

## Programming Languages and Compilers (CS 421)

---

Elsa L Gunter  
2112 SC, UIUC  
<http://www.cs.uiuc.edu/class/sp07/cs421/>

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

## Continuations

---

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an argument to which to pass its result; outer procedure “returns” no result
- Function receiving the result called a **continuation**

Elsa L. Gunter

## Continuation Passing Style

---

- Writing procedures so that they take a continuation to which to give (*pass*) the result, and return no result, is called **continuation passing style (CPS)**

Elsa L. Gunter

## Continuation Passing Style

---

- 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

Elsa L. Gunter

## 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

Elsa L. Gunter

## 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 passes its result to another function.
- Instead of returning the result to the caller, we pass it forward to another function.

Elsa L. Gunter

## Example

- Simple reporting continuation:  

```
# let report x =  
  (print_int x; print_newline( ) );;  
val report : int -> unit = <fun>
```
- Simple function using a continuation:  

```
# let plusk a b k = k (a + b)  
val plusk : int -> int -> (int -> 'a) -> 'a = <fun>  
# plusk 20 22 report;;  
42  
- : unit = ()
```

Elsa L. Gunter

## 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
```

Elsa L. Gunter

## Recursion Functions

```
# let rec factorialk n k =  
  if n = 0 then k 1 else factorialk (n - 1)  
  (fun m -> k (n * m));;  
val factorialk : int -> (int -> 'a) -> 'a =  
  <fun>  
# factorialk 5 report;;  
120  
- : unit = ()
```

Elsa L. Gunter

## Recursive Functions

- Notice: factorialk is now tail recursive
- To make recursive call, must build *intermediate* continuation to
  - take recursive value: m
  - build it to final result: n \* m
  - And pass it to final continuation:  
k (n \* m)

Elsa L. Gunter

## Nesting CPS

```
# let rec lengthk list k = match list with [ ] -> k 0  
  | x :: xs -> lengthk xs (fun r -> k (r + 1));;  
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>  
# let rec lengthk list k = match list with [ ] -> k 0  
  | x :: xs -> lengthk xs (fun r -> plusk r 1 k);;  
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>  
# lengthk [2;4;6;8] report;;  
4  
- : unit = ()
```

Elsa L. Gunter

## 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>
```

Elsa L. Gunter

## Exceptions - Example

```
# let rec 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.
```

Elsa L. Gunter

## 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 value are thrown away

Elsa L. Gunter

## 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>
```

Elsa L. Gunter

## 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) k0;;
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>
```

Elsa L. Gunter

## 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 = ()
```

Elsa L. Gunter

## Terminology

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

Elsa L. Gunter

## 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).
  - if (h x) then f x else (x + g x)
  - if (h x) then (fun x -> f x) else (x + g x)

Elsa L. Gunter

## 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.”

Elsa L. Gunter

## CPS Transformation

**Step 3:** Pass the current continuation to every function call in tail position

return f arg  $\Rightarrow$  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 built into a new continuation (containing the old continuation as appropriate)

return op (f arg)  $\Rightarrow$  f arg (fun r -> k(op r))

- op represents a primitive operation

Elsa L. Gunter

## 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 =  
  (* 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 *)
```

Elsa L. Gunter

## Continuations Example

```
let add a b k = print_string "Add "; k (a + b);;  
let sub a b k = print_string "Sub "; k (a - b);;  
let report n = print_string "Answer is: ";  
  print_int n;  
  print_newline ();;  
let idk n k = k n;;  
  
type calc = Add of int | Sub of int
```

Elsa L. Gunter

## A Small Calculator

```
# let rec eval lst k =  
  match lst with  
  (Add x) :: xs -> eval xs (fun r -> add r x k)  
| (Sub x) :: xs -> eval xs (fun r -> sub r x k)  
| [] -> k 0  
# eval [Add 20; Sub 5; Sub 7; Add 3; Sub 5]  
report;;  
Sub Add Sub Sub Add Answer is: 6
```

Elsa L. Gunter

## Continuations Can Take Multiple Arguments

```
# add 3 5 (fun r -> sub r 2 report);;
Add Sub Answer is: 6
# add 3 5 (fun r k -> sub r 2 k);;
Add
- : (int -> 'a) -> 'a = <fun>
# add 3 5 ((fun k r -> sub r 2 k) report);;
Add Sub Answer is: 6
```

Elsa L. Gunter

## Composing Continuations

**Problem:** Suppose we want to do all additions before any subtractions

```
let ordereval lst k =
let rec aux lst ka ks = match lst with
| (Add x) :: xs -> aux xs (fun r k -> add r x ka k) ks
| (Sub x) :: xs -> aux xs ka (fun r k -> sub r x ks k)
| [] -> ka 0 ks k
in
aux lst idk idk
```

Elsa L. Gunter

## Sample Run

```
# ordereval [Add 20; Sub 5; Sub 7; Add 3;
Sub 5] report;;
Add Add Sub Sub Sub Answer is: 6
```

Elsa L. Gunter

## Execution Trace

```
ordereval [Add 20; Sub 5; Sub 7] report
aux [Add 20; Sub 5; Sub 7] idk idk report
aux [Sub 5; Sub 7]
  (fun r1 k1 -> add 20 r1 idk k1) idk report
aux [Sub 7] (fun r1 k1 -> add r1 20 idk k1)
  (fun r2 k2 -> sub r2 5 idk k2) report
aux [] (fun r1 k1 -> add r1 20 idk k1)
  (fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
  report
```

Elsa L. Gunter

## Execution Trace

```
aux [] (fun r1 k1 -> add r1 20 idk k1)
  (fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
  report
(* Start calling the continuations *)
(fun r1 k1 -> add r1 20 idk k1)
0
(fun r3 k3 -> sub r3 7
  (fun r2 k2 -> sub r2 5 idk k2) k3)
report
```

Elsa L. Gunter

## Execution Trace

```
(fun r1 k1 -> add r1 20 idk k1)
0
(fun r3 k3 -> sub r3 7
  (fun r2 k2 -> sub r2 5 idk k2) k3)
report
add 0 20 idk (* remember idk n k = k n *)
  (fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
  report
```

Elsa L. Gunter

## Execution Trace

---

```
add 0 20 idk (* remember idk n k = k n *)
  (fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
  report
idk 20
  (fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
  report
```

Elsa L. Gunter

## Execution Trace

---

```
idk 20
  (fun r3 k3 -> sub r3 7
    (fun r2 k2 -> sub r2 5 idk k2) k3)
  report
  (fun r3 k3 -> sub r3 7 (fun r2 k2 -> sub r2 5 idk k2) k3)
  20 report
  sub 20 7 (fun r2 k2 -> sub r2 5 idk k2) report
  (fun r2 k2 -> sub r2 5 idk k2) 13 report
  sub 13 5 idk report
  idk 8 report ---> report 8
```

Elsa L. Gunter