Interactive Programming in Haskell

In this section we discuss the challenge of creating interactive applications in Haskell, and how Haskell addresses the problem.

Reading

- Sections 10.1-10.5

- Practice exercises (10.10): 1, 4, 5, 6

Interactive Programming in a Pure World

In Haskell all functions are **pure**, which means that their outputs are completely determined by their inputs. This is a wonderful feature that provides numerous guarantees. We can more easily argue about the correctness of a program if we know exactly what the results will be.

By contrast, interacting with the user requires that we break this purity. If the user is expected to type some string in, how can our function return a predictable result? And what about functions that print something to the screen? Each time we run the function, a new "effect" is produced, as a new string is printed to the screen. This is in addition to and distinct from whatever return value the function might produce.

So the question becomes how to represent such a situation, when our language does not allow such "side-effects". The answer is that we imagine the rest of the system as one big structure, of type World. A value of this type represents the state of the system, including what files are open, what is shown on the screen via the standard output, what has been read from the keyboard/standard input, and so on. An action that interacts with the user/system in some way takes us from one state of the world to a new state.

Actions

As an example, imagine the act of reading a single character from the standard input. We could imagine this act as a function:

```
getChar :: World -> (Char, World)
getChar w1 = .... read a char and return that char and the new state of the world.
```

Similarly we could imagine a function that prints a character. This takes as input a character, and also the current world state, and returns an updated world state (plus an empty tuple as a meaningless result — all functions must return a result and the updated world is not enough).

```
putChar :: Char -> World -> ((), World)
putChar c w1 = ((), w2)   -- w2 is the updated world state with the character printed
```

We have a specific type to represent these "functions":

```
type IO a = World -> (a, World)
```

So a value of type IO a interacts with the world in some way and also produces a value of type a. We can now write the types of the two functions above as:

```
getChar :: IO Char
putChar :: Char -> IO ()
```

Values of type IO a are called **actions**. They are "impure" in the sense that their results depend on external factors such as user interaction, and as such they cannot be used inside of "pure" functions. What this means in practical terms is that we cannot get the value of type a completely out of its IO context: There is no function of type IO a -> a that we can make. Once you are in IO you are stuck there.

So the main idea of working with interactive programming is as follows:

- Put as much of your code as possible in pure functions.

- Use small IO actions where you need user interaction.

The Do notation: Combining Actions

In order to write more complicated actions we need a way of combining primitive actions together. The do notation is very handy there. The do notation looks like this:

```
myNewAction = do
    action1
    x <- action2
    action3
    action4
```

So it consists of a series of actions, and it executes these actions in order. An action could instead have an assignment like x <- action2 above, meaning that the result of the action should be stored in the variable x, which then becomes available to the followup actions (action3 and action4 in the preceding example).

For example, let's write an action getLine. It reads characters until it encounters the newline character. It then creates a string from all those characters. It has type IO String. Here is the logic:

- We read a character.

- If the character is not the newline, then we recursively call ourselves to read the rest of the line, then prepend our character to the front.

- If the character is the newline, then we need to return an empty list/string, so that the process of prepending characters can begin.

In the above a key step is to be able to "return" a specific value, like a string. This means that we need to turn an actual value into an *action* that does not change the world but that contains this value as its value. This is done by a function aptly called return:

```
return :: a -> IO a
return v = \world -> (v, world)    -- Imaginary definition if IO a was really World ->...
```

The function return simply takes a value and turns it into the trivial action.

Now let's look at the code for our getLine function:

```
getLine :: IO String
getLine = do
    c <- getChar
    if c == '\n'
        then return ""
        else do
            cs <- getLine
            return (c:cs)
```

So this example consists of 2 actions. The first is a call to getChar. The second action is the result of the if−then−else call, which depends on the character c. It is the return "" action if the character is the newline character, or it is an action defined via a do as the combination of two other actions, for any other character.

Let us similarly write a function that prints a string. It must take as input a string, and returns an action with no value.

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (c:cs) = do putChar c
                   putStr cs
```

There is also a version that follows the string with a newline:

```
putStrLn :: String -> IO ()
putStrLn s = do putStr s
                putChar '\n'
```

All these are already implemented in Haskell, but even if they were we could have easily added them.

Before we move on, let us consider another useful function, called sequence_. It has type:

```
sequence_ :: [IO a] -> IO ()
```

It takes a list of actions, and it performs them in sequence, ignoring their results. It could be implemented as follows:

```
sequence_ []       = return ()
sequence_ (ac:acs) = do ac
                        sequence_ acs
```

You should notice a similarity between this function and putStr earlier. In fact, think of how you could implement putStr using sequence_ and a list comprehension (or map) to form a list of actions out of a list of characters.

Practice

1. Implement putStr using sequence_. The types should help guide you in the process.

2. Implement a function getUntil :: (Char –> Bool) –> IO Char –> IO String that keeps collecting characters as long as the predicate is True. It then returns the concatenation of those characters.

3. Use the function getUntil to provide an alternative implementation of getLine.

4. Implement the action sequence :: [IO a] –> IO [a]. It takes a list of actions, performs them all, and collects the results into one list.

5. Implement the action (<$>) :: (a –> b) –> IO a –> IO b. It performs the action, then passes it through the function before returning it.