

CS421 Lecture 21: Continuation Passing Style¹

Mark Hills
mhills@cs.uiuc.edu

University of Illinois at Urbana-Champaign

July 27, 2006

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Continuations

Continuation Passing Style

CPS by Hand

Question

Can we use functions to represent the control flow in a program?

Continuations

Yes, using the concept of a **continuation**.

- ▶ We will augment each procedure with an additional argument – a function to which it will pass the current computational result.
- ▶ The outer procedure “returns” no result – it will be kept in the function argument
- ▶ This function argument, receiving the result, will be called the continuation.
- ▶ Note: We saw the use of continuations briefly in the last lecture.

Continuation Passing Style

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

Continuation Passing Style

CPS provides a programming technique for all forms of “non-local” control flow:

- ▶ non-local jumps
- ▶ exceptions
- ▶ etc

CPS turns all non-tail calls into tail calls.

- ▶ Essentially a higher-order functional GOTO

Continuation Passing Style

- ▶ CPS also acts as a compilation technique to implement non-local control flow
- ▶ Especially useful in interpreters
- ▶ Also acts as a formalization of non-local control flow in denotational semantics (again, mentioned in last lecture)

CPS Terminology

- ▶ A function is in **direct style** when it returns its result back to the caller (this is a standard function)
- ▶ A **tail call** occurs when a function returns the result of another function call without any more computations (like in tail recursion, but not restricted to just recursive calls)
- ▶ A function is in **continuation passing style** when it passes its result to another function instead of back to its caller – essentially, we pass the result *forward*, not *backwards*

Example

A simple reporting continuation:

```
1 # let report x =
2   (print_int x; print_newline( ) );;
3 val report : int -> unit = <fun>
```

And a function that uses it:

```
1 # let plusk a b k = k (a + b)
2 val plusk : int -> int -> (int -> 'a) -> 'a = <fun>
3 # plusk 20 22 report;;
4 42
5 - : unit = ()
```

Recursive Functions

Recall our definition of factorial:

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

Recursive Functions

Now for our version with continuations:

```
1 # let rec factorialk n k =
2   if n = 0
3   then k 1
4   else factorialk (n - 1) (fun m -> k (n * m));;
5 val factorialk : int -> (int -> 'a) -> 'a = <fun>
6 # factorialk 5 report;;
7 120
8 - : unit = ()
```

- ▶ Note that our factorial calculation is now tail recursive
- ▶ To make it work, we had to
 - ▶ take a recursive value, m
 - ▶ build it to a final result, n * m
 - ▶ then pass it to the final continuation, k (n * m)

Another Example: Length, Take 1

```
1 # let rec lengthk list k =
2   match list with
3   | [] -> k 0
4   | x :: xs -> lengthk xs (fun r -> k (r + 1));;
5 val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
6 # lengthk [2;4;6;8] report;;
7 4
8 - : unit = ()
```

Another Example: Length, Take 2

```

1 # let rec lengthk list k =
2   match list with
3     | [] -> k 0
4     | x :: xs -> lengthk xs (fun r -> plusk r 1 k);;
5 val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
6 # lengthk [2;4;6;8] report;;
7 4
8 - : unit = ()

```

Example: Exceptions

```

1 # exception Zero;;
2 exception Zero
3 # let rec list_mult_aux list =
4   match list with [ ] -> 1
5   | x :: xs -> if x = 0 then raise Zero
6                 else x * list_mult_aux xs;;
7 val list_mult_aux : int list -> int = <fun>
8 # let rec list_mult list =
9   try list_mult_aux list with Zero -> 0;;
10 val list_mult : int list -> int = <fun>
11 # list_mult [3;4;2];;
12 - : int = 24
13 # list_mult [7;4;0];;
14 - : int = 0
15 # list_mult_aux [7;4;0];;
16 Exception: Zero.

```

Exceptions in OCaml

Exceptions in OCaml work like those we saw in Java in the last lecture:

- ▶ The current computation is aborted;
- ▶ control is “thrown” back up the call stack until a matching handler is found;
- ▶ all intermediate calls waiting for a return value are thrown away.

Continuations as Exceptions

```

1 # let multkp m n k =
2   let r = m * n in
3   (print_string "product result: ";
4    print_int r; print_newline ();
5    k r);;
6 val multkp : int -> int -> (int -> 'a) -> 'a = <fun>
7 # let rec list_multk_aux list k kexcp =
8   match list with [ ] -> k 1
9   | x :: xs -> if x = 0 then kexcp 0
10                else list_multk_aux xs
11                    (fun r -> multkp x r k) kexcp;;
12 val list_multk_aux : int list -> (int -> 'a) ->
13   (int -> 'a) -> 'a = <fun>
14 # let rec list_multk list k = list_multk_aux list k k;;
15 val list_multk : int list -> (int -> 'a) -> 'a = <fun>

```

Exceptions, Part 2

```

1 # list_multk [3;4;2] report;;
2 product result: 2
3 product result: 8
4 product result: 24
5 24
6 - : unit = ()
7 # list_multk [7;4;0] report;;
8 0
9 - : unit = ()

```

More Terminology

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

Performing the CPS Transform

Step 1 Add a continuation argument to any function definition:

$$\text{let } f \text{ args} = e \Rightarrow \text{let } f \text{ args } k = e$$

Step 2 A *simple* expression (one with no available function calls) in tail position should be passed a to a continuation instead of returned – if *a* is a constant or variable, we have:

$$\text{return } a \Rightarrow k \ a$$

where return just indicates that *a* is in a tail position

Performing the CPS Transform

Step 3 Augment every function call in tail position by passing it the current continuation – since the function won't return a value, we pass the continuation forward so we can do something with it.

$$\text{return } f \text{ args} \Rightarrow f \text{ args } k$$

Step 4 Now, each function call not in tail position needs to be built into a new continuation, containing the old continuation and the remainder of the operation that still needs to be performed. Here, *op* is a primitive operation, which could include application!

$$\text{op } (f \text{ args}) \Rightarrow f \text{ args } (\text{fun } r \text{ } \rightarrow k \ (\text{op } r))$$

Example

Before:

```
1
2 let rec add_list lst =
3   match lst with
4     [] -> 0
5   | 0 :: xs ->
6     add_list xs
7   | x :: xs ->
8     (+) x (add_list xs);;
```

After:

```
1 (* rule 1 *)
2 let rec add_listk lst k =
3   match lst with
4     [] -> k 0 (* rule 2 *)
5   | 0 :: xs ->
6     add_listk xs k (* rule 3 *)
7   | x :: xs ->
8     add_listk xs
9     (fun r -> k ((+) x r));;
10  (* rule 4 *)
```

CPS Evaluation Example

```
1 let add a b k = print_string "Add "; k (a + b);;
2 let sub a b k = print_string "Sub "; k (a - b);;
3 let report n = print_string "Answer is: ";
4                 print_int n;
5                 print_newline ();;
6 let idk n k = k n;;
7
8 type calc = Add of int | Sub of int
```

A Simple Calculator

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

Continuations with Multiple Arguments

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

Composing Continuations

What is we wanted to do all additions before any subtractions?

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

Continuation Composition: Sample Run

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

Sample Execution Trace

```
1 ordereval [Add 20; Sub 5; Sub 7] report
2 aux [Add 20; Sub 5; Sub 7] idk idk report
3 aux [Sub 5; Sub 7]
4   (fun r1 k1 -> add 20 r1 idk k1) idk report
5 aux [Sub 7] (fun r1 k1 -> add r1 20 idk k1)
6   (fun r2 k2 -> sub r2 5 idk k2) report
7 aux [] (fun r1 k1 -> add r1 20 idk k1)
8   (fun r3 k3 -> sub r3 7
9     (fun r2 k2 -> sub r2 5 idk k2) k3)
10    report
```

Sample Execution Trace

```
1 aux [] (fun r1 k1 -> add r1 20 idk k1)
2   (fun r3 k3 -> sub r3 7
3     (fun r2 k2 -> sub r2 5 idk k2) k3)
4   report
5 (* Start calling the continuations *)
6 (fun r1 k1 -> add r1 20 idk k1)
7   0
8   (fun r3 k3 -> sub r3 7
9     (fun r2 k2 -> sub r2 5 idk k2) k3)
10  report
11 add 0 20 idk (* remember idk n k = k n *)
12   (fun r3 k3 -> sub r3 7
13     (fun r2 k2 -> sub r2 5 idk k2) k3)
14  report
```

Sample Execution Trace

```
1 add 0 20 idk (* remember idk n k = k n *)
2   (fun r3 k3 -> sub r3 7
3     (fun r2 k2 -> sub r2 5 idk k2) k3)
4   report
5 idk 20
6   (fun r3 k3 -> sub r3 7
7     (fun r2 k2 -> sub r2 5 idk k2) k3)
8   report
9 (fun r3 k3 -> sub r3 7 (fun r2 k2 -> sub r2 5 idk k2) k3)
10 20 report
11 sub 20 7 (fun r2 k2 -> sub r2 5 idk k2) report
12 (fun r2 k2 -> sub r2 5 idk k2) 13 report
13 sub 13 5 idk report
14 idk 8 report ----> report 8
```