Functional Programming 2, week 2

Monads

20th April 2016

Instructions

To do these exercises, you will need to program a bit and write some things down. The easiest way to write your code is to place it in one big file. The code in these exercises is given in Clean. You may choose to use either Clean or Haskell for your solutions.

There are some bonus exercises. We encourage you to make them, but they do not count for your final mark. Hand in your answers as a plain text file or PDF document on BlackBoard before **Tuesday 26th of April, 23:59 CEST**.

Reference: the Monad Type Class and its Laws

This section contains a quick reference to the Monad type class and its laws. Refer to the slides for more information. The Monad type class is defined as follows:

```
class Monad m where
  pure          :: a -> m a
  (>>=) infixl 1 :: (m a) (a -> m b) -> m b
```

The following laws should hold for any instance of this class:

Left identity

```
pure a >>= \x -> f x == f a
```

Right identity

```
m >>= \x -> pure x == m
```

Associativity

```
(m >>= \x -> f x) >>= \y -> g y == m >>= (\x -> (f x >>= \y -> g y))
```

1 Getting started

If you use the Clean IDE, create a new project. Then download the file `MonadPractical.icl` from BlackBoard and change the name to `MonadPracticalYourNames.icl`. You can add all your solutions to this file.

2 The Maybe monad

The `Maybe` type constitutes a monad that models computations that might fail. As we discussed in the lecture, the type and its `Monad` class instance are defined as follows:

```
:: Maybe a = Nothing | Just a

instance Monad Maybe where
  (>>=) Nothing  _    = Nothing
  (>>=) (Just x) next = next x

  pure x = Just x
```

2.1 Safe division

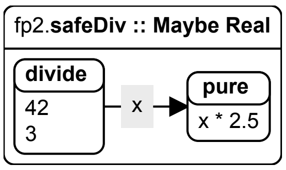
Dividing a `Real` by zero will result in an infinite value. The following Clean program will give as result `inf`:

```
Start = 42.0 / 0.0
```

If we don't want our division to fail when dividing by zero, a possible solution is to define a safe division using `Maybe`:

```
divide :: Real Real -> Maybe Real
divide x y
  | y == 0.0  = Nothing
  | otherwise = Just (x / y)
```

1. What will be the outcome of the next programs?
   a) `Start = divide 42.0 3.0`
   b) `Start = divide 42.0 0.0`
   c) `Start = (divide 42.0 3.0) * 2.5`



2. Using at most one bind (`>>=`), one `pure`, and one lambda, rewrite 1c such that it type checks correctly and produces the correct answer (`Just 35.0`).

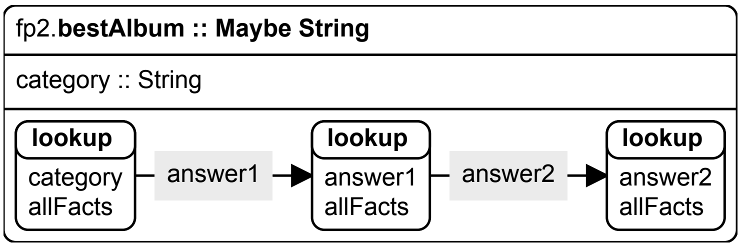
2.2 Music facts

Take a look at the following code about a (very) small music database:

```
allFacts :: [(String, String)]
allFacts =  [ (”Best singer”,      ”Ronnie James Dio”)
            , (”Ronnie James Dio”, ”Dio”)
            , (”Dio”,              ”Holy Diver”) ]

lookup :: String [(String, String)] -> Maybe String
lookup key []    = Nothing
lookup key [(key‘, value) : rest]
  | key‘ == key = Just value
  | otherwise   = lookup key rest

bestAlbum :: String -> Maybe String
bestAlbum category = lookup category allFacts
     >>= \answer1 -> lookup answer1  allFacts
     >>= \answer2 -> lookup answer2  allFacts
```



3. For the following programs, describe in your own words how they are evaluated, and what their result is:
   a) `Start = bestAlbum ”Best singer”`
   b) `Start = bestAlbum ”Ronnie James Dio”`

2.3 Fitting the laws

We already proved the left and right identity properties for `Maybe` during the lecture.

4. Prove the associativity property for `Maybe` using the definition on Page 2.

3 The Either monad

`Maybe` comes in handy when doing computations that may fail. The only problem is that if a computation fails, you never know **why** your program failed, you only know **that** it failed!

To remedy this, you may want to add some extra information when a computation fails, such as an error message. The `Either` type allows you to do this. It models a computation that might fail with an error message:

```
:: Either e a = Left e | Right a
```

By convention `Right` is used as a "right" answer and `Left` is used to signal a "not right" answer, or an error.

1. Implement the `Monad` instance for `Either`.

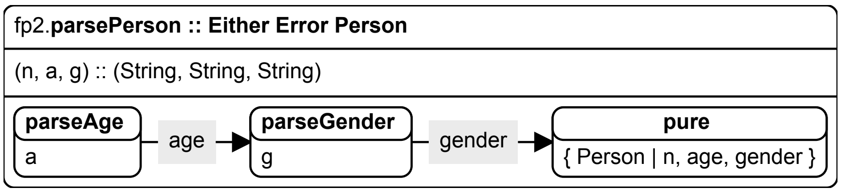
Lets take a look at a program which parses CSV[1] formatted data. Each row contains person details: a person's name, his or her age and gender. The rows and cells are already split in lists of tuples of strings.

```
:: Person  = {name :: String, age :: Int, gender :: Gender}
:: Gender  = Male | Female
:: Error :== String

parseAge :: String -> Either Error Int
parseAge ""               = Left "Nothing to parse as age"
parseAge str
  # x = toInt str
  | x == 0 && str <> "0" = Left (str +++ " is not a valid age")
  | otherwise             = Right x

parseGender :: String -> Either Error Gender
parseGender "m" = Right Male
parseGender "f" = Right Female
parseGender  _  = Left "Unknown gender"

parsePerson :: (String, String, String) -> Either Error Person
parsePerson (n, a, g) =
  case parseAge a of
    Left e    -> Left e
    Right age ->
      case parseGender g of
        Left e       -> Left e
        Right gender -> Right {name = n, age = age, gender = gender}
```


fp2.parsePerson :: Either Error Person
(n, a, g) :: (String, String, String)
parseAge
a
age
parseGender
g
gender
pure
{ Person | n, age, gender }

[1] Comma Separated Values

2. Describe in your own words how the follwoing program is evaluated and what its result is.

   ```
   Start = map parsePerson
             [ (”Alice”, ”37”,  ”f”)
             , (”Bob”,   ”2.5”, ”m”)
             , (”Carol”, ”18”,  ”o”)
             , (”Dave”,  ””,    ”f”) ]
   ```

3. Refactor this example using the monadic `(>>=)` and `pure` functions.

4. Bonus: Show that your instance fulfils the monad laws.

4 The List monad

In this section we will study the movements of a knight piece on a chess board. We will model the position of the knight with a pair of integers:

```
:: Position :== (Int, Int)
```

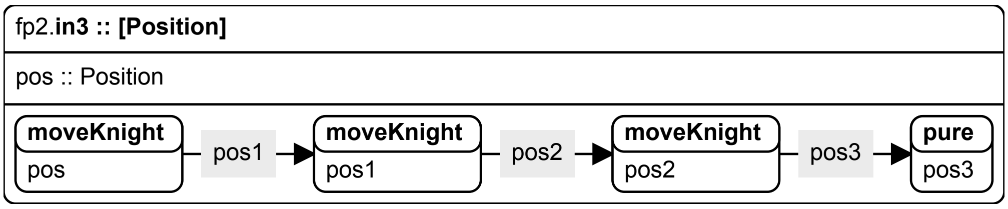
and calculating the possible moves of a knight on the chess board using:

```
moveKnight :: Position -> [Position]
moveKnight (x, y) = filter onBoard
  [ (x+2, y-1), (x+2, y+1), (x-2, y-1), (x-2, y+1)
  , (x+1, y-2), (x+1, y+2), (x-1, y-2), (x-1, y+2)
  ]
```

A chess board consists of just 8 rows and 8 columns. The application of `filter onBoard` in the definition of `moveKnight` removes any positions that may lie outside of the board's boundaries:

```
onBoard :: Position -> Bool
onBoard (x,y) = 0 < x && x < 9 &&
                0 < y && y < 9
```

1. Implement a function `in3 :: Position -> [Position]` that calculates the reachable positions of a knight after **three** moves. You may use any function from `StdEnv`.



The above exercise is an example of **non-deterministic** calculations. In this example we're not sure which move the player will make: we take a look into future **possible** moves. After that, we calculate possible moves out of possible moves, etcetera. The list monad conceptualises non deterministic calculations.

2. Take a look at your implementation of `in3`. Can you see a recurring pattern? Use your observations to implement the `Monad` instance for lists.
3. Refactor your `in3` function using the `(>>=)` and `pure` functions.
4. Bonus: Show that your instance fulfils the monad laws.

5 The State monad

The `State` monad can pass around any state `s` "under water". Instead of the simple type synonym we used at the lecture, now we use a real data type which wraps a function:

```
:: State s a = State (s -> (a, s))
```

In this way we can make `State` an instance of the `Monad` class, the compiler would complain if we had used a type synonym. Be aware that you have to **wrap and unwrap** the `State` constructor to use this definition for your stateful functions.

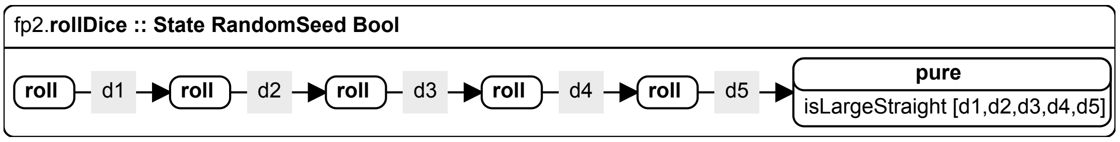
1. Implement the `Monad` instance for `State`. Get inspired by the instance of `IO` we discussed during the lecture.

You are playing a game of Yahtzee. You almost won. You only need a Large Straight. Unfortunately, your dog ate all but one of the dice, so you will have to roll your remaining dice five times. We use a function `random` like you used last week.

```
roll :: RandomSeed -> (Int, RandomSeed)
roll s
  # (x, seed) = random s
  = (x rem 6 + 1, seed)

isLargeStraight :: [Int] -> Bool
isLargeStraight dices = sorted == [1, 2, 3, 4, 5] ||
                        sorted == [2, 3, 4, 5, 6]
                        where sorted = sort dices

rollDice :: RandomSeed -> (Bool, RandomSeed)
rollDice seed
  # (d1, seed) = roll seed
  # (d2, seed) = roll seed
  # (d3, seed) = roll seed
  # (d4, seed) = roll seed
  # (d5, seed) = roll seed
  = (isLargeStraight [d1, d2, d3, d4, d5], seed)
```


fp2.rollDice :: State RandomSeed Bool
roll
d1
roll
d2
roll
d3
roll
d4
roll
d5
pure
isLargeStraight [d1,d2,d3,d4,d5]

While this code works as expected, it is tedious to manually pass the seed around all the time. We can use a state monad to do this for us.

2. Refactor `rollDice` in terms of `(>>=)` and `pure`, using the `State` monad. You may use the functions given below. Remember you should wrap and unwrap stateful functions.

```
eval :: (State s a) s -> a
eval (State f) s
  # (x, _) = f s
  = x

state :: (s -> (a, s)) -> State s a
state f = State f
```

3. Bonus: Show that your monad instance for State fulfils the monad laws.