

# Functional Programming 2014/2015

## Assignment 4: Monads

Ruud Koot

September 16, 2014

In this assignment we'll ask you to implement a probability monad and an instrumented state monad.

### 1 A Game of Chance

Here's a game I like to play: I toss a coin six times and count the number of heads I see, then I roll a dice; if the number of eyes on the dice is greater than or equal to the number of heads I counted then I win, else I lose. As I'm somewhat of a sore loser, I'd like to know my chances of winning of beforehand, though.

There are three ways to compute this probability:

1. Use a pen, paper (or, if you prefer, chalk and a blackboard) and some basic discrete probability theory to calculate the probability directly.
2. We draw or compute the complete decision tree of the game and count the number of wins and losses.
3. We write a computer program that simulates the game to approximate the probability.

As we're computer scientists, we'll leave the first option to the mathematicians and focus on the second and third possibilities. In fact, using monads, we'll see how both can be done at the same time.

#### 1.1 The Gambling Monad

Modeling a coin and a dice in Haskell shouldn't pose much difficulty for you anymore:

```
data Coin      = H | T
data Dice      = D1 | D2 | D3 | D4 | D5 | D6
data Outcome   = Win | Lose
```

The tossing of a *Coin* and rolling of a *Dice* is given by the monadic interface *MonadGamble*:

```
class Monad m => MonadGamble m where
  toss :: m Coin
  roll :: m Dice
```

**Exercise 1.** Write a function `game :: MonadGamble m => m Outcome` that implements the game above.

## 1.2 Simulation

Simulating probabilistic events requires a (pseudo)random number generator. Haskell has one available in the `System.Random` library. Random number generators need to have access to a piece of state called the *seed*, as such the random number generator runs in a monad, the *IO* monad to be exact.

**Exercise 2.** Give *Random* instances for *Coin* and *Dice*.

**Exercise 3.** Give a *MonadGamble* instance for the *IO* monad.

**Exercise 4.** Write a function `simulate :: IO Outcome -> Integer -> IO Rational` that runs a game of chance (given as the first parameter, not necessarily the game implemented in Exercise 1) *n* times (*n* > 0, the second parameter) and returns the fraction of games won.

You can now approximate to probability of winning using `simulate game 10000`. Would you care to take a guess what the exact probability of winning is?

## 1.3 Decision trees

One drawback of simulation is that the answer is only approximate. We can obtain an exact answer using decision trees. Decision trees of probabilistic games can be modeled as:

```
data DecisionTree α = Result α | Decision [DecisionTree α]
```

In the leaves we store the result and in each branch we can take one from several possibilities. As we don't store the probabilities of each decision, we'll have to assume they are uniformly distributed (i.e., each possibility has an equally great possibility of being taken). Fortunately for us, both fair coins and fair dice produce a uniform distribution.

**Exercise 5.** Give a *Monad* instance for *DecisionTree*. (Hint: Use the types of (`>>=`) and `return` as your guidance: it's the most straightforward, type-correct definition. The definition will somewhat resemble that of the *Monad* instance for *Eval*, as asked for in the last exercise in Hutton's monad material. In both cases they're so-called free monads.)

**Exercise 6.** Give a *MonadGamble* instance for *DecisionTree*.

**Exercise 7.** Write a function `probabilityOfWinning :: DecisionTree Outcome -> Rational` that, given a decision tree, computes the probability of winning.

You can find the exact probability of winning using `probabilityOfWinning game`. Was your earlier guess correct? If you know a bit of probability theory, you can double check the correctness by doing the pen-and-paper calculation suggested above.

Note that we used the same implementation of *game* to obtain both an approximate and an exact answer.

## 2 Instrumented State Monad

In Hutton’s monad material we encountered the state monad (which he called *ST*, as it are technically state transformers—functions mapping one state to another—that form a monad). We’ll now give a presentation of the state monad that is closer to how they are found in Haskell’s *Control.Monad.State* library.

A state monad is monad with additional monadic operations *get* and *put*:

```
class Monad m => MonadState m s | m -> s where
  get    :: m s
  put    :: s -> m ()
  modify :: (s -> s) -> m s
```

(The “ $| m \rightarrow s$ ” part of this class is called a *functional dependency*. You can ignore this. If you want to know exactly what it does, then you should follow the *Advanced Functional Programming* course during your Master’s. The short answer is that it helps the compiler figure out which particular state monad instance it needs to use for a given type.)

Apart from the usual three monad laws, state monads should also satisfy:

$$\begin{aligned} \text{put } s_1 &\gg \text{put } s_2 && \equiv \text{put } s_2 \\ \text{put } s &\gg \text{get} && \equiv \text{put } s \gg \text{return } s \\ \text{get} &\gg \text{put} && \equiv \text{return } () \\ \text{get} &\gg (\lambda s \rightarrow \text{get} \gg k \ s) && \equiv \text{get} \gg (\lambda s \rightarrow k \ s \ s) \end{aligned}$$

Check to see if you understand what these four laws say and if they make sense.

**Exercise 8.** Give default implementations of *get* and *put* in terms of *modify*, and a default implementation of *modify* in terms of *get* and *put*.

### 2.1 Instrumentation

We are now going to define our own, slightly modified state monad that, besides keeping track of a piece of state, has also been instrumented to count the number of ( $\gg$ ), *return*, *get* and *put* operations that have been performed during a monadic computation.

The counts are given by the type:

```
data Counts = Counts {
  binds    :: Int,
  returns  :: Int,
  gets     :: Int,
  puts     :: Int
}
```

**Exercise 9.** As a convenience, give a *Monoid* instance for *Count* that sums the counts pairwise. Define constants *oneBind*, *oneReturn*, *oneGet*, *onePut* :: *Counts* that represent a count of one ( $\gg$ ), *return*, *get* and *put* operation, respectively.

Our state transformer is now given by:

```
newtype State' s α = State' { runState' :: (s, Counts) → (α, s, Counts) }
```

Note that our *State'* corresponds to Hutton's *ST*, but that it has been parameterized over the type of state *s* (for which Hutton used the type synonym *State*). Additionally, we keep track of the *Counts* as an internal piece of state that is not exposed through the *get* and *put* interface.

**Exercise 10.** Give *Monad* and *MonadState* instances for *State'* that count the number of ( $\gg$ ), *return*, *get* and *put* operations.

## 2.2 Tree Labeling

Here is another tree data type:

```
data Tree α = Branch (Tree α) α (Tree α) | Leaf
```

This is a binary tree that stores values on the internal nodes only.

**Exercise 11.** Write a function *label* :: *MonadState m Int*  $\Rightarrow$  *Tree α*  $\rightarrow$  *m (Tree (Int, α))* that labels a tree with integers increasingly, using a depth-first in-order traversal.

**Exercise 12.** Write a function *run* :: *State' s α*  $\rightarrow$  *s*  $\rightarrow$  (*α*, *Counts*) that runs a state monadic computation in the instrumented state monad, given some initial state of type *s*, and returns the computed value and the number of operations counted.

## 3 Further Reading

If you want a bit more practice with implementing functions in the *IO* monad, then try implementing the Mastermind game from the Utrecht Summerschool on Applied Functional Programming.

If you want to know more about the probability monad then have look at “Probabilistic Functional Programming in Haskell”, Martin Erwig and Steve Kollmansberger, *Journal of Functional Programming*, Vol. 16, No. 1, pp. 21–34, 2006.