Lec 5 Datatypes

* Giving name to type

type point = int * int

↑ synonym

* Defining new type

Say we want a
enum {
  lessThan
  equal
  greaterThan
}

→ int ? works, but possible to have errors

datatype order = LESS | EQUAL | GREATER ← Convention is to capitalise or all caps

"constructers". they are value of type order!

LESS : order

we can also use these for pattern matching

Built in comparison already does this.

Int.compare : int * int → order

```
(case Int.compare (x, y) of
    LESS ⇒ ...
  | EQUAL ⇒ ...
  | GREATER ⇒ ...      )
```

datatype bool = true | false

↑ build in does this

Different type same name?

⚠ Beware this

type point = int * int
type point = int * int

Nothing much happens. Still same type

datatype order = ...
datatype order = ...

Two `order` but different

Consider

```
(* listmin int list -> ?
 :
```

? : → int ? what if listmin([]) ... uh oh
→ something like Option<i32> ? Yes!

extended integer

```
datatype extint = PosInf | NegInf | Finite of int
```

Finite of int: this constructer carries an int!

```
fun listmin ([]:int list):extint = PosInf
  | listmin (x::xs) =
    (case listmin xs of
        PosInf => Finite x
      | Finite y => Finite (Int.min(x,y))
      | NegInf => NegInf )
```

So:

PosInf : extint
NegInf : extint ← Not function

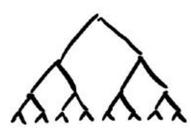
Finite 12 : extint
└ we say this whole thing is a value

Finite : int → extint
└ this is a function instead.

Consider:

```
fun listmin ([]:int list):extint = PosInf
  | listmin (x::xs) =
    (case listmin xs of
        PosInff => Finite x
      | Finite y => Finite (Int.min(x,y))
      | NegInf => NegInf )
```

PosInff ← then compiler thinks this is variable name so it matches everything.



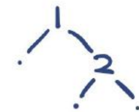
← Foreshadowing

Trees

```
datatype tree = Empty
              | Node of tree * int * tree
```

Ex.

Node(Empty, 1, Node(Empty, 2, Empty))



Depth of tree

Not really good contract to prove

```
(* depth tree -> int *)
fun depth (Empty : tree) = 0
  | depth (Node(t1, x, t2)) = 1 + Int.max(depth t1, depth t2)
```

Theorem: depth is total on T.

* total: for any value T : tree, depth $T \hookrightarrow v$ for some v.

Structural induction on T

(BC) T = Empty.
Well, depth T $\Rightarrow$ 0 [clause 1 of depth]
Coose $v = 0$. then depth T $\Rightarrow v$ as required.

(IC) T = Node(t_1, x, t_2) for some t_1 : tree, x : int, t_2 : tree
(IH) depth $t_1 \hookrightarrow v_1$ and depth $t_2 \hookrightarrow v_2$ for some value v_1, v_2.
(WTS) depth $T \hookrightarrow v$ for some value v

depth T $\Rightarrow$ 1 + Int.max(depth t1, depth t2) [clause 2 of depth]
$\cdots\Rightarrow$ 1 + Int.max(v_1, v_2) [by IH]
$\Rightarrow$ 1 + k [Int.max total]
Choose $v = 1 + k$

$\square$

More tree (with all data at leaf)

```
datatype tree = Leaf of int
              | Node of tree * tree

(* flatten : tree -> int list *)
fun flatten (Leaf x : tree) : int list = [x]
  | flatten (Node (t1, t2))            = (flatten t1) @ (flatten t2)
```

bad performance ... $O(n^2)$ worst case

```
(* flatten2 : tree * int list -> int list
   ENS: true
   REQ: flatten2 (T, acc) ≅ flatten T @ acc
*)
fun flatten2 ( Leaf(x): tree, acc: int list ): int list = x::acc
  | flatten2 ( Node (t1, t2), acc ) = flatten2(t1, flatten2(t2, acc))
```

A tail call

Not tail call !

⚠ Not tail recursive !