Computer Science 312

I/O and Side Effects

Functional “Programming”

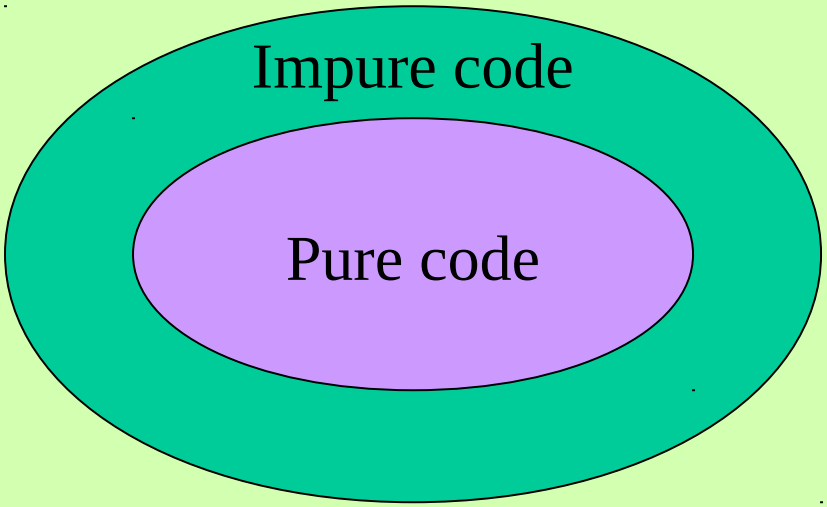
- Structure code in terms of cooperating functions and data types
- Computation is a transformation of values into other values
- This occurs via the evaluation of expressions, among which are function applications
- It’s a pure universe: nothing ever really changes, no side effects

Where Is the Program?

- Pure function applications are not really programs
- A real program takes input from the outside world, processes it, and sends output back to the outside world
- The parts of a program that simply perform computations on data might be pure functions, but the parts that deal with the outside world, or with changes of state more generally, cannot be pure in that sense

Isolate the Pure from the Impure

External world (files, I/O devices, network ports, etc.)



Example: Output with **`putStrLn`**

Like Python's **`print`**:

```
Prelude> putStrLn "3 more years of Trump!"
3 more years of Trump!

Prelude> :type putStrLn
putStrLn :: String -> IO ()
```

Takes a **`String`** as an argument, prints it, and returns nothing!

`IO` is a type class, and **`()`** is the empty type

Run only for its side effect, producing output on the terminal

Example: Input with **`getLine`**

Like Python's **`input`**:

```
Prelude> getLine   -- Input is in italic on next line
Some input text
"Some input text"
Prelude> :type getLine
getLine :: IO String
```

Takes no arguments, waits for input, and returns the string that the user enters at the keyboard

`IO` is a type class for types associated with I/O functions

Different calls can produce different results (not pure!)

Actions as Impure Functions

- **`putStrLn`** and **`getLine`** express ***actions***, which produce side effects

- They are ***impure***, in that they can produce different values with the same arguments

Extracting a Value from an Action

```
Prelude> name <- getLine   -- Input is in italic on next line
Ken
Prelude> putStrLn name ++ " Lambert"
Ken Lambert
```

Use the **<-** operator, not the **=** operator, to extract a value from an action and bind a variable to it

A Standalone Program

- Defines a function **`main IO ()`**
- Runs a sequence of actions in a **`do`** block
- These actions can take inputs, process them, and output results

Example: Entering One's Name

```
-- SimpleIO.hs

module Main (main) where

main :: IO ()
main = do
    putStr "Enter your first name: "
    firstName <- getLine
    putStr "Enter your last name: "
    lastName <- getLine
    let fullName = firstName ++ " " ++ lastName
    putStrLn ("Your full name is " ++ fullName)
```

A **do** block allows you to run a sequence of actions and pure function applications, such as **let**

Run in the GHCI to Test

```
Prelude> :load SimpleIO
[1 of 1] Compiling Main
Ok, one module loaded.

*Main> main
Enter your first name: Ken
Enter your last name: Lambert
Your full name is Ken Lambert
```

Compile Standalone to Deploy

```
$ ghc --make SimpleIO
[1 of 1] Compiling Main             ( SimpleIO.hs, SimpleIO.o )
Linking SimpleIO ...

$ ./SimpleIO
Ken
Lambert
Enter your first name: Enter your last name: Your full name is
Ken Lambert
```

The prompts and inputs appear to be out of sync

The terminal waits for a newline before displaying output, but the **`putStr`** function has not provided it; only the final call of **`putStrnLn`** does that

Remedy: Force Outputs with Newlines

```
-- SimpleIO.hs

module Main (main) where

main :: IO ()
main = do
    putStr "Enter your first name: "
    hFlush stdout
    firstName <- getLine
    putStr "Enter your last name: "
    hFlush stdout
    lastName <- getLine
    let fullName = firstName ++ " " ++ lastName
    putStrLn ("Your full name is " ++ fullName)
```

`hFlush` flushes the output buffer, forcing text to the terminal

Compile Standalone to Deploy

```
$ ghc --make SimpleIO
[1 of 1] Compiling Main         ( SimpleIO.hs, SimpleIO.o )
Linking SimpleIO ...

$ ./SimpleIO
Enter your first name: Ken
Enter your last name: Lambert
Your full name is Ken Lambert
```

Package I/O Details in a Function

```
module terminalIO (getString) where

getString :: String -> IO String    -- Note IO String here
getString prompt = do
    putStr prompt
    hFlush stdout
    inputString <- getLine
    return inputString
```

Like Python's **`input`** function, **`getString`** displays the prompt and waits for input

The **`<-`** operator extracts the input string from the action in **`getLine`**

The **`return`** function makes this value available to be extracted by the caller of **`getString`**

A Cleaner, Simpler I/O App

```
-- SimpleIO.hs

module Main (main) where

import TerminalIO (getString)

main :: IO ()
main = do
    firstName <- getString "Enter your first name: "
    lastName <- getString "Enter your last name: "
    let fullName = firstName ++ " " ++ lastName
    putStrLn ("Your full name is " ++ fullName)
```

The **`do`** block contains ***impure*** code for I/O and ***pure*** code (the **`let`**) for processing

Mixing the Pure and the Impure

- Impure functions (those that produce side effects) cannot be called from within pure functions
- Pure functions can be called within impure functions
- Therefore, any function that calls an impure function must also be impure, and its signature must specify this

What about Numeric Inputs?

Python:

```
>>> salary = float(input("Enter your salary: "))
Enter your salary: 42000.55
>>> salary
42000.55
```

Haskell:

```
Prelude> salaryString <- getString "Enter your salary: "
Enter your salary: 42000.55

Prelude> salary = read salaryString :: Float

Prelude> salary
42000.55
```

Package I/O Details in a Function

```
module terminalIO (getString, getFloat) where

getString :: String -> IO String    -- Note IO String here
getString prompt = do
    putStr prompt
    hFlush stdout
    inputString <- getLine
    return inputString

getFloat :: String -> IO Float      -- Note IO Float here
getFloat prompt = do
    inputString <- getString prompt
    return (read inputString :: Float)
```

A Simple Tax Calculator

```
{-File: TaxCode.hs-}

module Main (main) where

import TerminalIO (getInt, getFloat)

main :: IO ()
main = do
income <- getFloat "Enter your income: "
exemptionAmount <- getFloat "Enter the exemption amount: "
exemptions <- getInt "Enter the number of exemptions: "
taxRate <- getInt "Enter your tax rate as a percent: "
let tax = income * fromIntegral taxRate / 100.0 -

          exemptionAmount * exemptions
putStrLn ("Your tax is " ++ show tax))
```

Model/View Pattern

- Separate code that manages (transforms) the data from code that manages the I/O
- The model’s code is, for the most part, pure
- The view’s code is, for the most part, impure
- Factor these two areas of concern into different modules

Temperature Conversion: The Model

```
{-
File: Conversions.hs
Purpose: some conversion functions
-}

module Conversions (celsiusToFahr, fahrToCelsius) where

-- Temperature conversions

celsiusToFahr :: Float -> Float
celsiusToFahr degreesC = degreesC * 9 / 5 + 32

fahrToCelsius :: Float -> Float
fahrToCelsius degreesF = (degreesF - 32) * 5 / 9
```

Pure code!

Temperature Conversion: The View

```
{-
File: Conversions.hs
Purpose: some conversion functions
-}

module Main (main) where

import Conversions (fahrToCelsius)
import TerminalIO (getFloat, getString)
main :: IO ()
main = do
    degreesF <- getFloat "Enter the degrees Fahrenheit: "
    let degreesC = fahrToCelsius degreesF
    putStr "The degrees Celsius is "
    putStrLn (show degreesC)
```

Impure code!

Iterative Interaction

```
{-
File: Conversions.hs
Purpose: some conversion functions
-}

module Main (main) where

main :: IO ()
main = do
    degreesF <- getString ("Enter the degrees Fahrenheit " ++
                           "or return to quit: ")
    if degreesF == "" then
        return ()
    else do
        let degreesC = fahrToCelsius (read degreesF :: Float)
        putStr "The degrees Celsius is "
        putStrLn (show degreesC)
        main
```

Summary

- I/O functions are impure, in that they produce side effects
- Calls of pure functions can be embedded in impure functions, but not conversely
- Impure functions contain sequences of actions (in the imperative style)
- Values are extracted from actions with **`<-`**, and are made available for extraction with **`return`**

For next time

Working with files

Random numbers for non-determinism