Monad and QuickCheck Exercises

Warm-up: Reader-Writer-State monads

The Reader-Writer-State (RWS) monad encapsulates three separate pieces of state: one that can only be read, one that can only be appended to, and one that can be freely both read and written. (In applications where one of these behaviors is not relevant, we can make the corresponding data pieces be just `()`.)

The file `Warmup.hs` contains skeleton definitions for the RWS monad and associated operations, as well as a variant of the RWS monad that also supports throwing errors, together with a couple of simple tests. Complete the missing parts of the definitions, and check that the provided tests complete successfully. Don't change anything that is not explicitly marked as `undefined`.

Randomised Response

Randomised response is a statistical protocol that enable collecting sensitive information while protecting the privacy of the responders. The core idea is that you introduce random noise to protect the individual responders.

For a binary (yes/no) question the protocol is: first the responder flips a coin, if the coin shows head, then the responder gives a truthful answer (and flips the coin again to eliminate a side channel); if the coin show tail, then the responder flips the coin again and gives the result as answer. Thus, when we get an answer from an reponder we don't know if we got a random answer or a truthful answer.

Using the `System.Random` (Links to an external site.) module we can write the function `binary_random_response` that implements the randomised response protocol:

```
import qualified System.Random as R


binary_random_response :: R.StdGen -> Bool -> (Bool, R.StdGen)

binary_random_response g true_answer =

  if first_coin then (true_answer, g')

  else R.random g'

  where (first_coin, g') = R.random g
```

Where the user should call `binary_random_response` with a random number generator and the truthful answer as arguments, and then submit the boolean returned as result.

Warm-up exercise: Actually, the given `binary_random_response` function doesn't quite implement the protocol correctly. It has introduced a side-channel (the protocol specifies that the coin should always be flipped twice, but the program only sometimes call `R.random` twice). Correct the implementation.

Generalise exercise: Implement a function `random_response` that works for multiple-choice questions. That is, `random_response` should have the type:

```
random_response :: R.StdGen -> a -> [a] -> (a, R.StdGen)
```

Use monads exercise: In the previous functions it's somewhat cumbersome and error-prone to thread the random number generator around. To simplify the code we can use a monad.

- Which monad should we use?
- Rewrite your functions to use monadic style using the `do`-notation.

Maybe we have Options

In SML (and OCaml and F#) we have an `Option`-type constructor rather than the `Maybe`-type constructor as in Haskell. Declare a Haskell type `Option` to be an instance of `Functor`, `Applicative`, and `Monad`. Your `Option` type should have two value constructors, `NONE` and `SOME` (corresponding to `Nothing` and `Just`, respectively).

Try to write the functions `fmap`, `return` and `>>=` by yourself rather than looking them up. For reference, here are the type-class definitions for `Functor` and `Monad`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b


class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b
   fail   :: String -> m a
```

You might want to use with the following instance for `Applicative` if you don't want to write it by hand:

```
instance Applicative Option where
  pure = return
  (<*>) = ap             -- from Control.Monad
```

You could have invented Monads

The following two exercises are a rewording/ripoff of the exercises from Dan Piponi's blog post You Could Have Invented Monads! (And Maybe You Already Have) (Links to an external site.), go there if you want solutions, more exercises, and longer explanations.

Traceable functions

Define a monad for working with *traceable functions*. That is, functions that compute both a result and a trace of their computation. For instance, a normal function that takes a floating-point number and computes a new floating-point number will have the type:

```
f :: Float -> Float
```

but a traceable function should have a type more like:

```
ft :: Float -> (Float, String)
```

Thus, if we have the two traceable functions `ft` and `gt` we can then compose them, and compute the composed trace with code:

```
let (y, s) = gt x
    (z, t) = ft y in (z, s++t)
```

But keeping track of the trace like this is quite cumbersome, instead we would like use the `do`-notation:

```
do y <- gt x
   z <- ft y
   return z
```

Define a monad for working with traceable functions. That is, complete the following code:

```
newtype Trace a = T (a, String)


instance Monad Trace where
    -- (>>=) :: Trace a -> (a -> Trace b) -> Trace b
    (T p) >>= f = ...


    -- return :: a -> Trace a
    return x = ...
```

For testing your code you might want to use the following utility function for constructing traceable functions for normal functions:

```
traceable :: String -> (t -> a) -> t -> Trace a

traceable name f = \x -> T(f x, name ++" called.")
```

Multivalued Functions

Consider the the functions `sqrt` and `cbrt` that compute the square root and cube root, respectively, of a real number. These are straightforward functions of type:

```
Float -> Float
```

(although `sqrt` will thrown an exception for negative arguments, something we'll ignore).

Now consider a version of these functions that works with complex numbers. Every complex number, besides zero, has two square roots. Similarly, every non-zero complex number has three cube roots. So we'd like `sqrtC` and `cbrtC` to return lists of values. In other words, we'd like

```
sqrtC, cbrtC :: Complex Float -> [Complex Float]
```

We'll call these *multivalued* functions.

Suppose we want to find the sixth root of a real number. We can just concatenate the cube root and square root functions. In other words we can define `sixthroot` as follow:

```
sixthroot x = sqrt (cbrt x)
```

But how do we define a function that finds all six sixth roots of a complex number using `sqrtC` and `cbrtC`. We can't simply concatenate these functions. What we'd like is to first compute the cube roots of a number, then find the square roots of all of these numbers in turn, combining together the results into one long list. That is, by using `do`-notation we can define `sixthrootC` as:

```
sixthrootC :: Complex Float -> [Complex Float]

sixthrootC x = do

   cr <- cbrtC x

   sr <- sqrtC cr

   return sr
```

We have rediscovered the list monad. Try to define it for yourself, by completing the implementation:

```
newtype Multivalued a = MV [a]


instance Monad Multivalued where
    -- (>>=) :: Multivalued a -> (a -> Multivalued b) -> Multivalued b
    (MV p) >>= f = ...


    -- return :: a -> Multivalued a
    return x = ...


sqrtC, cbrtC, sixthrootC :: Complex Float -> Multivalued (Complex Float)
...
```

QuickCheck Exercises

Work-sheet Generation

When kids are learning maths in primary school it's custom to give them work-sheets with lots of problems to practise on. We use the type `Problem` for representing a practise problem that mixes addition and multiplication of integers:

```
type Problem = (Expr, Int)
data Expr = Const Int
          | Plus Expr Expr
          | Mult Expr Expr
          deriving (Show, Read, Eq)
```

That is, a `Problem` is a pair of an arithmetic expression and the value it evaluates to (the later is used for generating a grading sheet for the teacher).

Worksheet exercise: Write a function `generate_sheet` that can generate a list of *n* practise problems.

- What is the type of `generate_sheet`?
- You might want to start by writing a function `eval` that can evaluate an arithmetic expression. (Why?)

Worksheets continued

Based on our previous success with worksheet generation for addition and multiplication practise problems, we'll now try to generate quadratic equations practise problems. We'll use the types `Problem`, `Equality` and `QuadExpr` to represent practise problems for quadratic equations:

```
type Problem = (Equality, Maybe (Int, Int))


data Equality = Equal QuadExpr QuadExpr
              deriving (Show, Read, Eq)


data QuadExpr = Const Int
              | X
              | Plus QuadExpr QuadExpr
              | Mult QuadExpr QuadExpr
              | Squared QuadExpr
          deriving (Show, Read, Eq)
```

Notice that the type `QuadExpr` is quite (too?) expressive, it's possible to represent general polynomials and the constructor `Squared` is redundant. However, we'll ignore that aspect for this exercise.

Worksheet exercise: Write a function `generate_sheet` that can generate a list of *n* practise problems.

- What is the type of `generate_sheet`?
- How do you make sure that you generate equations of the right form? That is, equations that are quadratic.
- How do you make sure to generate interesting equations? That is, equations that have solutions and the solutions are (smallish) integers.
- Make `Equality` and instance of the `Arbitrary` type-class from the QuickCheck library.
- Write a function `solve` that can solve quadratic equations.
- Write some interesting properties and test them with QuickCheck.