Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421D

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

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

$\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{y \rightarrow 100, b \rightarrow 6\}$

$= \{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$

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 $\{x \rightarrow v\} + \rho$

Evaluating conditions expressions

- To evaluate a conditional expression:
  if b then e1 else e2
  - Evaluate b to a value v
  - If v is True, evaluate e1
  - If v is False, evaluate e2

Evaluation of Application with Closures

- Given application expression $f(e_1,\ldots,e_n)$
- Evaluate $(e_1,\ldots,e_n)$ to value $(v_1,\ldots,v_n)$
- In environment ρ, evaluate left term to closure, $c = <(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - $(x_1,\ldots,x_n)$ variables in (first) argument
- Update the environment ρ' to

  $\rho'' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
- Evaluate body b in environment ρ''

Evaluation of Application of plus_x;;

- Have environment:

  $\rho = \{plus\_x \rightarrow <y \rightarrow y + x, \rho_{plus\_x} >, \ldots ,$
  $y \rightarrow 3, \ldots\}$

  where $\rho_{plus\_x} = \{x \rightarrow 12, \ldots , y \rightarrow 24, \ldots\}$
- Eval (plus_x y, ρ) rewrites to
- App (Eval(plus_x, ρ) , Eval(y, ρ)) rewrites to
- App (Eval(plus_x, ρ) , 3) rewrites to
- App ($<y \rightarrow y + x, \rho_{plus\_x} >$, 3) rewrites to

  …

Evaluation of Application of plus_x;;

- Have environment:

  $\rho = \{plus\_x \to <y \to y + x, \rho_{plus\_x}>, \ldots ,$
  $y \to 3, \ldots\}$

  where $\rho_{plus\_x} = \{x \to 12, \ldots , y \to 24, \ldots\}$
- $App\ (<y \to y + x, \rho_{plus\_x}>, 3)$ rewrites to
- $Eval\ (y + x, \{y \to 3\} + \rho_{plus\_x})$ rewrites to
- $Eval\ (y, \{y \to 3\} + \rho_{plus\_x}) +$
  $Eval\ (x, \{y \to 3\} + \rho_{plus\_x})$ rewrites to
- $Eval\ (y, \{y \to 3\} + \rho_{plus\_x}) + 12$ rewrites to
- $3 + 12 = 15$

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3\ldots,$

$\quad plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair}>\} + \rho_{plus\_pair}$

- Eval (plus_pair (4,x), ρ)=
- App (Eval (plus_pair, ρ), Eval ((4,x), ρ)) =
- App (Eval (plus_pair, ρ), (Eval(4,ρ), Eval(x,ρ))) =
- App (Eval (plus_pair, ρ), (Eval(4,ρ), 3)) =
- App (Eval (plus_pair, ρ), (4,3)) =

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \rightarrow 3\ldots,$

$\quad plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair}>\} + \rho_{plus\_pair}$

- App (Eval (plus_pair, ρ), (4,3)) =
- App (<(n,m) $\rightarrow$ n + m, ρ_{plus_pair}>, (4,3)) =
- Eval (n + m, {n -> 4, m -> 3} + ρ_{plus_pair}) =
- Eval (4, {n -> 4, m -> 3} + ρ_{plus_pair}) +

  Eval (3 , {n -> 4, m -> 3} + ρ_{plus_pair}) = 4 + 3 = 7

Evaluation of Curried Functions

Assume ρ_{add_three} is the environment when add_three is defined, and ρ comes after add_three is defind.

Recall:

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

val add_three : int -> int -> int -> int = <fun>

let t = add_three 6 3 2;;

- Eval (((add_three 6) 3) 2, ρ) =
- App (Eval (((add_tree 6) 3), ρ), Eval(2, ρ)) =
- App (Eval (((add_tree 6) 3), ρ), 2) =
- App (App (Eval ((add 6), ρ), Eval(3, ρ)), 2) =

Evaluation of add_three 6 3 2

$\rho = \{x \rightarrow 3\ldots,$
$\quad plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{plus\_pair}>\}$
$\quad + \rho_{plus\_pair}$

- App(App(App(Eval(add_three, ρ),Eval(6, ρ)),3),2) =
- App(App(App(Eval(add_three, ρ),6),3),2) =
- App(App(App(< x ->fun y -> (fun z -> x + y + z), ρ_{add_three} >,6),3),2) =
- App(App(Eval(fun y -> (fun z -> x + y + z), {x -> 6} +ρ_{add_three}), 3),2) =

Evaluation of add_three 6 3 2

- App(App(Eval(fun y -> (fun z -> x + y + z),
  {x -> 6} $+\rho_{add\_three}$), 3),2) =
- App (App(<y -> (fun z -> x + y + z),
  {x -> 6} $+\rho_{add\_three}$ >), 3),2) =
- App (Eval(fun z -> x + y + z,
  {y -> 3, x -> 6} $+\rho_{add\_three}$),2) =
- App (<z -> x + y + z,
  {y -> 3, x -> 6} $+\rho_{add\_three}$ >),2) =
- Eval(x + y + z, {z -> 2, y -> 3, x -> 6} $+\rho_{add\_three}$)
  =

Evaluation of add_three 6 3 2

- Eval(x + y, {z -> 2, y -> 3, x -> 6} $+\rho_{add\_three}$)+ Eval(z, {z -> 2, y -> 3, x -> 6} $+\rho_{add\_three}$) =
- Eval(x + y, {z -> 2, y -> 3, x -> 6} $+\rho_{add\_three}$)+2=
- (Eval(x, {z -> 2, y -> 3, x -> 6} $+\rho_{add\_three}$) + Eval(y, {z -> 2, y -> 3, x -> 6} $+\rho_{add\_three}$))+2=
- (Eval(x, {z -> 2, y -> 3, x -> 6} $+\rho_{add\_three}$) + 3)+2=
- (6+3)+2= 9 + 2 = 11

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 *)
```

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

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

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; …; xn] == x1 :: x2 :: … :: xn :: [ ]

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]
```

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

Question

- Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

Answer

- Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]

- 3 is invalid because of last pair

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]
```

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"]
```

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

Question: Length of list

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

```
let length l =
```

Question: Length of list

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

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

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
```

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) ->
```

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) ->
```

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) ->
```

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
```

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

Same Length

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

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)
```

Higher-Order 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]
```

Recursing 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 = ()
```

The Primitive Recursion Fairy

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

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>
```

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>
```

Base Case | Operator | Recursive Call

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

Base Case | Operator | Recursive Call

Encoding Forward 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>
```

Base Case Operation Recursive Call

```
# 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]
```

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]
```

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

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
```

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)))

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
```

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
```

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
```

How long will it take?

- Remember the big-O notation from CS 225 and CS 374
- 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

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

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

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>
```

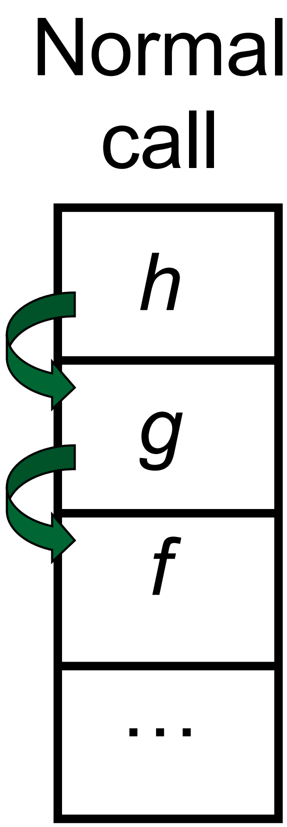
Exponential running time

- Poor worst-case 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

Exponential running time

```
# let rec slow n =
    if n <= 1
    then 1
    else 1+slow (n-1) + slow(n-2);;
val slow : int -> int = <fun>
# List.map slow [1;2;3;4;5;6;7;8;9];;
- : int list = [1; 3; 5; 9; 15; 25; 41; 67;
  109]
```

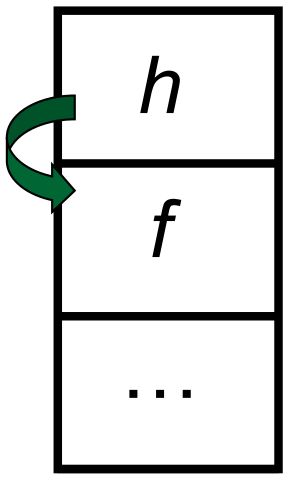
An Important Optimization



- 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*)?

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*

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

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?

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]

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]

Folding - Tail Recursion

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

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 = ()
```

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 $[x_1; x_2;\ldots;x_n]$ = $f(\ldots(f\ (f\ a\ x_1)\ x_2)\ldots)x_n$

```
# 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 $[x_1; x_2;\ldots;x_n]$ b = $f\ x_1(f\ x_2\ (\ldots(f\ x_n\ b)\ldots))$

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