Prog & Alg I (COMS10002)
Week 9 - Intro to Haskell

Dr. Oliver Ray
Department of Computer Science
University of Bristol

Monday 24^{th} November, 2014

Timetable for Weeks 9-12

<table>
<tr><th></th><th>Mon</th><th>Tue</th><th>Wed</th><th>Thu</th><th>Fri</th></tr>
<tr><td>9</td><td></td><td rowspan="3">LAB
(Group 1)</td><td></td><td></td><td></td></tr>
<tr><td>10</td><td></td><td>LEC
(Oliver)</td><td></td><td></td></tr>
<tr><td>11</td><td></td><td></td><td></td><td></td></tr>
<tr><td>12</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td rowspan="3">LAB
(Group 2)</td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>LEC
(Oliver)</td><td></td><td></td><td></td></tr>
<tr><td>4</td><td></td><td></td><td>TUT
(Group 2)</td><td></td></tr>
<tr><td>5</td><td></td><td></td><td></td><td>TUT
(Group 1)</td><td></td></tr>
</table>

Haskell

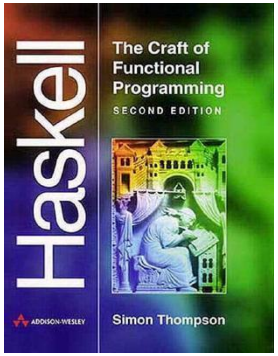
Coursework

Theory

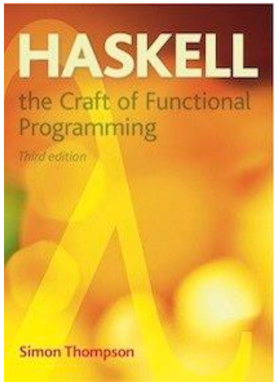
Worksheet

Course Materials

You'll need to get a copy of the course book



- ***Haskell: The Craft of Functional Programming*** by Simon Thompson (Addison-Wesley)
- Note that page numbers refer to the 2^{nd} edition, of which there are several copies in the library



We'll be using a state-of-the-art Haskell compiler

- The ***Glasgow Haskell Compiler (Interactive)***
- this is installed on the lab machines and is also available from http://www.haskell.org/ghc/
- Note that `ghci` replaces the `hugs` system used in the 1^{st} and 2^{nd} editions of the book

You may find these older slides (by Ian) useful:

- https://www.cs.bris.ac.uk/Teaching/Resources/COMS12400/lectures/

Functional Programming

- Functional programming is very high-level (no hacking around or trial-and-error programming like in C!)
- The focus is on formalising the relationship between the inputs and outputs of a task in a functional way
- There is no notion of state or variable assignment; there are only types and functions defined on them
- There is no notion of program execution; there is only the evaluation of functions
- There are no side-effects, no sequencing, and no loops; there are only pure functions
- A functional program is a set of function definitions and type declarations (in fact Haskell can usually work out the types, but it is very good practice to include them)

Function Declarations (p.10)

- A function declaration is a statement of the form

```
fname :: t1 -> t2 -> … -> tn -> t
```

- `fname` is the name of the function

  `n` (≥0) is the number of inputs to the function

  t_i is the type of the `i`'th input

  `t` is the type of the value returned by the function

Note that functional programmers tend to translate multi-argument functions $(t_1 \times t_2 \times \ldots \times t_n) \rightarrow t$ into a nesting of single argument functions $t_1 \rightarrow (t_2 \rightarrow (\ldots \rightarrow (t_n \rightarrow t)\ldots))$. This technique is known as *Currying* after Haskell B. Curry

Function Definitions (p.10,39)

- A function definition is a set of equations of the form

```
fname p1 p2 … pn = r0
```

or

```
fname p1 p2 … pn
    | g1    = r1
    | g2    = r2
    …
    | gm    = rm
```

indentation is required!

- p_i is an expression (parameter) of type t_i

  r_i is an expression (a result) of type t

  g_i is an expression (guard) of type `Bool`

  g_m can be keyword `otherwise` (which is always true)

As with any language there are built-in types

- `Bool` **(p.33-4)**

```
True False && || not == /=
```

- `Char` **(p.41-2)**

```
'a' 'b' … 'A' 'B' … '\t' '\n' '\\' '\'' '\"' ' '
```

- `String` **(p.92-3)**

```
"" "Haskell Rules OK\n" ++ !! show
```

- `Int` **(p.35-6)**

```
0 1 (-1) 2 … + - * ^ div mod divMod abs negate > >= <= <
== even odd
```

- `Float` **(p.43-4)**

```
0.123 1.2e3 … + - * / ^ ** abs sin asin ceiling floor
round fromIntegral exp pi log logBase negate signum sqrt
```

- `Integer / Double / Rational / …`

Example: Factorial (p.60)

- Consider the factorial function which maps a positive integer n to corresponding n integer n! defined thus:

$$n! = \begin{cases} 1 & if \quad n = 0 \\ n(n-1)! & if \quad n > 0 \end{cases}$$

- In Haskell we can implement this definition like so:

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

declaration

definition

guards

result

Example: Exponentiation

- Recall our definition of fast integer exponentiation:

$$x^n = \begin{cases} 1 & \text{if} \quad n = 0 \\ (x^2)^{n/2} & \text{if} \quad n \text{ is even} \\ x.x^{n-1} & \text{if} \quad n \text{ is odd} \end{cases}$$

- In Haskell we can implement this definition like so:

```
power :: Int -> Int -> Int
power x n
    | n==0   = 1
    | even n = power (x*x) (div n 2)
    | odd n  = x * power x (n-1)
```

- Note brackets are used for grouping, NOT function calls

Local Definitions (p.103-8)

- To make them easier to read, function definitions can include local definitions of two possible forms:

- "where" statements

```
where l₁ l₂ … lₖ
```

which are visible to all equations in a definition

- "let" statements

```
let l₁ ; l₂ ; … ; lₖ in e
```

which are visible only to the particular expression e

useful to avoid retyping an expression many times

Example: Exponentiation (revisited)

- We can also implement our previous example like so:

```
pow :: Int -> Int -> Int
pow x 0          = 1
pow x n
    | m == 0     = let y=x*x in pow y d
    | otherwise = let y=pow x (n-1) in x*y
    where (d,m) = divMod n 2
```

local to each expression

local to whole equation

- In this simple example the use of the let statements does not make the definition very much easier to read (but it often can make definitions much easier to read)
- Here the where statement using divMod is slightly more efficient than separately testing parity and halving n

Constants and Function calls

- Functions with no arguments are simply constants (once defined their value cannot be changed!):

```
greeting :: String
greeting = "Hello world!"
```

```
value :: Int
value = 5
```

- NOTE: when calling functions we don't use commas to separate arguments and we only use brackets for grouping arguments: i.e. we write (f a b) not f(a,b)

```
power value (value+1) = power 5 6 = 15625
power(value,value+1) = syntax error
power value value + 1 = (power 5 5) + 1 = 3126
```

Hello World!

- If we put the above definitions into a text file “intro.hs” and run “gchi intro.hs” on a lab machine, we can verify

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( intro.hs, interpreted )
Ok, modules loaded: Main.
*Main> greeting
"Hello world!"
*Main> "The answer is "++show(value*8+2)
"The answer is 42"
*Main> fac 5
120
*Main> power value (value+1)
15625
*Main> :quit
```

Some general tips

- reloading: after editing your script, remember to reload it into ghci

```
:reload
```

- comments: it is always useful to comment your code

```
-- single line comment
{-  multi line
     comment  -}
```

- debugging: it often helps to insert code that throws an errors to see if a particular function has been evaluated

```
...error ”got here!”...
```

Function Evaluation (p.17,60,340-343,345)

- Evaluation involves repeatedly replacing function calls by the appropriate instances of function definitions
- Evaluation of a sub-expression is only carried out if it is necessary (aka lazy computation)
- Repeated sub-expressions are evaluated at most once
- Each equation is tried in turn until one is found that conforms to the pattern of the call arguments
- Each guard is tried in turn until one is found to succeed
- The function call is then replaced by the guard result
- Some evaluation may be necessary to determine if the the call matches a pattern and/or if a guard is true
- For convenience we underline the call being evaluated

Example

top-level goal

guard testing

function replacement

built-in ops

etc.

```
fac 2
    ? 2==0 = False
    ? 2>0 = True
= (2 * fac (2-1))
= (2 * fac 1)
    ? 1==0 = False
    ? 1>0 = True
= (2 * (1 * fac (1-1)))
= (2 * (1 * fac 0))
    ? 0==0 = True
= (2 * (1 * 1))
= (2 * 1)
= 2
```