

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

# Programming Languages

Dan Grossman  
2013

*Optional: The Value Restriction and  
Other Type-Inference Challenges*

# *Two more topics*

- ML type-inference story so far is too lenient
  - Value restriction limits where polymorphic types can occur
  - See why and then what
- ML is in a “sweet spot”
  - Type inference more difficult without polymorphism
  - Type inference more difficult with subtyping

Important to “finish the story” but these topics are:

- A bit more advanced
- A bit less elegant
- Will not be on the exam

# The Problem

As presented so far, the ML type system is *unsound*!

- Allows putting a value of type `t1` (e.g., `int`) where we expect a value of type `t2` (e.g., `string`)

A combination of polymorphism and mutation is to blame:

```
val r = ref NONE (* val r : 'a option ref *)  
val _ = r := SOME "hi"  
val i = 1 + valOf (!r)
```

- Assignment type-checks because (infix) `:=` has type `'a ref * 'a -> unit`, so instantiate with `string`
- Dereference type-checks because `!` has type `'a ref -> 'a`, so instantiate with `int`

# What to do

To restore soundness, need a stricter type system that rejects at least one of these three lines

```
val r = ref NONE (* val r : 'a option ref *)  
val _ = r := SOME "hi"  
val i = 1 + valOf (!r)
```

- And cannot make special rules for reference types because type-checker cannot know the definition of all type synonyms
  - Module system coming up

```
type 'a foo = 'a ref  
val f = ref (* val f : 'a -> 'a foo *)  
val r = f NONE
```

# *The fix*

```
val r = ref NONE (* val r : ?.X1 option ref *)  
val _ = r := SOME "hi"  
val i = 1 + valOf (!r)
```

- Value restriction: a variable-binding can have a polymorphic type only if the expression is a variable or value
  - Function calls like **ref NONE** are neither
- Else get a warning and unconstrained types are filled in with dummy types (basically unusable)
- Not obvious this suffices to make type system sound, but it does

# *The downside*

As we saw previously, the value restriction can cause problems when it is unnecessary because we are not using mutation

```
val pairWithOne = List.map (fn x => (x,1))  
(* does not get type 'a list -> ('a*int) list *)
```

The type-checker does not know `List.map` is not making a mutable reference

Saw workarounds in previous segment on partial application

- Common one: wrap in a function binding

```
fun pairWithOne xs = List.map (fn x => (x,1)) xs  
(* 'a list -> ('a*int) list *)
```

# *A local optimum*

- Despite the value restriction, ML type inference is elegant and fairly easy to understand
- More difficult *without* polymorphism
  - What type should length-of-list have?
- More difficult *with* subtyping
  - Suppose pairs are supertypes of wider tuples
  - Then `val (y,z) = x` constrains `x` to have at least two fields, not exactly two fields
  - Depending on details, languages can support this, but types often more difficult to infer and understand
  - Will study subtyping later, but not with type inference