Functional Programming

3. Input/Output and Commands

Frank C Langbein
frank@langbein.org

Version 1.4.0

Mind vs Body

- Every function in Haskell is **"pure"**
  - It has no **side-effects**
  - With side-effects, the **order of evaluation** matters!
- Pure functions have a problem, though
  - How does Haskell actually **do something**?
- **Mind (thoughts) vs Actions (body)**
  - Haskell has got something to link thinking (computation) and acting

---

Commands

- **Print a character**

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

    - The command to print an exclamation mark, **if it is ever performed**

    ```
    putChar '!'
    ```

- `IO ()` is the type of **commands**
    - `()` is a 0-tuple (the only 0-tuple there is)
    - `putChar` **yields** a command, it **does not perform** the command
    - This is purely functional

Combining commands

```
(>>) :: IO () -> IO () -> IO () -- "then"
```

- For example

    ```
    putChar '?' >> putChar '!'
    ```

    - Represents the command that prints a question mark followed by an exclamation mark
    - **IF IT IS EVER PERFORMED**

```
done :: IO ()
```

- **`done` is the command**, that if it is actually ever performed, **will not do anything**.
    - Thinking about doing nothing vs. actually doing nothing
    - These are two different things!
- **'>>' constructs sequences of commands** (not too different from other operators, e.g. '++')

Performing a command

- The main function links thinking with performing (mind-body)

```haskell
main :: IO ()
main = putChar '?' >> putChar '!' >> putChar '\n'
```

- Note, there is `putStr` and `putStrLn` which creates a command to print a string (with or without newline)
- Also note, `putStr` takes a string only

```haskell
print x = putStr (show x)
```

- show converts anything (it knows about) to a string

Side-effects - Equational Reasoning Lost

- Assume a language with side effects that prints "haha"

```
print "ha"; print "ha";
```

- So this program only prints "ha" as side effect

```
let x = print "ha" in x; x    -- the side effect, not the value is relevant
```

- But this prints "haha" as side effect

```
let f () = print "ha" in f (); f ()    -- () is an evaluation
```

- THIS IS NOT HASKELL!
- With side-effects: each evaluation changes the state / executes IO / etc
- Without side-effects: no state needed; the expression and its result are equivalent

---

Equational Reasoning Regained - no State

- In Hasell `(1+2) * (1+2)` and `let x = 1 + 2 in x * x` are equivalent.
- Similarly, in Haskell

  ```
  putStr "ha" >> putStr "ha"
  ```

  and

  ```
  let m = putStr "ha" in m >> m
  ```

  are equivalent!
- The simple equivalence rule works in Haskell, even with commands that involve printing
  - **You can always use a variable to factor out a common sub-expression without changing the meaning**
  - A variable cannot actually be modified but is a symbol bound to an expression

---

Commands with Values

- A command to read a character

  ```
  getChar :: IO Char
  ```

  - `IO Char` indicates that this is a **command that yields a value of type `Char`**
  - **Performing** the command `getChar` on the input "abc" yields the value `'a'` and the remaining input "bc"
- Do nothing and return a value (similar to done)

  ```
  return :: a -> IO a
  ```

  - This performs the command

  ```
  return [] : IO String
  ```

  - When the input contains "bc" this yields the value `[]` and the unchanged input "bc"

Combining Commands with Values

```
(>>=) :: IO a -> (a -> IO b) -> IO b -- bind
```

- For example, performing the command

  ```
  getChar >>= \x -> putChar (toUpper x)
  ```

  - When the input is "abc" this produces the output "A" and the remaining input is "bc"
  - (This needs to import Data.Char to work)

Bind in Detail

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- If

  ```
  m :: IO a
  ```

  is a command yielding a value of type a, and

  ```
  k :: a -> IO b
  ```

  is a function from a value of type a to a command yielding a value of type `b`, then

  ```
  m >>= k :: IO b
  ```

  is the command that, if it is ever performed, behaves as follows:

  - first perform command `m` yielding a vlaue `x` of type a
  - then perform command `k x` yielding a value y of type `b`
  - then yield the final value y

General Operations on Commands

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- The command done is a special case of `return`

  ```
  done :: IO ()
  done = return ()
  ```

- The operator >> is a special case of >>=

  ```
  (>>) :: IO () -> IO () -> IO ()
  m >> n = m >>= \() -> n
  ```

- This starts to look like a pattern?! (more later)
  - `(>>=,return)` and `(>>,done)` similar to `(*,1)`, `(+,0)`, `(++,[])` etc.

Do Notation - getLine : States and Imperative Programming

- Reading a line

```
getLine :: IO String
getLine = getChar >>= \x -> if x == `\n` then
                                return []
                              else
                                getLine >>= \xs -> return (x:xs)
```

- In "do" notation (~imperative programming)

```haskell
getLine' :: IO String
getLine' = do {
               x <- getChar;
               if x == `\n` then
                 return []
               else do {
                 xs <- getLine;
                 return (x:xs)
               }
             }
```

- A way to define **imperative programming / computing with states**:
  - Each line `x <- e; (...)` becomes `e >>= \x -> (...)`
  - Each line `e; (...)` becomes `e >> (...)`