Poly-HO!

COS 326
Speaker: Andrew Appel
Princeton University

polymorphic, higher-order programming

slides copyright 2017 David Walker and Andrew Appel
permission granted to reuse these slides for non-commercial educational purposes
Some Design & Coding Rules

• Save some software-engineering effort:
  Never write the same code twice.

  “Ooh, I get it! I’ll write the code once, copy-paste it somewhere else . . . that way, I didn’t write the same code twice”

    – What’s wrong with that?
      • 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.

• Instead, a better practice:
  – factor out the common bits into a reusable procedure.
  – even better: use someone else’s (well-tested, well-documented, and well-maintained) procedure.
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
```
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
```
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, Alonzo Church wrote this function using \( \lambda \) 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

```
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)
```
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

```plaintext
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

```ml
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

```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
```
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

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
what does work is: ( * )
More on Anonymous Functions

Function declarations:

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

are *syntactic sugar* for:

```
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, \textit{returns a function} (fun y -> x+y) which:
  - when given a value y, returns x+y.
**curry**: verb

(1) to prepare or flavor with hot-tasting spices

(2) to encode a multi-argument function using nested, higher-order functions.

```
fun x -> (fun y -> x+y) (* curried *)
fun x y -> x + y       (* curried *)
fun (x,y) -> x+y       (* uncurried *)
```
Curried Functions

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
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:

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

```plaintext
let inc = add 1
```

Equivalent to writing:

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

which is equivalent to writing:

```plaintext
let inc y = 1+y
```

also:

```plaintext
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.

```
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):

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

Rewrite 2 (Substitution):

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

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

roughly: all occurrences of \( x \) replaced by \( \text{arg} \) (though getting this \textit{exactly right} is shockingly difficult)

Rewrite 3 (Eta-expansion):

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

if \( f \) has a function type

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

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

*Let’s use these rules*
*to prove that these two functions are equivalent*
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))
Expanding map square

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)
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 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 rec square_all ys =
    match ys with
    | [] -> []
    | hd::tl -> (square hd)::(square_all tl)
We saw this:

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

let square_all = 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