Durham University

Session 8: IO and interaction

COMP2221: Functional programming

Lawrence Mitchell[*]

[*]lawrence.mitchell@durham.ac.uk

Recap

- Discussed Haskell's implementation of expression evaluation: lazy evaluation
- Saw how lazy evaluation allows for programming with infinite data structures
- Discussed difference between *strict* and *lazy* evaluation, and how to implement strict functions in Haskell.

IO and side effects

Batch programs

- So far, we've only written *batch* programs
- That is, programs that take all their inputs at the start and provide output at the end.

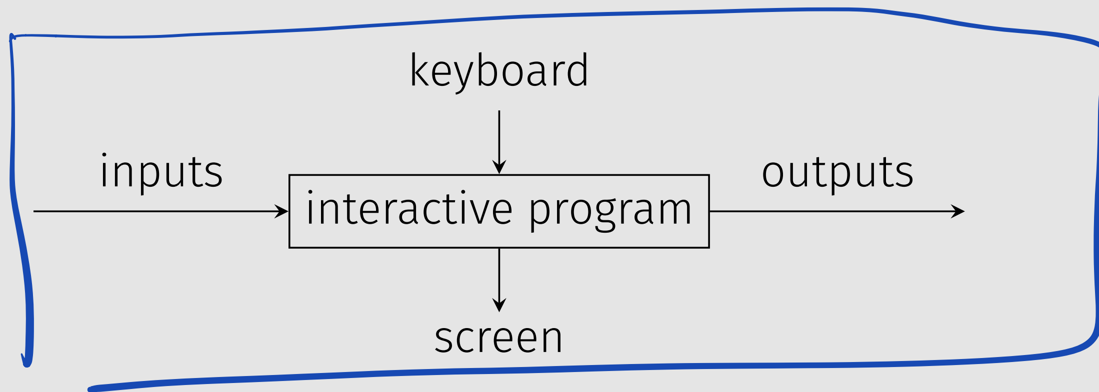
Batch programs

- To change what we compute, need to change source code and rerun.
- What if want to write programs that allow *interaction*?

Interactive programs

- What if we want to use Haskell to write interactive programs?
- These read from the “keyboard” and write to the “screen” *as they are running*

Interactive program



- Now the results can change depending on what input is provided (via the keyboard) ⇒ problem?

A problem

- Haskell programs are *pure* mathematical functions

⇒ Haskell programs therefore *have no side effects*

Definition (Side effect)

Modify some (internal/hidden) state as well as returning a value

- Reading from the keyboard and writing to the screen are side effects

⇒ Interactive programs *have side effects*

- How to square this circle?

Oh no!

y = (putChar 'x', putChar 'x')

y' = let x = putChar 'x' in (x, x)

Conceptual idea

- We can think of an interactive program as a pure function of type

```
World -> World
```

- That is, it takes the current *state of the world* as input and produces a *modified world* as output

⇒ new `World` object reflects any side effects that were performed

`IO` actions

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

Input/output eats the world and produces a result of type `a`, along with a new world.

output:

input

main :: IO ()

readString :: IO String

A solution: *actions*

- Copying the world is too expensive in practice

$\Rightarrow$ Introduce *new types* to distinguish pure expressions from *impure actions*

$\Rightarrow$ Use the concept, but Haskell uses a *primitive type*: implementation details are hidden.

- These actions may have side effects
- Now we can write interactive programs in Haskell and “hide” the side effects behind a type.

The `IO` type

```haskell
data IO a = ... -- "Opaque" implementation
```

The type of actions that return a value of type `a`.

Basic actions

Read and echo a character.

Send output of getChar with putChar.

Reading

```
getChar :: IO Char
getChar = ...
```

Read a character from the keyboard, echo it to the screen, and return it

type mismatch

Writing

```
putChar :: Char -> IO ()
putChar c = ...
```

Write a character to the screen and return nothing (indicated by the empty tuple)

Bridging from expressions into actions

- For type safety, we need a way of “wrapping” values into actions
- Allows us to bring side-effect-free expressions into the “action” world.

From pure to impure

```
return :: a -> IO a
return v = ...
```

“Lift” a pure expression into an impure action.

Note: no way of turning an action back again.

can't unpack (safely).

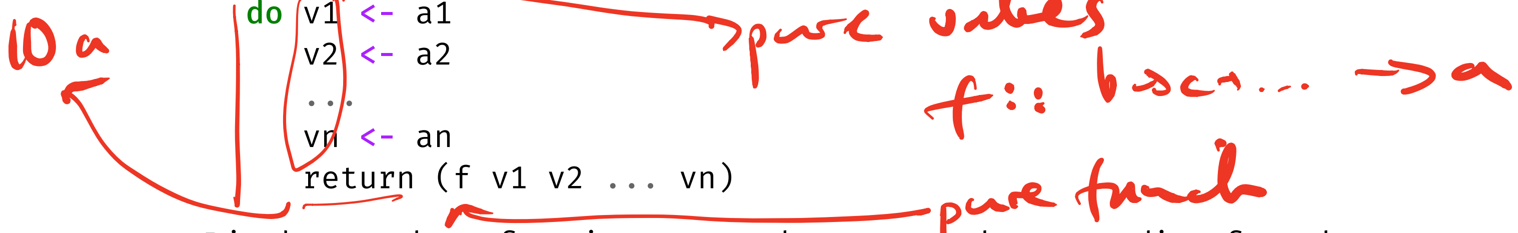
WARNING!

The name `return` is rather misleading when coming from imperative languages. Calling `return` *does not* affect control flow.

Sequencing actions

- We can combine a sequence of `IO` actions using `do` notation

```
do v1 <- a1
   v2 <- a2
   ...
   vn <- an
   return (f v1 v2 ... vn)
```



- Binds results of actions to values and then applies f to the values and lifts into "action-land" with return.

Similarity with list comprehensions

- Each expression `vi <- ai` is called a *generator*
- If we want to execute an action, but don't care about the result, we can use `_ <- ai` or just `ai`

Example: reading characters

A first action

```
act :: IO (Char, Char)
act = do x <- getChar
         getChar
         y <- getChar
         return (x, y)
```

- Read three characters, discard the second, and return the first and third.
- Note use of `return`, without it we would get a type error

$\Rightarrow$ `(x, y) :: (Char, Char)`, but we need an `IO (Char, Char)`.

More primitives

Read a string

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

More primitives

Write a string

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Write a string with a new line

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

When is an action performed?

- Actions never require arguments: `act :: IO a` is not a function
- Just specify that something will be done

⇒ Must be "run" to execute

- GHCi knows to run actions at the prompt

```
Prelude> x = putStrLn "hello"
Prelude> x
hello
Prelude> x
hello
```

x :: IO()

- Conversely, when writing a program to be compiled, GHC only ends up running the `main` action.

⇒ Compare **`main`** function in C/Java.

module Main where

⋮

main :: IO ()

Why these complications?

- One might wonder why we can't write actions as functions
- They would then behave like we're "used to"

Why not this?

```
getChar :: () -> Char
getChar _ = ...
```

"`getChar` ignores its argument and returns a `Char`"

- The problem is one of purity and *referential transparency*

Pure vs. Impure

Pure	Impure
• Always produces same result when applied to the same arguments	• *May* produce different results when applied to the same arguments
• Never has side effects	• *May* have side effects
• Never alters state	• *May* alter state

- Impure functions are not *referentially transparent*

Definition (Referential transparency)

Replacing an expression by its *value* does not change the behaviour of the program

- Not possible with `getChar`: which `Char` should we substitute?

$\Rightarrow$ Can't treat them as normal (pure) functions

Actions as promises

- To fix the issue of referential transparency, **`IO`** is introduced
- We can think then of a type **`IO Char`** as a *placeholder* for a **`Char`** that will only materialise once the program executes
- Moreover, it encapsulates a *promise* that this **`Char`** will actually appear.

⇒ manipulating an **`IO Char`** is equivalent to setting up "plans" to be executed when the **`Char`** materialises.

- This way, we maintain type safety "inside" the action.

IO Monad, oh it's easy, just think of the list Monad.

An example interactive program

Hangman

- Let's write a simple "hangman" game:
- Player A secretly enters a word
- Player B tries to figure out the word with a sequence of guesses
- For each guess, the program indicates which letters of the secret word are in the guess
- Game is over once the guess is correct
- Let's implement this "top down"

Hangman I

- We start by importing useful IO functions
- The `main` function will just run the game

```
import System.IO
-- Set terminal output buffering so we see prints immediately
main = do hSetBuffering stdout NoBuffering
          hangman
```

- We prompt for a word, read it secretly (without echoing) and then run the play loop.

```
hangman :: IO ()
hangman = do putStrLn "Think of a word: "
             word <- secretlyGetLine
             play word
```

Hangman II

- Now we want to read input from the terminal, but without echoing
- `getLine` does the former, but also echos as we type.
- Here we turn off the echoing and instead print hyphens

```
secretlyGetLine :: IO String
secretlyGetLine = do hSetEcho stdin False
                     xs <- getLine
                     putChar (replicate (length xs) '-')
                     hSetEcho stdin True
                     return xs
```

- Notice how *inside* the `do` block, the *results* of actions are just normal pure types.

Hangman III

- Finally, we define how to play the game
- We repeatedly ask for a guess, either it was correct

```
play :: String -> IO ()
play word = do putStr "What is your guess? "
               guess <- getLine
               if guess == word then
                 do putStr "Correct! The word was "
                    putStrLn word
```

- Or it was not, in which case we show which letters matched and prompt again.

```
               else
                 do putStrLn (match word guess)
                    play word

match :: String -> String -> String
match xs ys = [if x `elem` ys then x else '-'| x <- xs]
```

Building block summary

- Prerequisites: none
- Content
  - Saw `IO` action, and how this allows side-effectful input and output in Haskell programs
  - Discussed difference between pure and impure functions
  - Saw *sequencing* and `do` syntax for `IO` actions
  - Saw how to write interactive programs that prompt for input from terminal.
- Expected learning outcomes
  - Student can *explain* how Haskell deals conceptually with side-effectful IO.
  - Student can *write* simple interactive programs
- Self-study
  - None