Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Functional Programming

WS 2019/20

Benjamin Dietrich, Denis Hirn

Assignment #10

Submission Deadline: Thu, 23.1.2020

Exercise 1: Exceptions with Monadic Either (4 Points)

In `sequence-Either.hs` you can find function `chinese'` which translates English numerals to Chinese digits. It makes use of the *Kleisli* composition `(>=>)` for *exception-generating functions* of type `Exc b` to propagate exceptions of type `Error` thrown by any of the functions it is composed of.

Instead of using `(>=>)` we could also provide a `Monad` instance for `Exc` and the `bind` operator `(>>=)` to compose `chinese'` from `numeralToDigit`, `digitToVal` and `chineseNumeral`.

Open file `monadic-Either.hs` and complete the missing parts:

1. Implement `Functor`, `Applicative` and `Monad` instances for the type `data Exc a = Exc Error | Val a`
2. Implement `chinese'` using `(>>=)` instead of `(>=>)`.

Exercise 2: Desugaring of Monadic Code (6 Points)

In Haskell and other pure functional languages, functions can only interact with the arguments passed to them. A global state or global variables do not exist.

The `Reader` monad (also called the Environment monad) is the standard way to handle functions which need to read an environment. Such an environment often contains configuration or context information. In order to access this information deep inside of a program, we could use a normal function parameter. However, that would lead to a lot of code restructuring, because every intermediate function needs to get an additional parameter.

In contrast to the `Maybe` and `Either` monads, which are defined in terms of an explicit data type, the `Reader` monad takes the form of a function instead: `newtype Reader e a = R { runReader :: e -> a }`.

This effectively adds an environment parameter to all functions that are defined using the `Reader` monad. The bind operator >>= of this monad saves us from explicitily passing that additional environment argument around. If a function needs access to the environment, it can explicitly invoke `ask` to do so.

In this task, you are provided with a monadic interpreter for a simple language with numbers, addition and variables. You have to perform a sequence of refactorings in order to obtain a semantically equivalent version of the interpreter, that reveals the formerly hidden environment passing.

Proceed as follows:

1. Complete `evalInline` by inlining the given definitions of `ask`, `return` and `bind (>>=)`. That is, replace calls to functions with their body, replacing occurrences of formal parameters with the corresponding actual arguments.

   Hint: The following example shows the original monadic and inlined versions of `bind (>>=)` in case of the `Maybe` monad:

```
instance Monad Maybe where
  return x = Just x

  (>>=) m g = case m of
                Nothing -> Nothing
                Just x  -> g x

chinese' :: String -> Maybe Char
chinese' n = numeralToDigit n >>= digitToVal >>= chineseNumeral

-- Inlining of bind (>>=)
chinese'' :: String -> Maybe Char
chinese'' n = case numeralToDigit n of
                Nothing -> Nothing
                Just x  -> case digitToVal x of
                                Nothing -> Nothing
                                Just y  -> case chineseNumeral y of
                                                Nothing -> Nothing
                                                Just z  -> Just z
```

2. Complete `evalDesugar` by removing all usages of the data type `Reader` in `evalInline`.

3. Complete `evalFinalSimplified` by reducing function applications in `evalDesugar`.

After each of the three steps, make sure that you obtain a syntactically valid Haskell program that still computes the same result as the original program we provided.

Exercise 3: Guessing Numbers (10 Points)

With monads, we finally have the tools to express and combine computations which perform I/O. We will implement a simple interactive application. The interaction is defined by the following rules:

1. The computer picks a random number between 1 and 2^{10}.
2. The *player* has 13 attempts to guess the right number.
3. After every wrong guess, the computer tells whether the guessed number was too small or too large.
4. If the player can not guess the correct number in 13 attempts, he loses and the computer makes fun of him/her.
5. If the player manages to guess the correct number, he/she wins the game.

Example interactions in GHCi

```
*Guess> main
Guess a number:
invalid
Guess a number:
512
Your guess is too large!
Guess a number:
42
You won within 2 attempts.

*Guess> main
Guess a number:
1
Your guess is too small!
Guess a number:
2
Your guess is too small!
Guess a number:
3
Your guess is too small!
[...]
Guess a number:
14
Your guess is too small!
You ran out of attempts. Go back to your algorithms class!
```

Open `Guess.hs` and complete the missing function definitions to implement the game according to the rules described above:

- Function `prompt :: IO Int` – Ask the player for a guess. Parse his input string (use function `Text.Read.readMaybe`). If the input is invalid, ask again until a valid input is provided. Otherwise return the guessed number. A basic example how to use I/O combinators `getLine` and `putStrLn` is given below.
- Function `hint :: Int -> Int -> IO ()` – Given a target number and a guess, print a hint to the player whether his guess is to small, to large or correct.
- Function `gameLoop :: Guess Bool` – Interact with the player and finally return whether the player won or not. Ask the player for a guess. If he is correct or is out of attempts, end the game. Otherwise enter the loop again.

- Function `main :: IO ()` – Pick a random number (use function `System.Random.randomRIO`[1]) and enter the game loop with an initial `GameState`.

Example how to use basic I/O combinators, namely `getLine` and `putStrLn` (you can also find it in `io-example.hs`):

```
check :: String -> String -> IO Bool
check q a = putStrLn (q ++ "Wait...") >>
            return (a == "42")

main :: IO ()
main =
  putStrLn "Please give a question:" >>
  getLine >>= \q ->
  putStrLn "And an answer:" >>
  getLine >>= \a ->
  if last q == '?'
    then check q a >>= \res ->
         putStrLn $ "Your answer is " ++ show res
    else putStrLn "Your question should end with an ?. Retry!" >>
         main
```

[1] To import `System.Random` you may have to install package `random` first: `cabal install --lib random` or `stack install random`