OCaml Higher Order Functions

CSC 310 - Programming Languages

Anonymous Functions

- In functional programming, passing around functions is common, so we often do not need to give them names
- Anonymous function syntax: `fun p1 ... pn -> e`
  - `p1 .. pn` are the paramters
  - `e` is the body
- Example

```
# (fun x -> x + 1) 7;;
- : int = 8
```

Functions and Binding

- Functions are first-class, so you can bind them to other names as you like
- In fact, `let` for functions is a syntactic shorthand:
  - `let f x = body`
  - `let f = fun x -> body`
- Function definitions can occur anywhere

First-Class Function Examples

```
# let f b x =
    let dec x = x - 1 in
    let inc x = x + 1 in
    if b then dec x
    else      inc x;;
val f : bool -> int -> int = <fun>

# let f' b x = (* equivalent to f *)
    if b then (fun y -> y - 1) x
    else      (fun y -> y + 1) x;;
val f' : bool -> int -> int = <fun>
```

Pattern Matching with `fun`

- `match` can be used within `fun`
  - Idiom: use named functions if the match is complicated
- Examples

```
# (fun lst -> match lst with (h::_) -> h) [1;2];;
Warning 8 [partial-match]: this pattern-matching is
not exhaustive.
Here is an example of a case that is not matched:
[]

- : int = 1
# (fun (x,y) -> x+y) (1,2);;
- : int = 3
```

Passing Functions as Arguments

- In OCaml functions can be be passed as arguments to a function
- Example:

```
# let add_one x = x + 1;;
val add_one : int -> int = <fun>

# let twice f x = f (f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>

# twice add_one 1;;
- : int = 3
```

The `map` Function

- `map f lst` takes a function `f` and a list `lst` and applies the function `f` to each element of `lst` returning a list of the results

```
map f [v1; ..., vn]
    = [f v1; ..., f vn]
```

- Example

```
# let add_one x = x + 1;;
val add_one : int -> int = <fun>
# map add_one [1; 2; 3];;
- : int list = [2; 3; 4]
```

map Implementation

```
# let rec map f lst =
  match lst with
  | [] -> []
  | h::t -> (f h)::(map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Another `map` Example

- Apply a list of functions to a list of integer values

```
let neg = x = -x;;
let add_one x = x + 1;;
let double x = x + x;;
let fs = [neg; add_one; double];;
let lst = [1;2;3];;

map (fun f -> map f lst) fs
- int list list [[-1;-2;-3]; [2;3;4]; [2;4;6]]
```

Recursive Function Examples

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h::t -> h + (sum t)
```

```
let rec concat lst =
  match lst with
  | [] -> ""
  | h::t -> h ^ (concat t)
```

The foldr Function

```
# let rec foldr f acc lst =
    match lst with
    | [] -> acc
    | h::t -> f h (foldr f acc t);;
val foldr : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun

# let sum lst = foldr (+) 0 lst;;
val sum : int list -> int = <fun>

# let concat lst = foldr (^) "" lst;;
val concat : string list -> string = <fun>
```

The `foldr` Function (continued)

- `foldr` is a function that
  - takes a function of two arguments, a final value, and a list
  - processes the list by applying the function to the head and the recursive application of the function to the rest of the list, returning the final value for the empty list

```
foldr f v [v1; ...; vn] =
    f v1 (... (f vn v) ...)
```

The Standard Library `foldr`

- The `List` module has a function `List.fold_right` is similar to the `foldr` function above except that the order of the last two parameters is reversed

```
fold_right f [v1; ...; vn] v =
    f v1 (... (f vn v) ...)
```

Fold

- The List module also defines a function fold_left, here we will call it fold

```
# let rec fold f acc lst =
    match lst with
    | [] -> acc
    | h::t -> fold f (f acc h) t;;
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <f
```

- Similar to fold_right, but the changes the order of operations

Fold (continued)

- `let add a x = a + x`
- `fold add 0 [1; 2; 3]`
- `fold add (add 0 1) [2; 3]`
- `fold add 1 [2; 3]`
- `fold add (add 1 2) [3]`
- `fold add 3 [3]`
- `fold add (add 3 3) []`
- `fold add 6 []`
- `6`

Fold Right vs. Fold Left

- Fold right

  ```
  foldr add 0 [1;2;3;4] =
    add 1 (add 2 (add 3 (add 4 0))) = 10
  ```

- Fold

  ```
  fold add 0 [1;2;3;4] =
    add (add (add (add 0 1) 2) 3) 4) = 10
  ```

Which Fold to Use?

- Many problems lend themselves to `fold_right`
- But `fold_right` has a performance problem: the recursion allocates a new stack frame for each recursive call of `fold_right`
- Tail call optimization allows `fold_left` to use no stack frames for each recursive call.

Combining `map` and `fold`

- Idea: map a list to another list and then fold over it to compute the final result.
- Example:

```
let count_one lst =
  fold (fun a h -> if h=1 then a+1 else a) 0 lst

let count_ones lst =
  let counts = map count_one lst in
  fold (fun a c -> a+c) 0 counts
```