Declaratieve Talen

Haskell 3

1 Maybe Monad

In this exercise you'll be using the `Maybe` monad, one of the easiest monads to grok, to familiarise yourself with the `(>>=)`-operator (bind) and the `do`-notation.

Given a list of `(String, Int)` pairs, where the `String` is the variable name and the `Int` is the value assigned to that variable, write a function `sumABC :: [(String, Int)] -> Maybe Int` that calculates the sum of the values assigned to the variables `"A"`, `"B"`, and `"C"`. The return type of the function is `Maybe Int` and not `Int`, because it is possible that the list of variables is missing some assignments (for instance when the list is empty). Only if all variable assignments are available in the list, a `Just` with the sum should be returned, otherwise `Nothing`.

Use the `lookup :: Eq a => a -> [(a, b)] -> Maybe b` function to find assignments in the list.

1. Write the function `sumABC` by pattern matching on the results of the lookups using nested `case ... of`s.
2. Rewrite the function (call it `sumABCBind`) by using the monadic `(>>=) :: m a -> (a -> m b) -> m b` operator (bind) and `return :: a -> m a`. If you don't know how to start, have a look at the lecture slides or go to the **hint** at the end of this exercise.
3. Rewrite the function (call it `sumABCDo`) by using the `do`-notation (and `return :: a -> m a`).

Examples The examples below should work with `sumABCBind` and `sumABCDo` too.

```
> sumABC []
Nothing

> sumABC [("A", 1), ("B", 2)]
Nothing

> sumABC [("C", 100), ("A", 1), ("B", 2)]
Just 103
```

Hint Remember that all three expressions below (where `mb` is a `Maybe Int` and `x` will be the `Int` inside the `Maybe`) are equivalent:

```
-- Pattern matching
case mb of
    Just x -> ...
    Nothing -> Nothing

-- Bind-operator
mb >>= \x -> ...

-- Do-notation
do x <- mb
   ...
```

2 Functors

Write `Functor` instances for the following three data types (don't forget to look at the **hint** at the end of this exercise!)

- Write a `Functor` instance for `Identity` which just wraps a value.

    ```
    data Identity a = Identity a deriving Show
    ```

    Example

    ```
    > fmap not (Identity False)
    Identity True
    ```

- Write a `Functor` instance for `Pair a` which combines two types. Note that `fmap` only looks at the second element of the pair.

    ```
    data Pair a b = Pair a b deriving Show
    ```

    Example

    ```
    > fmap length (Pair 'z' "zet")
    Pair 'z' 3
    ```

- Write a `Functor` instance for `Unit` which contains no value, so its type parameter can be chosen freely, while the same constructor can be used.

    ```
    data Unit a = Unit deriving Show
    ```

    Example

    ```
    > fmap (++ [True]) Unit
    Unit
    > fmap length Unit
    Unit
    ```

Hint Recall that a `Functor` instance has the following form (where everything between `<...>` must be replaced):

```
instance Functor <type> where
    fmap f (<constructor> <0 or more arguments>) = <result>
```

For example:

```
instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

3 MayFail Monad

The following datatype `MayFail` can be used as a result for computations that may fail to produce a well-defined result. It is a simple extension of the `Maybe` type, in the sense that in case of failure, a value of type `e` is returned that explains the failure. `safeDiv`, for example, uses it to report a division by zero.

```
data MayFail e a = Error e | Result a
  deriving (Show)

safeDiv :: Int -> Int -> MayFail String Int
safeDiv a b
  | b == 0    = Error "Division by zero"
  | otherwise = Result (div a b)
```

Failure of computations is a side-effect that can be handled using Monads.

- Write a `Functor` instance for the `MayFail e` type.
- Write a `Monad` instance for the `MayFail e` type.
- Implement an evaluator `eval` for the expression datatype `Exp` that correctly reports division by zero. Do not use syntactic sugar (`do`-notation).

  ```
  data Exp = Lit Int | Add Exp Exp | Mul Exp Exp | Div Exp Exp
    deriving (Show)

  eval :: Exp -> MayFail String Int
  ```

  Example

  ```
  ghci> eval (Add (Lit 1) (Lit 3))
  Result 4
  ghci> eval (Add (Div (Lit 3) (Lit 0)) (Lit 1))
  Error "Division by zero"
  ```

- Implement the evaluator again, this time using do-notation.

Note: since `MayFail` generalises the `Maybe` type, you might find it useful to look at the `Monad` instance for the `Maybe` type when you define the `Monad` instance for `MayFail e`.

```
instance Monad Maybe where
  return  = Just
  m >>= f = case m of
              Nothing -> Nothing
              Just x  -> f x
```

4 Writer Monad

Whereas `MayFail` is used for values with an added context of failure and the `State` monad is used for representing state-passing, the `Writer` monad is used for values that have an additional value attached, that acts like a log value. Hence, one of the many uses of the `Writer` monad is for *tracing*.

- For the (simplified) type `Exp`[1], write a function `evalTrace` that evaluates an expression, while keeping track of the constructors that are used in the expression in a newline-separated string:

  ```
  data Exp = Lit Int | Add Exp Exp | Mul Exp Exp
    deriving (Show)

  evalTrace :: Exp -> (Int, String)
  ```

  Example

  ```
  ghci> evalTrace (Add (Lit 1) (Mul (Lit 2) (Lit 3)))
  (7, "Add\nLit\nMul\nLit\nLit\n")
  ```

- Write a `Functor` instance for the type `Writer`.

  ```
  data Writer a = Writer a String
    deriving (Show)
  ```

- Write a `Monad` instance for the type `Writer` (Tip: For the bind function (`>>=`) try to figure out the repeated pattern and abstract over it, like we did in the class for the `Maybe` instance).

- Implement a function `trace` that logs a string:

[1] Make this exercise in a new file so that it doesn't clash with last exercise's definition of `Exp`.

```
trace :: String -> Writer ()
```

- Reimplement function `evalTrace` in a monadic style, using the `Writer` type instead of a tuple.

  ```
  evalTraceM :: Exp -> Writer Int
  ```

  Example

  ```
  ghci> evalTraceM (Add (Lit 1) (Mul (Lit 2) (Lit 3)))
  Writer 7 "Add\nLit\nMul\nLit\nLit\n"
  ```

5 Implementation of Monadic Functions

The `Prelude` and in particular the `Control.Monad`[2] module export many useful operations over monads. For instance, the `sequence` combinator:

```
sequence :: Monad m => [m a] -> m [a]
```

that evaluates each computation from left to right and returns the results.

Examples

In the examples, we use the `putStrLn :: String -> IO ()` function which prints a string (including a newline) and returns `()`. Because this function has a side effect, i.e. printing, it is in the `IO` monad.

```
> sequence [putStrLn "hello", putStrLn "world"] :: IO [()]
hello
world
[(), ()]
> sequence [Just 3, Just 5, Just 2] :: Maybe [Int]
Just [3,5,2]
> sequence [Nothing, Just 5, Just 2]
Nothing
```

- Now reimplement `sequence` yourself as `sequence'` (so it doesn't conflict with the predefined version which you may of course not use in the reimplementation).

There are also monadic variants of the pure `map`, `zipWith` and `replicate` combinators.

```
mapM       :: Monad m => (a -> m b) -> [a] -> m [b]
zipWithM   :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
replicateM :: Monad m => Int -> m a -> m [a]
```

[2] See the documentation at `http://hackage.haskell.org/package/base-4.6.0.1/docs/Control-Monad.html`, add `import Control.Monad` to the top of your file to use it.

Examples

```
> mapM putStrLn ["hello", "world"]
hello
world
[(),()]
> zipWithM (\x y -> putStrLn (show (x + y))) [1, 2] [100, 1000]
101
1002
[(),()]
> replicateM 3 (putStrLn "hello world")
hello world
hello world
hello world
[(),(),()]
```

- Reimplement `mapM`, `zipWithM`, and `replicateM` (add a `'` to their names) in terms of `sequence` and the pure variants. Use the examples above to guide your implementation.

6 Alternative Approaches to Monads

In the lecture we saw that a monad can be viewed as a container or a way to represent side effects. Now you have to show that these approaches are essentially equivalent.

- In the lecture we defined `State` as a monad, by providing functions `return` and `>>=` (side-effect approach). Implement functions `fmap`, `unit` and `join` for the `State` monad (container approach).

```
fmapState :: (a -> b) -> State s a -> State s b
unitState :: a -> State s a
joinState :: State s (State s a) -> State s a
```

- In the lecture we also defined list as a monad, by providing functions `fmap`, `unit` and `join` (container approach). Implement functions `return` and `>>=` for the list monad (side-effect approach):

```
returnList :: a -> [a]
bindList   :: [a] -> (a -> [b]) -> [b]
```

- Implement functions `fmap`, `unit` and `join`, in terms of `return` and `>>=`, for any monad `m`.

```
fmap' :: Monad m => (a -> b) -> m a -> m b
unit' :: Monad m => a -> m a
join' :: Monad m => m (m a) -> m a
```

- Implement functions `return` and bind (`>>=`), in terms of `fmap`, `unit` and `join`, for any monad `m`.

  ```
  return' :: Monad m => a -> m a
  bind'   :: Monad m => m a -> (a -> m b) -> m b
  ```

Note: On older GHC version (< 7.10), *such as those in the PC-labs*, a monad `m` does not need to be an instance of the `Functor` class. Hence, you may have to qualify some of the types of the above functions with an additional `Functor m` constraint in places where you use the `fmap` function.

7 Optional: Writer & MayFail Monads

In practice, it is common to use multiple effects in a program. For example, the following datatype can be used to represent computations that can fail and also trace the computation (i.e., it combines the `MayFail` and the `Writer` monads).

```
data Log e a = Error e | Result a String
  deriving (Show)
```

- Write a `Functor` instance for the `Log` type.
- Write a `Monad` instance for the `Log` type.
- Write an evaluator `evalLog` for the `Exp` type of the first exercise that evaluates an expression, correctly reports division by zero, and keeps log of the constructors used in the expression.

  ```
  evalLog :: Exp -> Log String Int
  ```

  Example

  ```
  ghci> evalLog (Div (Lit 1) (Lit 0))
  Error "Division by zero"

  ghci> evalLog (Add (Lit 1) (Mul (Lit 2) (Lit 3)))
  Result 7 "Add\nLit\nMul\nLit\nLit\n"
  ```

- We could also have defined `Log` as

  ```
  data Log' e a = Log' (MayFail e a) String
  ```

  Would that make a difference?