COMP105 Lecture 23

IO

Outline

Today

- The `IO` type
- Reading and writing to the console
- `do` blocks

Relevant book chapters

- Programming In Haskell Chapter 10
- Learn You a Haskell Chapter 9

Recap: pure functions

So far, we have studied **pure** functional programming

Pure functions

- Have no side effects
- Always return a value
- Are deterministic

All **computation** can be done in pure functional programming

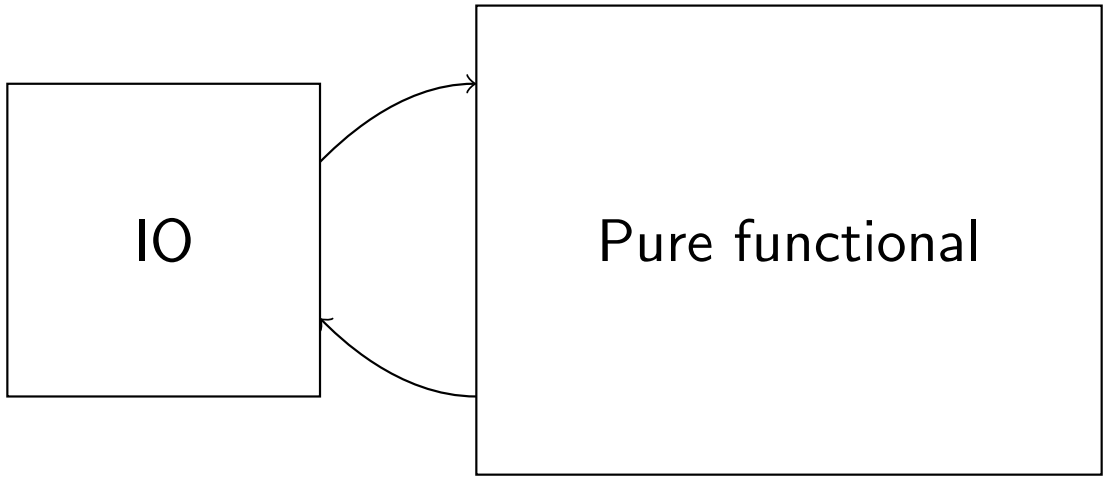
Input and output

Sometimes programs need to do **non-pure** things

- Print something to the screen
- Read or write a file
- Communicate over a network
- Create a GUI
- ...

Haskell includes mechanisms to do these impure things

IO vs Pure functional



- Impure IO code talks to the **outside world**
- Pure functional code does the **interesting computation**

IO code can call pure functions; Pure functions cannot call IO code

getLine

`getLine` reads a line of input from the console

```
ghci> getLine
hello
"hello"
```

```
ghci> :t getLine
getLine :: IO String
```

The IO type

The IO type marks a value as being **impure**

```
ghci> :t getLine
getLine :: IO String
```

```
ghci> :t getChar
getChar :: IO Char
```

If a function returns an IO type then it is impure

- It may have side effects
- It may return different values for the same inputs

The IO type

The IO type should be thought of as a **box**

- The box holds a value from an impure computation
- We can use <- to get the value out

```
ghci> x <- getLine
hello

ghci> x
"hello"

ghci> :t x
x :: String
```

The IO type

Values **must** be unboxed before they are used in pure functions

```
ghci> head (getLine)

Couldn't match expected type [a]
    with actual type IO String
```

```
ghci> x <- getLine
hello

ghci> head x
'h'
```

putStrLn

`putStrLn` prints a string onto the console

```
ghci> putStrLn "hello"
hello
```

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
```

The return type indicates that it returns nothing useful

- It has the IO type, indicating that it has a side effect

Exercise

What do these ghci queries do?

```
ghci> x <- getLine
ghci> y <- getLine
ghci> putStrLn (x ++ " " ++ y)
```

```
ghci> n <- getLine
ghci> let num = (read n) :: Int
ghci> putStrLn (show (num + 1))
```

```
ghci> putStrLn (getLine)
```

Writing our own IO code

We can write **our own** IO actions

```
print_two :: String -> String -> IO ()
print_two s1 s2 = putStrLn (s1 ++ s2)
```

```
ghci> print_two "abc" "def"
abcdef
```

Note that the return type is `IO ()`

Combining multiple IO calls

The **do** syntax allows us to combine multiple IO actions

```
get_and_print :: IO ()
get_and_print =
    do
        x <- getLine
        y <- getLine
        putStrLn (x ++ " " ++ y)
```

The do syntax

A `do` block has the following syntax

```
do
    v1 <- [IO action]
    v2 <- [IO action]
    ...
    vk <- [IO action]
    [IO action]
```

- `v1` through `vk` unbox the results of IO actions
- The final IO action is the return value

The do syntax

The `v <-` portion can be **skipped** if you don't want to unbox

```
echo_two :: IO ()
echo_two =
    do
        x <- getLine
        putStrLn x
        y <- getLine
        putStrLn y
```

The do syntax

`let` expressions can be used inside do blocks

```
add_one :: IO ()
add_one =
    do
        n <- getLine
        let num = (read n) :: Int
            out = show (num + 1)
        putStrLn out
```

This is useful to do **pure** computation between IO actions

The do syntax

`if` expressions can be used inside do blocks

```
guess :: IO ()
guess = do
    x <- getLine
    if x == "42"
        then putStrLn "correct!"
        else putStrLn "try again"
```

Both branches of the if must have the same type

`do` blocks

`do` blocks let you sequence multiple actions

- Works with IO actions
- Will **not work** in pure functional code

Functional programs consist of

- a small amount of IO code
- a large amount of pure functional code

Don't try to write your entire program in IO code!

Putting values in the IO box

Sometimes we need to put a pure value **into** `IO`

- We can use the `return` function to do this

```
ghci> :t "hello"
"hello" :: [Char]
```

```
ghci> :t return "hello"
IO [Char]
```

`return` example

```
print_if_short :: String -> IO ()
print_if_short str =
    if length str <= 2
        then putStrLn str
        else return ()
```

Both sides of the `if` must have type `IO ()`

- So we use `return ()` in the else part

return

Note that `return` does **not** stop execution

- It just converts pure values to IO values
- It is nothing like `return` from imperative languages

```
print123 =
    do
        x <- return 1
        y <- return 2
        z <- return 3
        putStrLn (show x ++ show y ++ show z)
```

Monads

The type of `return` mentions monads

```
ghci> :t return
return :: Monad m => a -> m a
```

This is because IO is a **monad**

- Whenever you see `Monad m =>` substitute IO for m
- So `return :: a -> IO a`

You **don't need to know** anything about monads for COMP105

Exercise

What does this function do?

```
import Data.Char

mystery :: IO ()
mystery =
    do
        x <- getLine
        if x == ""
        then return ()
        else do
            putStrLn (map toUpper x)
            mystery
```

Exercises

1. Write a function `get_two_lines :: IO (String, String)` reads two different lines from the console, and returns an IO tuple containing those lines.

2. Write a function `write_two_lines :: String -> String -> IO ()` that takes two strings, and prints them on different lines.

3. Write a function `io_mult2 :: IO ()` that reads a number from the console, and then prints two-times that number.

Exercises

4. Write a function `add_two :: IO ()` that reads two lines, each containing a number, and then prints out the sum of the two numbers.

5. Write a function `print_if_small :: Int -> IO ()` that takes an integer. If the integer is less than ten, it should be printed out, otherwise nothing should be printed.

6. Write a function `add_two_repeat :: IO ()` that does the same thing as `add_two`, but repeats until the user enters `""` for either of the two numbers.

Summary

- The `IO` type
- Reading and writing to the console
- `do` blocks

Next time: Writing programs