Propagating failure with the `Option` type

Mark Armstrong

November 4, 2020

Contents

1 Introduction

These notes were created during and after the tutorial on November 2nd.

2 Motivation

We have used the `Option` type previously to represent some sort of *failure*, but you have likely found it tedious to work with the results of functions which return `Option`'s.

Today, we investigate this tediousness, and offer a way to avoid it, using the idea of a method which propagates failure.

3 Using `Option` to represent failure

The `Option` type in Scala, called `Maybe` type in some other languages, lets us represent *in the type system* the possibility of failure for a computation. (See the aside below for a discussion of other types for representing failure in Scala.)

For instance, consider division. Usually, division by 0 results in an exception, which breaks control flow, and unless caught, results in a crash. Such breaks in control flow make programs harder to reason about; they are essentially unconditional jump statements.

We can write a safe division method using `Option`:

```
def safediv(x: Int, y: Int): Option[Int] = {
  if (y == 0)
    None
  else
    Some(x / y)
}
```

The use of `safediv` requires the user to unwrap the result. The returned value is an `Option[Int]`, not an `Int`, and cannot be used as if it were just an `Int`.

This is by design; we want to force the user to handle the possibility of failure. But notice that to use `safediv` twice in a row, we have to unwrap the result of the first call before making the second call, and if it's `None`, we just propagate the error and just return `None`.

```
def half_of_third(x: Int): Option[Int] =
  // First, try to take a third.
  safediv(x,3) match {
    // If that succeeds, try to halve the result.
    case Some(y) => safediv(y,2)
    // Otherwise, propagate the failure.
    case None => None
  }
```

This is a lot more to write than the pseudocode

```
safediv(safediv(x,3),2)
```

we might have in mind.

And if we wanted to compose this safe division together more than two times, the situation gets even worse.

```
def half_of_third_of_quarter(x: Int): Option[Int] =
  // Try to quarter the argument.
  safediv(x,4) match {
    // If that succeeds, try to take a third of the result.
    case Some(y) => safediv(y,3) match {
      // If that succeeds, halve the result.
      case Some(z) => safediv(z,2)
      // Otherwise, if taking a third failed, propagate the
          failure.
      case None => None
    }
    // Otherwise, if taking a quarter failed, propagate the
        failure.
    case None => None
  }
```

this is **much** worse to write than the already lengthy

```
safediv(safediv(safediv(x,4),3),2)
```

It would be nice if at each step, we could just assume that the operation was successful, and only write the code for that case (since the code to handle failure is routine; just propogate the `None`.) We cannot do this directly, but we can write a method which does the tedious part for us!

The idea is that this method with take an `Option` valuew and a *function* that specifies what to do in the case of success. The method has builtin what to do with failure, so we only need to give it the code to handle

So, this method we want has the type

```
Option[A] => (A => Option[B]) => Option[B]
```

And here it is (with a bit of a lengthy name.)

```
def propagateFailure[A,B](ma: Option[A],
                          f: A => Option[B]): Option[B] =
  ma match {
    case Some(a) => f(a)
    case None => None
  }
```

Now we can write the two methods we had before, which had long, multi-line definitions, in a fairly succicent way.

```
def half_of_third(x: Int): Option[Int] =
  propagateFailure(safediv(x,3), y => safediv(y,2))

def half_of_third_of_quarter(x: Int): Option[Int] =
  propagateFailure(propagateFailure(safediv(x,4),
                                    (y: Int) => safediv(y,3)),
                   (z: Int) => safediv(z,2))
```

4 Propagating the empty list

We can carry out a similar exercise on `List`'s.

In this case, since the return value must be a list, we concatenate the results in the case of the non-empty list.

```
def propagateEmpty[A,B](xs: List[A],
                        f: A => List[B]): List[B] =
  xs match {
    case y :: ys => f(y) ++ propagateEmpty(ys,f)
    case Nil => Nil
  }
```

In the tutorial, I could only think of this contrived example (even more contrived than the previous ones!) to use this.

```
// Bunny invasion example from the Haskell wiki
// Replicate each element in a list 3 times.

// First, we give a method to repeat a value 3 times in a
    list.
def replicate[A](x: A): List[A] =
  List(x,x,x)

def bunnyInvasion[A](xs: List[A]): List[A] =
  propagateEmpty(xs, (y: A) => replicate(y))
```

5 Improving the syntax

We can vastly improve the user-friendliness of our methods above by using a better name. In fact, we should use an *operator*; a name consisting of symbols.

A common name for what we have been calling propagation is `>>=`, which is read/pronounced as *bind*. This operator evokes the idea of piping successful computations into the next expression. (So far as I know, in order for this operator to be written using infix notation, it must be wrapped in a `class`; we use an implicit class, essentially as a wrapper to `Option`. If anyone knows how to avoid this, please let me know.)

```
implicit class OptionBind[A](private val ma: Option[A])
    extends AnyVal {
  def >>=[B](f: A => Option[B]): Option[B] = ma match {
    case Some(a) => f(a)
    case None => None
  }
}
```

Now our code becomes even more readable. Notice how this syntax emphasizes the sequential nature of applying operations; the `>>=` is almost like a semicolon.

```
def half_of_third(x: Int): Option[Int] =
  safediv(x,3) >>= (y => safediv(y,2))

def half_of_third_of_quarter(x: Int): Option[Int] =
  safediv(x,4) >>=
    ((y: Int) => safediv(y,3) >>=
      ((z: Int) => safediv(z,2)))
```

6 It's a monad!

:TODO:

7 Aside Other means to represent failure

There are at least two types other than `Option` which are commonly used to represent failure in Scala;

- the `Either` type and
- the `Try` type.

7.1 The `Either` type

We have used `Either` before; it is an instance of a sum or disjoint union type, which can represent two possible types.

Compared to the `Option[A]` type, which has constructors

- `Some`, which carries an `A` value, and
- `None`, which does not carry a value,

the `Either[A,B]` type has constructors

- `Left`, which carries an `A` value, and
- `Right`, which carries a `B` value.

So `Either` can be used to represent a possible failure, by deciding one of the two constructors represents failure and the other success. But compared to the `Option` type, failure values using an `Either` can carry some value, probably providing some information about the failure, such as a string that could be printed to the user, or some numerical or enum value representing an error type, such as a HTTP status code (e.g., a `404` standing for file not found.)

By convention, when the `Either` type is used to represent possible failure, the `Left` case is used for failure, and the `Right` case is used for success (since it is right, as in correct.)

7.2 The `Try` type

The `Try` type is somewhat in between an `Option` and an `Either` type.

Like an `Either`, the `Try` type's two constructors, `Success` and `Failure`, each carry a value.

However, unlike the two constructors of an `Either`, the `Failure` constructor of a `Try` can *only* carry a `Throwable` value; that is, an exception value.

The `Try` constructor then takes an expression that may result in an exception; if it does, that exception value is returned wrapped in a `Failure`. Otherwise, it returns the resulting value wrapped in a `Success`.

```
import scala.util.{Try, Success, Failure}

def safediv(x: Int, y: Int): Try[Int] = Try(x/y)
```

```
safediv(4,0)   // Returns
    Failure(java.lang.ArithmeticException: / by zero)
safediv(4,2)   // Returns Success(2)
```

7.3 Which to use? `Option`, `Either` or `Try`?

Compared to the use of the `Option` type, the `Try` type does take a bit less work; we don't have to specifically check for the cases that will cause an exception.

However, the `Try` type does not allow us to represent failures that are not rooted in a possible exception. An `Option` can be used to represent the lack of a result, regardless of the reason for that lack of result.

And of course, since it can carry any two types in its two constructors, the `Either` type offers the most flexibility of all. In fact, it could be used to represent both the `Option[A]` type (`Either[Unit,A]`) and the `Try[A]` type (`Either[Throwable,A]`.)