then changes in the value are seen inside the called function. But in this course, we normally use immutable data structures and `val` declarations.

A call-by-name parameter is called *nonstrict* because the called function does not always require that parameter's value for its execution. The corresponding argument can thus be evaluated *lazily*, that is, evaluated only if and when its value is needed.

We look at more implications of strict and nonstrict functions in Chapter 5 [Chiusano 2015].

Note: See `ClosureExample.scala` and `ThunkExample.scala` for examples of using closures and thunks, respectively.

4.3.4 Implementing the `Option` methods

Now, let's define the methods in the `Option` data type.

Above we defined the trait `Option` as follows. It is parameterized by covariant type `A`.

```
import scala.{Option => _, Either => _, _} // hide standard
sealed trait Option[+A] {
```

```scala
    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]
```

The `Option` data type is similar to a list that is either empty or has one element. As with the `List` algebraic data type from Chapter 3, we *follow the types to implementations*. (This is sometimes called *type-driven development*.) That is, we use the form of the data to guide us to design the form of the function.

The `map` method applies a function to its implicit `Option[A]` argument. If the implicit argument is a `Some`, the method applies the function to the wrapped value and returns the resulting `Some`. If it is a `None`, the method just returns `None`. We can implement `map` using pattern matching directly as follows:

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

Similarly, we can use pattern matching directly to implement the `getOrElse` function. If the implicit argument is of type `Some`, this function returns the value it wraps. If the implicit argument is of type `None`, this function returns the value denoted by the `default` argument. By passing `default` by name, the argument is only evaluated when its value is needed.

```scala
def getOrElse[B >: A](default: => B): B = this match {
    case None    => default // evaluate the thunk
    case Some(a) => a
}
```

Function `flatMap` applies its argument function `f`, which might fail, to its implicit `Option` argument when this value is not `None`. We can define `flatMap` in terms of `map` and `getOrElse` as shown below. (Reminder: If we apply method names as operators in an infix manner, they associate to the left. The leftmost operator implicitly operates on `this` object.)

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

We can also define `flatMap` using pattern matching directly.

```scala
def flatMap_1[B](f: A => Option[B]): Option[B] = this match {
   case None    => None
   case Some(a) => f(a)
}
```

Function `orElse` returns the implicit `Option` argument if is not `None`; otherwise, it returns the explicit `Option` argument. We can define `orElse` in terms of `map` and `getOrElse` or by directly using pattern matching.

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

def orElse_1[B>:A](ob: => Option[B]): Option[B] =
    this match {
        case None => ob
        case _    => this
    }
```

The `filter` function converts its implicit argument from `Some` to `None` if it does not satisfy the boolean function `p`. We can define `filter` by using pattern matching directly or by using `flatMap`.

```
def filter(p: A => Boolean): Option[A] =
    this match {
        case Some(a) if p(a) => this
        case _               => None
    }

def filter_1(p: A => Boolean): Option[A] =
    flatMap(a => if (p(a)) Some(a) else None)
```

4.3.5 Using `Option` for statistical `mean` and `variance`

Consider a function to calculate and return the *mean* (i.e. average value) of a list of numbers. This function must sum the list of numbers and divide by the number of elements. It might have a signature such as:

```
def mean(xs: List[Double]): Double
```

But what should be returned for empty lists?

We can modify the signature to use `Option` and define the function as follows:

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

The return type now allows the possibility that the mean may be undefined. We thus extend a partial function to a total function in a meaningful way.

Above we also generalize the `List` type to its supertype `Seq` from the Scala library. Type `Seq` denotes a family of sequential data types that includes the

`List` type, array-like collections, etc. This type defines the methods `isEmpty`, `sum`, and `length`.

If the mean of a sequence s is m, then the (statistical) *variance* of the sequence s is the mean of the sequence formed by the terms $(x - m)^2$ for each $x \in s$ (perserving the order).

Using the `mean` function defined above, we can compute the variance of a sequence of numbers as follows:

```
def variance(xs: Seq[Double]): Option[Double] =
    mean(xs) flatMap
        (m => mean(xs.map(x => math.pow(x - m, 2))))
```

4.3.6 Using `Option` in the labelled digraph

In the doubly labelled `Digraph` case study, we define the following method to return the label on a vertex:

```
def getVertexLabel(ov: A): B
```

In the case study's `DigraphList` implementation, we define this function as follows. The function terminates with an error when the the argument vertex is not present in the digraph.

```
def getVertexLabel(ov: A): B =
    (vs dropWhile (w => w._1 != ov)) match {
        case Nil =>
            sys.error("Vertex " + ov + " not in digraph")
        case ((_,l)::_) => l
    }
```

We can avoid the error termination in this function by changing the method signature to return an `Option[B]` instead of a `B`.

```
def getVertexLabelOption(ov: A): Option[B] =
    (vs dropWhile (w => w._1 != ov)) match {
        case Nil        => None
        case ((_,l)::_) => Some(l)
}
```

Code that uses this new `Digraph` method can call the various `Option` methods (or directly use pattern matching) to process the result appropriately.

For example, if the vertex label is a string, it may be appropriate in some scenarios to just use a null string for the label of a nonexistent vertex. Let `g` be a `Digraph` and be `v` be a possible vertex, then

```
(g getVertexLabelOption v) getOrElse ""
```

would either return the label string for `v` if it exists or the null string if `v` does not exist.

Similarly, the code

```
(g getVertexLabelOption v) getOrElse
    (sys.error("undefined vertex " + v))
```

would still terminate with an error. However, this design enables the user of the `Digraph` library to decide under what circumstances and at what point in the code to terminate.

Idiom: A common pattern for computing with the `Option` type is to use `map`, `flatMap`, and `filter` to transform values generated by a function like `getVertexLabelOption` and then to use `getOrElse` for error handling at the end.

4.3.7 Lifting

It seems that that deciding to use of `Option` could cause lots of changes to ripple through our code, much like the introduction of extensive exception-handling would. However, we can avoid that somewhat by using a technique called *lifting*.

For example, the `Option` type's `map` function enables us to transform values of type `Option[A]` into values of type `Option[B]` using a function of type `A => B`.

Alternatively, we could consider `map` as transforming a function of type `A => B` into a function of type `Option[A] => Option[B]`. That is, we *lift* an ordinary function into a function on `Option`.

We can formalize this alternative view with the following function:

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

Thus any existing function can be transformed to work within the context of a single `Option` value. For example, we can lift the square root function from type `Double` to work with `Option[Double]` as follows:

```
def sqrtO: Option[Double] => Option[Double] =
    lift(math.sqrt _)
```

We can now use `sqrtO` such as `sqrtO(Some(4))`. This evaluates to `Some(2)`.

Chapter 4 of *Functional Programming in Scala* [Chiusano 2015] gives several examples where `Option` types can be used effectively in realistic scenarios. One of these examples illustrates how to wrap an exception-oriented API to provide users with `Option` results in error situations.

TODO: Add example of wrapping an exception-oriented API.

Note: See `WrapException.scala` for examples of how to wrap exception-throwing functions to return `Option` and `Either` objects, respectively.

4.3.8 For comprehensions

Another useful function on `Option` data types is the function `map2` that combines two `Option` values by lifting a binary function. If either of the arguments are `None`, then the result should also be `None`. We can define `map2` as follows:

```
def map2[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C):
    Option[C] =
        a flatMap (aa =>
            b map (bb =>
                f(aa, bb)))
```

This function applies a series of `map` and `flatMap` calls.

Because lifting is so common in functional programming, Scala provides a syntactic construct called a *for-comprehension* to facilitate its use. This construct is really *syntactic sugar* for a series of applications of `map`, `flatMap`, and `withFilter`. (Method `withFilter` works like `filter` except it filters on demand, without creating a new collection as a result.)

Here is the same code expressed as a for-comprehension:

```
def map2fc[A,B,C](a: Option[A], b: Option[B])(f: (A, B) => C):
    Option[C] =
        for {
            aa <- a
            bb <- b
        } yield f(aa, bb)
```

Of course, for-comprehensions are more general than just their use with `Option`. They can be used for the lists, arrays, iterators, ranges, streams, and other types in the Scala standard library that support the `map`, `flatMap`, and `withFilter` (or `filter`) operations.

Consider a list `persons` of person objects with `name` and `age` fields. We can collect the names of all persons who are age 21 and above as follows:

```
for (p <- persons if p.age >= 21) yield p.name
```

This is equivalent to the following `List` expression

```
filter (p => p.age >= 21) map (p => p.name)
```

4.3.9 Translating (desugaring) for-comprehensions

In general, a for-comprehension

```
for (enums) yield e
```

evaluates expression `e` for each binding generated by the enumerator sequence `enums` and collects the results. An enumerator sequence begins with a generator and may be followed by additional generators, value definitions, and guards.

- A *generator* `p <- e` produces a sequence of zero or more bindings from expression `e` by matching each value against pattern `p`.
- A *value definition* `p = e` binds the names in pattern `p` to the result of evaluating the expression `e`.
- A *guard* `if e` restricts the bindings to those that satisfy the boolean expression `e`.

We can translate (or *desugar*) for-comprehension (more or less) as follows:

1. We replace every generator `p <- e`, where `p` is a pattern and `e` is an expression, by:

   ```
   p <- e.withFilter { case p => true ; case _ => false }
   ```

   Here we use `withFilter` to filter out those items that do not match the pattern `p`.

2. While all comprehension have not been eliminated, repeat the following:

   a. Translate for-comprehension

   ```
   for (p <- e) yield e1
   ```

   to the expression:

   ```
   e.map { case p => e1 }
   ```

   b. Translate for-comprehension

   ```
   for (p <- e; p1 <- e1; ... ) yield e2
   ```

   where `...` is a (possibly empty) sequence of generators, definitions, or guards, to the expression:

   ```
   e.flatMap { case p => for (p1 <- e1; ... ) yield e2 }
   ```

   c. Translate a generator `p <- e` followed by a guard `if g` to a single generator

   ```
   p <- e.withFilter((x1,...,xn) => g)
   ```

   where `x,...,xn` are the free variables of `p`.

   d. Translate a generator `p <- e` followed by a value definition `p1 = e1` to the following generator of pairs of values, where `x` and `x1` are fresh names:

   ```
   (p, p1) <- for (x@p <- e) yield
       { val x1@p1 = e1; (x, x1) }
   ```

The Scala notation `x@p` means that name `x` is bound to the value of the expression `p`.

Note: Above we do not consider the imperative for-loops. These can also be translated as above except that the imperative method `forEach` is also needed.

As an example, we can translate (desugar) the for-comprehension

```
for(x <- e1; y <- e2; z <- e3) yield {...}
```

into the expression:

```
e1.flatMap(x => e2.flatMap(y => e3.map(z => {...})))
```

As a second example, we can also translate the for-comprehension

```
for(x <- e; if p) yield {...}
```

into the expression:

```
e.withFilter(x => p).map(x => {...})
```

If no `withFilter` method is available, we can instead use:

```
e.filter(x => p).map(x => {...})
```

As a third example, we can translate for-comprehension

```
for(x <- e1; y = e2) yield {...}
```

into the expression:

```
e1.map(x => (x, e2)).map((x,y) => {...})
```

4.3.10 Adding for-comprehensions to data types

The Scala language has no typing rules for the for-comprehensions themselves. The Scala compiler first translates for-comprehensions into calls on the various method and then checks the types. It does not require that methods `map`, `flatMap`, and `withFilter` have particular type signatures. However, a particular setup for some collection type `C` with elements of type `A` is the following:

```
trait C[A] {
    def map[B](f: A => B): C[B]
    def flatMap[B](f: A => C[B]): C[B]
    def withFilter(p: A => Boolean): C[A]
}
```

We can define our own data types to support for-comprehension by providing one or more of the required operations above.

- If the data type defines just `map`, Scala allows for-comprehensions consisting of a single generator.

- If the data type defines both `flatMap` and `map`, Scala allows for-comprehensions consisting of several generators.
- If the data type defines `withFilter`, Scala allows for-comprehensions with guards. (If `withFilter` is not defined but `filter` is, Scala will currently use `filter` instead. However, this gives a deprecation warning, so this fallback feature may be eliminated in a future release of Scala.)

We added for-comprehensions to our own `Option` type earlier. We do the same for the `Either` type in the next section.

Note: A for-comprehension is, in general, convenient syntactic sugar for expressing compositions of monadic operators. If time allows, we will discuss monads later in the semester.

4.4 An `Either` Algebraic Data Type

We can use data type `Option` to encode that a failure or exception has occurred. However, it does not give any information about what went wrong.

We can encode this additional information using the algebraic data type `Either`.

```scala
import scala.{Option => _, Either => _, _} // hide builtin
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, AA >: A](b: => Either[EE,AA]):
        Either[EE,AA]
    def map2[EE >: E, B, C](b: Either[EE, B])(f: (A,B) => C):
        Either[EE, C]
}
case class Left[+E](get: E) extends Either[E,Nothing]
case class Right[+A](get: A) extends Either[Nothing,A]
```

By convention, we use the constructor `Right` to denote success and constructor `Left` to denote failure.

We can implement `map`, `flatMap`, and `orElse` directly using pattern matching on the `Either` type.

```scala
def map[B](f: A => B): Either[E, B] =
    this match {
        case Left(e)  => Left(e)
        case Right(a) => Right(f(a))
    }

def flatMap[EE >: E, B](f: A => Either[EE, B]):
    Either[EE, B] = this match {
```

```
        case Left(e)  => Left(e)
        case Right(a) => f(a)
    }

def orElse[EE >: E, AA >: A](b: => Either[EE, AA]):
    Either[EE, AA] = this match {
        case Left(_)  => b
        case Right(a) => Right(a)
    }
```

The availability of `flatMap` and `map` enable us to use for-comprehension generators with `Either`. We can thus implement `map2` using a for-comprehension, as follows:

```
def map2[EE >: E, B, C](b: Either[EE, B])(f: (A, B) => C):
    Either[EE, C] =
        for { a <- this; b1 <- b } yield f(a,b1)
```

Let's again use `mean` as an example and use a `String` to describe the failure in `Left`.

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

We can use the `Left` value to encode more information, such as the location of the error in the program. For example, we might catch and return the value of an `Exception` generated as we do in the `safeDiv` function below.

```
def safeDiv(x: Int, y: Int): Either[Exception, Int] =
    try Right(x / y)
        catch { case e: Exception => Left(e) }
```

We can abstract the computational pattern in the `safeDiv` function as function `Try` defined below:

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

Chapter 4 of *Functional Programming in Scala* [Chiusano 2015] describes other functions for type `Either`.

4.5 Standard Library

Both `Option` and `Either` appear in the standard library.

The standard library `Option` type is similar to the one developed here, but the library version is missing some of the extended functions described in Chapter 4 [Chiusano 2015].

The standard library `Either` type is similar but more complicated, using projections on the left and right. It is also missing some of the extended functions from Chapter 4 [Chiusano 2015].

Study the Scala API documentation for more information on these data types.

4.6 Summary

The big idea in this chapter is to use ordinary values to represent exceptions and use higher-order functions for handling and propagating errors. As examples, we considered the algebraic data types `Option` and `Either` and functions such as `map`, `flatMap`, `filter`, and `orElse` to process their values.

We will use this general technique of using values to represent effects in the subsequent studies in this course.

We introduced the idea of nonstrict functions. We examine the implications and use of these more in Chapter 5.

4.7 Source Code for Chapter

- `Option2.scala`
- `Either2.scala`

4.8 Exercises

TODO: Add

4.9 Acknowledgements

In Spring 2016, I wrote this set of notes to accompany my lectures on Chapter 4 of the book *Functional Programming in Scala* [Chiusano 2015] (i.e. the Red Book). I constructed the notes around the ideas, general structure, and Scala examples from that chapter and its associated materials.

I also patterned some of the discussion of for-comprehensions on Chapter 10 of the document *Scala by Example* by Martin Odersky [Odersky 2014], on Chapter 23 of 2nd Edition of the book *Programming in Scala* [Odersky 2016], on the relevant parts of the Scala language specification, and on a relevant FAQ in the Scala documentation.

In 2018 and 2019, I updated the format of the document to be more compatible with my evolving document structures and corrected a few errors. In 2019, I also added the sections on null references and method chaining.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

4.10 References

[**Chiusano 2015**]**:** Paul Chiusano and Runar Bjarnason. *Functional Programming in Scala*, Manning, 2015. This book is sometimes called the Red Book. The chapter notes for this book are available on GitHub at https://github.com/fpinscala/fpinscala/wiki.

[**Cunningham 2019a**]**:** H. Conrad Cunningham. *Notes on Scala for Java Programmers*, 2019.

[**Cunningham 2019b**]**:** H. Conrad Cunningham. *Recursion Styles, Correctness, and Efficiency (Scala Version)*, 2019.

[**Cunningham 2019c**]**:** H. Conrad Cunningham. *Type System Concepts*, 2019.

[**Cunningham 2019d**]**:** H. Conrad Cunningham. *Functional Data Structures*, 2019.

[**Hoare 2009**]**:** Tony Hoare. Null References: The Billion Dollar Mistake, *InfoQ.com*, August 25, 2009, Available at https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare, referenced 27 February 2019.

[**Odersky 2014**]**:** Martin Odersky. *Scala by Example*, EPFL, 2014. Available at https://www.scala-lang.org/docu/files/ScalaByExample.pdf.

[**Odersky 2016**]**:** Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*, 3rd Edition, Artima Inc, 2016; 1st Edition, 2007; 2nd Edition, 2010.

[**SourceMaking 2019**]**:** SourceMaking. Null Object Design Pattern, *Design Patterns*, Available at https://sourcemaking.com/design/_patterns/null/_object, referenced 27 February 2019.

[**Wikipedia 2019**]**:** Wikipedia. Articles on Null Object Pattern Nullable Type](https://en.wikipedia.org/wiki/Nullable/_type), accessed 28 February 2019.

[**Woolf 1998**]**:** Bobby Woolf. Null Object, Chapter 1, In, Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Addison Wesley Longman, 1998.

4.11 Terms and Concepts

Big idea: Using ordinary data values to represent failures and exceptions in programs. This preserves referential transparency and type safety while also preserving the benefit of exception-handling mechanisms, that is, the consolidation of error-handling logic.

Concepts: Error handling, exceptions referential transparency, type safety, null reference, Null Object design pattern, option types, nullabile types, `Option` and `Either` algebraic data types in Scala, method chaining, type variance, covariant and contravariant, parameter passing (by-value, by-name), thunk, free variables, closure, strict and nonstrict parameters/functions, eager and lazy evaluation, follow the types to implementation (type-driven development), lifting, for-comprehensions, syntactic sugar, (generators, definitions, guards), desugaring.