"Functional Programming in Scala"
Chapter 6
Functional Management of State

Nathaniel Osgood

Important Topics

- Returning state representations rather than mutating them in place
- Central abstraction as a mapping: State transition (action, transformation)
- Chaining computations
  - Passing results from previous to next
  - flatMap as
    - Sequencer
    - Cleaning up syntax: Eliminating obvious passing via flatMap
  - Use of for comprehensions to hide calls to flatMap, map & filter(With)
- Compatibility of functional perspective and imperative semantics

Central Role of Flatmap (Bind)

- It is easy to fall prey to the misunderstanding that *flatMap* (bind) is…
  - A hack to deal with the uncomfortable nesting of data structures that sometimes comes about from map
  - Just another slightly more powerful tweak to *map*
  - Just something that allows us to “short circuit” a computation by returning error codes from failing calculations
- In fact, flatMap (bind) is the key sequencing operation that opens up a huge potential functionality using monads
  - Allows us to compose successive actions in a general way
  - Supports easy definition of other methods
  - Permits easy custom sequencing

A New Look at flatmap

flatMap itself performs the sequencing of M[A] and $A \rightarrow M[B]$

When we invoke flatMap on a monad in object-oriented style (m.flatMap(f) as in Scala), this is implicit

- flatMap/“bind”: $M[A] \times (A \rightarrow M[B]) \rightarrow M[B]$

This argument to flatMap indicates “what to do next” (using the value of type A coming out of the current monad (here, Rand[A])

This return value Is a combination of (when monad is a fn, composing) M[A]&f

Example of Sequencing of Computations

```
Try(scala.io.StdIn.readLine().toInt).flatMap(a =>
    Try(scala.io.StdIn.readLine().toInt).flatMap(b =>
        Try(a/b).map(quotient => quotient)))
```

Example Using Flatmap Behind the Scenes

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

Additional Points on Flatmap

- Exactly how the monads M[A] and f are “combined” to yield the result M[B] varies from monad to monad
  - Rand[A] and many other “action” monads: composition
  - Option[A]: m.flatMap(f) only calls f if m is some value; if m is None, f is not even called!
  - Try[A]: m.flatMap(f) only calls f if m has succeeded; if m indicates failure, f is not even called!
- The semantics of flatMap are constrained by the need to adhere to monad laws

Mapping Abstractions: "Rand"/"State": Examples from "Functional Programming in Scala"

- type Rand[T] = (RNG => (T, RNG)) // really a "computation using randomness"
- def unit[T](v:T) = rng:RNG => (v, rng)
- def int: Rand[Int] = rng:RNG => rng.nextInt
- def nonNegativeInt: Rand[Int] = rng:RNG => { val (vInt, rng2) = rng.nextInt;

  if (vInt == Int.MinValue) nonNegativeInt(rng2) else abs(vInt) }
- def map[T,U](action: Rand[T])(f: T => U): Rand[U] = (rng1:RNG) => { val (v, rng2) = action(rng1); (f(v), rng2) }
- def flatMap[T,U](action: Rand[T])(nextAction: T => Rand[U]): Rand[U] = (rng1:RNG) => { val (v2, rng2) = action(rng1); nextAction(v2)(rng2) }
- nonNegativeEven: Rand[Int] = map(nonNegativeInt)(k => k – (k % 2))
- nonNegativeLessThan(n: Int): Rand[Int] = flatMap(nonNegativeInt)(i => val mod = i % n; if ((i + n-1) – mod >= 0) (rng => (mod, rng)) else NonNegativeLessThan(n))

flatMap serves to **compose** the two actions given to it

"what comes next"

Rand[A] here represents an **Action**

- In FPiS, Rand[A] itself represents a transition / action / computation that
  - Takes in a RNG state
  - Returns a pair of a value and an RNG state
- The result of r.flatMap(f) represents the *composition* (sequencing) of the actions represented in r (Rand[Int]) and that resulting from applying f (itself of type Int => Rand[Int])
  - This action resulting from flatMap represents the combination of considering both actions (most often, by *sequencing* those operations)

FlatMap as “chaining together” behind the scenes
Part 1: Traditional Code Combinator “Pipeline”

Code from Chiusano & Bjarnason, “Functional Programming in Scala” page 89

```
val ns: Rand[List[Int]] =
    int.flatMap(x =>
        int.flatMap(y =>
            ints(x).map(xs =>
                xs.map(_ % y))))
```

Returns a Rand[List[Int]]

These are both Rand[Int]

Returns a List[Int]

Returns a Rand[List[Int]]

Returns a List[Int]

Returns an Int

FlatMap as “chaining together” behind the scenes
Part 2: For Comprehension

Code from Chiusano & Bjarnason, “Functional Programming in Scala” page 89

```
val ns: Rand[List[Int]] = for {
    x <- int          flatMap over this
    y <- int          flatMap over this
    xs <- ints(x)     map over this
    } yield xs.map(_ % y)
```

Design Space Option 1: Lowest Level

```
def unit[T](v:T) = rng:RNG => (v, rng)
def int(rng:RNG): (Int, RNG) = rng.nextInt
// There is no "map" or "flatMap" because there is no abstraction/structure holding values
```

Pros:

Conceptually straightforward

Cons:

It is awkward to:

"Chain": to pass the rng along to later exprs

string together expressions that operate on the values returned in nice ways

Use

```
val (v1, rng1) = int(rngInitial)
val (v2, rng2) = int(rng1)
val (v3, rng3) = int(rng2)

(v1+v2+v3, rng3)
```

Example: nextInt(rngInitial).map((v1, rng1) => (v1*v1, p._2))
.map((v1, rng1) => { (v2,rng2) = nextInt(rng1); (v1+v2, rng2)})

The above lead to much "cruft", impeding transparent understanding of the key things taking place

Design Space Option 2: Object Oriented Approach

case class ValueWithRNGState(val value: Int, val rngState: RNG)

int(): ValueWithRNGState

map(f:Int =>Int) : ValueWithRNGState

flatMap(f:Int => ValueWithRNGState) : ValueWithRNGState // because f is purely a function of double this CANNOT be used for actions depending on state

// other functions "lifted" here b/c flatMap is not an option

Use:

val vrs1 = x.int()

... // use vrs.state

val vrs2 = vrs1.int()

...

val vrs3 = vrs2.int()

ValueWithRNGState(vrs1.value + vrs2.value + vrs3.value, vrs3.rngState)

Pros:

- Conceptually straightforward
- Slightly reduced syntactic ugliness

Cons:

- Can't chain together computation in "pipeline" that is very general or neat (critically, would need to operate on both the values and the state, and f in flatMap can't access state)
- Can't define general other functions in terms of nice flatMap
- Can't use flatMap to chain in for comprehensions

Design Space Option 3: Approach Seen Earlier

type Rand[T] = (RNG => (T, RNG)) // really a “computation using randomness”

def unit[T](v:T) = rng:RNG => (v, rng)

def int: Rand[Int] = rng:RNG => rng.nextInt

def map[T,U](action: Rand[T])(f: T => U): Rand[U] = (rng1:RNG) => { val (v, rng2) = action(rng1); (f(v), rng2) }

def flatMap[T,U](action: Rand[T])(f: T => Rand[U]): Rand[U] = { val (v2, rng2) = action(rng1); f(v)(rng2) }

Use:

```
for {
    x <- int
    y <- int
    z <- int
    } yield (x+y+z)
```

Pros

- Conceptually straightforward
- Greatly enhanced transparency
  - The rng maintenance essentially “goes away”

Cons:

- Conceptually a bit confusing in that the monad represents an ACTION/TRANSITION/COMPUTATION
- The handling of the functions accepting the rngs is hidden and not obvious

Design Space Option 4: Variant on Approach Seen Earlier

case class ValueWithRNGStateAction(run: RNG => (Int, RNG))

int: ValueWithRNGStateAction

map(f:Int =>Int) : ValueWithRNGStateAction

flatMap(f:Int => ValueWithRNGStateAction) : ValueWithRNGStateAction // because returning something that takes in an RNG, f has access to the state component of the monad

Use:

```
for {
    x <- int
    y <- int
    z <- int
    } yield (x+y+z)
```

Pros

- Conceptually straightforward
- Greatly enhanced transparency
  - The rng maintenance essentially "goes away"

Cons:

- Conceptually a bit confusing in that the Abstract (monad) represents an ACTION/TRANSITION/COMPUTATION
- The handling of the functions accepting the rngs is hidden and not obvious