Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/sp2023

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

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.

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>
```

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>
```

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)
```

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)
```

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)
```

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
```

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
```

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 = ()
```

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)

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 = ()
```

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?

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
```

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?

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 (r,1) k);;
val lengthk : 'a list -> (int -> 'b) -> 'b = <fun>
# lengthk [2;4;6;8] report;;
4
- : unit = ()
```

CPS for sum

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

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);;
```

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

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?

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 =
```

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
```

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
```

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) ->
```

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
```

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
    )
```

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>
```

Terminology: Review

- 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.
- A Tail Call occurs when a function returns the result of another function call without any more computations (eg tail recursion)
- Instead of returning the result to the caller, we pass it forward to another function giving the computation after the call.

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."

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.

CPS Transformation

- 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) $\Rightarrow$ f arg (fun r -> k(op r))
  - op represents a primitive operation
  - return g(f arg) $\Rightarrow$ f arg (fun r-> g r k)

Example

Before:

```
let rec mem (y,lst) =
match lst with
  [ ] -> false
| x :: xs ->
  if (x = y)
   then true
   else mem(y,xs);;
```

After:

```
let rec memk (y,lst) k =
                     (* rule 1 *)
```

Example

Before:

```
let rec mem (y,lst) =
match lst with
  [ ] -> false
| x :: xs ->
  if (x = y)
   then true
   else mem(y,xs);;
```

After:

```
let rec memk (y,lst) k =
                              (* rule 1 *)

                     k false (* rule 2 *)



                   k true (* rule 2 *)
```

Example

Before:

```
let rec mem (y,lst) =
match lst with
  [ ] -> false
| x :: xs ->
  if (x = y)
   then true
   else mem(y,xs);;
```

After:

```
let rec memk (y,lst) k =
                    (* rule 1 *)

          k false (* rule 2 *)



      k true (* rule 2 *)
        memk (y, xs) k (* rule 3 *)
```

Example

Before:

```
let rec mem (y,lst) =
match lst with
  [ ] -> false
| x :: xs ->
  if (x = y)
   then true
   else mem(y,xs);;
```

After:

```
let rec memk (y,lst) k =
                    (* rule 1 *)

          k false (* rule 2 *)

eqk (x, y)
 (fun b ->   b (* rule 4 *)
      k true (* rule 2 *)
        memk (y, xs) (* rule 3 *)
```

Example

Before:

```
let rec mem (y,lst) =
match lst with
  [ ] -> false
| x :: xs ->
  if (x = y)
   then true
   else mem(y,xs);;
```

After:

```
let rec memk (y,lst) k =
                          (* rule 1 *)

             k false (* rule 2 *)

 eqk (x, y)
  (fun b ->if b (* rule 4 *)
then k true (* rule 2 *)
   else memk (y, xs) (* rule 3 *)
```

Example

Before:

```
let rec mem (y,lst) =
match lst with
  [ ] -> false
| x :: xs ->
  if (x = y)
   then true
   else mem(y,xs);;
```

After:

```
let rec memk (y,lst) k =
                        (* rule 1 *)
match lst with
| [ ] -> k false (* rule 2 *)
| x :: xs ->
 eqk (x, y)
  (fun b ->if b (* rule 4 *)
then k true (* rule 2 *)
   else memk (y, xs) k (* rule 3 *)
```

Example

Before:

```
let rec mem (y,lst) =
match lst with
  [ ] -> false
| x :: xs ->
  if (x = y)
   then true
   else mem(y,xs);;
```

After:

```
let rec memk (y,lst) k =
                        (* rule 1 *)
match lst with
| [ ] -> k false (* rule 2 *)
| x :: xs ->
 eqk (x, y)
  (fun b ->if b (* rule 4 *)
then k true (* rule 2 *)
   else memk (y, xs) k (* rule 3 *)
```

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 *)
```