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

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)

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
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

Example

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

Simple function using a continuation:
```ocaml
# let addk (a, b) k = k (a + b);
val addk : int * int -> (int -> 'a) -> 'a = <fun>
# addk (22, 20) report;;
2
- : unit = ()
```

Simple Functions Taking Continuations

- Given a primitive operation, can convert it to pass its result forward to a continuation

Examples:
```ocaml
# 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

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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:

```ml
# 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

```ml
# let rec factorial n =  
    let b = (n = 0) in (* First computation *)  
    if b then 1 (* Passed value *)  
    else let s = n – 1 in (* Second computation *)  
         let r = factorial s in (* Third computation *)  
        odel s r (* Returned value *);;  
val factorial : int -> int = <fun>
# factorial 5;;  
- : int = 120
```

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)`

```
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?
```
Example: CPS for length

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

What is the CSP version of this?
```

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

```ocaml
#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 allk : (a -> bool) -> 'a list -> bool = <fun>

What is the CPS version of this?
```

Example: all

```ocaml
#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

```ocaml
#let rec all (p, l) = match l with 
  | [] -> true
  | (x :: xs) -> let b = p x in 
    if b then all (p, xs) k else k
false)
val allk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> 
  (bool -> 'b) -> 'b = <fun>
```

What is the CPS version of this?

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

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, 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.
**Terminology**

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

**CPS Transformation**

- **Step 1**: Add continuation argument to any function definition:
  - let $f \ arg = e \implies let f \ arg \ k = e$
  - 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 $a \implies k \ a$
  - Assuming $a$ is a constant or variable.
  - “Simple” = “No available function calls.”

**Example**

**Before:**

```ocaml
let rec add_list lst =
  match lst with
  [ ] -> 0
| 0 :: xs -> add_list xs
| x :: xs -> (+) x
     (add_list xs);
```

**After:**

```ocaml
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 ((+) r));
     (* rule 4 *)
```
CPS for sum

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

CPS for sum

```ocaml
# let rec sum list = match list with 
  | []   -> 0 
  | x :: xs -> let r1 = sum xs  in x + r1;;
val sum : int list -> int = <fun>
```

CPS for sum

```ocaml
# 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>
```

Exceptions - Example

```ocaml
# 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>
```

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
### Exceptions - Example

```ocaml
# let 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.
```

### 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 values are thrown away

### Implementing Exceptions

```ocaml
# 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>
```

```ocaml
# 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) kexcp);
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>
```

```ocaml
# 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 = ()
```