Reviewing Haskell

COMP2600 / COMP6260

Dirk Pattinson
Australian National University

Semester 2, 2015

Why Haskell?

- Declarative, not imperative.
- Functions specify values, not sequences of updates to the store.
- Evaluate the program by rewriting to normal form.
- Much easier to reason about.
- example: `fac` $n = n!$

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

- example: `mult` $m\ n = m + m + \ldots + m$ (n times)

```
mult m 0 = 0
mult m n = m + mult m (n - 1)
```

Purity

Rewriting a pure program in a different order does not change its final value.

tfac $m\ n = m * n!$

```
tfac m 0 = m
tfac m n = tfac (m * n) (n - 1)
```

`tfac 3 2`	`tfac 3 2`
$\Longrightarrow$ `tfac (3 * 2) (2 - 1)`	$\Longrightarrow$ `tfac (3 * 2) (2 - 1)`
$\Longrightarrow$ `tfac 6 (2 - 1)`	$\Longrightarrow$ `tfac (3 * 2) 1`
$\Longrightarrow$ `tfac 6 1`	$\Longrightarrow$ `tfac (3 * 2 * 1) (1 - 1)`
$\Longrightarrow$ `tfac (6 * 1) (1 - 1)`	$\Longrightarrow$ `tfac (3 * 2 * 1) 0`
$\Longrightarrow$ `tfac 6 (1 - 1)`	$\Longrightarrow$ `3 * 2 * 1`
$\Longrightarrow$ `tfac 6 0` $\Longrightarrow$ `6`	$\Longrightarrow$ `6 * 1` $\Longrightarrow$ `6`

Purity makes a program easier to understand, for both people and compilers.
Purity gives us the freedom to choose which evaluation order to use.

Contrast with C/Java/Python/Perl/PHP ....

The ‘value’ of an imperative program is totally dependent on evaluation order.

```
fac n
 = do result = 1
      while n > 1
          result := result * n
          n      := n - 1
      return result
```

The variables `result` and `n` are *destructively updated* during each iteration of the loop. They take new values, and the old values are destroyed.
We will learn how to reason about both pure and impure programs.

Types

- The function `fac` accepts a value, and produces a value — but not all work.

```
fac "toast" = "toast" * fac ("toast" - 1)
            = ???
```

- The rewriting is *stuck*, because there is no rule to subtract from ”toast”.
- A type is a set of values. `Int = {.., -2, -1, 0, 1, 2, ..}`
- Our `fac` function can accept an `Int` and return an `Int`

```
fac :: Int -> Int
```

Partial Application

Functions take their arguments one at a time.

```
mult :: Int -> (Int -> Int)
(mult x) y = x * y
```

Parentheses shown are default, can be omitted.
Function definition is right-associative, but application is left-associative.

We can *partially apply* the `mult` function by providing just one argument.

```
multTwo :: Int -> Int
multTwo = mult 2
```

`multTwo` is the same as the function `double`

```
double :: Int -> Int
double y = 2 * y
```

Partial Application

Partial application makes sense when we think about rewriting.

```
multTwo :: Int -> Int
multTwo = mult 2
```

```
multTwo 3
  ⟹  (mult 2) 3
  ⟹  mult 2 3
  ⟹  2 * 3
  ⟹  6
```

Tuples

We can collect together multiple values, of arbitrary type, with *tuples*.

```
item :: (String, Float)
item =  ("cola", 3.00)
```

The Prelude defines some useful functions for extracting the components.

```
fst (x, y)    = x
snd (x, y)    = y
```

We can also use pattern matching directly.

```
addPair :: (Int, Int) -> Int
addPair (x, y) = x + y
```

Lists

Lists collect together values of the *same type*.

Lists can be empty, or be constructed from an element and another list.

The following lists all have the same value:

`[1, 2, 3]` `(1 : [2, 3])` `(1 : 2 : 3 : [])`

This one indicates how the list is stored: `(1 : (2 : (3 : [])))`

We use *pattern matching* and *recursion* to write functions that deal with lists.

```
length []     = 0
length (x:xs) = 1 + length xs
```

Accumulating Parameters

An alternative definition of `length` is:

```
length :: [a] -> Int
length                = length' 0
length' acc []        = acc
length' acc (x:xs)    = length' (acc + 1) xs
```

```
length ["red", "rabbit", "rodeo"]
  ⟹   length' 0 ["red", "rabbit", "rodeo"]
  ⟹   length' 1 ["rabbit", "rodeo"]
  ⟹   length' 2 ["rodeo"]
  ⟹   length' 3 []
  ⟹   3
```

Polymorphism

The length function does not inspect the elements of a list. It is only concerned about the list's structure.

```
length [2, 3, 5]                  = 3
length ["red", "green", "blue"] = 3
```

We use type variables to indicate that `length` works on lists of any element type. We usually leave out the quantifier `forall a.`

```
length :: forall a. [a] -> Int
length []       = 0
length (x:xs) = 1 + length xs
```

Notice that `length` does not mention `x` in its body.

Type Classes

Our type for `fac` isn't as general as it could be.

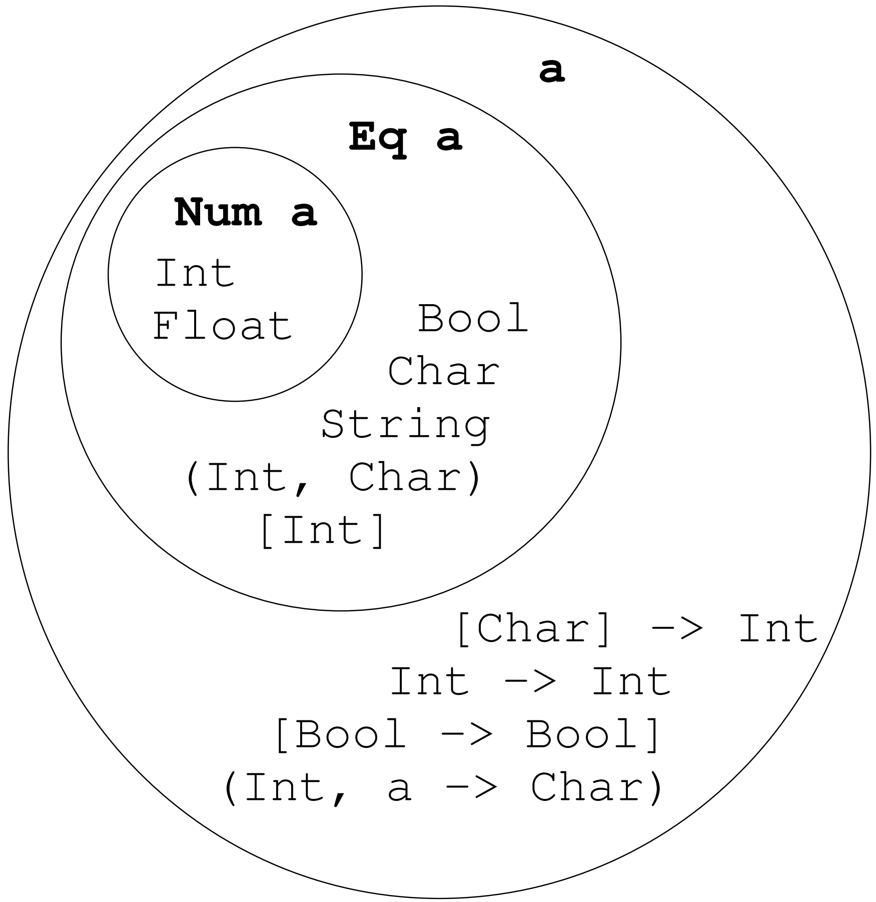
```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n - 1)
```

The argument `n` must be a number, because we use multiplication and subtraction on it, but it doesn't nessesarally have to be an `Int`.

```
fac :: Num a => a -> a
```

`fac` takes an argument of type `a`, and produces a result of the same type, as long as `a` is a *number type*, ie `Int`, `Integer`, `Float`, `Double`

Type Classes



Higher-order functions

Functions can take other functions as arguments.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
map double [1, 2, 3]
```

$\Longrightarrow$ `double 1 : map double [2, 3]`

$\Longrightarrow$ `1 * 2 : double 2 : map double [3]`

$\Longrightarrow$ `...`

$\Longrightarrow$ `1 * 2 : 2 * 2 : 3 * 2 : []`

$\Longrightarrow$ `[2, 4, 6]`

Curried and Uncurried Functions — Example

Curried (one argument at a time)

```haskell
mult :: Int -> Int -> Int
mult x y = x * y
```

Uncurried (multiple arguments)

```haskell
multp :: (Int, Int) -> Int
multp (x, y) = x * y
```

`map (mult 2) [3,4,5]` *equals* `[6,8,10]`
`map multp [(2,3), (4,5), (6,7)]` *equals* `[6,20,42]`

Guards

When an equation has *guards* they will be tried from first to last.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []          = []
filter p (x : xs)
     | p x           = x : filter p xs
     | otherwise     = filter p xs
```

```
filter isEven (1 : 2 : 3 : [])
  ⟹ filter isEven (2 : 3 : [])
  ⟹ 2 : filter isEven (3 : [])
  ⟹ 2 : filter isEven []
  ⟹ 2 : []
```

Algebraic Data Types

We can define our own type by specifying how its values are constructed.

```
data Shape = Circle    Float
           | Rectangle Float Float
           deriving (Eq, Show)
```

`Circle` takes a `Float` and produces a `Shape`.
`Rectangle` takes a `Float`, another `Float` and produces a `Shape`.

Thus `Circle 2.0 :: Shape`, and `Rectangle 2.0 3.0 :: Shape`

```
Circle    :: Float -> Shape
Rectangle :: Float -> Float -> Shape
```

`Circle` and `Rectangle` are *term constructors*, and can be used in *patterns*.

Pattern matching with ADTs

```
isRound :: Shape            -> Bool
isRound (Circle r)           = True
isRound (Rectangle l b)      = False

area :: Shape               -> Float
area (Circle r)              = pi * r^2
area (Rectangle l b)         = l * b
```

area (Rectangle 2 3)

$\Longrightarrow$ 2 * 3

$\Longrightarrow$ 6

Tuples and Lists (again)

An item list using the built-in types:

```
items :: [(String, Float)]
items = [("cola", 3.00), ("tuna", 2.45),
         ("bread", 3.20)]
```

We can define our own types which behave the same way

```
data Tuple2 a b   = T2 a b
data List a       = Nil   | Cons a (List a)

items :: List (Tuple2 String Float)
items = Cons (T2 "cola" 3.00)
             (Cons (T2 "tuna" 2.45)
                   (Cons (T2 "bread" 3.20) Nil))
```

Binary Trees

A binary tree is like a list, but with two tails.

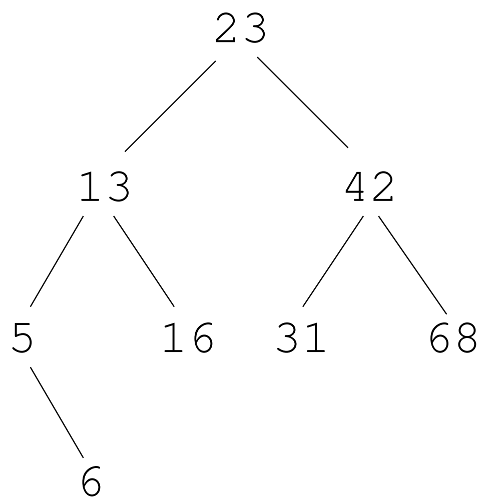
```
data Tree a
       = Null
       | Node a (Tree a) (Tree a)
```

Binary Search Tree Invariant

For any given node, call its key **k**.

- The keys in the left hand subtree of that node are always *less than* **k**.
- The keys in the right hand subtree are always *more than* **k**.

Trees




```
(Node 23  (Node 13  (Node 5  Null
                             (Node 6 Null Null))
                    (Node 16 Null Null))
          (Node 42  (Node 31 Null Null)
                    (Node 68 Null Null)))
```

Exercises

Write functions to:

- Find the smallest and largest elements in a tree.
- Find the number of nodes in a tree.
- Find the maximum depth of the tree.
- Reverse the order of nodes in the tree (inverting the invariant).
- Test whether a particular element is in a tree.
- Insert a new element into the appropriate place in a tree.
- Convert a tree to a list (with elements in increasing order).
- Count how many keys in a tree match a predicate, eg `isEven`