Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://www.cs.uiuc.edu/class/fa06/cs421/

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

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

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

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

Forward Recursion

- In structural recursion, you split your input into components
- In forward recursion, you first call the function recursively on all the recursive components, and then build the final result from the partial results
- Wait until the whole structure has been traversed to start building the 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>
```


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]
```


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
```

- 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
```


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

How long will it take?

Common big-O times:

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

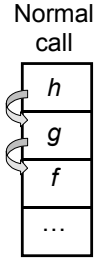

Exponential running time

- 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

Exponential running time

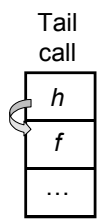
```
# let rec naiveFib n = match n
  with 0 -> 0
  | 1 -> 1
  | _ -> naiveFib (n-1) + naiveFib (n-2);;
val naiveFib : int -> int = <fun>
```


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



- 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]