ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Monads

Principles of Reactive Programming

Martin Odersky

Monads

Data structures with `map` and `flatMap` seem to be quite common.

In fact there's a name that describes this class of a data structures together with some algebraic laws that they should have.

They are called *monads*.

What is a Monad?

A monad `M` is a parametric type M[T] with two operations, `flatMap` and `unit`, that have to satisfy some laws.

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

In the literature, `flatMap` is more commonly called `bind`.

Examples of Monads

- `List` is a monad with `unit(x) = List(x)`
- `Set` is monad with `unit(x) = Set(x)`
- `Option` is a monad with `unit(x) = Some(x)`
- `Generator` is a monad with `unit(x) = single(x)`

`flatMap` is an operation on each of these types, whereas `unit` in Scala is different for each monad.

Monads and map

`map` can be defined for every monad as a combination of `flatMap` and `unit`:

```
m map f  ==  m flatMap (x => unit(f(x)))
         ==  m flatMap (f andThen unit)
```

Monad Laws

To qualify as a monad, a type has to satisfy three laws:

Associativity:

```
m flatMap f flatMap g  ==  m flatMap (x => f(x) flatMap g)
```

Left unit

```
unit(x) flatMap f  ==  f(x)
```

Right unit

```
m flatMap unit  ==  m
```

Checking Monad Laws

Let's check the monad laws for Option.

Here's `flatMap` for `Option`:

```
abstract class Option[+T] {

  def flatMap[U](f: T => Option[U]): Option[U] = this match {
    case Some(x) => f(x)
    case None => None
  }
}
```

Checking the Left Unit Law

Need to show: `Some(x) flatMap f == f(x)`

```
Some(x) flatMap f
```

Checking the Left Unit Law

Need to show: `Some(x) flatMap f == f(x)`

```
       Some(x) flatMap f

==     Some(x) match {
         case Some(x) => f(x)
         case None => None
       }
```

Checking the Left Unit Law

Need to show: `Some(x) flatMap f == f(x)`

```
      Some(x) flatMap f

==    Some(x) match {
        case Some(x) => f(x)
        case None => None
      }

==    f(x)
```

Checking the Right Unit Law

Need to show: `opt flatMap Some == opt`

```
opt flatMap Some
```

Checking the Right Unit Law

Need to show: `opt flatMap Some == opt`

```
      opt flatMap Some

==    opt match {
        case Some(x) => Some(x)
        case None => None
      }
```

Checking the Right Unit Law

Need to show: `opt flatMap Some == opt`

```
        opt flatMap Some

==      opt match {
          case Some(x) => Some(x)
          case None => None
        }

==      opt
```

Checking the Associative Law

Need to show: `opt flatMap f flatMap g == opt flatMap (x => f(x) flatMap g)`

```
opt flatMap f flatMap g
```

Checking the Associative Law

Need to show: `opt flatMap f flatMap g == opt flatMap (x => f(x) flatMap g)`

```
      opt flatMap f flatMap g

==    opt match { case Some(x) => f(x) case None => None }
          match { case Some(y) => g(y) case None => None }
```

Checking the Associative Law

Need to show: `opt flatMap f flatMap g == opt flatMap (x => f(x) flatMap g)`

```
        opt flatMap f flatMap g

  ==    opt match { case Some(x) => f(x) case None => None }
            match { case Some(y) => g(y) case None => None }

  ==    opt match {
          case Some(x) =>
            f(x) match { case Some(y) => g(y) case None => None }
          case None =>
            None match { case Some(y) => g(y) case None => None }
        }
```

Checking the Associative Law (2)

```
==    opt match {
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None
      }
```

Checking the Associative Law (2)

```
==    opt match {
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None
      }

==    opt match {
        case Some(x) => f(x) flatMap g
        case None => None
      }
```

Checking the Associative Law (2)

```
==    opt match {
        case Some(x) =>
          f(x) match { case Some(y) => g(y) case None => None }
        case None => None
      }

==    opt match {
        case Some(x) => f(x) flatMap g
        case None => None
      }

==    opt flatMap (x => f(x) flatMap g)
```

Significance of the Laws for For-Expressions

We have seen that monad-typed expressions are typically written as `for` expressions.

What is the significance of the laws with respect to this?

1. Associativity says essentially that one can "inline" nested for expressions:

```
      for (y <- for (x <- m; y <- f(x)) yield y
           z <- g(y)) yield z

==    for (x <- m;
           y <- f(x)
           z <- g(y)) yield z
```

Significance of the Laws for For-Expressions

2. Right unit says:

```
     for (x <- m) yield x

==   m
```

3. Left unit does not have an analogue for for-expressions.

Another type: Try

In the later parts of this course we will need a type named `Try`.

`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]
```

`Try` is used to pass results of computations that can fail with an exception between threads and computers.

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`:

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

Composing `Try`

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

```
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

```
abstract class Try[T] {
  def flatMap[U](f: T => Try[U]): Try[U] = this 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] = this 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?

```
O      Yes
O      No, the associative law fails
O      No, the left unit law fails
O      No, the right unit law fails
O      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 raise a non-fatal exception whereas the right-hand side will raise 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.