

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

9/17/24

1

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 -> 'b list -> 'a =  
  <fun>  
# fold_left  
  (fun () -> print_string)  
  ()  
  ["hi"; "there"];;  
hithere- : unit = ()
```

9/17/24

2

Your turn: length, fold_left

```
let length list =
```

9/17/24

3

Tail Recursion - length

- How can we write length with tail recursion?

```
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
```

9/17/24

4

Your turn: length, fold_left

```
let length list =  
  fold_left (fun acc -> fun x -> 1 + acc) 0 list
```

9/17/24

5

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) -> 'a -> 'b list -> 'a =  
  <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)...))
```

9/17/24

10

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

9/17/24

11

Mapping Recursion

```
# 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]
```

9/17/24

12

Map is forward recursive

```
# let rec map f list =  
  match list  
  with [] -> []  
       | (h::t) -> (f h) :: (map f t);;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# let map f list =  
  List.fold_right (fun h -> fun r -> (f h) :: r)  
    list [];;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

9/17/24

13

Mapping Recursion

- Can use the higher-order recursive map function instead of explicit 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]
```

9/17/24

14

Mapping Recursion

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

9/17/24

15

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

9/17/24

17

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

9/17/24

18

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)

9/17/24

19

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

9/17/24

20

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

9/17/24

21

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

9/17/24

22

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

9/17/24

23

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>
```

9/17/24

24

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_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>
```

9/17/24

25

## add\_three: a different order

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

9/17/24

26

## add\_three: a different order

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

9/17/24

27

## 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
```

9/17/24

29

## Terms

- A function is in **Direct Style** when it returns its result back to the caller.
- 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 giving the computation after the call.

9/17/24

30

## Recursive Functions

```
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 (* Returned value *) ;;
val factorial : int -> int = <fun>
factorial 5;;
- : int = 120
```

9/17/24

31

## Recursive Functions

```
let rec factorialk n k =
 eqk (n, 0)
 (fun b -> (* First computation *)
 if b then k 1 (* Passed value *)
 else subk (n, 1) (* Second computation *)
 (fun s -> factorialk s (* Third computation *)
 (fun r -> timesk (n, r) k))) (* Passed value *)
val factorialk : int -> (int -> 'a) -> 'a = <fun>
factorialk 5 report;;
120
- : unit = ()
```

9/17/24

32

## 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:
    - $\text{times}(n, r) k = k(n * r)$

9/17/24

33

425 minutes

9/17/24

34

## 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?

9/17/24

35

## 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 n = length bs in 1 + n
```

```
Let lengthk list k = match list with [] -> k 0
 | (a :: bs) -> lengthk bs (fun n -> addk (1,n)
 k)
```

9/17/24

36

### 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
```

9/17/24

37

### Example: CPS for length

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

What is the CSP version of this?

9/17/24

38

### Example: CPS for length

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

What is the CSP version of this?

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

```
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
```

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

```
4
```

```
- : unit = ()
```

9/17/24

39

450 minutes

### CPS for sum

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

9/17/24

41

- Let sum\_all list =
- let rec sum\_aux (lst, a) =
- match lst with [ ] -> a
- | (x::xs) -> sum\_aux (xs, (x + a))
- in sum\_aux (list, 0)
- Let sum\_allk list k = let rec sum\_auk (lst,a) k1=
- match lst with [ ] -> k1 a
- | (x::xs) -> addk(x, a) (fun r -> sum\_auk (xs,r) k1)
- in sum\_auk(list,0) k

9/19/24

42

## CPS for sum

```
let rec sum list = match list with [] -> 0
 | x :: xs -> x + sum xs ;;
val sum : int list -> int = <fun>
let rec sum list = match list with [] -> 0
 | x :: xs -> let r1 = sum xs in x + r1;;
val sum : int list -> int = <fun>
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 ;;
val sum : int list -> int = <fun>
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 ;;
val sum : int list -> int = <fun>
let rec sum list = match list with [] -> 0
 | x :: xs -> let r1 = sum xs in x + r1;;
val sum : int list -> int = <fun>
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>
sumk [2;4;6;8] report;;
20
- : unit = ()
```

## 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

9/17/24

49

## 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?

9/17/24

50

## 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?

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

9/17/24

51

## 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?

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

9/17/24

52

## 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?

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

9/17/24

53

## 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?

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

9/17/24

54



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

9/17/24

55

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

9/17/24

56

## 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?
#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>
```

9/17/24

57

475 minutes

9/17/24

58