PROGRAMMING III (COMP 2209)
SEMESTER ONE, 2018

Home

Exercise Sheet Nine : Applicatives and Monads in Haskell

The aim of this tutorial is to understand the use of Applicatives and Monads in Haskell beyond that of IO.

Attempt the exercsise in your own time before Tuesday's lab and use that session for obtaining feedback and for asking questions related to the exercises.

Exercise One

Similar to Exercise Four on Sheet 8 we can define a function `sequenceM :: Monad m => [ m a ] -> m [ a ]` that takes a list of monadic actions for any monad `m`, executes them and returns a monadic list of values.

Provide a definition for `sequenceM` using "do" notation and use it on these list of Maybe values:

```
[ Just 3, Just 5, Just 7 ]

[ Just 3, Nothing, Just 7 ]
```

and check that you get `Just [3,5,7]` in the former and `Nothing` in the latter. Now apply `sequenceM` to the list of lists `[ [1,2] , [3,4] ]`. Can you explain the result?

Exercise Two

Consider the Knight's Tour example in the Lecture Notes on Monads. Write a function `inN :: Int -> KnightPos -> [ KnightPos ]` to find all possible board positions that can be reached in a given number of moves. Write a function `minMoves :: KnightPos -> KnightPos -> Int` that returns minimum number of moves needed to reach a destination position from a given start position.

Exercise Three

The library Control.Applicative provides an alternative List applicative called `ZipList` where instead of `pure x` returning a singleton list `[x]` it returns an infinite stream of `x` values. Then the `<*>` operator can apply a list of functions to a list of arguments by using index-wise application, i.e. the nth function is applied to the nth argument. Complete the definition of this applicative below

```
newtype ZipList a = Z [a] deriving (Eq,Show,Read)

instance Functor ZipList where
 -- fmap :: (a -> b) -> ZipList a -> ZipList b
 fmap g (Z xs) = ...

instance Applicative ZipList where
 -- pure :: a -> ZipList a
 pure x = ...

 -- <*> :: ZipList (a -> b) -> ZipList a -> ZipList b
 (Z gs) <*> (Z xs) = ...
```

Exercise Four

Given the data type

```
data Expr a = Var a | Val Int | Add (Expr a) (Expr a)
  deriving (Eq,Show,Read)
```

show how to make this type in to an instance of the Monad class. What does the bind operation (>>=) do? Use "do" notation for this monad to define a substitution function `subst :: Eq a => a -> Expr a -> Expr a -> Expr a` such that `subst v e1 e2` substitutes all occurrences of the variable named `v` for `e1` within expression `e2`.

Exercise Five

Using similar ideas from the previous question, rewrite the tree relabelling function given in the lecture on Monads but using both the State monad for the indexing and a monad corresponding to

```
data LTree a = Leaf a | Node (LTree a) (LTree a)
  deriving (Eq,Show,Read)
```

Hint: this is not as straightforward as it sounds. You will find it useful to define an auxilliary function `dist :: LTree (ST a) -> ST (LTree a)` that distributes the Tree monad over the State monad. That is, it defines the interaction between the two monads.

Page maintained by Julian Rathke.