

Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

<http://courses.engr.illinois.edu/cs421>

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

8/31/15

1

Warm-up Scoping Question

Consider this code:

```
let x = 27;;  
let f x =  
  let x = 5 in  
    (fun x -> print_int x) 10;;  
f 12;;
```

What value is printed?

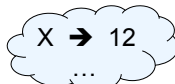
5
10
12
27

8/31/15

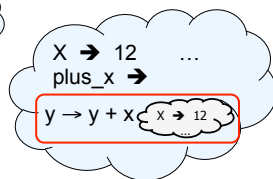
2

Recall: let plus_x = fun x => y + x

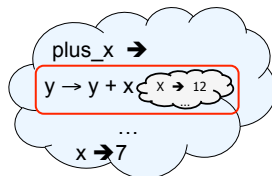
let x = 12



let plus_x = fun y => y + x



let x = 7



8/31/15

3

Closure for plus_x

- When plus_x was defined, had environment:

$\rho_{\text{plus_x}} = \{\dots, X \rightarrow 12, \dots\}$

- Recall: let plus_x y = y + x

is really let plus_x = fun y -> y + x

- Closure for fun y -> y + x:

$\langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle$

- Environment just after plus_x defined:

$\{\text{plus_x} \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle\} + \rho_{\text{plus_x}}$

8/31/15

4

Functions on tuples

```
# let plus_pair (n,m) = n + m;;  
val plus_pair : int * int -> int = <fun>  
# plus_pair (3,4);;  
- : int = 7  
# let double x = (x,x);;  
val double : 'a -> 'a * 'a = <fun>  
# double 3;;  
- : int * int = (3, 3)  
# double "hi";;  
- : string * string = ("hi", "hi")
```

8/31/15

5

Your turn now

Try Problem 1 on ML1

8/31/15

6

Save the Environment!

- A *closure* is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:
$$\langle (v_1, \dots, v_n) \rightarrow \text{exp}, \rho \rangle$$
- Where ρ is the environment in effect when the function is defined (for a simple function)

8/31/15

7

Closure for plus_pair

- Assume $\rho_{\text{plus_pair}}$ was the environment just before `plus_pair` defined
- Closure for `fun (n,m) -> n + m`:
$$\langle (n,m) \rightarrow n + m, \rho_{\text{plus_pair}} \rangle$$
- Environment just after `plus_pair` defined:
$$\{ \text{plus_pair} \rightarrow \langle (n,m) \rightarrow n + m, \rho_{\text{plus_pair}} \rangle \} + \rho_{\text{plus_pair}}$$

8/31/15

8

Your turn now

Try (* 1 *) from HW2

8/31/15

9

Functions with more than one argument

```
# let add_three x y z = x + y + z;;  
val add_three : int -> int -> int -> int = <fun>  
# let t = add_three 6 3 2;;  
val t : int = 11  
# let add_three =  
  fun x -> (fun y -> (fun z -> x + y + z));;  
val add_three : int -> int -> int -> int = <fun>
```

Again, first syntactic sugar for second

8/31/15

10

Your turn now

Try Problem 3 on ML1

8/31/15

11

Curried vs Uncurried

- Recall
`val add_three : int -> int -> int -> int = <fun>`
- How does it differ from
`# let add_triple (u,v,w) = u + v + w;;`
`val add_triple : int * int * int -> int = <fun>`
- `add_three` is *curried*;
- `add_triple` is *uncurried*

8/31/15

12

Curried vs Uncurried

```
# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `';
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
```

8/31/15

13

Partial application of functions

```
let add_three x y z = x + y + z;;
```

```
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

8/31/15

14

Your turn now

Try (* 2 *) from HW2

Caution!

Know what the argument is
and what the body is

8/31/15

15

Functions as arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

8/31/15

16

Your turn now

Try Problem 4 on ML1

8/31/15

17

Evaluating declarations

- Evaluation uses an environment ρ
- To evaluate a (simple) declaration `let x = e`
 - Evaluate expression e in ρ to value v
 - Update ρ with x v : $\{x \rightarrow v\} + \rho$
- Update: $\rho_1 + \rho_2$ has all the bindings in ρ_1 and all those in ρ_2 that are not rebound in ρ_1
 $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{"hi"}\} + \{y \rightarrow 100, b \rightarrow 6\}$
 $= \{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{"hi"}, b \rightarrow 6\}$

8/31/15

18

Evaluating expressions

- Evaluation uses an environment ρ
- A constant evaluates to itself
- To evaluate an variable, look it up in ρ ($\rho(v)$)
- To evaluate uses of $+$, $-$, etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: `let x = e1 in e2`
 - Eval `e1` to `v`, eval `e2` using $\{x \rightarrow v\} + \rho$

8/31/15

19

Evaluation of Application with Closures

- In environment ρ , evaluate left term to closure, $c = \langle (x_1, \dots, x_n) \rightarrow b, \rho \rangle$
- (x_1, \dots, x_n) variables in (first) argument
- Evaluate the right term to values, (v_1, \dots, v_n)
- Update the environment ρ to $\rho' = \{x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n\} + \rho$
- Evaluate body `b` in environment ρ'

8/31/15

20

Evaluation of Application of `plus_x`;

- Have environment:
 $\rho = \{\text{plus_x} \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle, \dots, y \rightarrow 3, \dots\}$
where $\rho_{\text{plus_x}} = \{x \rightarrow 12, \dots\}$
- Eval `(plus_x y, ρ)` rewrites to
- App `($\langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle, 3$)` rewrites to
- Eval `($y + x, \{y \rightarrow 3\} + \rho_{\text{plus_x}}$)` rewrites to
- Eval `($3 + 12, \rho_{\text{plus_x}}$) = 15`

8/31/15

21

Evaluation of Application of `plus_pair`

- Assume environment
 $\rho = \{x \rightarrow 3, \dots, \text{plus_pair} \rightarrow \langle (n,m) \rightarrow n + m, \rho_{\text{plus_pair}} \rangle\} + \rho_{\text{plus_pair}}$
- Eval `(plus_pair (4,x), ρ)` =
- App `($\langle (n,m) \rightarrow n + m, \rho_{\text{plus_pair}} \rangle, (4,3)$)` =
- Eval `($n + m, \{n \rightarrow 4, m \rightarrow 3\} + \rho_{\text{plus_pair}}$)` =
- Eval `($4 + 3, \{n \rightarrow 4, m \rightarrow 3\} + \rho_{\text{plus_pair}}$) = 7`

8/31/15

22

Your turn now

Try `(* 3 *)` from HW2

8/31/15

23

Closure question

- If we start in an empty environment, and we execute:
`let f = fun n -> n + 5;;`
`(* 0 *)`
`let pair_map g (n,m) = (g n, g m);;`
`let f = pair_map f;;`
`let a = f (4,6);;`
What is the environment at `(* 0 *)`?

8/31/15

24

Answer

```
let f = fun n -> n + 5;;
```

```
 $\rho_0 = \{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$ 
```

8/31/15

25

Closure question

- If we start in an empty environment, and we execute:

```
let f = fun => n + 5;;
```

```
let pair_map g (n,m) = (g n, g m);;
```

```
(* 1 *)
```

```
let f = pair_map f;;
```

```
let a = f (4,6);;
```

What is the environment at (* 1 *)?

8/31/15

26

Answer

```
 $\rho_0 = \{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$ 
```

```
let pair_map g (n,m) = (g n, g m);;
```

```
 $\rho_1 = \{pair\_map \rightarrow$   
     $\langle g \rightarrow fun (n,m) -> (g n, g m),$   
     $\{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\},$   
     $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$ 
```

8/31/15

27

Closure question

- If we start in an empty environment, and we execute:

```
let f = fun => n + 5;;
```

```
let pair_map g (n,m) = (g n, g m);;
```

```
let f = pair_map f;;
```

```
(* 2 *)
```

```
let a = f (4,6);;
```

What is the environment at (* 2 *)?

8/31/15

28

Evaluate pair_map f

```
 $\rho_0 = \{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$ 
```

```
 $\rho_1 = \{pair\_map \rightarrow \langle g \rightarrow fun (n,m) -> (g n, g m), \rho_0 \rangle,$   
     $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$ 
```

```
let f = pair_map f;;
```

8/31/15

29

Evaluate pair_map f

```
 $\rho_0 = \{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$ 
```

```
 $\rho_1 = \{pair\_map \rightarrow \langle g \rightarrow fun (n,m) -> (g n, g m), \rho_0 \rangle,$   
     $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$ 
```

```
Eval(pair_map f,  $\rho_1$ ) =
```

8/31/15

30

Evaluate pair_map f

```
 $\rho_0 = \{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$   
 $\rho_1 = \{\text{pair\_map} \rightarrow \langle g \rightarrow \text{fun } (n,m) \rightarrow (g \ n, g \ m), \rho_0 \rangle,$   
   $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$   
Eval(pair_map f,  $\rho_1$ ) =  
Eval(app ( $\langle g \rightarrow \text{fun } (n,m) \rightarrow (g \ n, g \ m), \rho_0 \rangle,$   
   $\langle n \rightarrow n + 5, \{ \} \rangle$ ),  $\rho_1$ ) =
```

8/31/15

31

Evaluate pair_map f

```
 $\rho_0 = \{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$   
 $\rho_1 = \{\text{pair\_map} \rightarrow \langle g \rightarrow \text{fun } (n,m) \rightarrow (g \ n, g \ m), \rho_0 \rangle,$   
   $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$   
Eval(pair_map f,  $\rho_1$ ) =  
Eval(app ( $\langle g \rightarrow \text{fun } (n,m) \rightarrow (g \ n, g \ m), \rho_0 \rangle,$   
   $\langle n \rightarrow n + 5, \{ \} \rangle$ ),  $\rho_1$ ) =  
Eval(fun (n,m)  $\rightarrow (g \ n, g \ m), \{g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\} + \rho_0)$   
=  $\langle (n,m) \rightarrow (g \ n, g \ m), \{g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\} + \rho_0 \rangle$   
=  $\langle (n,m) \rightarrow (g \ n, g \ m), \{g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle$   
   $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$ 
```

8/31/15

32

Answer

```
 $\rho_1 = \{\text{pair\_map} \rightarrow$   
 $\langle g \rightarrow \text{fun } (n,m) \rightarrow (g \ n, g \ m), \{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\},$   
   $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}$   
let f = pair_map f;;  
 $\rho_2 = \{f \rightarrow \langle (n,m) \rightarrow (g \ n, g \ m),$   
   $\{g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle,$   
   $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\},$   
  pair_map  $\rightarrow \langle g \rightarrow \text{fun } (n,m) \rightarrow (g \ n, g \ m),$   
     $\{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}\}$ 
```

8/31/15

33

Closure question

- If we start in an empty environment, and we execute:

```
let f = fun => n + 5;;
```

```
let pair_map g (n,m) = (g n, g m);;
```

```
let f = pair_map f;;
```

```
let a = f (4,6);;
```

(* 3 *)

What is the environment at (* 3 *)?

8/31/15

34

Final Evaluation?

```
 $\rho_2 = \{f \rightarrow \langle (n,m) \rightarrow (g \ n, g \ m),$   
   $\{g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle,$   
   $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\},$   
  pair_map  $\rightarrow \langle g \rightarrow \text{fun } (n,m) \rightarrow (g \ n, g \ m),$   
     $\{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}\}$   
let a = f (4,6);;
```

8/31/15

35

Evaluate f (4,6);;

```
 $\rho_2 = \{f \rightarrow \langle (n,m) \rightarrow (g \ n, g \ m),$   
   $\{g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle,$   
   $f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\},$   
  pair_map  $\rightarrow \langle g \rightarrow \text{fun } (n,m) \rightarrow (g \ n, g \ m),$   
     $\{f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle\}\}$   
Eval(f (4,6),  $\rho_2$ ) =
```

8/31/15

36

Evaluate f (4,6);;

```
ρ2 = {f → <(n,m) →(g n, g m),
      {g → <n → n + 5, { }>,
      f → <n → n + 5, { }>>},
      pair_map → <g → fun (n,m) -> (g n, g m),
      {f → <n → n + 5, { }>>}
Eval(f (4,6), ρ2) =
Eval(app(<(n,m) →(g n, g m),
      {g → <n → n + 5, { }>,
      f → <n → n + 5, { }>>},(4,6)), ρ2) =
```

8/31/15

37

Evaluate f (4,6);;

```
Eval(app(<(n,m) →(g n, g m),
      {g → <n → n + 5, { }>,
      f → <n → n + 5, { }>>},(4,6)), ρ2) =
Eval((g n, g m), {n → 4, m → 6} +
      {g → <n → n + 5, { }>,
      f → <n → n + 5, { }>>}) =
Eval((app(<n → n + 5, { }>, 4),
      app (<n → n + 5, { }>, 6)),
      {n → 4, m → 6, g → <n → n + 5, { }>,
      f → <n → n + 5, { }>>}) =
```

8/31/15

38

Evaluate f (4,6);;

```
ρ3 = {n → 4, m → 6, g → <n → n + 5, { }>,
      f → <n → n + 5, { }>>}
Eval((app(<n → n + 5, { }>, 4),
      app (<n → n + 5, { }>, 6)), ρ3) =
Eval((Eval(n + 5, {n → 4} + { })),
      (Eval(n + 5, {n → 6} + { })), ρ3) =
Eval((Eval(4 + 5, {n → 4} + { })),
      (Eval(6 + 5, {n → 6} + { })), ρ3) =
Eval((9, 11), ρ3) = (9, 11)
```

8/31/15

39

Your turn now

Try (* 4 *) from HW2

8/31/15

40

Match Expressions

```
# let triple_to_pair triple =
  match triple
  with (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int =
  <fun>
```

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause

8/31/15

41

Recursive Functions

```
# let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
# (* rec is needed for recursive function
  declarations *)
```

8/31/15

42

Your turn now

Try Problem 5 on ML1

8/31/15

43

Recursion Example

Compute n^2 recursively using:
$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec nthsq n = (* rec for recursion *)
  match n (* pattern matching for cases *)
  with 0 -> 0 (* base case *)
  | n -> (2 * n - 1) (* recursive case *)
        + nthsq (n - 1);; (* recursive call *)
val nthsq : int -> int = <fun>
# nthsq 3;;
- : int = 9
```

Structure of recursion similar to inductive proof

8/31/15

44

Recursion and Induction

```
# let rec nthsq n = match n with 0 -> 0
  | n -> (2 * n - 1) + nthsq (n - 1) ;;
```

- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- **if** or **match** must contain base case
- Failure of these may cause failure of termination

8/31/15

45

Lists

- First example of a recursive datatype (aka algebraic datatype)
- Unlike tuples, lists are homogeneous in type (all elements same type)

8/31/15

46

Lists

- List can take one of two forms:
 - Empty list, written []
 - Non-empty list, written $x :: xs$
 - x is head element, xs is tail list, $::$ called "cons"
 - Syntactic sugar: $[x] == x :: []$
 - $[x1; x2; \dots; xn] == x1 :: x2 :: \dots :: xn :: []$

8/31/15

47

Lists

```
# let fib5 = [8;5;3;2;1;1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
# (8::5::3::2::1::1::[ ]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
```

8/31/15

48

Lists are Homogeneous

```
# let bad_list = [1; 3.2; 7];;
```

Characters 19-22:

```
let bad_list = [1; 3.2; 7];;  
                ^^^
```

This expression has type float but is here used with type int

8/31/15

49

Question

- Which one of these lists is invalid?

- [2; 3; 4; 6]
- [2,3; 4,5; 6,7]
- [(2.3,4); (3.2,5); (6,7.2)]
- [["hi"; "there"]; ["wahcha"]; []; ["doin"]]

8/31/15

50

Answer

- Which one of these lists is invalid?

- [2; 3; 4; 6]
- [2,3; 4,5; 6,7]
- [(2.3,4); (3.2,5); (6,7.2)]
- [["hi"; "there"]; ["wahcha"]; []; ["doin"]]

- 3 is invalid because of last pair

8/31/15

51

Functions Over Lists

```
# let rec double_up list =  
  match list  
  with [ ] -> [ ] (* pattern before ->,  
                  expression after *)  
       | (x :: xs) -> (x :: x :: double_up xs);;  
val double_up : 'a list -> 'a list = <fun>  
# let fib5_2 = double_up fib5;;  
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1;  
1; 1; 1]
```

8/31/15

52

Functions Over Lists

```
# let silly = double_up ["hi"; "there"];;  
val silly : string list = ["hi"; "hi"; "there"; "there"]  
# let rec poor_rev list =  
  match list  
  with [ ] -> []  
       | (x::xs) -> poor_rev xs @ [x];;  
val poor_rev : 'a list -> 'a list = <fun>  
# poor_rev silly;;  
- : string list = ["there"; "there"; "hi"; "hi"]
```

8/31/15

53

Question: Length of list

- Problem: write code for the length of the list
 - How to start?

```
let length l =
```

8/31/15

54

Question: Length of list

- Problem: write code for the length of the list
 - How to start?

```
let rec length l =  
  match l with
```

8/31/15

55

Question: Length of list

- Problem: write code for the length of the list
 - What patterns should we match against?

```
let rec length l =  
  match l with
```

8/31/15

56

Question: Length of list

- Problem: write code for the length of the list
 - What patterns should we match against?

```
let rec length l =  
  match l with [] ->  
  | (a :: bs) ->
```

8/31/15

57

Question: Length of list

- Problem: write code for the length of the list
 - What result do we give when `l` is empty?

```
let rec length l =  
  match l with [] -> 0  
  | (a :: bs) ->
```

8/31/15

58

Question: Length of list

- Problem: write code for the length of the list
 - What result do we give when `l` is not empty?

```
let rec length l =  
  match l with [] -> 0  
  | (a :: bs) ->
```

8/31/15

59

Question: Length of list

- Problem: write code for the length of the list
 - What result do we give when `l` is not empty?

```
let rec length l =  
  match l with [] -> 0  
  | (a :: bs) -> 1 + length bs
```

8/31/15

60

Same Length

- How can we efficiently answer if two lists have the same length?

8/31/15

61

Same Length

- How can we efficiently answer if two lists have the same length?

```
let rec same_length list1 list2 =  
  match list1 with [] ->  
    (match list2 with [] -> true  
     | (y::ys) -> false)  
  | (x::xs) ->  
    (match list2 with [] -> false  
     | (y::ys) -> same_length xs ys)
```

8/31/15

62

Higher Order Functions

- A function is *higher-order* if it takes a function as an argument or returns one as a result
- Example:

```
# let compose f g = fun x -> f (g x);;  
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

- The type `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b` is a higher order type because of `('a -> 'b)` and `('c -> 'a)` and `-> 'c -> 'b`

8/31/15

63

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;  
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

8/31/15

64

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;  
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

```
# let thrice f = compose f (compose f f);;  
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- Is this the only way?

8/31/15

65

Partial Application

```
# (+);;  
- : int -> int -> int = <fun>  
# (+) 2 3;;  
- : int = 5  
# let plus_two = (+) 2;;  
val plus_two : int -> int = <fun>  
# plus_two 7;;  
- : int = 9
```

- Patial application also called *sectioning*

8/31/15

66

Lambda Lifting

- You must remember the rules for evaluation when you use partial application

```
# let add_two = (+) (print_string "test\n"; 2);;
test
val add_two : int -> int = <fun>
# let add2 = (* lambda lifted *)
  fun x -> (+) (print_string "test\n"; 2) x;;
val add2 : int -> int = <fun>
```

8/31/15

67

Lambda Lifting

```
# thrice add_two 5;;
- : int = 11
# thrice add2 5;;
test
test
test
- : int = 11
```

- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied

8/31/15

68

Partial Application and “Unknown Types”

- Recall `compose plus_two`:

```
# let f1 = compose plus_two;;
val f1 : ('a -> int) -> 'a -> int = <fun>
```

- Compare to lambda lifted version:

```
# let f2 = fun g -> compose plus_two g;;
val f2 : ('a -> int) -> 'a -> int = <fun>
```

- What is the difference?

8/31/15

69

Partial Application and “Unknown Types”

- 'a can only be instantiated once for an expression

```
# f1 plus_two;;
- : int -> int = <fun>
# f1 List.length;;
```

Characters 3-14:

```
f1 List.length;;
^^^^^^^^^^^^^^
```

This expression has type 'a list -> int but is here used with type int -> int

8/31/15

70

Partial Application and “Unknown Types”

- 'a can be repeatedly instantiated

```
# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : 'a list -> int = <fun>
```

8/31/15

71

Functions Over Lists

```
# let rec map f list =
  match list
  with [] -> []
  | (h::t) -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

8/31/15

72

Iterating over lists

```
# let rec fold_left f a list =
  match list
  with [] -> a
   | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a =
  <fun>
# fold_left
  (fun () -> print_string)
  ()
  ["hi"; "there"];;
hithere- : unit = ()
```

8/31/15

73

Iterating over lists

```
# let rec fold_right f list b =
  match list
  with [] -> b
   | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
  <fun>
# fold_right
  (fun s -> fun () -> print_string s)
  ["hi"; "there"]
  ();;
therehi- : unit = ()
```

8/31/15

74

Structural Recursion

- Functions on recursive datatypes (eg lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
 - Recursive calls made to components of structure of the same recursive type
 - Base cases of recursive types stop the recursion of the function

8/31/15

75

Structural Recursion : List Example

```
# let rec length list = match list
  with [ ] -> 0 (* Nil case *)
   | x :: xs -> 1 + length xs;; (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [] is base case
- Cons case recurses on component list xs

8/31/15

76

Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion form of Structural Recursion
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
- Wait until whole structure has been traversed to start building answer

8/31/15

77

Forward Recursion: Examples

```
# let rec double_up list =
  match list
  with [ ] -> [ ]
   | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>

# let rec poor_rev list =
  match list
  with [ ] -> [ ]
   | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

8/31/15

78

Encoding Recursion with Fold

```
# let rec append list1 list2 = match list1 with
  [ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>

# let append list1 list2 =
  fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Base Case

Operation

Recursive Call

8/31/15

79

Mapping Recursion

- One common form of structural recursion applies a function to each element in the structure

```
# let rec doubleList list = match list
  with [ ] -> [ ]
  | x::xs -> 2 * x :: doubleList xs;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

8/31/15

80

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```
# let doubleList list =
  List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

- Same function, but no rec

8/31/15

81

Folding Recursion

- Another common form “folds” an operation over the elements of the structure

```
# let rec multList list = match list
  with [ ] -> 1
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

- Computes $(2 * (4 * (6 * 1)))$

8/31/15

82

Folding Recursion

- multList folds to the right
- Same as:

```
# let multList list =
  List.fold_right
    (fun x -> fun p -> x * p)
    list 1;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

8/31/15

83

How long will it take?

- Remember the big-O notation from CS 225 and CS 273
- Question: given input of size n , how long to generate output?
- Express output time in terms of input size, omit constants and take biggest power

8/31/15

84

How long will it take?

Common big-O times:

- Constant time $O(1)$
 - input size doesn't matter
- Linear time $O(n)$
 - double input \Rightarrow double time
- Quadratic time $O(n^2)$
 - double input \Rightarrow quadruple time
- Exponential time $O(2^n)$
 - increment input \Rightarrow double time

8/31/15

85

Linear Time

- Expect most list operations to take linear time $O(n)$
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: `multList`, `append`
- Integer example: `factorial`

8/31/15

86

Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

```
# let rec poor_rev list = match list
  with [] -> []
      | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

8/31/15

87

Exponential running time

- Hideous running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear

8/31/15

88

Exponential running time

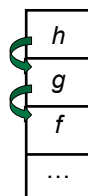
```
# let rec naiveFib n = match n
  with 0 -> 0
      | 1 -> 1
      | _ -> naiveFib (n-1) + naiveFib (n-2);;
val naiveFib : int -> int = <fun>
```

8/31/15

89

An Important Optimization

Normal call



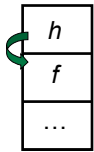
- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if `f` calls `g` and `g` calls `h`, but calling `h` is the last thing `g` does (a *tail call*)?

8/31/15

90

An Important Optimization

Tail
call



- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if f calls g and g calls h , but calling h is the last thing g does (a *tail call*)?
- Then h can return directly to f instead of g

8/31/15

91

Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls
- Tail recursion generally requires extra “accumulator” arguments to pass partial results
 - May require an auxiliary function

8/31/15

92

Tail Recursion - Example

```
# let rec rev_aux list revlist =
  match list with [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
val rev_aux : 'a list -> 'a list -> 'a list = <fun>
```

```
# let rev list = rev_aux list [ ];;
val rev : 'a list -> 'a list = <fun>
```

- What is its running time?

8/31/15

93

Comparison

- $\text{poor_rev } [1,2,3] =$
- $(\text{poor_rev } [2,3]) @ [1] =$
- $((\text{poor_rev } [3]) @ [2]) @ [1] =$
- $((((\text{poor_rev } []) @ [3]) @ [2]) @ [1]) =$
- $(([] @ [3]) @ [2]) @ [1] =$
- $([3] @ [2]) @ [1] =$
- $(3 :: ([] @ [2])) @ [1] =$
- $[3,2] @ [1] =$
- $3 :: ([2] @ [1]) =$
- $3 :: (2 :: ([] @ [1])) = [3, 2, 1]$

8/31/15

94

Comparison

- $\text{rev } [1,2,3] =$
- $\text{rev_aux } [1,2,3] [] =$
- $\text{rev_aux } [2,3] [1] =$
- $\text{rev_aux } [3] [2,1] =$
- $\text{rev_aux } [] [3,2,1] = [3,2,1]$

8/31/15

95

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
  [ ] -> 0 | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let rec prodlist list = match list with
  [ ] -> 1 | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```

8/31/15

96

Folding

```
# let rec fold_left f a list = match list
  with [] -> a | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a =
  <fun>
fold_left f a [x1; x2; ...; xn] = f(...(f (f a x1) x2)...)xn
# let rec fold_right f list b = match list
  with [] -> b | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
  <fun>
fold_right f [x1; x2; ...; xn] b = f x1(f x2(...(f xn b)...))
```

8/31/15

97

Folding - Forward Recursion

```
# let sumlist list = fold_right (+) list 0;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let prodlist list = fold_right ( * ) list 1;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```

8/31/15

98

Folding - Tail Recursion

```
- # let rev list =
-   fold_left
-     (fun l -> fun x -> x :: l) //comb op
-     [] //accumulator cell
-     list
```

8/31/15

99

Folding

- Can replace recursion by fold_right in any forward primitive recursive definition
 - Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition

8/31/15

100