CSCI 365 Problem Set 9: Monads

Due Saturday, 13 April 2024

Specification

To receive credit for this problem set:

- Complete at least 7 out of 11 exercises.
- Any code you write must adhere to the Haskell style guide linked from the course web page.

Standard library

Exercise 1 Visit the documentation for the `Control.Monad` module at `https://hackage.haskell.org/package/base-4.19.1.0/docs/Control-Monad.html`. Pick at least three functions or operators you are unfamiliar with, explain what they do, and give an example of their use.

Functor, Applicative, and Monad

Each of the following exercises asks you to implement one of the `Functor`, `Applicative`, or `Monad` methods in terms of others. For example, the below example shows how we can implement `fmap` given only `(>>=)` and `return` (proving that "every `Monad` is a `Functor`"):

```
fmapFromBind ::
  (forall a b. m a -> (a -> m b) -> m b) ->    -- If we have bind...
  (forall a. a -> m a) ->                      -- and return...
  (forall a b. (a -> b) -> m a -> m b)         -- we can implement fmap.
fmapFromBind (>>=) return = \f ma -> ma >>= (return . f)
```

Note that `fmapFromBind` has a *higher-rank type*, that is, instead of just having `forall` at the very outermost level, it has some `foralls` inside parentheses, in particular on the left side of `->`. This just ensures that `fmapFromBind` can only be given polymorphic functions as inputs, and must produce a polymorphic function as output.

Exercise 2 Give a detailed explanation why the definition

```
fmapFromBind (>>=) return = \f ma -> ma >>= (return . f),
```

given in the example above, has the correct type.

Exercise 3 Implement `(<*>)` in terms of `(>>=)` and `fmap`. ("Every Monad is an Applicative".)

```
apFromBind ::
  (forall a b. (a -> b) -> m a -> m b) ->         -- If we have fmap...
  (forall a b. m a -> (a -> m b) -> m b) ->       -- and bind...
  (forall a b. m (a -> b) -> m a -> m b)          -- we can implement (<*>).
apFromBind fmap (>>=) = _
```

Exercise 4 Implement `(>>=)` in terms of `join` and `fmap`. ("`join` is an alternative way to define `Monad`".)

```
bindFromJoin ::
  (forall a b. (a -> b) -> m a -> m b) ->
  (forall a. m (m a) -> m a) ->
  (forall a b. m a -> (a -> m b) -> m b)
bindFromJoin fmap join = _
```

Exercise 5 Implement `join` in terms of `(>>=)`. ("`join` and `(>>=)` are equivalent in power.")

```
joinFromBind ::
  (forall a b. m a -> (a -> m b) -> m b) ->
  (forall a. m (m a) -> m a)
joinFromBind (>>=) = _
```

Exercise 6 Implement `(>>=)` in terms of `(>=>)`. ("`(>=>)` is another alternative way to define `Monad`.")

```
bindFromFish ::
  (forall a b c. (a -> m b) -> (b -> m c) -> (a -> m c)) ->
  (forall a b. m a -> (a -> m b) -> m b)
bindFromFish (>=>) = _
```

Risk

The game of *Risk* involves two or more players, each vying to "conquer the world" by moving armies around a board representing the world and using them to conquer territories. The central mechanic of the game is that of one army attacking another, with dice rolls used to determine the outcome of each battle.

The rules of the game make it complicated to determine the likelihood of possible outcomes. You will write a *simulator* which could be used by Risk players to estimate the probabilities of different outcomes before deciding on a course of action.

The `Rand StdGen` monad

Since battles in Risk are determined by rolling dice, your simulator will need some way to access a source of randomness. Many languages include standard functions for getting the output of a pseudorandom number generator. For example, in Java one can write

```
   Random randGen = new Random();
   int dieRoll = 1 + randGen.nextInt(6);
```

to get a random value between 1 and 6 into the variable `dieRoll`. It may seem like we can't do this in Haskell, because the output of `randGen.nextInt(6)` may be different each time it is called—and Haskell functions must always yield the same outputs for the same inputs.

However, if we think about what's going on a bit more carefully, we can see how to successfully model this in Haskell. The Java code first creates a `Random` object called `randGen`. This represents a *pseudorandom number generator*, which remembers a bit of state (a few numbers), and every time something like `nextInt` is called, it uses the state to (deterministically) generate an `Int` and then updates the state according to some (deterministic) algorithm. So the numbers which are generated are not truly random; they are in fact completely deterministic, but computed using an algorithm which generates random-seeming output. As long as we initialize (*seed*) the generator with some truly random data, this is often good enough for purposes such as simulations.

In Haskell we can cerainly have pseudorandom number generator objects. Instead of having methods which mutate them, however, we will have functions that take a generator and return the next pseudorandom value *along with a new generator*. That is, the type signature for `nextInt` would be something like

```
nextInt :: Generator -> (Int, Generator)
```

However, using `nextInt` would quickly get annoying: we have to manually pass around generators everywhere. For example, consider some code to generate three random `Int`s:

```
threeInts :: Generator -> ((Int, Int, Int), Generator)
threeInts g = ((i1, i2, i3), g''')
  where (i1, g')   = nextInt g
```

```
        (i2, g'')  = nextInt g'
        (i3, g''') = nextInt g''
```

Ugh! Fortunately, there is a much better way. The `MonadRandom` package[1] defines a *monad* which encapsulates this generator-passing behavior (very similar to the `State` monad we explored in class). Using it, `threeInts` can be rewritten as follows:

[1] `http://hackage.haskell.org/package/MonadRandom`

```
threeInts :: Rand StdGen (Int, Int, Int)
threeInts = do
  i1 <- getRandom
  i2 <- getRandom
  i3 <- getRandom
  return (i1,i2,i3)
```

The type signature says that `threeInts` is a computation in the `Rand StdGen` monad which returns a triple of `Int`s. `Rand StdGen` computations implicitly pass along a pseudorandom generator of type `StdGen` (which is defined in the standard Haskell library `System.Random`).

Visit the documentation for the `MonadRandom` package on Hackage (`http://hackage.haskell.org/package/MonadRandom`). Take a look at the `Control.Monad.Random.Lazy` module, which defines various ways to "run" a `Rand` computation; in particular you will eventually (at the very end of the assignment) need to use the `evalRandIO` function. Take a look also at the `Control.Monad.Random.Class` module, which defines a `MonadRandom` type class containing methods you can use to access the random generator in a `Rand` computation. For example, this is where the `getRandom` function (used above in the `threeInts` example) comes from. However, you probably won't need to use these methods directly in this assignment.

In `Risk.hs` I have provided a type

```
newtype DieValue = DV { unDV :: Int }
```

for representing the result of rolling a six-sided die. I have also provided an instance of `Random` for `DieValue` (allowing it to be used with `MonadRandom`), and a definition

```
die :: Rand StdGen DieValue
die = getRandom
```

which represents the random outcome of rolling a fair six-sided die.

The Rules

The rules of attacking in Risk are as follows.

- There is an attacking army (containing some number of units) and a defending army (containing some number of units).
- The attacking player may attack with up to three units at a time. However, they must always leave at least one unit behind. That is, if they only have three total units in their army they may only attack with two, and so on.
- The defending player may defend with up to two units (or only one if that is all they have).
- To determine the outcome of a single battle, the attacking and defending players each roll one six-sided die for every unit they have attacking or defending. So the attacking player rolls one, two, or three dice, and the defending player rolls one or two dice.
- The attacking player sorts their dice rolls in descending order. The defending player does the same.
- The dice are then matched up in pairs, starting with the highest roll of each player, then the second-highest.
- For each pair, if the attacking player's roll is higher, then one of the defending player's units die. If there is a tie, or the defending player's roll is higher, then one of the attacking player's units die.

For example, suppose player A has 3 units and player B has 5. A can attack with only 2 units, and B can defend with 2 units. So A rolls 2 dice, and B does the same. Suppose A rolls a 3 and a 5, and B rolls a 4 and a 3. After sorting and pairing up the rolls, we have

A	B
5	4
3	3

A wins the first matchup (5 *vs.* 4), so one of B's units dies. The second matchup is won by B, however (since B wins ties), so one of A's units dies. The end result is that now A has 2 units and B has 4. If A wanted to attack again they would only be able to attack with 1 unit (whereas B would still get to defend with 2—clearly this would give B an advantage because the *higher* of B's two dice rolls will get matched with A's single roll.)

Exercise 7 Given the definitions

```
type Army = Int

data Battlefield = Battlefield { attackers :: Army, defenders :: Army }
```

(which are also included in `Risk.hs`), write a function with the type

```
battle :: Battlefield -> Rand StdGen Battlefield
```

which simulates a single battle (as explained above) between two opposing armies. That is, it should simulate randomly rolling the appropriate number of dice, interpreting the results, and updating the two armies to reflect casualties. You may assume that each player will attack or defend with the maximum number of units they are allowed.

You should add your function to `Risk.hs`. You can test it by typing `cabal repl` at a command line and then testing your function at the resulting GHCi prompt.

Exercise 8 Of course, usually an attacker does not stop after just a single battle, but attacks repeatedly in an attempt to destroy the entire defending army (and thus take over its territory).

Now implement a function

```
invade :: Battlefield -> Rand StdGen Battlefield
```

which simulates an entire invasion attempt, that is, repeated calls to `battle` until there are no defenders remaining, or fewer than two attackers.

Exercise 9 Finally, implement a function

```
successProb :: Battlefield -> Rand StdGen Double
```

which runs `invade` 1000 times, and uses the results to compute a `Double` between 0 and 1 representing the estimated probability that the attacking army will completely destroy the defending army. For example, if the defending army is destroyed in 300 of the 1000 simulations (but the attacking army is reduced to 1 unit in the other 700), `successProb` should return `0.3`.

Further Exploration

Exercise 10 Take a look at the paper *Monads for Functional Programming*, available from `https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf`.[2] This is a version of one of the papers which originally introduced monads into functional programming. Read as much as you can and explain one thing you learned from reading the paper.

[2] Wadler, Philip. "Monads for functional programming." In Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1, pp. 24-52. Springer Berlin Heidelberg, 1995.

Exercise 11 Take a look at the *Typeclassopedia*, `https://wiki.haskell.org/Typeclassopedia`. Read one of the sections and explain something you learned.