Lec 9 Polymorphism

Observations: nil : t list for any t
x::xs : t list if x : t, xs : t list

* Notice things have flexible type?

What SML does:

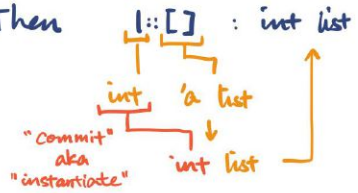
datatype 'a list = nil | :: of 'a * 'a list

[and a command that makes :: an infix op]

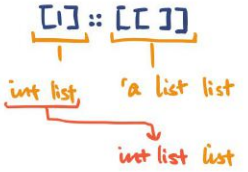
So what's type of nil? — 'a list ! → or 'b list, etc.

↑ takes the type of 'a

Then 1::[] : int list

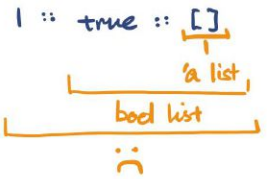


Consider [[]] ≅ [] :: []



Also true :: [] : bool list

But 1 :: true :: []



(☹)

([], []) : 'a list * 'b list

Alpha tree

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* trav : 'a tree → 'a list
   REQ  ...
   ENS  flatten tree into list
*)
fun trav (Empty : 'a tree) : 'a list = []
  | trav (Node (L, x, R)) = trav L @ x :: trav R
```

↳ Built-in @ (op@) is 'a list * 'a list → 'a list

Type analysis

trav (Empty) : 'a list ?

But [] :: [] is value cuz [] is val and :: is constructor. Empty is not constructor. For this class constructor applied to val is val

← compiler not happy if polymorphic things are not value.

trav (Node (Empty, 1, Empty)) : int list

Zip

```
(* zip : 'a list * 'b list → ('a * 'b) list
   REQ  true
   ENS  [ zip and drop extra in longer list ]
*)
fun zip ([] : 'a list, _ : 'b list) : ('a * 'b) list = []
  | zip (_, []) = []
  | zip (x::xs, y::ys) = (x,y) :: zip (xs, ys)

val L = zip ([1, 2, 3, 4, 5], ["a", "b", "c", "d"]) : (int * string) list
```

↳ [(1, "a"), (2, "b"), (3, "c"), (4, "d")]

Look up

```
(* lookup : ('a*'a -> bool) * 'a * ('a*'b) list -> 'b option *)
fun lookup (_:'a*'a -> bool, _:'a, []:('a*'b) list) : 'b option = NONE
  | lookup (eq, x, (a,b)::rest) =
      if eq(x, a) then SOME b else lookup (eq, x, rest)
```

tail recursive

```
datatype 'a option = NONE
  | SOME of 'a
```

Also, we can't always pattern match to check equality

e.g. `lookup ((op =), 2, L) : string option`
↪ `SOME "b"`

`lookup ((op =), 42, L) : string option`
↪ `NONE`

Type inference

two types, neither most general. # usually defaults to int addition.

sometimes there are corner cases, such as `trav(Empty)` and `+`

* ML almost always tries to come up with the most general type of expressions

Def: 1. t is most general type for e if $e:t$
2. $\forall t'$, $e:t'$, t' is an instance of t.

Def: if t' can be obtained from t by consistently instantiating type variables

→ this t is unique up to type variable choice i.e. 'a ⇔ 'b

Type inference example

```
- fun square y = y*y*1 : int -> int
```

int/real? ? int!

- `fun first (x,y) = x : 'a * 'b -> 'a`
- `fun sqrf (f, x) = square (f x) : ('a -> int) * 'a -> int`
- `fun g x = g x : 'a -> 'b` ← tail recursive. will loop forever
  Do we even have any interesting 'a -> 'b?
  Turns out we can prove no. Take 312.
- `fun h x = h (h x) : 'a -> 'a` ← not tail recursive. will loop until run out of memory
- `fun id x = x : 'a -> 'a`
  - `id square 12 : int`
  - `square (id 12) : int`
  - `square id 12 : ill-typed`

`id square 12` ≅ `(id square) 12` left associative