Programming Languages and Compilers (CS 421)

Sasa Misailovic

4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

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

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?

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]

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

- 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]

Your turn now

Write a function

```
map_tail : (‘a -> ‘b) -> ‘a list -> ‘b list
```

that takes a function and a list of inputs and gives the result of applying the function on each argument, but in tail recursive form.

```
let map_tail f lst =
```

Folding - Tail Recursion

```
# let rec rev_aux list revlist =
  match list with
    [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
# let rev list = rev_aux list [ ];;

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


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

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>
```


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

Simplest CPS Example

Identitiy function:

- `let ident x = x`

Identity function in CPS:

- `let identk x ret = ret x`

Example

- 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;;
42
- : unit = ()
```

- Simple reporting continuation:

```
# let report x = (print_int x; print_newline();
                  exit 0);;
val report : int -> unit = <fun>
```


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:

```
# let report x = (print_int x; print_newline();
                  exit 0);;
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;;
42
- : 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>

# let add_triple (x,y,z) = let p = x + y in p + z;;
val add_three : 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>
```


add_three: a different order

```
# let add_triple_k (x, y, z) k =
   addk (x, y) (fun p -> addk (p, z) k );;
```

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

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 $\boxed{\text{x + 2}}$ else $\boxed{\text{x - 4}}$
  - let x = 5 in $\boxed{\text{x + 4}}$
- Tail Call: A function call that occurs in tail position
  - if (h x) then $\boxed{\text{f x}}$ else $\boxed{\text{(x + g x)}}$

Recursive 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
```

Recursive Functions

```
# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);;

# let rec factorial n =
    let b = (n = 0) in   (* 1st computation *)
    if b then 1           (* Returned value *)
    else let s = n – 1 in (* 2nd computation *)
           let r = factorial s in  (* 3rd computation *)
           n * r (* Returned value *) ;;
val factorial : int -> int = <fun>

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


Recursive Functions

```
# let rec factorialk n k =
   eqk (n, 0)
   (fun b ->  (* 1st computation *)
    if b then
      k 1 (* Passed val *)
    else
      subk (n,1)   (* 2nd computation *)
       (fun s -> factorialk s  (* 3rd computation*)
         (fun r -> timesk (n, r) k) (* Passed val*)
       )
   )
val factorialk : int -> int = <fun>

# factorialk 5 report;;
120
```


Recursive Functions

- 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
```


Example: CPS for length

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

What is the CPS version of this?

Example: CPS for length

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

What is the CPS version of this?

```
#let rec lengthk list k = match list with
     [] -> k 0
   | x :: xs -> lengthk xs
                  (fun r -> addk (r,1) k);;
```

```
# lengthk [2;4;6;8] report;;
4
```


Order of Evaluation Matters!

- Your turn (MP2): Write a function quaddk that takes three integer arguments, a,b, and c, and "returns" the result of the expression (2∗(a*b) + 5∗b) + c
- `# let quaddk (a, b, c) k = ... ;;`

- Is the CPS form the same as for 2∗a*b + 5∗b + c ?
  - Refresher: Eval/App slides & Madhu's notes posted on Piazza
  - **MP2 solutions implement the order of evaluation of arithmetic operators from right to left**

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 CPS version of this?

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
```

- What is the CPS version of this?

```
#let rec allk (pk, l) k =
```


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
```

- What is the CPS version of this?

```
#let rec allk (pk, l) k = match l with
         [] ->     true
```

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
```

- What is the CPS version of this?

```
#let rec allk (pk, l) k = match l with
          [] ->  k  true
```


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
```

- What is the CPS version of this?

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


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
```

- What is the CPS version of this?

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


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
```

- What is the CPS version of this?

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


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
```

- What is the CPS version of this?

```
#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           )
```


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
```

- What is the CPS version of this?

```
#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)

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))

Not available

CPS Transformation

- Step 1: Add continuation argument to any function definition:
  - `let f arg = e  ⇒ 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 ⇒ k a`
  - Assuming a is a constant or variable.
  - "Simple" = "No available function calls."

CPS Transformation

- Step 3: Pass the current continuation to every function call in tail position
  - `return f arg ⇒ f arg k`
  - The function "isn't going to return," so we need to tell it where to put the result.

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 op (f arg) ⇒ f arg (fun r -> k(op r))`
  - op represents a primitive operation
  - `return  f(g arg) ⇒ g arg (fun r-> f r k)`

Example

Step 1: Add continuation argument to any function definition
Step 2: A simple expression in tail position should be passed to a continuation instead of returned
Step 3: Pass the current continuation to every function call in tail position
Step 4: Each function call not in tail position needs to be converted to take a new continuation (containing the old continuation as appropriate)

Before:

```
let rec add_list lst =

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

After:

```
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

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


CPS for sum

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

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


CPS for sum

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

# let rec sum list = match list with
      [ ] -> 0
    | x :: xs -> let r1 = sum xs  in x + r1;;

# let rec sumk list k = match list with
       [ ] -> k 0
     | x :: xs -> sumk xs  (fun r1 -> addk x r1 k);;
```


CPS for sum

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

# let rec sum list = match list with
      [ ] -> 0
    | x :: xs -> let r1 = sum xs  in x + r1;;

# let rec sumk list k = match list with
       [ ] -> k 0
     | x :: xs -> sumk xs  (fun r1 -> addk x r1 k);;

# sumk [2;4;6;8] report;;
20
```


Other Uses for Continuations

- CPS designed to **preserve evaluation order**
- **Continuations** used to **express** order of evaluation

- Can also 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 - Example

```
# let list_mult list =
    try list_mult_aux list with Zero -> 0;;
val list_mult : int list -> int = <fun>

# list_mult [3;4;2];;
- : int = 24

# list_mult [7;4;0];;
- : int = 0

# list_mult_aux [7;4;0];;
Exception: Zero.
```


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 -> (int -> 'a) -> 'a =
  <fun>
```


Implementing Exceptions

```
# 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) k)
                        kexcp;;

# let rec list_multk list k =
       list_multk_aux list k
              (fun x -> print_string "nil\n");;
```


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;;
nil
- : unit = ()
```