Monads

Principles of Functional 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.

```
extension [T, U](m: M[T])
  def flatMap(f: T => M[U]): M[U]

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

In the literature, `flatMap` is also called `bind`. It can be an extension method, or be defined as a regular method in the monad class `M`.

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)`

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)
```

Note: `andThen` is defined function composition in the standard library.

```
extension [A, B, C](f: A => B)
  @infix def andThen(g: B => C): A => C =
    x => g(f(x))
```

Monad Laws

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

Associativity:

```
m.flatMap(f).flatMap(g)  ==  m.flatMap(f(_).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`:

```
extension [T](xo: Option[+T])
  def flatMap[U](f: T => Option[U]): Option[U] = xo 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(f(_).flatMap(g))`

```
opt.flatMap(f).flatMap(g)
```

Checking the Associative Law

Need to show: `opt.flatMap(f).flatMap(g)  ==`
`opt.flatMap(f(_).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(f(_).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))
```

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))

==    opt.flatMap(f(_).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.