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

Mapping Functions Over Lists

```ocaml
# 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]
```

Mapping Recursion

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

```ocaml
# 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]
```

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

```ocaml
# 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]
```

Your turn now

Write a function

```ocaml
make_app : (('a -> 'b) * 'a) list -> 'b list
```

that takes a list of function – input pairs and gives the result of applying each function to its argument. Use map, no explicit recursion.

```ocaml
let make_app l =
```

Folding Recursion

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

```ocaml
# 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 Functions over Lists

How are the following functions similar?

```ocaml
# 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 multList list = match list with
[ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 24
```

- Base Case
- Recursive Call
- Head Element
- Combining Operation

Rec value

Rec value

Combining Operation
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>

Folding Recursion

- multList folds to the right
- Same as:

# let multList list =
  List.fold_right (fun x -> fun rv -> x * rv) list 1;;
val multList : int list -> int = <fun>

Encoding Recursion with Fold

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

Your turn now

Try Problem 1 on MP2

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?

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 r -> r + 1) list 0
Map from Fold

```ocaml
# 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?

Iterating over lists

```ocaml
# 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 = ()
```

Encoding Tail Recursion with fold_left

```ocaml
# 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>
```

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

Question

```ocaml
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?

Folding

```ocaml
# 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>
```

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

```ocaml
# 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?

Quadratic Time

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

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

Tail Recursion - Example

```ocaml
# 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>
```

```ocaml
# 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]

Folding - Tail Recursion

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

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]

9/15/16 25
9/15/16 26
9/15/16 27
9/15/16 28
9/15/16 29
9/15/16 30
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

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)

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

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 easily identified
- Strict forward recursion converted to tail recursion
  - At the expense of building large closures in heap
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:
  ```haskell
  # let report x = (print_int x; print_newline( ) );; val report : int -> unit = <fun>
  ```
- Simple function using a continuation:
  ```haskell
  # 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:
  ```haskell
  # 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

```haskell
# let add_triple (x, y, z) = (x + y) + z;;
val add_triple : int * int * int -> int = <fun>
# let add_triple_k (x, y, z) k = addk (x, y) (fun p -> addk (p, z) k );
val add_triple_k: int * int * int -> (int -> 'a) -> 'a = <fun>
```

Try Problem 7 on MP2

Your turn now

Try consk

add_three: a different order

```haskell
# 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 =
```
Your turn now

Try Problem 8 on MP4

Recursion Functions

Recall:

```
let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
```

```
factorial 5;;
- : int = 120
```

```
let rec factorial n =
  let b = (n = 0) in (* First computation *)
  if b then 1 (* Returned value *)
  else let s = n - 1 in (* Second computation *)
        let r = factorial s in (* Third computation *)
        n * r in (* Returned value *) ;;
val factorial : int -> int = <fun>
```

```
factorial 5;;
- : int = 120
```

```
let rec factorialk n k =
  eqk (n, 0) (fun b ->
     (* First computation *)
     if b then k 1
     (* Passed value *)
     else subk (n,) 1
              (fun s -> factorialk s
               (* Third computation *)
               (fun r -> timesk (n, r) k)))
     (* Passed value *)
val factorialk : int -> int = <fun>
```

```
factorialk 5 report;;
120
- : unit = ()
```

To make recursive call, must build intermediate continuation to

- take recursive value: r
- build it to final result: n * r
- And pass it to final continuation:
  - times (n, r) k = k (n * r)

Example: CPS for length

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

What is the let-expanded version of this?
Example: CPS for length

let rec length list = match list with
  | [] -> 0
  | (a :: bs) -> 1 + length bs
What is the let-expanded version of this?
let rec length list = match list with
  | [] -> 0
  | (a :: bs) -> let r1 = length bs in 1 + r1

Your turn now

Try Problem 12 on MP2

Example: CPS for Higher Order Functions

- In CPS, every procedure / function takes a continuation to receive its result
- Procedures passed as arguments take continuations
- Procedures returned as results take continuations
- CPS version of higher-order functions must expect input procedures to take continuations

Example: all

# let rec all (p, l) = match l with
#  | [] -> true
#  | (x :: xs) -> let b = p x in
#    if b then all (p, xs) else false
val all : (‘a -> bool) -> ‘a list -> bool = <fun>
# What is the CSP version of this?
Example: all

```ocaml
# let rec all (p, l) = match l with [] -> true
| (x :: xs) -> let b = p x in
  if b then all (p, xs) else false
val all : ('a -> bool) -> 'a list -> bool = <fun>

- What is the CPS version of this?
- let rec allk (pk, l) k =

9/15/16 55
```

Example: all

```ocaml
# let rec allk (pk, l) k = match l with [] -> k true
| (x :: xs) -> pk x

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

9/15/16 56
```

Example: all

```ocaml
# let rec all (p, l) = match l with [] -> true
| (x :: xs) -> let b = p x in
  if b then all (p, xs) else false
val all : ('a -> bool) -> 'a list -> bool = <fun>

- What is the CPS version of this?
- let rec allk (pk, l) k = match l with [] -> k true

9/15/16 57
```

Example: all

```ocaml
# let rec allk (pk, l) k = match l with [] -> k true
| (x :: xs) -> (fun b -> if b then allk pk xs k else k false)

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

9/15/16 58
```

Example: all

```ocaml
# let rec all (p, l) = match l with [] -> true
| (x :: xs) -> let b = p x in
  if b then all (p, xs) else false
val all : ('a -> bool) -> 'a list -> bool = <fun>

- What is the CPS version of this?
- let rec allk (pk, l) k = match l with [] -> k true

9/15/16 59
```

Example: all

```ocaml
# let rec allk (pk, l) k = match l with [] -> k true
| (x :: xs) -> pk x

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

9/15/16 60
```
Example: all

```ocaml
# let rec all (p, l) = match l with [] -> true
| (x :: xs) -> let b = p x in
  if b then all (p, xs) else false
val all : ('a -> bool) -> 'a list -> bool = <fun>
```

What is the CPS version of this?

```ocaml
# let rec allk (pk, l) k = match l with [] -> k true
| (x :: xs) -> pk x
  (fun b -> if b then allk (pk, xs) k else k false)
val allk : ('a -> (bool -> 'b) -> 'b) * 'a list -> (bool -> 'b) -> 'b = <fun>
```

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, and every function call in it, passes its result to another function.
- Instead of returning the result to the caller, we pass it forward to another function.

Terminology

- **Tail Position**: A subexpression \( s \) of expressions \( e \), such that if evaluated, will be taken as the value of \( e \)
  - if \( x > 3 \) then \( x + 2 \) else \( x - 4 \)
  - let \( x = 5 \) in \( x + 4 \)
- **Tail Call**: A function call that occurs in tail position
  - if \( (h x) \) then \( f x \) else \( x + g x \)

CPS Transformation

- **Step 1**: Add continuation argument to any function definition:
  - let \( f \) arg = \( e \) \( \Rightarrow \) let \( f \) arg k = \( e \)
  - Idea: Every function takes an extra parameter saying where the result goes
- **Step 2**: A simple expression in tail position should be passed to a continuation instead of returned:
  - return \( a \) \( \Rightarrow \) \( k a \)
  - Assuming \( a \) is a constant or variable.
  - “Simple” = “No available function calls.”
CPS Transformation

- Step 4: Each function call not in tail position needs to be converted to take a new continuation (containing the old continuation as appropriate)
  - return \( \text{op} (f \text{ arg}) \Rightarrow f \text{ arg} (\text{fun} r \Rightarrow k(\text{op} r)) \)
  - \( \text{op} \) represents a primitive operation
  - return \( f(g \text{ arg}) \Rightarrow g \text{ arg} (\text{fun} r \Rightarrow f \ r \ k) \)

Example

Before:

```ocaml
let rec add_list lst =
  match lst with
  | [] -> 0
  | 0 :: xs -> add_list xs
  | x :: xs -> (+) x
  (add_list xs);;
```

After:

```ocaml
let rec add_listk lst k =
  (* rule 1 *)
  match lst with
  | [] -> k 0 (* rule 2 *)
  | 0 :: xs -> add_listk xs k
  (* rule 3 *)
  | x :: xs -> add_listk xs
    (fun r -> k ((+) x r));;
  (* rule 4 *)
```

CPS for sum

```ocaml
# let rec sum list = match list with
| [] -> 0
| x :: xs -> x + sum xs ;;
val sum : int list -> int = <fun>
```

```ocaml
# let rec sum list = match list with
| [] -> 0
| x :: xs -> let r1 = sum xs in x + r1;;
val sum : int list -> int = <fun>
```

```ocaml
# let rec sumk list k = match list with
| [] -> k 0
| x :: xs -> sumk xs (fun r1 -> addk (x, r1) k);;
val sumk : int list -> (int -> 'a) -> 'a = <fun>
```

CPS for sum

```ocaml
# let rec sum list = match list with
| [] -> 0
| x :: xs -> x + sum xs ;;
val sum : int list -> int = <fun>
```

```ocaml
# let rec sum list = match list with
| [] -> 0
| x :: xs -> let r1 = sum xs in x + r1;;
val sum : int list -> int = <fun>
```

```ocaml
# let rec sumk list k = match list with
| [] -> k 0
| x :: xs -> sumk xs (fun r1 -> addk (x, r1) k);;
val sumk : int list -> (int -> 'a) -> 'a = <fun>
```

```ocaml
# sumk [2;4;6;8] report;;
20
- : unit = ()
```
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

Exceptions - Example

# exception Zero;;
exception Zero

# let rec list_mult_aux list =
   match list with [ ] -> 1
   | x :: xs ->
      if x = 0 then raise Zero
      else x * list_mult_aux xs;;
val list_mult_aux : int list -> int = <fun>

Exceptions
- When an exception is raised
  - The current computation is aborted
  - Control is “thrown” back up the call stack until a matching handler is found
  - All the intermediate calls waiting for a return values are thrown away

Implementing Exceptions

# let multkp (m, n) k =
    let r = m * n in
    (print_string "product result: ";
     print_int r; print_string "\n";
     k r);;
val multkp : int ( int -> 'a) -> 'a = <fun>

# let rec list_multk_aux list k kexcp =
    match list with [ ] -> k 1
    | x :: xs -> if x = 0 then kexcp 0
      else list_multk_aux xs
         (fun r -> multkp (x, r) kexcp);
val list_multk_aux : int list -> (int -> 'a) -> (int -> 'a)
      -> 'a = <fun>

# let rec list_multk list k = list_multk_aux list k k;;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
Implementing Exceptions

# list_multk [3;4;2] report;;
product result: 2
product result: 8
product result: 24
24
- : unit = ()
#
# list_multk [7;4;0] report;;
0
- : unit = ()