Prof. Dr. Peter Thiemann
Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23
Issued 2022-12-12
Discussion on 2022-12-16

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exercise Sheet 8

Exercise 1 (Template rendering)

The file `Template.hs` linked on the course homepage contains the definition of the **`Template`** data type. A web server, for example, might contain a template rendering engine to produce dynamic HTML. The engine would parse the template file into a representation similar to **`Template`**.

The goal of this exercise is to write a rendering function, that is a function to turn **`Template`** values into **`String`**s. The renderer will make use of the function arrow monad `(->) e`.[1] The type variable `e` stands for an *environment* which is passed to each computation. This monad is also called the *reader* monad.

1. Write the supporting functions

   ```
   lookupVariable  :: String -> Environment -> Maybe String
   lookupTemplate  :: String -> Environment -> Maybe Template
   updateVariables :: [(String, String)] -> Environment -> Environment
   ```

   Which of these functions can be used as actions inside the `(->) Environment` monad? How many arguments do they have to be applied to to become valid actions?

2. A reader monad does not keep state between consecutive actions. But we can write a function to adapt an action so that it receives a modified environment. Write the function

   ```
   local :: (e -> e') -> (e' -> a) -> (e -> a)
   ```

   Rewrite the second argument's arrow to prefix form. Do the same for the return type.

3. Implement the two functions

   ```
   resolveDef :: Definition -> Environment -> (String, String)
   resolve    :: Template   -> Environment -> String
   ```

   Write these using the `(->) Environment` monad. That is, don't bind the **`Environment`** argument to a variable.

Exercise 2 (Joining monads)

In the future we will be able to give a different, but equally powerful, definition for monads using `join` from `Control.Monad` and another function we will learn later. `join` has some interesting properties by itself:

1. Implement `join :: Monad m => m (m a) -> m a`.
2. What type do you get when you instantiate `m` to some function monad? How does the result behave? Write QuickCheck properties to demonstrate.
3. What is the type and behaviour of `join (.)`? Write QuickCheck properties to demonstrate.

[1] The arrow operator `->` has a prefix form just like any other operator in Haskell. The type `e -> a` is equivalent to the type `(->) e a`. If we're talking about the monad, as we do here, we have to use the latter version so that we can construct a type of kind `* -> *`.

Exercise 3 (Random number generator)

A common way to generate pseudo-random numbers is through the use of a series. For example, Donald Knuth in The Art of Computer Programming presents the following series:

$$x_n = (6364136223846793005 * x_{n-1} + 1442695040888963407) \mod 2^{64}$$

This style of a pseudo-random number generator is known as a linear congruential generator.

It can easily be implemented using a state monad. Instead of writing our own type and instances we will now make use of a library which provides everything we need. Add the transformers package to your dependencies. You can find the **State** monad and its documentation in `Control.Monad.Trans.State.Strict`. For example, to increment the state and return the old value we can write

```
incr :: State Integer Integer
incr = do
  n <- get
  put (n + 1)
  return n
```

Use the above formula to implement the following API:

```
type Random a = State Integer a
fresh :: Random Integer
runPRNG :: Random a -> Integer -> a
```