

Programming Languages and Compilers (CS 421)

Grigore Rosu
2110 SC, UIUC

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

Slides by Elsa Gunter, based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

2/4/2014

1

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

2/4/2014

2

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

2/4/2014

3

Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion is a 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

2/4/2014

4

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>
```

2/4/2014

5

Question

- How do you write length with forward recursion?

let rec length l =

2/4/2014

6

Question

- How do you write length with forward recursion?

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

2/4/2014

7

Question

- How do you write length with forward recursion?

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

2/4/2014

8

Question

- How do you write length with forward recursion?

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

2/4/2014

9

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 $('a \rightarrow 'b) \rightarrow ('c \rightarrow 'a) \rightarrow 'c \rightarrow 'b$ is a higher order type because of $('a \rightarrow 'b)$ and $('c \rightarrow 'a)$ and $\rightarrow 'c \rightarrow 'b$

2/4/2014

10

Thrice

- Recall:

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

- How do you write thrice with compose?

2/4/2014

11

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?

2/4/2014

12

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>
```

2/4/2014 13

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

2/4/2014 14

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
```

- Partial application also called *sectioning*

2/4/2014 15

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]
```

2/4/2014 19

Mapping 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]
```

2/4/2014 20

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

2/4/2014 21

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

2/4/2014

22

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
```

2/4/2014

23

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 = ()
```

2/4/2014

24

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
```

2/4/2014

25

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

2/4/2014

26

Question

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

- How do you write length with fold_right, but no explicit recursion?

2/4/2014

27

Question

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

- How do you write length with fold_right, but no explicit recursion?

```
let length list =
  List.fold_right (fun x -> fun n -> n + 1) list 0
```

2/4/2014

28

Map from Fold

```
# let map f list =
  fold_right (fun x -> fun y -> f x :: y) list [];;

val map : ('a -> 'b) -> 'a list -> 'b list =
  <fun>

# map ((+)1) [1;2;3];;
- : int list = [2; 3; 4]
```

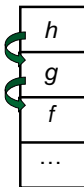
- Can you write fold_right (or fold_left) with just map? How, or why not?

2/4/2014

29

An Important Optimization

Normal
call



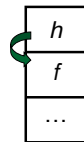
- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if f calls g and g calls h , but calling h is the last thing g does (a *tail call*)?

2/4/2014

30

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

2/4/2014

31

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

2/4/2014

32

Example of Tail Recursion

```
# let rec prod l =
  match l with [] -> 1
  | (x :: rem) -> x * prod rem;;


val prod : int list -> int = <fun>

# let prod list =
  let rec prod_aux l acc =
    match l with [] -> acc
    | (y :: rest) -> prod_aux rest (acc * y)
  (* Uses associativity of multiplication *)
  in prod_aux list 1;;

val prod : int list -> int = <fun>
```

2/4/2014


33

 Question

- How do you write length with tail recursion?

```
let length l =
```

2/4/2014 34


 Question

- How do you write length with tail recursion?

```
let length l =  
  let rec length_aux list n =
```

in

2/4/2014 35


 Question

- How do you write length with tail recursion?

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

in

2/4/2014 36


 Question

- How do you write length with tail recursion?

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

in

2/4/2014 37


 Question

- How do you write length with tail recursion?

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

in

2/4/2014 38

 Question

- How do you write length with tail recursion?

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

in

2/4/2014 39

Question

- How do you write length with tail recursion?

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

2/4/2014

40

Question

- How do you write length with tail recursion?

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

2/4/2014

41

Iterating over lists

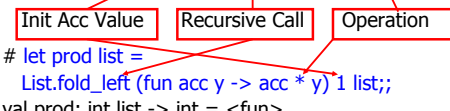
```
# let rec fold_left f a list =  
  match list  
  with [] -> a  
  | (x :: xs) -> fold_left f (f a x) xs;;  
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a =  
  <fun>  
# fold_left  
  (fun () -> print_string)  
  ()  
  ["hi"; "there"];;  
hithere- : unit = ()
```

2/4/2014

42

Encoding Tail Recursion with fold_left

```
# let prod list = let rec prod_aux l acc =  
  match l with [] -> acc  
  | (y :: rest) -> prod_aux rest (acc * y)  
  in prod_aux list 1;;  
val prod : int list -> int = <fun>  
# let prod list =  
  List.fold_left (fun acc y -> acc * y) 1 list;;  
val prod : int list -> int = <fun>  
# prod [4;5;6];;  
- : int = 120
```



2/4/2014

43

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)...))
```

2/4/2014

44

Question

- ```
let length l =
 let rec length_aux list n =
 match list with [] -> n
 | (a :: bs) -> length_aux bs (n + 1)
 in length_aux l 0
```
- How do you write length with fold\_left, but no explicit recursion?

2/4/2014

45

## Question

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

- How do you write length with fold\_left, but no explicit recursion?

```
let length list =
 List.fold_left (fun n -> fun x -> n + 1) 0 list
```

2/4/2014

46

## Recall

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

- What is its running time?

2/4/2014

47

## 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>
```

2/4/2014

48

## 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?

2/4/2014

49

## Comparison

- poor\_rev [1,2,3] =
- (poor\_rev [2,3]) @ [1] =
- ((poor\_rev [3]) @ [2]) @ [1] =
- ((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]

2/4/2014

50

## Comparison

- rev [1,2,3] =
- rev\_aux [1,2,3] [ ] =
- rev\_aux [2,3] [1] =
- rev\_aux [3] [2,1] =
- rev\_aux [ ] [3,2,1] = [3,2,1]

2/4/2014

51



### Folding - Tail Recursion

```

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

```

2/4/2014 52

### 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

2/4/2014 53

### Continuation Passing Style

- A programming technique for all forms of “non-local” control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially it's a higher-order function version of GOTO

2/4/2014 54

### Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an argument to which to pass its result; outer procedure “returns” no result
- Function receiving the result called a continuation
- Continuation acts as “accumulator” for work still to be done

2/4/2014 55

### Example of Tail Recursion

```

let rec app fl x =
 match fl with [] -> x
 | (f :: rem_fs) -> f (app rem_fs x);;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
let app fs x =
 let rec app_aux fl acc =
 match fl with [] -> acc
 | (f :: rem_fs) -> app_aux rem_fs
 (fun z -> acc (f z));;
 in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>

```

2/4/2014 56

### Continuation Passing Style

- Writing procedures so that they take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)

2/4/2014 57

## Example of Tail Recursion & CSP

```
let app fs x =
 let rec app_aux fl acc =
 match fl with [] -> acc
 | (f :: rem_fs) -> app_aux rem_fs
 (fun z -> acc (f z))
 in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
let rec appk fl x k =
 match fl with [] -> k x
 | (f :: rem_fs) -> appk rem_fs x (fun z -> k (f z));;
val appk : ('a -> 'a) list -> 'a -> ('a -> 'b) -> 'b
```

2/4/2014

58

## Continuation Passing Style

- A compilation technique to implement non-local control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics

2/4/2014

59

## Terms

- A function is in Direct Style when it returns its result back to the caller.
- A Tail Call occurs when a function returns the result of another function call without any more computations (eg tail recursion)
- A function is in Continuation Passing Style when it passes its result to another function.
- Instead of returning the result to the caller, we pass it forward to another function.

2/4/2014

60

## Example

- Simple reporting continuation:

```
let report x = (print_int x; print_newline());;
val report : int -> unit = <fun>
```

- Simple function using a continuation:

```
let plusk a b k = k (a + b)
val plusk : int -> int -> (int -> 'a) -> 'a = <fun>
plusk 20 22 report;;
42
- : unit = ()
```

2/4/2014

61

## Simple Functions Taking Continuations

- Given a primitive operation, can convert it to pass its result forward to a continuation
- Examples:  
# let subk x y k = k(x - y);;  
val subk : int -> int -> (int -> 'a) -> 'a = <fun>  
# let eqk x y k = k(x = y);;  
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>  
# let timesk x y k = k(x \* y);;  
val timesk : int -> int -> (int -> 'a) -> 'a = <fun>

2/4/2014

62

## Nesting Continuations

```
let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
let add_three x y z = let p = x + y in p + z;;
val add_three : int -> int -> int -> int = <fun>
let add_three_k x y z k =
 addk x y (fun p -> addk p z k);;
val add_three_k : int -> int -> int -> (int -> 'a)
-> 'a = <fun>
```

2/4/2014

63