Functional and Logic Programming

Lecture 8 — Programming with IO

Pedro Vasconcelos
DCC/FCUP

2025

Motivation

- We have so far only defined pure functions: computations that transform an input into an output
- Referentially transparent: given the same input, we always get back the same output
- However, this is not enough to express computations that interact with the outside world:
  - read from stdin or files;
  - write to stdout or files;
  - open network connections;
  - ...

Side effects

- Most languages solve this problem by allowing side effects during evaluation

```
// this is C
char c = getchar();
putchar(c);
putchar('\n');
```

- Evaluating `getchar` will read the next character as a side effect
- Evaluating `putchar` will write a character as a side effect
- Calling `getchar()` twice may give different characters
- The order of the evaluation of these functions is important

The problem with side effects

- Side effects do not work with lazy evaluation:

```
-- this pseudo-Haskell
-- getchar :: () → Char
getTwo  :: [Char]
getTwo = [getchar (), getchar ()]
```

- What are the side effects?

```
getTwo        -- reads the 1st and 2nd chars
getTwo !! 0   -- reads the 1st character
getTwo !! 1   -- also reads the 1st character
length getTwo -- reads no characters
```

- Side effects are not useful when we are not sure which ones will occur!

The Haskell solution

- Introduce a special type for IO actions
- This allows distinguishing

```
'a'  :: Char          -- a character
getChar :: IO Char    -- an IO action that
                      -- will yield a character
```

- `IO a` is the type of actions that may perform IO and yield a value of type *a*
- `putChar` is a function that takes a character and gives an IO action:

```
putChar :: Char → IO ()  -- output a character
```

- `putChar 'A'` is the action that outputs a single 'A' to stdout

Combining IO actions

- We can combine actions in sequence using a special do-notation:

```
do putChar 'A'
   putChar '\n'
```

- The above expression is an action that outputs 2 characters in sequence and has type `IO ()`
- You could also use braces and semicolons instead of indentation:

```
do { putChar 'A'; putChar '\n' } :: IO ()
```

Naming actions

- IO actions are values; we can give them names just like any other value:

```
action :: IO ()
action = do putChar 'A'
            putChar 'B'
            putChar 'C'
```

- Evaluating an IO action in GHCi will perform it:

```
ghci> action
ABCghci>
```

Getting values from actions

- Inside a do-block, we can use <− to get the value returned by an action:

```
echo :: IO ()
echo=  do c ← getChar
          putChar c
          putChar c
```

- `echo` reads a character from stdin and outputs twice to stdout
- You really need `do` and <− to compose actions; the following gives an error:

```
putChar getChar -- couldn't match 'Char'
                -- with actual type 'IO Char'
```

The return action

- There is special action that does not perform any IO but simply returns a value
- This is useful when you want to return a value that is a combination of previous values

```
getTwo :: IO [Char]
getTwo = do c1 ← getChar
            c2 ← getChar
            return [c1,c2]
```

- The type of return is

```
return :: a → IO a
```

- Unlike C/Java/etc, return only makes sense as the final action in a do-block (it does *not* perform an "early exit")

Programming with IO

Basic IO actions defined in the Prelude:

```
putChar :: Char → IO ()
putStr :: String → IO ()       -- print a string
putStrLn :: String → IO ()     -- print a string
print :: Show a ⇒ a → IO () -- print a value
getChar :: IO Char
getLine :: IO String       -- get a line
getContents :: IO String -- get the standard input
```

We can combine these using do-blocks to write IO programs.

Programming with IO (cont.)

```haskell
main :: IO ()
main = do
    putStrLn "What is your name? "
    name ← getLine
    putStrLn ("Hello, " ++ name ++ "!")
```

- The `main` action in module `Main` is the entry point for a Haskell program
- We can compile this with GHC and get a binary executable

```
$ ghc Main.hs -o main
[1 of 1] Compiling Main
[2 of 2] Linking main
$ ./main
What is your name? Pedro
Hello, Pedro!
```

Reflection

Doesn't this just re-invent imperative programming. . . ?

Two major advantages:

- It is generally a good idea to decouple the "business logic" from the "IO handling" but in Haskell we are explicit about this separation in the types
- Because IO actions are first class, we can define our own control structures
- The do-notation can also for things other than IO (we will see its use for writing parsers in a future lecture)

Imperative shell, functional core

```
-- imperative shell
main :: IO ()
main = do
   txt ← getContents
   putStr (count txt)
   -- alternative: interact count
   -- interact :: (String → String) → IO ()

-- functional core
count :: String → String
count txt
  = let nlines = length (lines txt)
        nwords = length (words txt)
    in "lines: " ++ show nlines ++ "\n" ++
       "words: " ++ show nwords ++ "\n"
```

A larger example

- Let us write a program to pretty-print JSON data[1]
- A slightly simplified version of the example described the *Haskell Unfolder #46*: `https://www.youtube.com/live/5W0ZUY_l1dU?si=DVi0kcdaHGNkFJwf`

[1] `https://www.json.org/`

Sample JSON input


```
{"student": {"name": "Alex Johnson","age": 14,
"grade": 9,"subjects": ["Math", "Science",
"History"],"isEnrolled": true, "contact":
{"email": "alex.johnson@example.com", "phone":
"555-1234"}}}
```

Sample pretty-printed output

```
{
    "student": {
        "name": "Alex Johnson"
        ,"age": 14
        ,"grade": 9
        ,"subjects": [
            "Math"
            , "Science"
            , "History"
        ]
        , "isEnrolled": true
        , "contact": {
             "email": "alex.johnson@example.com"
            , "phone": "555-1234"
        }
    }
}
```

Implementation

- The functional core is a function `String -> String`
- Process each character at a time
- Keep track of the current identation level
- Introduce a newline at brackets (`[`), braces (`{`) and commas (`,`)
- Increase indentation when we see an open bracket or brace
- Decrease indentation when we see a close bracket or brace

Implementation (cont.)

```
main :: IO ()
main = interact prettyPrint

prettyPrint :: String → String
prettyPrint txt = prettyAt 0 txt
  where
    prettyAt :: Int → [Char] → [Char]
    prettyAt _ []      = []
    prettyAt i (c:cs)
      | elem c "{[" = [c] ++ newline (i + 1)
                        ++ prettyAt (i + 1) cs
      | elem c "}]" = newline (i − 1) ++ [c]
                        ++ prettyAt (i − 1) cs
      | c == ','    = newline i ++ [c]
                        ++ prettyAt i cs
      | otherwise   = c : prettyAt i cs
```

Implementation (cont.)

```
newline :: Int → String
newline i = "\n" ++ replicate (i * indent) ' '

indent :: Int
indent = 4
```

User-defined control structures

- IO actions are first class values
- We can pass them around to functions freely
- Evaluation is separate from performing the action
- This allows defining our own control structures

User-defined control structures (cont.)

```
-- run a list of actions
seqn :: [IO ()] → IO ()
seqn [] = return ()
seqn (act:rest) = do act
                     seqn rest

main :: IO ()
main = seqn [print i | i←[1..10]]
-- prints 1, 2, ..., 10
```

NB: `sequence_` is a more general function from the Prelude that does the same thing as `seqn`.