Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

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

9/9/21

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

```ocaml
# 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>
```

Encoding Forward Recursion with Fold

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

Mapping Recursion
- 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]
```
Mapping Recursion

- 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]
```

- Same function, but no rec

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 \(2 \times (4 \times (6 \times 1))\)

Folding Recursion

- multList folds to the right
- Same as:

``` Ocaml
# 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?

``` 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 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

``` Ocaml
# 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
```
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)?

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.

Comparison

\[
\text{poor\_rev} [1,2,3] = \\
(poor\_rev [2,3]) @ [1] = \\
(((poor\_rev [3]) @ [2]) @ [1] = \\
(((poor\_rev [ ] ) @ [3]) @ [2]) @ [1] = \\
((( ] @ [3]) @ [2]) @ [1] = \\
([3] @ [2]) @ [1] = \\
(3:: ([ ] @ [2])) @ [1] = \\
[3,2] @ [1] = \\
3 :: ([2] @ [1]) = \\
3 :: (2:: ([ ] @ [1]))) = [3, 2, 1]
\]

Tail Recursion - Example

```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 = <fun>

# let rev list = rev_aux list [ ];;
val rev : 'a list -> 'a list = <fun>
```

What is its running time?

Comparison

\[
\text{rev} [1,2,3] = \\
\text{rev\_aux} [1,2,3] [ ] = \\
\text{rev\_aux} [2,3] [1] = \\
\text{rev\_aux} [3] [2,1] = \\
\text{rev\_aux} [ ] [3,2,1] = [3,2,1]
\]
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 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 -> 'b) -> 'a list -> 'b -> 'b = <fun>

  fold_left f a [x_1; x_2;...;x_n] = f(...(f (f a x_1) x_2)...)x_n

  fold_right f [x_1; x_2;...;x_n] b = f x_1(f x_2(...(f x_n b)...))

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 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:
  ```ocaml
  # let report x = (print_int x; print_newline( ));
  val report : int -> unit = <fun>
  # addk (22, 20) report;;
  2-: unit = ()
  ```

- Simple function using a continuation:
  ```ocaml
  # 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:
    ```ocaml
    # 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

```ocaml
# 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_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>
```

### add_three: a different order

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

```

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

### Recursive Functions

**Recall:**

```ocaml
# let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

```ocaml
# let rec factorialk n k =
  eqk (n, 0) (fun b ->
    if b then k 1
    else subk (n, 1) (fun s -> factorialk s n r)
  )
val factorialk : int -> (int -> 'a) -> 'a = <fun>
# factorialk 5 report;;
- : unit = ()
```
Recursive Functions

To make recursive call, must build intermediate continuation to
- take recursive value: \( n \)
- build it to final result: \( n \times r \)
- And pass it to final continuation:
  \[ \text{times}(n, r) k = k(n \times r) \]

Example: CPS for length

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

```ocaml
let rec length list = match list with [] -> 0 | (a :: bs) -> let r1 = length bs in 1 + r1
```

What is the let-expanded version of this?

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

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

```ml
#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?

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

9/9/21 44

Example: all

```ml
#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?

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

9/9/21 47

Example: all

```ml
#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?

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

9/9/21 48
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>
```

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 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 list -> 'a = <fun>
```

```ocaml
# sumk [2;4;6;8] report;;
val sumk : int list -> (int -> 'a) -> 'a list -> 'a = <fun>
- : unit = ()
```
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 (e.g., 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 = $e$ ⇒ 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.”

**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.

**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 $r$ -> $f$ $r$ $k$)

Not available
Example

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 =
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 *)

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

- 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

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>

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>
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;;

val list_multk_aux : int list -> (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 = ()