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 all : ('a -> bool) -> 'a list -> bool
```

What is the CPS version of this?

```ocaml
#let rec allk(pk, l) k =
```

2/9/23
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 -> '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 (e.g., 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 ⇒ 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 ⇒ 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 \to 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) \to f arg (fun r -> k(op r))
- op represents a primitive operation
- return g(f arg) \to 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 *)
match lst with
[ ] -> k false (* rule 2 *)
| x :: xs ->
  if (x = y)
    then true
    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 ->
  if (x = y)
    then true
    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) (* rule 3 *)
  )
  (* rule 4 *)

Example

Before:
let rec add_list lst =
match lst with
[ ] -> 0
| x :: xs ->
  if (x = y)
    then true
    else add_list xs
  (* rule 3 *)

After:
let rec add_listk lst k =
(* rule 1 *)
match lst with
[ ] -> k 0 (* rule 2 *)
| x :: xs ->
  eqk (x, y)
  (fun b -> if b (* rule 4 *)
    then k true (* rule 2 *)
    else add_listk xs k (* rule 3 *)
  )
  (* rule 4 *)

Extra Material

- 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
# 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>
```

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 -> '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) k) 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 = ()
```

```
# 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.
```

```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
```
Data type in Ocaml: lists

- Frequently used lists in recursive program
- Matched over two structural cases
  - `[]` - the empty list
  - `(x :: xs)` a non-empty list
- Covers all possible lists
- `type 'a list = [ ] | (::)` of `‘a * ‘a list`  
  - Not quite legitimate declaration because of special syntax

Variants - Syntax (slightly simplified)

- `type name = C_1 [of ty_1] | . . . | C_n [of ty_n]`
- Introduce a type called `name`
- `(fun x -> C_i x) : ty_i -> name`
- `C_i` is called a constructor, if the optional type argument is omitted, it is called a constant
- Constructors are the basis of almost all pattern matching

Enumeration Types as Variants

An enumeration type is a collection of distinct values

In C and Ocaml they have an order structure; order by order of input

```
# type weekday = Monday | Tuesday | Wednesday |
| Thursday | Friday | Saturday | Sunday;;
type weekday =
  Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
| Sunday
```

Functions over Enumerations

```
# let day_after day = match day with
  Monday -> Tuesday
| Tuesday -> Wednesday
| Wednesday -> Thursday
| Thursday -> Friday
| Friday -> Saturday
| Saturday -> Sunday
| Sunday -> Monday;;
val day_after : weekday -> weekday = <fun>
```
Functions over Enumerations

```ocaml
# let rec days_later n day =
  match n with
  | 0 -> day
  | _ -> if n > 0
    then day_after (days_later (n - 1) day)
    else days_later (n + 7) day;;
val days_later : int -> weekday -> weekday = <fun>
```

```ocaml
Problem:

# type weekday = Monday | Tuesday | Wednesday
  | Thursday | Friday | Saturday | Sunday;;
- Write function is_weekend : weekday -> bool
let is_weekend day =

```

```ocaml
Example Enumeration Types

# type bin_op = IntPlusOp | IntMinusOp
  | EqOp | CommaOp | ConsOp
# type mon_op = HdOp | TlOp | FstOp
  | SndOp
```

Disjoint Union Types

- Disjoint union of types, with some possibly occurring more than once

- We can also add in some new singleton elements
Disjoint Union Types

```ocaml
# type id = DriversLicense of int | SocialSecurity of int | Name of string;;
type id = DriversLicense of int | SocialSecurity of int | Name of string
# let check_id id = match id with
    DriversLicense num -> not (List.mem num [13570; 99999])
    | SocialSecurity num -> num < 900000000
    | Name str -> not (str = "John Doe");;
val check_id : id -> bool = <fun>
```

Problem

- Create a type to represent the currencies for US, UK, Europe and Japan

```ocaml
type currency =
    Dollar of int
    | Pound of int
    | Euro of int
    | Yen of int
```

Example Disjoint Union Type

```ocaml
# type const =
    BoolConst of bool
    | IntConst of int
    | FloatConst of float
    | StringConst of string
    | NilConst
    | UnitConst
```

- How to represent 7 as a const?
  - Answer: IntConst 7

Polymorphism in Variants

- The type 'a option is gives us something to represent non-existence or failure
- Used to encode partial functions
- Often can replace the raising of an exception
Functions producing option

```ocaml
# let rec first p list = 
  match list with 
  | [] -> None 
  | (x::xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>

# first (fun x -> x > 3) [1;3;4;2;5];;
- : int option = Some 4

# first (fun x -> x > 5) [1;3;4;2;5];;
- : int option = None
```

Problem

- Write a `hd` and `tl` on lists that doesn't raise an exception and works at all types of lists.

```ocaml
# let hd list = 
  match list with 
  | [] -> None 
  | (x::xs) -> Some x
val hd : 'a list -> 'a option = <fun>

# let tl list = 
  match list with 
  | [] -> None 
  | (x::xs) -> Some xs
val tl : 'a list -> 'a option = <fun>
```

Mapping over Variants

```ocaml
# let optionMap f opt = 
  match opt with 
  | None -> None 
  | Some x -> Some (f x);;
val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>

# optionMap 
  (fun x -> x - 2) 
  (first (fun x -> x > 3) [1;3;4;2;5]);;
- : int option = Some 2
```

Folding over Variants

```ocaml
# let optionFold someFun noneVal opt = 
  match opt with 
  | None -> None 
  | Some x -> Some (someFun x);;
val optionFold : ('a -> 'b) -> 'a option -> 'b option = <fun>

# optionFold (fun x -> Some (f x)) None opt;;
val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>
```
Recursive Types

- The type being defined may be a component of itself

```
  ty  ty'  ty  .  .
```

Recursive Data Types

```ocaml
# type int_Bin_Tree =
   Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree);
```

```
type int_Bin_Tree = Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

Recursive Data Type Values

```ocaml
# let bin_tree =
   Node(Node(Leaf 3, Leaf 6), Leaf (-7));;
```

```
val bin_tree : int_Bin_Tree = Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```

Recursive Functions

```ocaml
# let rec first_leaf_value tree =
   match tree with (Leaf n) -> n
   | Node (left_tree, right_tree) ->
     first_leaf_value left_tree;;
```

```
val first_leaf_value : int_Bin_Tree -> int = <fun>
```

```ocaml
# let left = first_leaf_value bin_tree;;
val left : int = 3
```