A Few More Thoughts on Types & Lists

COS 326
David Walker
Princeton University
Java has a paucity of types
  – There is no type to describe just the pairs
  – There is no type to describe just the triples
  – There is no type to describe the pairs of pairs
  – There is no type ...

OCaml has many more types
  – use option when things may be null
  – do not use option when things are not null
  – OCaml types describe data structures more precisely
    • programmers have fewer cases to worry about
    • entire classes of errors just go away
    • type checking and pattern analysis help prevent programmers from ever forgetting about a case
Java has a paucity of types
- There is no type to describe just the pairs
- There is no type to describe just the triples
- There is no type...

OCaml has many more types
- use `option` when things may be null
- do not use `option` when things are not null
- OCaml types describe data structures more precisely
  - programmers have fewer cases to worry about
  - errors just go away
  - type checking and program analysis help prevent programmers from ever forgetting about a case

SCORE: OCAML 1, JAVA 0
Java has a paucity of types
   – but at least when you forget something,
     it *throws an exception* instead of *silently going off the trolley*!

If you forget to check for null pointer in a C program,
   – no type-check error at compile time
   – no exception at run time
   – it might crash right away (that would be best), or
   – it might permit a buffer-overrun (or similar) vulnerability
   – so the hackers pwn you!
Java has a paucity of types
  – but at least when you forget something, it *throws an exception* instead of silently going off a trolley!

If you:
  – no type
  – it might permit a buffer overrun (or similar, vulnerability)
  – so the hackers pwn you!

**SCORE:**
OCAML 1, JAVA 0, C -1
MORE THOUGHTS ON LISTS
• Recall that a list is either:
  – [ ] (the empty list)
  – v :: vs (a value v followed by a *previously constructed list vs*)

• Some examples:

```
let l0 = [];;  (* length is 0 *)
let l1 = 1::l0;; (* length is 1 *)
let l2 = 2::l1;; (* length is 2 *)
let l3 = 3::l2;; (* length is 3 *)
...
```
Consider the following picture. How long is the linked structure?
- Can we build a value with type `int list` to represent it?
Consider This Picture

- How long is it? **Infinitely long?**
- Can we build a value with type `int list` to represent it? **No!**
  - all values with type `int list` have finite length

```
1
- 2
- 4
- 3
```
The List Type

• Is it a good thing that the type list does not contain any infinitely long lists? Yes!

• A terminating list-processing scheme:

```plaintext
let rec f (xs : int list) : int =
    match xs with
    | [] -> ... do something not recursive ...
    | hd:@tail -> ... f tail ...
```

terminates because f only called recursively on smaller lists
A Loopy Program

let rec loop (xs : int list) : int =
  match xs with
  [] -> 0
  | hd::tail -> hd + loop (0::tail)

Does this program terminate?
let rec loop (xs : int list) : int =
    match xs with
    [] -> []
  | hd::tail -> hd + loop (0::tail)

Does this program terminate? **No!** Why not? We call loop recursively on (0::tail). This list is the same size as the original list -- not smaller.
Take-home Message

ML has a *strong type system*

• ML *types say a lot* about the set of values that inhabit them

In this case, the tail of the list is *always* shorter than the whole list

This makes it easy to write functions that terminate; *it would be harder if you had to consider more cases*, such as the case that the tail of a list might loop back on itself. *Moreover OCaml hits you over the head to tell you what the only 2 cases are!*

Note: Just because the list type excludes cyclic structures does not mean that an ML program can't build a cyclic data structure if it wants to. *ML is better than other languages because it gives you control* over the values you want to program with via types!
Rant #2: Imperative lists

• One week from today, ask yourself: Which is easier:
  – Programming with immutable lists in ML?
  – Programming with pointers and mutable cells in C/Java
  – I guarantee you are going to say ML.
• there are so many more cases to worry about in C/Java
• so many more things that can go wrong

SCORE: OCAML 2, JAVA 0
C: why bother?
Do not believe his lies.
let rec xs : int list = 0::xs
SCORE: OCAML 1.8, JAVA 0
C: why bother?
Poly-HO!

COS 326
David Walker
Princeton University

polymorphic, higher-order programming
Some Design & Coding Rules
Some Design & Coding Rules

• *Laziness* can be a really good force in design.

• Never write the same code twice.
  – factor out the common bits into a reusable procedure.
  – better, use someone else’s (well-tested, well-documented, and well-maintained) procedure.

• Why is this a good idea?
  – why don’t we just cut-and-paste snippets of code using the editor instead of creating new functions?
Some Design & Coding Rules

• **Laziness** can be a really good force in design.

• Never write the same code twice.
  
  – factor out the common bits into a reusable procedure.
  
  – better, use someone else’s (well-tested, well-documented, and well-maintained) procedure.

• Why is this a good idea?
  
  – why don’t we just cut-and-paste snippets of code using the editor instead of creating new functions?
  
  – find and fix a bug in one copy, have to fix in all of them.
  
  – decide to change the functionality, have to track down all of the places where it gets used.
Factoring Code in OCaml

Consider these definitions:

```ocaml
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)

let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```
Factoring Code in OCaml

Consider these definitions:

```ocaml
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)

let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```

The code is almost identical – factor it out!
A *higher-order* function captures the recursion pattern:

```ocaml
let rec map (f:int->int) (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```
A *higher-order* function captures the recursion pattern:

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

Uses of the function:

```ocaml
let inc x = x+1
let inc_all xs = map inc xs
```
Factoring Code in OCaml

A higher-order function captures the recursion pattern:

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

Uses of the function:

```ocaml
let inc x = x+1
let inc_all xs = map inc xs

let square y = y*y
let square_all xs = map square xs
```

Writing little functions like inc just so we call map is a pain.
Factoring Code in OCaml

A higher-order function captures the recursion pattern:

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

Uses of the function:

```ocaml
let inc_all xs = map (fun x -> x + 1) xs
let square_all xs = map (fun y -> y * y) xs
```

We can use an anonymous function instead. Originally, Church wrote this function using λ instead of fun:

\( (\lambda x. x+1) \) or \( (\lambda x. x^2) \)
Another example

```ocaml
let rec sum (xs:int list) : int =
    match xs with
    | [] -> 0
    | hd::tl -> hd + (sum tl)

let rec prod (xs:int list) : int =
    match xs with
    | [] -> 1
    | hd::tl -> hd * (prod tl)
```

**Goal:** Create a function called `reduce` that when supplied with a few arguments can implement both `sum` and `prod`. Define `sum2` and `prod2` using `reduce`.

**(Try it)**

**Goal:** If you finish early, use `map` and `reduce` together to find the sum of the squares of the elements of a list.

**(Try it)**
Another example

```ocaml
let rec sum (xs:int list) : int =  
  match xs with  
  | [] -> 0  
  | hd::tl -> hd + (sum tl)

let rec prod (xs:int list) : int =  
  match xs with  
  | [] -> 1  
  | hd::tl -> hd * (prod tl)
```
Another example

```ocaml
let rec sum (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> hd OP (RECURSIVE CALL ON tl)

let rec prod (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> hd OP (RECURSIVE CALL ON tl)
```
Another example

```ml
let rec sum (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (RECURSIVE CALL ON tl)

let rec prod (xs:int list) : int =
  match xs with
  | [] -> b
  | hd::tl -> f hd (RECURSIVE CALL ON tl)
```
let add x y = x + y
let mul x y = x * y

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

let sum xs = reduce add 0 xs
let prod xs = reduce mul 1 xs
Using Anonymous Functions

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

let sum xs = reduce (fun x y -> x+y) 0 xs
let prod xs = reduce (fun x y -> x*y) 1 xs
Using Anonymous Functions

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

let sum xs = reduce (fun x y -> x+y) 0 xs
let prod xs = reduce (fun x y -> x*y) 1 xs

let sum_of_squares xs = sum (map (fun x -> x * x) xs)
let pairify xs = map (fun x -> (x,x)) xs
```
Using Anonymous Functions

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

let sum xs = reduce (+) 0 xs
let prod xs = reduce ( * ) 1 xs

let sum_of_squares xs = sum (map (fun x -> x * x) xs)
let pairify xs = map (fun x -> (x,x)) xs
```
Using Anonymous Functions

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

let sum xs = reduce (+) 0 xs
let prod xs = reduce (*) 1 xs

let sum_of_squares xs = sum (map (fun x -> x * x) xs)
let pairify xs = map (fun x -> (x,x)) xs
```

Wrong
Using Anonymous Functions

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

let sum xs = reduce (+) 0 xs
let prod xs = reduce (*) 1 xs

let sum_of_squares xs = sum (map (fun x -> x * x) xs)
let pairify xs = map (fun x -> (x,x)) xs
```

wrong -- creates a comment! ug. OCaml -0.1
Function declarations:

``` OCaml
let square x = x*x
let add x y = x+y
```

are *syntactic sugar* for:

``` OCaml
let square = (fun x -> x*x)
let add = (fun x y -> x+y)
```

In other words, *functions are values* we can bind to a variable, just like 3 or “moo” or true.

Functions are 2\textsuperscript{nd} class no more!
One argument, one result

Simplifying further:

```ml
let add = (fun x y -> x+y)
```

is shorthand for:

```ml
let add = (fun x -> (fun y -> x+y))
```

That is, add is a function which:

- when given a value x, *returns a function* (fun y -> x+y) which:
  - when given a value y, returns x+y.
**Curried Functions**

**Currying**: verb. gerund or present participle

(1) to prepare or flavor with hot-tasting spices
(2) to encode a multi-argument function using nested, higher-order functions.

\[
\begin{align*}
\text{fun} & \ x \rightarrow \ (\text{fun} \ y \rightarrow \ x+y) \quad (* \ \text{curried} \ *) \\
\text{fun} & \ x \ y \rightarrow \ x + y \quad (* \ \text{curried} \ *) \\
\text{fun} & \ (x,y) \rightarrow x+y \quad (* \ \text{uncurried} \ *)
\end{align*}
\]
Named after the logician Haskell B. Curry (1950s).

– was trying to find minimal logics that are powerful enough to encode traditional logics.
– much easier to prove something about a logic with 3 connectives than one with 20.
– the ideas translate directly to math (set & category theory) as well as to computer science.
– Actually, Moses Schönfinkel did some of this in 1924
  • thankfully, we don't have to talk about Schönfinkelled functions
What is the type of add?

```
let add = (fun x -> (fun y -> x+y))
```

Add’s type is:

```
int -> (int -> int)
```

which we can write as:

```
int -> int -> int
```

That is, the arrow type is right-associative.
What’s so good about Currying?

In addition to simplifying the language, currying functions so that they only take one argument leads to two major wins:

1. We can *partially apply* a function.
2. We can more easily *compose* functions.
Curried functions allow defs of new, *partially applied* functions:

```ml
let add = (fun x -> (fun y -> x+y))
```

Equivalent to writing:

```ml
let inc = add 1
```

which is equivalent to writing:

```ml
let inc = (fun y -> 1+y)
```

also:

```ml
let inc2 = add 2
let inc3 = add 3
```
SIMPLE REASONING ABOUT HIGHER-ORDER FUNCTIONS
We can factor this program

```haskell
let square_all ys = 
  match ys with
  | [] -> []
  | hd::tl -> (square hd)::(square_all tl)
```

into this program:

```haskell
let square_all = map square
```

assuming we already have a definition of map
Reasoning About Definitions

**Goal**: Rewrite definitions so my program is simpler, easier to understand, more concise, ...

**Question**: What are the reasoning principles for rewriting programs without breaking them? For reasoning about the behavior of programs? About the equivalence of two programs?

I want some *rules* that never fail.

```ocaml
let square_all ys = match ys with |
| [] -> [] |
| hd::tl -> (square hd)::(square_all tl)

let square_all = map square
```
**Simple Equational Reasoning**

**Rewrite 1 (Function de-sugaring):**

\[
\text{let } f \ x = \text{body} \quad \equiv \quad \text{let } f = (\text{fun } x \rightarrow \text{body})
\]

**Rewrite 2 (Substitution):**

\[
(\text{fun } x \rightarrow \ldots \ x \ldots) \ \text{arg} \quad \equiv \quad \ldots \ \text{arg} \ldots
\]

if \( \text{arg} \) is a value or, when executed, will always terminate without effect and produce a value

**Rewrite 3 (Eta-expansion):**

\[
\text{let } f = \text{def} \quad \equiv \quad \text{let } f \ x = (\text{def}) \ x
\]

if \( f \) has a function type

chose name \( x \) wisely so it does not shadow other names used in \( \text{def} \)

roughly: all occurrences of \( x \) replaced by \( \text{arg} \) (though getting this exactly right is shockingly difficult)
Eta-expansion is an example of Leibniz’s law

Gottfried Wilhelm von Leibniz
German Philosopher
1646 - 1716

Leibniz’s law:

If every predicate possessed by x is also possessed by y and vice versa, then entities x and y are identical. Frequently invoked in modern logic and philosophy.

Rewrite 3 (Eta-expansion):

```
let f = def
```

```
let f = fun x -> (def)x
```

if f has a function type

chose name x wisely so it does not shadow other names used in def
let rec map f xs =

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

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

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

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

let square_all =
  (fun f ->
    (fun xs ->
      match xs with
      | [] -> []
      | hd::tl -> (f hd)::(map f tl)
      ) square)
Substitute map definition into square_all

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

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

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

let square_all =
    (fun xs ->
        match xs with
        | [] -> []
        | hd::tl -> (square hd)::(map square tl)
    )

argument square substituted for parameter f
let rec map =
  (fun f ->
    (fun xs ->
      match xs with
      | [] -> []
      | hd::tl -> (f hd):::(map f tl))

let square_all ys =
  (fun xs ->
    match xs with
    | [] -> []
    | hd::tl -> (square hd):::(map square tl)
  ) ys

add argument
via eta-expansion
let rec map =
  (fun f ->
    (fun xs ->
     match xs with
     | [] -> []
     | hd::tl -> (f hd)::(map f tl)))

let square_all ys =
  match ys with
  | [] -> []
  | hd::tl -> (square hd)::(map square tl)
So Far

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

let square_all xs = map square xs
```

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

let square_all xs = map square xs

let square_all ys =
  match ys with
  | [] -> []
  | hd::tl -> (square hd)::(map square tl)

let square_all ys =
  match ys with
  | [] -> []
  | hd::tl -> (square hd)::(map square tl)

proof by simple rewriting unrolls definition once

proof by induction eliminates recursive function map
We saw this:

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

let square_all ys = map square
```

Is equivalent to this:

```ocaml
let square_all ys =
  match ys with
  | [] -> []
  | hd::tl -> (square hd)::(map square tl)
```

Morals of the story:
(1) OCaml’s *hot* (higher-order, typed) functions capture recursion patterns
(2) we can figure out what is going on by *equational reasoning*.
(3) ... but we typically need to do *proofs by induction* to reason about recursive (inductive) functions
POLY-HO!
Here’s an annoying thing

```ocaml
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

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

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

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"]
Type of the undecorated map?

```ocaml
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 α or β to represent type variables.
We can say this explicitly

```ocaml
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.

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

```ocaml
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)
```
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)
let comp f g x = f (g x)

let mystery = comp (add 1) square

let comp = fun f -> (fun g -> (fun x -> f (g x)))

let mystery = comp (add 1) square

let mystery = (fun f -> (fun g -> (fun x -> f (g x)))) (add 1) square

let mystery = fun x -> (add 1) (square x)

let mystery x = add 1 (square x)
What does this program do?

\[
\text{map } f \ (\text{map } g \ [x_1; x_2; ...; x_n])
\]

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

\[
\text{map } f \ ([g \ x_1; g \ x_2; ...; g \ x_n])
\]

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

\[
[f \ (g \ x_1); f \ (g \ x_2); ...; f \ (g \ x_n)]
\]

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

\[
\text{map } (\text{comp } f \ g) \ [x_1; x_2; ...; x_n]
\]
Deforestation

\[ \text{map } f \ (\text{map } g \ [x_1; x_2; \ldots; x_n]) \]

This kind of optimization has a name:

**deforestation**

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

\[ \text{map} \ (\text{comp } f \ g) \ [x_1; x_2; \ldots; x_n] \]
What is the type of `comp`?

```ocaml
let comp f g x = f (g x)
```
What is the type of \texttt{comp}?

\begin{verbatim}
let \texttt{comp} \ f \ g \ x = f \ (g \ x) \n\end{verbatim}

\texttt{comp} : ('b -> 'c) -> ('a -> 'b) -> ('a -> 'c)
What is the type of `comp`?

```ocaml
let comp f g x = f (g x)
```

```
comp : ('b -> 'c) ->
      ('a -> 'b) ->
      ('a -> 'c)
```
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 generally

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

```ocaml
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 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.
How about reduce?

```ocaml
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?

Furthermore, hd came from xs, so f must take an 'a value as its first argument.
How about reduce?

```ocaml
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.
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?
let rec reduce (f:'a -> 'b -> ?) u (xs: 'a list) : 'b =
matching 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?

```ocaml
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?
How about reduce?

```ocaml
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
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)
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?

```ocaml
define (reduce f u xs) = match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)
define mystery0 = reduce (fun x y -> 1+y) 0
define 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) []
And this one?

```ocaml
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!
```
We coded map in terms of reduce:

- ie: we showed we can compute \( \text{map } f \text{ xs} \) using a call to \( \text{reduce } ? ? ? \) just by passing the right arguments in place of \( ? ? ? \)

Can we code reduce in terms of map?
Some Other Combinators: List Module

http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

```ocaml
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'a list

val mapi : (int -> 'a -> unit) -> 'a list -> unit
List.mapi f [a0; ...; an] == f 0 a0; ... ; f n an

val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
List.map2 f [a0; ...; an] [b0; ...; bn] == [f a0 b0 ; ... ; f an bn]

val iter : ('a -> unit) -> 'a list -> unit
List.iter f [a0; ...; an] == f a0; ... ; f an
```
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