Haskell exercise: Monads

Department of Mathematics and Computer Science
University of Southern Denmark

October 25, 2017

1. The *Maybe*-monad, and a monad for *Expr*

You are given a data type for expressions:

$$\textbf{data}\ \mathit{Expr}\ a = \mathit{Var}\ a \mid \mathit{Add}\ (\mathit{Expr}\ a)\ (\mathit{Expr}\ a)\ \textbf{deriving}\ \mathit{Show}$$

1. To turn *Expr* into a monad, we should give valid definitions for return and bind:

   $$\begin{aligned} &\mathit{return} :: a \to \mathit{Expr}\ a \\ &(\gg\!=)\ \ :: \mathit{Expr}\ a \to (a \to \mathit{Expr}\ b) \to \mathit{Expr}\ b \end{aligned}$$

   Fill out the function bodies with appropriate definitions:

   $$\begin{aligned} &\textbf{instance}\ \mathit{Monad}\ \mathit{Expr}\ \textbf{where} \\ &\quad \mathit{return}\ x \qquad\quad = \bot \\ &\quad (\mathit{Var}\ a) \gg\!= f \quad = \bot \\ &\quad (\mathit{Add}\ x\ y) \gg\!= f = \bot \end{aligned}$$

2. We would like to define a function

   $$\mathit{replace} :: \mathit{Eq}\ a \Rightarrow [(a, b)] \to \mathit{Expr}\ a \to \mathit{Expr}\ (\mathit{Maybe}\ b)$$

   which replaces occurences of type a with something of type $\mathit{Maybe}\ b$. Example:

   - $\mathit{replace}\ [\,]\ (\mathit{Var}\ \texttt{'a'}) = \mathit{Var}\ \mathit{Nothing}$
   - $\mathit{replace}\ [(\texttt{'a'}, 3)]\ (\mathit{Var}\ \texttt{'a'}) = \mathit{Var}\ (\mathit{Just}\ 3)$

- *replace* $[(\texttt{'a'}, 3)]$ $(Add\ (Var\ \texttt{'a'})\ (Var\ \texttt{'b'})) = Add\ (Var\ (Just\ 3))\ (Var\ Nothing)$

You should use the functionality of the *Expr*-monad to implement *replace*. You can use

$$lookup :: Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b$$

from the Prelude in your implementation of replace.

3. Now we would like to make a function

$$convert :: Expr\ (Maybe\ a) \rightarrow Maybe\ (Expr\ a)$$

which returns *Nothing* if there is an occurrence of *Nothing* inside the input expression e, otherwise it returns $Just\ e'$, where e' is a new expression where the internal values of type a are not wrapped in *Just*.

You should use the functionality of the *Maybe* monad to implement your convert function.

2. Random numbers and the State Monad

Functions involving randomness in Haskell take a seed $g :: StdGen$ as input, and returns an output and a new seed $g' :: StdGen$.

As an example, the built-in function

$$randomR :: (Int, Int) \rightarrow StdGen \rightarrow (Int, StdGen)$$

on inputs $randomR\ (a, b)\ g$, returns (x, g') (an integer x and a new seed g'), where x is chosen uniformly, with the condition $a \leqslant x \leqslant b$.

To get a random seed, one needs the *IO* environment.

A complete program to simulate two die rolls and return the sum of the die-values is given below. Make sure that you understand how this works, before going on to the rest of the exercise.

```
import System.Random
die6 :: StdGen -> (Int, StdGen)
die6 g = randomR (1,6) g
twoDie :: StdGen -> (Int, StdGen)
twoDie g = let (d1, g') = die6 g
               (d2, g'') = die6 g'
   in          (d1 + d2, g'')
test :: IO (Int, StdGen)
```

```
test = do g ← newStdGen
          return (twoDie g)
```

We would like to give a nicer definition for *twoDie* which does not explicitly handle the random seed.

The *State*-monad libary provides the following functions to wrap and unwrap functions in the state monad.

```
state :: (s → (a, s))        → State s a
runState :: State s a → s → (a, s)
```

You are provided the following stub of a program:

```
import Control.Monad.State

die6' :: State StdGen Int
die6' = ⊥

twoDie' :: State StdGen Int
twoDie' = ⊥

test' :: IO (Int, StdGen)
test' = do g ← newStdGen
           return (runState twoDie' g)
```

1. Using the function *state* and your definition *die6*, provide a definition of *die6′*.

2. Implement *twoDie′* only referring to *die6′* and the monadic functionality of State Monad.