Polymorphism

COS 326
Speaker: Andrew Appel
Princeton University
POLY-HO!

polymorphic, higher-order programming
Here’s an annoying thing

let rec map (f:int->int) (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl);;

What if I want to increment a list of floats?
Alas, I can’t just call this map. It works on ints!
Here’s an annoying thing

let rec map (f:int->int) (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl);;

What if I want to increment a list of floats?
Alas, I can’t just call this map. It works on ints!

let rec mapfloat (f:float->float) (xs:float list) :
  float list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(mapfloat f tl);;
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)

let ints = map (fun x -> x + 1) [1; 2; 3; 4]

let floats = map (fun x -> x +. 2.0) [3.1415; 2.718]

let strings = map String.uppercase ["sarah"; "joe"]
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)

map : ('a -> 'b) -> 'a list -> 'b list
Type of the undecorated map?

```
let rec map f xs =
  match xs with
  | []  -> []
  | hd::tl -> (f hd)::(map f tl)

map : ('a -> 'b) -> 'a list -> 'b list
```

Read as:
• for any types 'a and 'b,
• if you give map a function from 'a to 'b,
• it will return a function
  – which when given a list of 'a values
  – returns a list of 'b values.

We often use greek letters like \( \alpha \) or \( \beta \) to represent type variables.
We can say this explicitly

```
let rec map (f:'a -> 'b) (xs:'a list) : 'b list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

map : ('a -> 'b) -> 'a list -> 'b list

The OCaml compiler is smart enough to figure out that this is the most general type that you can assign to the code. (technical term: principal type)

We say map is polymorphic in the types 'a and 'b – just a fancy way to say map can be used on any types 'a and 'b.

Java generics derived from ML-style polymorphism (but added after the fact and more complicated due to subtyping)
More realistic polymorphic functions

```ml
let rec merge (lt:'a->'a->bool) (xs:'a list) (ys:'a list) : 'a list =
    match (xs,ys) with
    | ([],_) -> ys
    | (_,[]) -> xs
    | (x::xst, y::yst) ->
        if lt x y then x::(merge lt xst ys)
        else y::(merge lt xs yst)

let rec split (xs:'a list) (ys:'a list) (zs:'a list) : 'a list * 'a list =
    match xs with
    | [] -> (ys, zs)
    | x::rest -> split rest zs (x::ys)

let rec mergesort (lt:'a->'a->bool) (xs:'a list) : 'a list =
    match xs with
    | ([] | _::[]) -> xs
    | _ -> let (first,second) = split xs [] [] in
        merge lt (mergesort lt first) (mergesort lt second)
```
More realistic polymorphic functions

mergesort : ('a->'a->bool) -> 'a list -> 'a list

mergesort (<) [3;2;7;1]  
  == [1;2;3;7]

mergesort (>) [2; 3; 42]  
  == [42; 3; 2]

mergesort (fun x y -> String.compare x y < 0) ["Hi"; "Bi"]  
  == ["Bi"; "Hi"]

let int_sort = mergesort (<)  
let int_sort_down = mergesort (>)  
let str_sort = mergesort (fun x y -> String.compare x y < 0)
Another Interesting Function

let comp f g x = f (g x)

let mystery = comp (add 1) square
Function composition!

```ocaml
let comp f g x = f (g x)
let mystery = comp (add 1) square
```

$$(f \circ g)(x) = f \left( g(x) \right)$$

mystery = (add 1) $\circ$ square

mystery(x) = (add 1) (square (x))
What is the type of \( \text{comp} \)?

\[
\text{let } \text{comp} \ f \ g \ x = f \ (g \ x)
\]

\[
\text{let } \text{comp} \ (f: 'b\rightarrow'c) \ (g: 'a\rightarrow'b) \ (x: 'a) : 'c = f \ (g \ x)
\]

\[
\text{comp} : ('b \rightarrow 'c) \rightarrow ('a \rightarrow 'b) \rightarrow ('a \rightarrow 'c)
\]
What does this program do?

```haskell
map f (map g [x1; x2; ...; xn])
```

For each element of the list $x_1, x_2, x_3 \ldots x_n$, it executes $g$, creating:

```haskell
map f ([g x_1; g x_2; ...; g x_n])
```

Then for each element of the list $[g x_1, g x_2, g x_3 \ldots g x_n]$, it executes $f$, creating:

```haskell
[f (g x_1); f (g x_2); ...; f (g x_n)]
```
What does this program do?

map f (map g )

map f

reclaimed by garbage collector
What does this program do?

```
map f (map g [x1; x2; ...; xn])
```

For each element of the list \(x_1, x_2, x_3 \ldots x_n\), it executes \(g\), creating:

```
map f ([g x1; g x2; ...; g xn])
```

Then for each element of the list \([g x_1, g x_2, g x_3 \ldots g x_n]\), it executes \(f\), creating:

```
[f (g x1); f (g x2); ...; f (g xn)]
```

Is there a faster way? Yes! (And query optimizers for SQL do it for you.)

```
map (comp f g) [x1; x2; ...; xn]
```
This kind of optimization has a name: \textit{deforestation}

(because it eliminates intermediate lists and, um, trees...)

\texttt{map f (map g [x1; x2; ...; xn])}

\texttt{map (comp f g) [x1; x2; ...; xn]}
let rec reduce f u xs =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)

What’s the most general type of reduce?
let rec reduce f u xs =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)

What's the most general type of reduce?

Based on the patterns, we know xs must be a ('a list) for some type 'a.
How about reduce?

```ml
let rec reduce f u (xs: 'a list) =
    match xs with
    | []  -> u
    | hd::tl -> f hd (reduce f u tl)
```

What’s the most general type of reduce?
let rec reduce f u (xs: 'a list) =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)

What’s the most general type of reduce?

f is called so it must be a function of two arguments.
let rec reduce (f:? -> ? -> ?) u (xs: 'a list) =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)

What’s the most general type of reduce?
How about reduce?

```ocaml
let rec reduce (f:T -> T -> T) u (xs: 'a list) =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)
```

What’s the most general type of reduce?

Furthermore, `hd` came from `xs`, so `f` must take an `'a` value as its first argument.
let rec reduce (f:'a -> ? -> ?) u (xs: 'a list) =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)

What’s the most general type of reduce?
let rec reduce (f:'a -> ? -> ?) u (xs: 'a list) =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)

What’s the most general type of `reduce`?

The second argument to `f` must have the same type as the result of `reduce`. Let’s call it 'b.
How about reduce?

```ocaml
let rec reduce (f:'a -> 'b -> ?) u (xs: 'a list) : 'b =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)
```

What’s the most general type of reduce?

The result of f must have the same type as the result of reduce overall: 'b.
How about reduce?

```ocaml
let rec reduce (f:'a -> 'b -> 'b) u (xs: 'a list) : 'b =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)
```

What’s the most general type of reduce?
How about reduce?

```ocaml
let rec reduce (f:'a -> 'b -> ?) u (xs: 'a list) : 'b =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)
```

What’s the most general type of reduce?

If `xs` is empty, then `reduce` returns `u`. So `u`’s type must be `'b`.
How about reduce?

```haskell
let rec reduce (f:'a -> 'b -> ?) (u:'b) (xs: 'a list) : 'b =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)
```

What’s the most general type of reduce?
let rec reduce (f:'a -> 'b -> ?) (u:'b) (xs: 'a list) : 'b =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)

What’s the most general type of reduce?

reduce returns the result of f. So f’s result type must be 'b.
let rec reduce (f:'a -> 'b -> 'b) (u:'b) (xs: 'a list) : 'b =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)

What’s the most general type of reduce?
let rec reduce (f:'a -> 'b -> 'b) (u:'b) (xs: 'a list) : 'b =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)

What’s the most general type of reduce?

('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
What does this do?

let rec reduce f u xs =
  match xs with
  | []    -> u
  | hd::tl -> f hd (reduce f u tl)

let mystery0 = reduce (fun x y -> 1+y) 0
let rec reduce f u xs =
  match xs with
  | []    -> u
  | hd::tl -> f hd (reduce f u tl);

let mystery0 = reduce (fun x y -> 1+y) 0;;

let rec mystery0 xs =
  match xs with
  | []    -> 0
  | hd::tl ->
    (fun x y -> 1+y) hd (reduce (fun ... ) 0 tl)
What does this do?

```
let rec reduce f u xs =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl);

let mystery0 = reduce (fun x y -> 1+y) 0;;

let rec mystery0 xs =
    match xs with
    | [] -> 0
    | hd::tl ->
        (fun x y -> 1+y) hd (reduce (fun ... ) 0 tl)
```
let rec reduce f u xs =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl);

let mystery0 = reduce (fun x y -> 1+y) 0;;

let rec mystery0 xs =
    match xs with
    | [] -> 0
    | hd::tl ->
        (fun y -> 1+y) (reduce (fun ... ) 0 tl)
What does this do?

```
let rec reduce f u xs =
    match xs with
    | []  -> u
    | hd::tl -> f hd (reduce f u tl)

let mystery0 = reduce (fun x y -> 1+y) 0

let rec mystery0 xs =
    match xs with
    | []  -> 0
    | hd::tl -> 1 + reduce (fun ... ) 0 tl
```
let rec reduce f u xs =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)

let mystery0 = reduce (fun x y -> 1+y) 0

let rec mystery0 xs =
  match xs with
  | [] -> 0
  | hd::tl -> 1 + mystery0 tl
What does this do?

```
let rec reduce f u xs =
    match xs with
    | [] -> u
    | hd::tl -> f hd (reduce f u tl)

let mystery0 = reduce (fun x y -> 1+y) 0

let rec mystery0 xs =
    match xs with
    | [] -> 0
    | hd::tl -> 1 + mystery0 tl List Length!
```
let rec reduce f u xs =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl);;

let mystery1 = reduce (fun x y -> x::y) []
What does this do?

```ocaml
let rec reduce f u xs =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)

let mystery1 = reduce (fun x y -> x::y) []

let rec mystery1 xs =
  match xs with
  | [] -> []
  | hd::tl -> hd::(mystery1 tl)  Copy!
```
let rec reduce f u xs =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)

let mystery2 g =
  reduce (fun a b -> (g a)::b) []
let rec reduce f u xs =
    match xs with
    | [] => u
    | hd::tl => f hd (reduce f u tl)

let mystery2 g =
    reduce (fun a b => (g a)::b) []

let rec mystery2 g xs =
    match xs with
    | [] => []
    | hd::tl => (g hd)::(mystery2 g tl) map!
Map and Reduce

We coded **map** in terms of **reduce**:

- **ie**: we showed we can compute `map f xs` using a call to `reduce ? ? ?` just by passing the right arguments in place of `? ? ?`

Can we code **reduce** in terms of **map**?
let reduce \( f \) \( u \) \( xs \) = ... map (...)(...) ...

(use only: map, \( f \), \( u \), \( xs \); don’t use rec)

reduce (+) 0 [1;2;3] = ... map (...) (...) ...
### Some Other Combinators: List Module

[https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html](https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html)

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val fold_left : ('a -&gt; 'b -&gt; 'a) -&gt; 'a -&gt; 'b list -&gt; 'a</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>val fold_right : ('a -&gt; 'b -&gt; 'b) -&gt; 'a list -&gt; 'b -&gt; 'b</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>val mapi : (int -&gt; 'a -&gt; 'b) -&gt; 'a list -&gt; 'b list</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| \[
\text{List.mapi } f \ [a_0; \ldots; a_n] = [f \ 0 \ a_0; \ldots; f \ n \ a_n]
\]
| `val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list` |
| \[
\text{List.map2 } f \ [a_0; \ldots; a_n] \ [b_0; \ldots; b_n] = [f \ a_0 \ b_0; \ldots; f \ a_n \ b_n]
\]
| `val iter : ('a -> unit) -> 'a list -> unit` |
| \[
\text{List.iter } f \ [a_0; \ldots; a_n] = f \ a_0; \ldots; f \ a_n
\]
Summary

• Map and reduce are two higher-order functions that capture very, very common recursion patterns.

• Reduce is especially powerful:
  – related to the “visitor pattern” of OO languages like Java.
  – can implement most list-processing functions using it, including things like copy, append, filter, reverse, map, etc.

• We can write clear, terse, reusable code by exploiting:
  – higher-order functions
  – anonymous functions
  – first-class functions
  – polymorphism
Using map, write a function that takes a list of pairs of integers, and produces a list of the sums of the pairs.

- e.g., `list_add [(1,3); (4,2); (3,0)] = [4; 6; 3]
- Write `list_add` directly using `reduce`.

Using map, write a function that takes a list of pairs of integers, and produces their quotient if it exists.

- e.g., `list_div [(1,3); (4,2); (3,0)] = [Some 0; Some 2; None]
- Write `list_div` directly using `reduce`.

Using reduce, write a function that takes a list of optional integers, and filters out all of the None’s.

- e.g., `filter_none [Some 0; Some 2; None; Some 1] = [0;2;1]
- Why can’t we directly use `filter`? How would you generalize `filter` so that you can compute `filter_none`? Alternatively, rig up a solution using `filter` + `map`.

Using reduce, write a function to compute the sum of squares of a list of numbers.

- e.g., `sum_squares = [3,5,2] = 38`