Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
https://courses.engr.illinois.edu/cs421/sp2023

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

### 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) -> let r = poor_rev xs in r @ [x];;
val poor_rev : 'a list -> 'a list = <fun>

#### Base Case

#### Operator

#### Recursive Call

#### 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) -> 'a list -> 'b = <fun>

#### \# fold_right

#### (fun s -> fun () -> print_string s)

#### ['"hi"; "there"]

#### ()

#### therehi- : unit = ()

#### Folding Recursion : Length Example

#### \# let rec length list =
match list
with [ ] -> 0 (* Nil case *)
| a :: bs -> 1 + length bs; (* Cons case *)
val length : 'a list -> int = <fun>

#### \# let length list =
fold_right (fun a -> fun r -> 1 + r) list 0;;
val length : 'a list -> int = <fun>

#### # length [5; 4; 3; 2];;
- : int = 4

#### Encoding Forward Recursion with Fold

#### \# let rec multList_fr list =

#### ACT 2
let rec multList_fr list = match list with [] -> 1 | (x::xs) -> let r = (multList_fr xs) in (x * r)

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

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>

Extra Material

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

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>
Encoding Forward Recursion with Fold

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

Base Case
```

```ocaml
# let append list1 list2 = fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

```ocaml
# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```
Terminology

- **Available**: An operation that can be executed by the current expression
- The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).
  - if \( h\ x \) then \( f\ x \) else \((x + g\ x)\)
  - if \( h\ x \) then (fun \( x \rightarrow f\ x \)) else \((g\ (x + x))\)

Not available

Tail Position: A subexpression \( s \) of expressions \( e \), which is available and such that if evaluated, will be taken as the value of \( e \)
- if \((x>3)\) then \( x + 2 \) else \( x - 4 \)
- let \( x = g\ 5 \) in \( x + 4 \)

Tail Call: A function call that occurs in tail position
- if \((h\ x)\) then \( f\ x \) else \((x + g\ x)\)

End of Extra Material

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 - length

- How can we write length with tail recursion?

```ml
let length list =
    let rec length_aux list acc_length =
        match list with
        | [] -> acc_length
        | (x::xs) ->
            length_aux xs (1 + acc_length)
    in length_aux list 0
```

Extra Material

Your turn: num_neg – tail recursive

```ml
# let num_neg list =

let rec num_neg_aux list curr_neg =
    match list with
    | [] -> curr_neg
    | (x::xs) ->
        in num_neg_aux  ?  ?
```

Your turn: num_neg – tail recursive

```ml
# let num_neg list =

let rec num_neg_aux list curr_neg =
    match list with
    | [] ->
        match list with [] ->
        | (x::xs) ->
            in num_neg_aux  ?  ?
```
Your turn: num_neg – tail recursive

```ocaml
# let num_neg list =
let rec num_neg_aux list curr_neg =
  match list with
  | [] -> curr_neg
  | (x :: xs) ->
    num_neg_aux xs
in num_neg_aux
```

End of Extra Material
# Iterating over lists

```ml
# 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
```

```ml
# fold_left
  (fun () -> (fun s -> print_string s))
  ()
  ["hi"; "there"];

hithere: unit = ()
```

---

## length, fold_left

```ml
let length list =
  fold_left
    (fun acc -> fun x -> 1 + acc) // comb op
    0 // initial accumulator cell value
    list
```

---

## Extra Material

Your turn: num_neg, fold_left

```ml
let num_neg list =
  fold_left
    ? // comb op
    ? // initial accumulator cell value
```

```ml
let num_neg list =
  fold_left
    (fun curr_neg -> fun x ->
      if x < 0 then 1 + curr_neg else curr_neg)
    // comb op
    0 // initial accumulator cell value
```

```ml
let num_neg list =
  fold_left
    (fun curr_neg -> fun x ->
      if x < 0 then 1 + curr_neg else curr_neg)
    // comb op
    0 // initial accumulator cell value
```
Your turn: `num_neg`, `fold_left`

```ocaml
let num_neg list = fold_left (fun curr_neg -> fun x ->
  if x < 0 then 1 + curr_neg else curr_neg)
  // comb op
  0 // initial accumulator cell value
  list
```

End of Extra Material

```
poor_rev – forward recursive

# let rec poor_rev list =
  match list with [] -> []
  | (x :: xs) -> poor_rev xs @ [x]
```

Extra Material

```
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]

Folding - Tail Recursion

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

End of Extra Material

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 list = <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)...)b)

Folding

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

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

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

```ocaml
# 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]
```

Recall: 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

Terminology

- Available: A function call that can be executed by the current expression
- The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).
  - if (h x) then f x else (x + g x)
  - if (h x) then (fun x -> f x) else (g (x + x))

An Important Optimization

- When a function call is made, the return address needs to be saved to the stack so we know 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)?

```
  Normal call  
  h
  g
  f
  ...
```

```
  Tail call  
  h
  f
  ...
```

- When a function call is made, the return address needs to be saved to the stack so we know 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
Continuations

- 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

Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an extra 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

Continuation Passing Style

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

Why CPS?

- Makes order of evaluation explicitly clear
- Allocates variables (to become registers) for each step of computation
- Essentially converts functional programs into imperative ones
- Major step for compiling to assembly or byte code
- Tail recursion (and forward recursion) easily identified

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
- Possible intermediate state in compiling functional code
Other Uses for Continuations

- CPS designed to preserve order of evaluation
- Continuations used to express order of evaluation
- Can be used to change order of evaluation
- Implements:
  - Exceptions and exception handling
  - Co-routines
  - (pseudo, aka green) threads

Example

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

- Simple function using a continuation:
  ```
  # let addk (a, b) k = k (a + b);
  val addk : int * int -> (int -> 'a) -> 'a = <fun>
  # addk (22, 20) report;
  2
  - : unit = ()
  ```

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>
  ```

Nesting Continuations

- ```
  # let add_triple (x, y, z) = (x + y) + z;;
  val add_triple : int * int * int -> int = <fun>
  ```

- How do we write `add_triple_k` to use a different order?
  ```
  # let add_triple_k (x, y, z) k =
  ```