Lecture 9

Review: What have we seen so far?

- (Monomorphic) Types
- Functions (incl. Lambdas)
- Recursion (incl. Tail)
- Datatypes
- Pattern Matching
- Work & Span
- Equivalence
- Totality
- Induction
- Proof of Correctness

Preview: Where are we heading?

- Polymorphism (Staring today)
- Higher Order Functions
- Modular Systems

Polymorphism

A Polymorphic List

Question. How do we define a list type that can take any types of elements?

> We need a "**type variable**".

```
datatype 'a list = nil | :: of 'a * 'a list
(* The cons operator takes a 'a type and cons it to a 'a list. *)
infixr :: (* The normal definition of cons. *)
```

Consider the type of the following lists:

```
[1] : int list
[] : 'a list (* The most general type of this list. *)
1 :: [] (* The 'a is instantiated as int in this expression. *)
```

> Note: `int list` is an "**instance**" of `'a list` .

Consider: `[[]] : 'a list list` .

A Polymorphic Tree

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

(* trav : 'a tree -> 'a list *)
fun trav (Empty : 'a tree) : 'a list = []
  | trav (Node(L, x, R)) = (trav L) @ (x :: (trav v))
```

> Note: `@` is polymorphic, so we're good. This definition is purely structural.

Zip

```
(* zip : 'a list * 'b list -> ('a * 'b) list
 * REQ: true
 * ENS: zip ([a1,..., am], [b1,..., bn]) === [(a1, b1),..., (ak, bk)], k = min(n, m)
 *)
fun zip (nil : 'a list, _ : 'b list) : ('a * 'b) list = []
  | zip (_, nil) = []
  | zip (a::xs, b::ys) = (a, b) :: (zip (xs, ys))
(* This is a demonstration of double polymorphism. *)
```

Option

Question. If you want to find something, but it's not necessarily there, what do we do?

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

(* 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) :: L) =
    case EQ (x, a) of
      true => SOME b
    | false => lookup (EQ, x, L)
```

> Note: If we simply use `=` in place of `EQ` function, ML will give warning and force using **equality types**. If the above type annotation exists, it will thus conflict with ML warning and ML will give a **type error**.

Type Inference

In deciding whether an expression `e` has a type ML solves a "bunch" of type constraint equations and determines the most general type consistent with the constraints.

Consider: `fun f x = x + 1` has type `fn : int -> int`. This is because while `+` is **overloaded**, `1` is definitely `int`.

Consider: `fun g (x, y) = x + y` could be both `fn : int -> int` and `fn : real -> real`, BUT ML defaults to `fn : int -> int`.

> Note: `+` is simply **overloaded**, NOT polymorphic.

Consider: `fun f x = f x` has type `fn : 'a -> 'b`.

> Note: Function applications are associative **to the left**.

Thus, consider the following function application:

```
fun id x = x
id id 42 (* This is same as the following line. *)
(id id) 42
```

The left most `id` has type `('a -> 'a) -> ('a -> 'a)`.