REVIEW OF HASKELL

A lightening tour in 45 minutes

What is a Functional Language?

Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments;
- A functional language is one that supports and encourages the functional style.

Example

Summing the integers 1 to 10 in Java:

```
total = 0;
for (i = 1; i ≤ 10; ++i)
   total = total+i;
```

The computation method is variable assignment.

Example

Summing the integers 1 to 10 in Haskell:

```
sum [1..10]
```

The computation method is function application.

This Lecture

A series of six micro-lectures on Haskell:

- First steps;
- Types in Haskell;
- Defining functions;
- List comprehensions;
- Recursive functions;
- Declaring types.

REVIEW OF HASKELL

1 - First Steps

Glasgow Haskell Compiler

- GHC is the leading implementation of Haskell, and comprises a compiler and interpreter;

- The interactive nature of the interpreter makes it well suited for teaching and prototyping;

- GHC is freely available from:

`www.haskell.org/downloads`

Starting GHC

The GHC interpreter can be started from the Unix command prompt % by simply typing ghci:

```
% ghci

GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help

Prelude>
```

The GHCi prompt > means that the interpreter is ready to evaluate an expression.

For example:

```
> 2+3*4
14

> (2+3)*4
20

> sqrt (3^2 + 4^2)
5.0
```

Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

```
f(a,b) + c d
```

Apply the function f to a and b, and add the result to the product of c and d.

In Haskell, function application is denoted using space, and multiplication is denoted using *.

```
f a b + c*d
```

As previously, but in Haskell syntax.

Moreover, function application is assumed to have higher priority than all other operators.

`f a + b`

Means (f a) + b, rather than f (a + b).

REVIEW OF HASKELL

2 - Types in Haskell

What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

`False` `True`

Types in Haskell

- If evaluating an expression e would produce a value of type t, then e has type t, written

```
e :: t
```

- Every well formed expression has a type, which can be automatically calculated at compile time using a process called type inference.

Basic Types

Haskell has a number of basic types, including:

`Bool` - logical values

`Char` - single characters

`String` - strings of characters

`Int` - fixed-precision integers

List Types

A list is sequence of values of the same type:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```

In general:

[t] is the type of lists with elements of type t.

Tuple Types

A tuple is a sequence of values of different types:

```
(False,True)     :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

In general:

(t1,t2,…,tn) is the type of n-tuples whose ith components have type ti for any i in 1…n.

Function Types

A function is a mapping from values of one type to values of another type:

```
not     :: Bool → Bool

isDigit :: Char → Bool
```

In general:

t1 → t2 is the type of functions that map values of type t1 to values to type t2.

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

```
length :: [a] → Int
```

for any type a, length takes a list of values of type a and returns an integer.

REVIEW OF HASKELL

3 - Defining Functions

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
abs   :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

```
not       :: Bool → Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called “cons” that adds an element to the start of a list.

```
[1,2,3,4]
```

Means 1:(2:(3:(4:[]))).

Functions on lists can be defined using x:xs patterns.

```
head        :: [a] → a
head (x:_)   = x

tail        :: [a] → [a]
tail (_:xs)  = xs
```

head and tail map any non-empty list to its first and remaining elements.

Lambda Expressions

A function can be constructed without giving it a name by using a lambda expression.

$$\lambda x \rightarrow x+1$$

The nameless function that takes a number x and returns the result x+1.

Why Are Lambda's Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

```
add x y = x+y
```

means

```
add = λx → (λy → x+y)
```

REVIEW OF HASKELL

4 - List Comprehensions

Lists Comprehensions

In Haskell, the comprehension notation can be used to construct new lists from old lists.

```
[x^2 | x ← [1..5]]
```

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

Note:

- The expression x ← [1..5] is called a generator, as it states how to generate values for x.
- Comprehensions can have multiple generators, separated by commas. For example:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x,y are elements of the list [1..3] and $y \geq x$.

Using a dependant generator we can define the library function that concatenates a list of lists:

```
concat     :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

For example:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

```
[x | x ← [1..10], even x]
```

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors   :: Int → [Int]
factors n =
   [x | x ← [1..n], n `mod` x == 0]
```

For example:

```
> factors 15

[1,3,5,15]
```

REVIEW OF HASKELL

5 - Recursive Functions

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

factorial maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.

For example:

```
    factorial 3
=
    3 * factorial 2
=
    3 * (2 * factorial 1)
=
    3 * (2 * (1 * factorial 0))
=
    3 * (2 * (1 * 1))
=
    3 * (2 * 1)
=
    3 * 2
=
    6
```

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions.
- As we shall see, however, many functions can naturally be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product         :: [Int] → Int
product []       = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

For example:

```
    product [2,3,4]
=
    2 * product [3,4]
=
    2 * (3 * product [4])
=
    2 * (3 * (4 * product []))
=
    2 * (3 * (4 * 1))
=
    24
```

REVIEW OF HASKELL

6 - Declaring Types

Data Declarations

A new type can be declared by specifying its set of values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown]

flip         :: Answer → Answer
flip Yes      = No
flip No       = Yes
flip Unknown  = Unknown
```

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be <u>recursive</u>.

```
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat → Nat.

Note:

- A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

```
Zero

Succ Zero

Succ (Succ Zero)
```

⋮

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int          :: Nat → Int
nat2int Zero      = 0
nat2int (Succ n) = 1 + nat2int n

int2nat  :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add     :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```