CMSC 330: Organization of Programming Languages

OCaml
Higher Order Functions
Anonymous Functions

- Recall code blocks in Ruby
  
  \[(1..10).each \{ |x| \text{print } x \}\]
  
  - Here, we can think of \{ |x| \text{print } x \} as a function

- We can do this (and more) in OCaml
Anonymous Functions

- Passing functions around is very common
  - So often we don’t want to bother to give them names

- Use `fun` to make a function with no name

```
fun x -> x + 3
```

```
(fun x -> x + 3) 5 = 8
```
Anonymous Functions

Syntax

• \texttt{fun \ x_1 \ldots \ x_n \rightarrow e}

Evaluation

• An anonymous function is an expression
• In fact, \textit{it is a value} – no further evaluation is possible
  ➢ As such, it can be passed to other functions, returned from them, stored in a variable, etc.

Type checking

• \((\texttt{fun \ x_1 \ldots \ x_n \rightarrow e}) : (t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u)\)
  if \(e : u\) under assumptions \(x_1 : t_1, \ldots, x_n : t_n\).
  ➢ (Same rule as \texttt{let \ f \ x_1 \ldots \ x_n = e})
All Functions Are Anonymous

- Functions are first-class, so you can bind them to other names as you like
  
  ```
  let f x = x + 3;;
  let g = f;;
  g 5   = 8
  ```

- In fact, *let* for functions is syntactic shorthand
  
  ```
  let f x = body
  ↓
  is semantically equivalent to
  let f = fun x -> body
  ```
Example Shorthands

- let next x = x + 1
  - Short for let next = fun x -> x + 1

- let plus x y = x + y
  - Short for let plus = fun x y -> x + y

- let rec fact n =
  if n = 0 then 1 else n * fact (n-1)
  - Short for let rec fact = fun n ->
    (if n = 0 then 1 else n * fact (n-1))
Defining Functions Everywhere

let move l x = 
  let left x = x - 1 in (* locally defined fun *) 
  let right x = x + 1 in (* locally defined fun *) 
  if l then left x 
  else right x 

;;

let move’ l x = (* equivalent to the above *) 
  if l then (fun y -> y - 1) x 
  else (fun y -> y + 1) x
Calling Functions, Generalized

Syntax \( e_0 e_1 \ldots e_n \)

Evaluation

- Evaluate arguments \( e_1 \ldots e_n \) to values \( v_1 \ldots v_n \)
  - Order is actually right to left, not left to right
  - But this doesn’t matter if \( e_1 \ldots e_n \) don’t have side effects

- Evaluate \( e_0 \) to a function \( \text{fun } x_1 \ldots x_n \rightarrow e \)

- Substitute \( v_i \) for \( x_i \) in \( e \), yielding new expression \( e' \)

- Evaluate \( e' \) to value \( v \), which is the final result
Calling Functions, Generalized

- Syntax $e_0 \ e_1 \ \ldots \ e_n$
- Type checking (almost the same as before)
  - If $e_0 : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u$ and $e_1 : t_1, \ldots, e_n : t_n$
    then $e_0 \ e_1 \ \ldots \ e_n : u$

- Example:
  - $(\text{fun } x \rightarrow x+1) \ 1 : \text{int}$
  - since $(\text{fun } x \rightarrow x+1) : \text{int} \rightarrow \text{int}$ and $1 : \text{int}$
Pattern Matching With Fun

- `match` can be used within `fun`
  
  ```
  (fun l -> match l with (h::_) -> h) [1; 2]
  = 1
  ```

- But use named functions for complicated matches
- May use standard pattern matching abbreviations
  
  ```
  (fun (x, y) -> x+y) (1,2)
  = 3
  ```
Passing Functions as Arguments

- In OCaml you can pass functions as arguments (akin to Ruby code blocks)

```ocaml
let plus_three x = x + 3 (* int -> int *)
let twice f z = f (f z) (* ('a->'a) -> 'a -> 'a *)
twice plus_three 5 = 11
```

- Ruby’s `collect` is called `map` in OCaml
  - `map f l` applies function `f` to each element of `l`, and puts the results in a new list (preserving order)

```ocaml
map plus_three [1; 2; 3] = [4; 5; 6]
map (fun x -> (-x)) [1; 2; 3] = [-1; -2; -3]
```
The Map Function

Let’s write the map function

• Takes a function and a list, applies the function to each element of the list, and returns a list of the results

```ocaml
let rec map f l = match l with
  | [] -> []
  | (h::t) -> (f h)::(map f t)
```

```ocaml
let add_one x = x + 1
let negate x = -x

map add_one [1; 2; 3] = [2; 3; 4]
map negate [9; -5; 0] = [-9; 5; 0]
```

Type of map?
The Map Function (cont.)

What is the type of the map function?

```
let rec map f l = match l with
    [] -> []
  | (h::t) -> (f h)::(map f t)
```

- ('a -> 'b) → 'a list
- 'b list

\[ f \]
\[ l \]
The Fold Function

- Common pattern
  - Iterate through list and apply function to each element, keeping track of partial results computed so far

```ocaml
let rec fold f a l = match l with
  | [] -> a
  | (h::t) -> fold f (f a h) t
```

- \(a\) = “accumulator”
- Usually called \(\text{fold left}\) to remind us that \(f\) takes the accumulator as its first argument

- What's the type of \(\text{fold}\)?
  \[= ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a\]
Example

\[
\text{let rec fold } f \ a \ l = \text{match } l \text{ with} \\
[] \rightarrow a \\
| (h::t) \rightarrow \text{fold } f (f \ a \ h) \ t \\
\]

let add \ a \ x = a + x
fold add 0 [1; 2; 3; 4] \rightarrow 
fold add 1 [2; 3; 4] \rightarrow 
fold add 3 [3; 4] \rightarrow 
fold add 6 [4] \rightarrow 
fold add 10 [] \rightarrow 
10

We just built the \text{sum} function!
Another Example

```ml
let rec fold f a l = match l with
  [] -> a
| (h::t) -> fold f (f a h) t
```

```ml
let next a _ = a + 1
fold next 0 [2; 3; 4; 5] →
fold next 1 [3; 4; 5] →
fold next 2 [4; 5] →
fold next 3 [5] →
fold next 4 [] →
4
```

We just built the `length` function!
Using Fold to Build Reverse

Let’s build the reverse function with fold!

```ocaml
let rec fold f a l = match l with
  | [] -> a
  | (h::t) -> fold f (f a h) t

let prepend a x = x::a

fold prepend [] [1; 2; 3; 4] →
fold prepend [1] [2; 3; 4] →
fold prepend [2; 1] [3; 4] →
fold prepend [3; 2; 1] [4] →
fold prepend [4; 3; 2; 1] [] →
[4; 3; 2; 1]
```
Summary

- map $f$ $[v_1; v_2; \ldots; v_n]$
  
  \[= [f \ v_1; f \ v_2; \ldots; f \ v_n]\]
  
  - e.g., $\text{map (fun x -> x+1)} [1;2;3] = [2;3;4]\]

- fold $f$ $v$ $[v_1; v_2; \ldots; v_n]$
  
  \[= \text{fold} \quad f \quad (f \ v \ v_1) \quad [v_2; \ldots; v_n]\]
  
  \[= \text{fold} \quad f \quad (f (f \ v \ v_1) \ v_2) \quad [\ldots; v_n]\]
  
  \[= \ldots\]
  
  \[= f \ (f \ (f \ (f \ v \ v_1) \ v_2) \ \ldots) \ v_n\]
  
  - e.g., $\text{fold add 0} \ [1;2;3;4] = $
    
    add (add (add (add 0 1) 2) 3) 4 = 10
Quiz 1: What does this evaluate to?

\[
\text{let id } x = x \text{ in } \\
(f \text{un } f \ y \rightarrow f \ (y+1)) \ id \ 1
\]

A. Error
B. 2
C. 1
D. (id 2)
Quiz 1: What does this evaluate to?

```ml
let id x = x in
(fun f y -> f (y+1)) id 1
```

A. Error
B. 2
C. 1
D. (id 2)
Quiz 2: What does this evaluate to?

\[
\text{map (fun x -> x *. 4) [1;2;3]}
\]

A. [ 1.0; 2.0; 3.0 ]
B. [ 4.0; 8.0; 12.0 ]
C. Error
D. [4; 8; 12 ]
Quiz 2: What does this evaluate to?

\[
\text{map } (\text{fun } x \rightarrow x * . 4) \ [1;2;3]
\]

A. [ 1.0; 2.0; 3.0 ]
B. [ 4.0; 8.0; 12.0 ]
C. Error
D. [4; 8; 12 ]
Quiz 3: What does this evaluate to?

\[
\text{fold \ (fun \ a \ y \ \rightarrow \ y :: a)} \ \ [] \ \ [2;3;4]
\]

A. [ 9 ]
B. [ 2;5;9 ]
C. [ 4;3;2 ]
D. Error
Quiz 3: What does this evaluate to?

\[ \text{fold } (\text{fun } a \ y \rightarrow y :: a) \ [\] \ [2;3;4] \]

A. [ 9 ]
B. [ 2;5;9 ]
C. [ 4;3;2 ]
D. Error
Quiz 4: What does this evaluate to?

```
let is_even x = (x mod 2 = 0) in
map is_even [1;2;3;4;5]
```

A. [false;true;false;true;false]
B. [0;1;1;2;2]
C. [0;0;0;0;0]
D. false
Quiz 4: What does this evaluate to?

let is_even x = (x mod 2 = 0) in
map is_even [1;2;3;4;5]

A. [false;true;false;true;false]
B. [0;1;1;2;2]
C. [0;0;0;0;0]
D. false