Types and Recursion

COMP2600 / COMP6260

Dirk Pattinson
Australian National University

Semester 2, 2015

Induction and Recursion

The main theme:

- Inductive types
- Recursively defined functions
- Proofs by structural induction

The above are closely linked!

Enumerated Types

Examples:

```
data Temp   = Cold   | Hot
data Season = Spring | Summer | Autumn | Winter
```

A function from one enumerated type to another:

```
weather :: Season -> Temp
weather Summer = Hot
weather _      = Cold
```

- Note the keyword `data`
- Enumerated type: finite number of nullary data constructors
- _ is a *wildcard*: matches everything but binds nothing
- What happens if we swap the last two lines of `weather`?

Datatypes and Constructors

A voter is distinguished by full name and address

```
data Voter = MkVoter String String
```

`MkVoter` is a data constructor. It takes 2 arguments and constructs a `Voter`

```
MkVoter :: String -> String -> Voter
MkVoter "Alan Turing" "Milton Keynes, UK"
```

`Voter` is a type constructor. It has no arguments.

```
Voter :: *
```

Values have types, and types have *kinds*.
A type which takes no arguments has kind `*`.

Datatypes with Alternatives

```haskell
data Box = Square    Float
         | Rectangle Float Float

Square :: Float -> Box
Rectangle :: Float -> Float -> Box

boxArea :: Box -> Float
boxArea (Square a)      = a * a
boxArea (Rectangle a b) = a * b
```

- Which are the *type constructors* and which are the *data constructors*?
- When we write functions over types with altenatives, we must remember to handle each case.
- What would happen if we left off the last line of `boxArea`?

Recursive and Polymorphic Types

Lists are a recursive data type. We could define them as follows:

```
data List    = Nil
             | Cons Int List
```

```
 someList = Cons 2 (Cons 3 (Cons 5 Nil))
```

However, we'd prefer to have lists of any type, not just integers.

```
data List a = Nil
            | Cons a (List a)
```

Now `List` is a type constructor which takes another type as its argument.

```
 Nil :: List a
 Cons :: a -> List a -> List a
```

Kinds

An example value of type `String` is:

```
"I am a String!"
```

An example value of type `List Char` is:

```
Cons 'b' (Cons 'r' (Cons 'a' (Cons 'p' Nil)))
```

What is an example value of type `List` ? . . . (There isn't one).

A list has to be a list of things of *some type*

The `List` type constructor takes a type and returns a new type. It has *kind*:

```
List :: * -> *
```

It is like a *function* in the world of *types*

Haskell List Syntax

Haskell has a special syntax for lists.

The structure is the same, but the names are different.

```
data [a] =   []
           |   a : [a]
```

- `[]` as a data constructor, is similar to `Nil`.
- `(:)` is similar to `Cons`, but we use it *infix*.
- `[ ]` as a type constructor is similar to `List`, but we use it *outfix*.

This special syntax is an example of *syntactic sugar*.

For example, try the following at a HUGS/GHCi console:

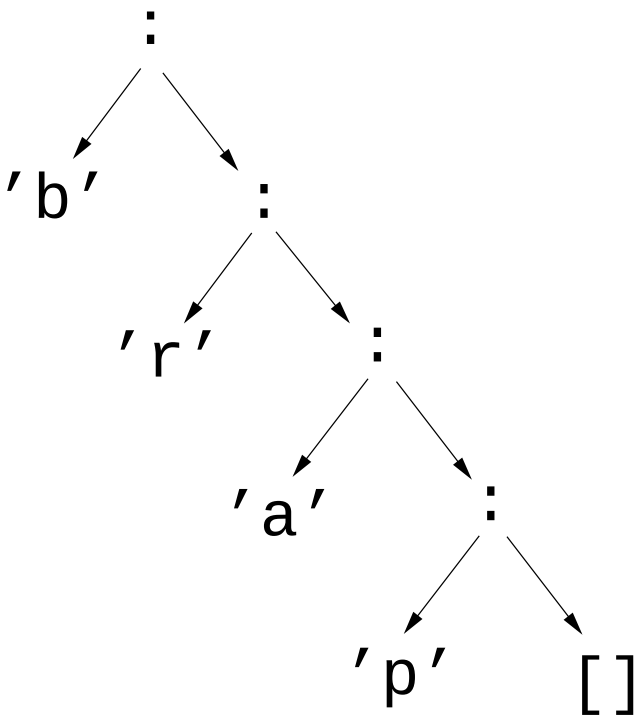
```
:type (:)
```

Lists and Strings

In Haskell:

```
type String = [Char]
```

To the left of a cons is an element, and to the right is *another list*.

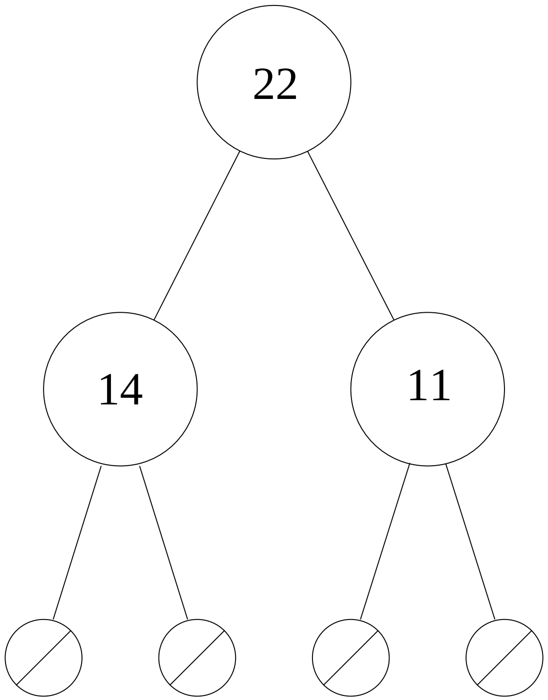



```
list = "brap"
     = ['b', 'r', 'a', 'p']
     = 'b' : ('r' : ('a' : ('p' : [])))
```

Binary Trees

Binary trees are *multiply recursive*:

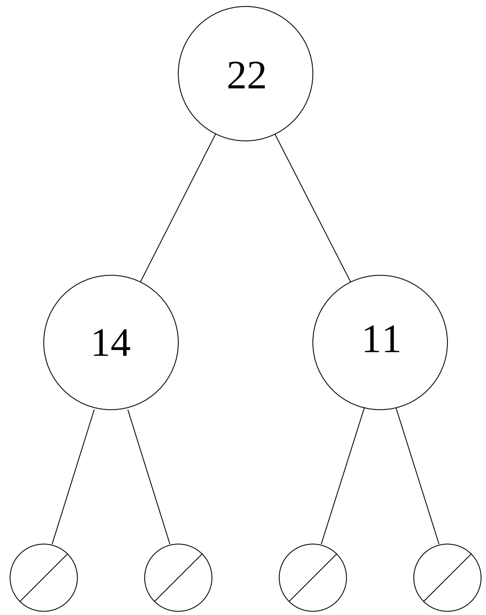
```
data Tree a
          =  Nul | Node a (Tree a) (Tree a)
```




```
Node 22 (Node 14 Nul Nul) (Node 11 Nul Nul)
```

Summing the elements of a tree

```
sum :: Tree Int -> Int
sum  Nul           = 0
sum (Node n t1 t2)
   = n + sum t1 + sum t2
```



Using our example tree:

```
sum (Node 22 (Node 14 Nul Nul) (Node 11 Nul Nul))
  ⟹  22 + sum (Node 14 Nul Nul) + sum (Node 11 Nul Nul)
  ⟹  22 + (14 + sum Nul + sum Nul) + (11 + sum Nul + sum Nul)
  ⟹  22 + (14 + 0 + 0) + (11 + 0 + 0)
```

Flattening a tree

```
flatten :: Tree a -> [a]
flatten Nul                = []
flatten (Node a t1 t2)
     = flatten t1 ++ [a] ++ flatten t2
```

Or a nicer version, using an accumulator: this avoids multiple uses of `(++)` which take time proportional to the length of the first list.

`flatten' t acc` flattens tree `t` and appends it to list `acc`

```
flatten :: Tree a -> [a]
flatten tree                = flatten' tree []

flatten' :: Tree a -> [a] -> [a]
flatten' Nul            acc = acc
flatten' (Node a t1 t2) acc
   = flatten' t1 (a : flatten' t2 acc)
```

Flattening a tree: example

Example of `flatten'`, using $\emptyset$ for `Nul` for brevity:

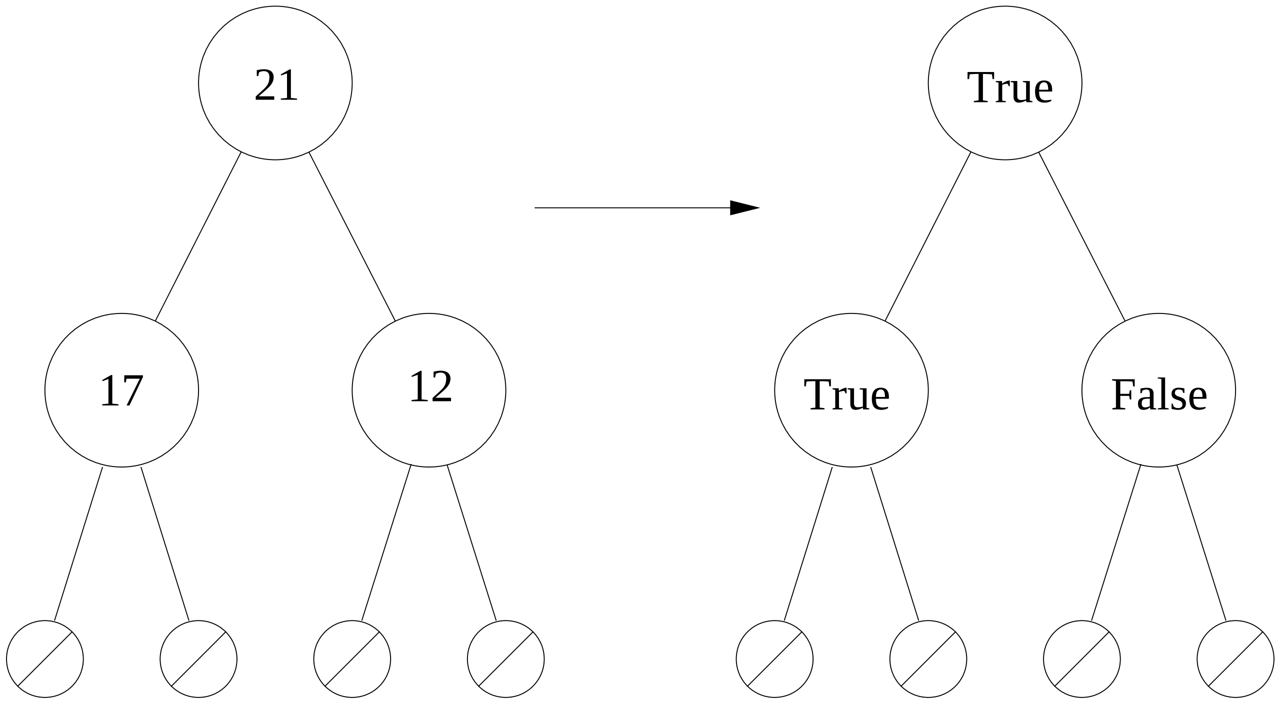
```
flatten' (Node 22 (Node 14 ∅ ∅) (Node 11 ∅ ∅)) []
  ⟹  flatten' (Node 14 ∅ ∅) (22 : flatten' (Node 11 ∅ ∅) [])
  ⟹  flatten' (Node 14 ∅ ∅) (22 : flatten' ∅ (11 : flatten' ∅ []))
  ⟹  flatten' (Node 14 ∅ ∅) (22 : (11 : []))
  ⟹  flatten' ∅ (14 : flatten' ∅ (22 : (11 : [])))
  ⟹  (14 : flatten' ∅ (22 : (11 : [])))
  ⟹  (14 : (22 : (11 : [])))
```

Mapping over Trees

The `map` function for trees is similar to the list version.

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Nul = Nul
mapTree f (Node x t1 t2)
      = Node (f x) (mapTree f t1) (mapTree f t2)
```

For example, we could map the oddness predicate over a tree of integers to give a tree of Boolean values (with the same structure).



Expression Trees

Expressions in most languages (and compilers) are defined inductively.

The Glasgow Haskell Compiler uses an expression tree in the same spirit as:

```
data Op    = OAdd | OSub | OMul
           | OAnd | ONot | OEq | OGt

data Lit   = LBool Bool | LInt Int

data Exp   = XLit  Lit
           | XPrim Op  Exp Exp
           | XIf   Exp Exp Exp
           ...
```

Example

An example expression:

```
if (5 * 3) > 14 then (7 - 3) else (2 + 5)
```

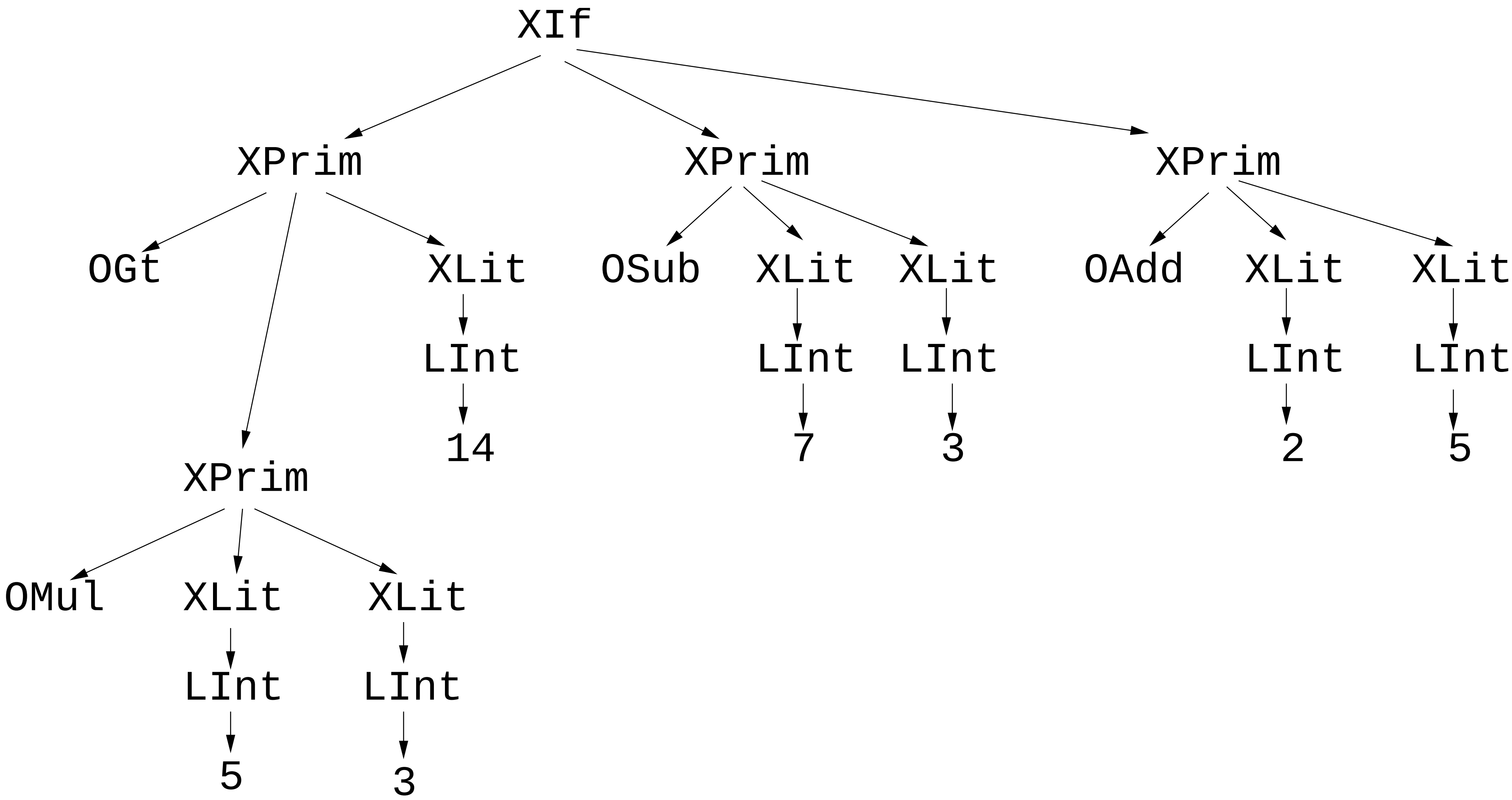
Represented as a tree:

```
exp1
= XIf (XPrim OGt
         (XPrim OMul (XLit (LInt 5)) (XLit (LInt 3)))
         (XLit (LInt 14)))
      (XPrim OSub
         (XLit (LInt 7))
         (XLit (LInt 3)))
      (XPrim OAdd
        (XLit (LInt 2))
        (XLit (LInt 5)))
```

Example

The same expression:

```
if (5 * 3) > 14 then (7 - 3) else (2 + 5)
```



Evaluation

```
evalX :: Exp -> Lit
evalX (XLit l)                = l
evalX (XPrim OAdd x1 x2)
    = LInt  (takeI (evalX x1) + takeI (evalX x2))
...
evalX (XIf x1 x2 x3)
    | takeB (evalX x1)  = evalX x2
    | otherwise         = evalX x3
```

```
takeI :: Lit -> Int
takeI (LInt i) = i
takeI _         = error "takeI: not an int!"
```

Expression Trees – Rewrites

Many useful compiler optimisations can be represented as local rewrites on the expression tree representing the program.

Here is a rewrite which evaluates a literal addition:

```
shortX :: Exp -> Exp
shortX (XPrim OAdd (XLit (LInt i1)) (XLit (LInt i2)))
= XLit (LInt (i1 + i2))
```

For example, we would like

```
if (5 * 3) > 14 then (7 - 3) else (2 + 5)
```

to rewrite to:

```
if (5 * 3) > 14 then (7 - 3) else 7
```

Applying rewrites in a more general manner

We would prefer not to write code to walk over the tree every time we define a new rewrite.

Solution: use a map function to apply a simple rewrite to all sub-expressions:

```
mapX :: (Exp -> Exp) -> Exp -> Exp

mapX f (XLit l)
    = f (XLit l)

mapX f (XPrim op x1 x2)
    = f (XPrim op (mapX f x1) (mapX f x2))

mapX f (XIf x1 x2 x3)
    = f (XIf (mapX f x1) (mapX f x2) (mapX f x3))
```

Example: see Expression.hs, compare `pprX exp1` and `pprX exp1r`.