Functional Programming
Retake Assignment: Monads

Ruud Koot

January 16, 2024

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 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. Draw or compute the complete decision tree of the game and count the number of wins and losses.
3. 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:

$$\begin{aligned}&\textbf{data}\ \mathit{Coin} &&= \mathit{H} \mid \mathit{T}\\ &\textbf{data}\ \mathit{Dice} &&= \mathit{D1} \mid \mathit{D2} \mid \mathit{D3} \mid \mathit{D4} \mid \mathit{D5} \mid \mathit{D6}\\ &\textbf{data}\ \mathit{Outcome} &&= \mathit{Win} \mid \mathit{Lose}\end{aligned}$$

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. Read the description of the game very carefully: it is easy to make an off-by-one error; furthermore, as tossing and rolling are side-effects the order in which you perform them matters.*

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 of 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* $(\gg\!=)$ *and return for guidance: it's the most straightforward, type-correct definition that isn't an infinite loop. This is an example of a so-called* free monad*.)*

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 this course, we have encountered the idea of a state monad, which lets us emulate an imperative style of programming with mutable variables in a functional language. We'll now take a moment to study some of the operations that constitute a state monad.

A state monad is monad with additional monadic operations *get* and *put*, and some optional compound operations such as the operation *modify* we include below (which is slightly different from the operation modify in *Control.Monad.State*):

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ m\ s \mid m \to s\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{get} & :: & m\ s \\
\mathit{put} & :: s & \to m\ () \\
\mathit{modify} & :: (s \to s) & \to m\ s
\end{array}
\end{array}
$$

(The "$\mid m \to s$" 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:

$$
\begin{array}{lll}
\mathit{put}\ s_1 \gg \mathit{put}\ s_2 & \equiv \mathit{put}\ s_2 \\
\mathit{put}\ s \gg \mathit{get} & \equiv \mathit{put}\ s \gg \mathit{return}\ s \\
\mathit{get} \gg\!= \mathit{put} & \equiv \mathit{return}\ () \\
\mathit{get} \gg\!= (\lambda s \to \mathit{get} \gg\!= k\ s) & \equiv \mathit{get} \gg\!= (\lambda s \to k\ s\ s)
\end{array}
$$

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. Your implementation of modify should not use return.*

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:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Counts} = \mathit{Counts}\ \{ \\
\quad \mathit{binds} :: \mathit{Int},
\end{array}
$$

$$
\begin{aligned}
&\quad \mathit{returns} :: \mathit{Int},\\
&\quad \mathit{gets} \quad :: \mathit{Int},\\
&\quad \mathit{puts} \quad :: \mathit{Int}\\
&\}
\end{aligned}
$$

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

Our state monad is now given by:

$$\textbf{newtype}\ \mathit{State}'\ s\ \alpha = \mathit{State}'\ \{\mathit{runState}' :: (s, \mathit{Counts}) \to (\alpha, s, \mathit{Counts})\}$$

Note that our State' is like the usual *State* monad, but that it has been parameterized over the type of state s. 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:

$$\textbf{data}\ \mathit{Tree}\ \alpha = \mathit{Branch}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha) \mid \mathit{Leaf}$$

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

Exercise 11. *Write a function* $\mathit{label} :: \mathit{MonadState}\ m\ \mathit{Int} \Rightarrow \mathit{Tree}\ \alpha \to m\ (\mathit{Tree}\ (\mathit{Int}, \alpha))$ *that labels a tree with integers increasingly, using a depth-first in-order traversal. In your implementation, make use of the operation modify rather than using get and put.*

Exercise 12. *Write a function* $\mathit{run} :: \mathit{State}'\ s\ \alpha \to s \to (\alpha, \mathit{Counts})$ *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.*

For example, the expression

$$
\begin{aligned}
&\textbf{let}\ \mathit{tree} = \mathit{Branch}\ (\mathit{Branch}\ \mathit{Leaf}\ \texttt{"B"}\ \mathit{Leaf})\ \texttt{"A"}\ \mathit{Leaf}\\
&\textbf{in}\ \ \mathit{run}\ (\mathit{label}\ \mathit{tree})\ 42
\end{aligned}
$$

should evaluate to

$$
\begin{aligned}
&(\mathit{Branch}\ (\mathit{Branch}\ \mathit{Leaf}\ (42, \texttt{"B"})\ \mathit{Leaf})\ (43, \texttt{"A"})\ \mathit{Leaf}\\
&, \mathit{Counts}\ \{\mathit{binds} = 10, \mathit{returns} = 5, \mathit{gets} = 4, \mathit{puts} = 2\})
\end{aligned}
$$

3 Further reading

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.