Programming Languages and Compilers (CS 421)

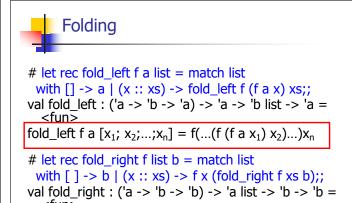


Elsa L Gunter 2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

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

2/6/24



2/6/24 15

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



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

2/6/24



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]
```



2/6/24

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>
```



18

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]
```

2/6/24 19



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

2/6/24

20

23

25



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

2/6/24 22



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

2/6/24



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)

2/6/24



Continuation Passing Style

- A compilation technique to implement nonlocal control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics
- Possible intermediate state in compiling functional code

2/6/24



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

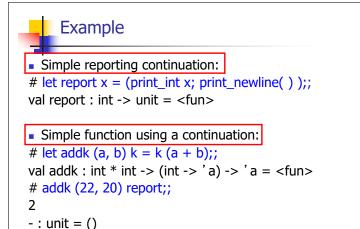
2/6/24 26



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

2/6/24



2/6/24 28



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>
```

2/6/24



27

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>
```

2/6/24 30



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 =



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)

2/6/24

2/6/24

31

32

```
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
```



- 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.

2/6/24 35

```
4
```

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
```

```
# 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 = ()
```



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)

2/6/24



```
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)

2/8/24
```

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

#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 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 = ()
```



```
# 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 -> 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);;
```

2/6/24 48

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

49

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

2/6/24 50



- : unit = ()

Example: all

2/6/24 51

```
#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 =
```

```
#let rec all (p, l) = match | 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 | with [] -> true
```

#let rec all (p, l) = match I 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 I with [] -> k true

2/6/24 54

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

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

