Input and Output

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University

Semester 1, 2006

Assignment 1 hint, clarification, advice

Tutors and I have been asked by lots of students for help with transformation functions that take more than one argument, such as

```
threshold :: Int -> Image -> Image
```

In `TestBed.hs`, you can't pass `threshold` to the `transform` test function because it's the wrong type.

Partial application is the key idea.
`(threshold 100)` is a function of type `Image -> Image`, which is the right type for `transform`. You should evaluate

```
transform (threshold 100)
```

The same advice holds for the function arguments to `map`.

If you have a threshold function for pixels, like:

```
thresholdPixel :: Int -> Pixel -> Pixel
```

which you want to `map` over every element of a list of pixels (i.e. a `Row`), again you need to partially apply that function to get the right type: `(thresholdPixel 100)` is a function of type `Pixel -> Pixel`.

I don't want to give too much away, but:

```
thresholdRow :: Int -> Row -> Row
thresholdRow level rows = map (thresholdPixel level) rows
```

What’s a *Program?*

So far, we have concentrated on **pure functions** — they take arguments, perform a **computation**, and return a result.

To use such functions we have used the **program** GHCi which ***interacts*** with the user.

A **program** running on a computer ***interacts with its environment***.

The most basic kinds of actions are **input** and **output**:

- **input** from the keyboard or a file (e.g. `ppm` image files)
- **output** to the screen or a file (e.g. `ppm` image files)

(Other interactions include communication with devices, networks etc.)

A Haskell *Program*

- A module called `Main`, containing
- a function `main` of type `IO ()`

```
module Main where

main :: IO ()
main = print (map factorial [0..10])

factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Compiling Programs

We use the **GHC** compiler to *translate* the Haskell program to machine language:

```
ghc Factorials.hs -o factorials
```

This produces a *machine-language program* (in the file `factorials`) which can be run independently:

```
./factorials
[1,1,2,6,24,120,720,5040,40320,362880,3628800]
```

`print` is a function which prints any value which has a text representation.

What happens if we apply `print` to a string?

```
print "Hello World"
```

If we want to print the *contents* of a string we use

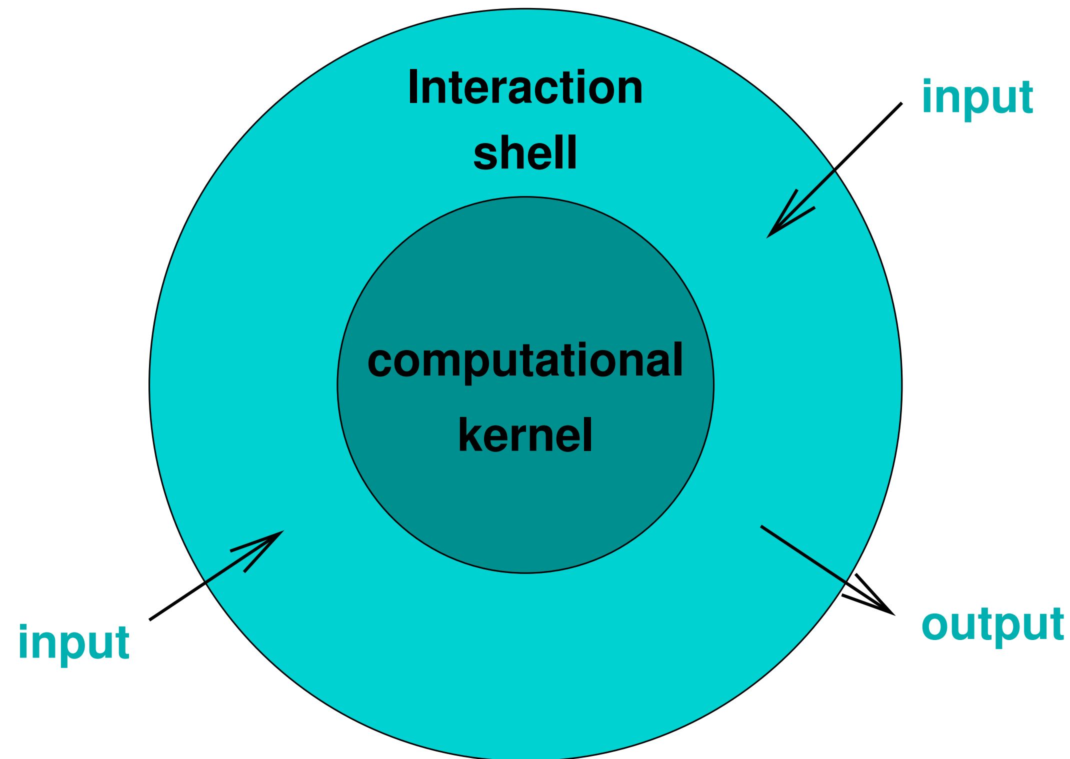
```
putStr "Hello World"
```

or

```
putStrLn "Hello World"
```

Interaction and Computation

Programs are generally composed of an inner computational kernel and an outer interaction shell:



Haskell Separates Interaction from Computation

Most languages don't clearly distinguish these two aspects.

Haskell does it by means of **types**.

All the types we have seen so far are "computation" types.
For example, the expression

```
"Hello " ++ "World!"
```

denotes a computation, and the function

```
(++) :: [a] -> [a] -> [a]
```

is a **pure** function — it only computes a result string from two input strings.

`IO` types

Input/output actions always have a type of the form `IO a` where `a` is the result of the I/O operation.

For example, the expression

```
putStr "Hello World!"
```

has type `IO ()` where the `()` indicates that no result is returned.

The type of `putStr` is

```
putStr :: String -> IO ()
```

The result of the `putStr` function is not only a value `()`, but also an IO interaction.

Combining Actions

Unlike pure functions, we **cannot** combine IO functions by composing them.

```
putStr getLine
```

doesn't work, because `getLine` has type `IO String` but `putStr` expects a pure `String`, not an IO action that produces a `String`.

Haskell distinguishes between computations and interactions by *types.*

Order of execution

In pure computations, there is some freedom for the system to choose the order in which subcomputations are performed. That is not the case with IO actions. We must specify the sequence.

The `do` Notation

When we want to combine IO actions we use the `do` notation:

```
do
  <statement #1>
  <statement #2>
  ...
  <statement #n>
```

where each statement is an IO action.

Whenever we want the result of an action, we can use a statement of the form:

```
v <- <action>
```

to bind the result of the action to some variable v.

Now we can combine `getLine` and `putStr` as follows:

```
do
  input <- getLine
  putStr input
```

Since `getLine` has type `IO String`, the variable `input` has type `String`, which can be passed to `putStr`.

Of course, IO types can be used in function definitions:

```
echoLine :: IO ()
echoLine = do
             input <- getLine
             putStr input
```

Defining Interaction Functions

As with computations, we can define functions that encapsulate useful interaction patterns. The following `ask` function poses a question and waits for the user's response.

The question is passed as a `String` argument, and the result is an IO action that produces a `String`:

```
ask :: String -> IO String
ask question = do
                 putStr question
                 getLine
```

An example using `ask` is in `HelloYou.hs`

Showing and Reading Values

IO actions `getLine` and `putStr` allow us to read and write `Strings`.

The functions `readLn` and `print` read and write lots of other types of values.

```
askNumber :: String -> IO Float
askNumber question = do
                     putStr question
                     readLn
```

Examples using `askNumber` include `Sum2Nums.hs` and `MaleMetabolic.hs`

Conditions and Repetition in IO Actions

We want to write a program that reads a line of text, then outputs it after converting all characters to upper case (see week 4 lab exercises).

The program should repeat the process until an empty line is entered.

The computational part should be familiar:

```
stringToUpper :: String -> String
stringToUpper chs = map toUpper chs
```

The whole program is `Upper.hs`

Notice how the computational part (`stringToUpper`) is quarantined from the interaction part.

The `upperLine` Function

The repeated interaction is:

1. get a line

   ```
   line <- getLine
   ```

2. if it's empty, return (that is, escape the repetition)

   ```
   if line == "" then return ()
   ```

3. otherwise convert to upper case, print, and repeat the process (recurse)

   ```
   else do
     putStrLn (allUpper line)
     upperLine                 -- recursive call
   ```

Notes on `upperLine` syntax

- The `then` and `else` branches are indented because they belong to the statement starting with `if`.
  (Remember the off-side rule.)
- As there are two statements in the `else` branch, we need another `do` to put them together.

`SumNums.hs` is another program, similar to `Upper.hs`, which adds up a sequence of numbers entered at the keyboard.

File I/O

All examples so far have been about reading from the keyboard and writing to the screen.

Programs must also be able to access and modify **files** on disk, etc.
(For example, assignment 1 manipulates `ppm` files.)

The three simplest standard functions are:

```
type FilePath = String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath           -> IO String
```

- Values of the `FilePath` denote files using the usual path syntax, eg `"/home/clem/words.txt"`
- `WriteFile path str` creates a new file called `path` and writes `str` into the file.

  If a file with that name already exists, it is overwritten.
- `appendFile path str` adds `str` to the end of an already existing file called `path`.
- `readFile path` reads the entire contents of the file called `path` and returns it as a string.

Example

A simple program to read the name of a file from the keyboard, read the file, then print it to the screen:

```
main :: IO ()
main = do
       putStr "Name of file: "          -- to screen
       path <- getLine                  -- from keyboard
       contents <- readFile path        -- from file
       putStrLn contents                -- to screen
```

Example: `Cat.hs`

File contents as a single string...

`readFile` returns a single string being the contents of the file.

Demonstration: `Contents.hs`

Often we want to deal with each line separately. The standard, pure function:

```
lines :: String -> [String]
```

breaks a string into a list of lines at each new line character `'\n'`

The dual of `lines` is

```
unlines :: [String] -> String
```

Example: `PickLine.hs`

There is also a pure function that breaks a string into separate “words” by looking for space or newline characters:

```
words :: String -> [String]
unwords :: [String] -> String
```

Example: `Alpha.hs`

Showing and Reading Values . . . Revisited

Recall the functions `readLn` and `print` that can read and write non-string values of various types.

In fact they are defined in terms of string I/O functions and two standard **pure** functions, `show` and `read`.

- `show` converts values to their string representation.
  e.g. `show 42` $\Longrightarrow$ `"42"`
- `read` is the dual of `show`: it converts string representations to values.
  e.g. `read "42"` $\Longrightarrow$ `42`

Not every type of value has a string representation so the types of `show` and `read` are *not*:

```
show :: a -> String
read :: String -> a
```

because not every type can be substituted for `a`. `show` and `read` are not *polymorphic*. They are *overloaded*, so their types are:

```
show :: Show a => a -> String
read :: Read a => String -> a
```

`print` is defined using `show` and `putStrLn`:

```
print :: Show a => a -> IO ()
print value = putStrLn (show value)
```

`readLn` is defined using `read` and `getLine`:

```
readLn :: Read a => IO a
readLn = do
           line <- getLine
           return (read line)
```

Advice: Don't always try to write programs from scratch.

Re-use code that you have written, I have written, from textbooks, etc.