CMSC 330: Organization of Programming Languages

OCaml

Higher Order Functions
Anonymous Functions

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

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

- As with Ruby, passing around functions is 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
```

Parameter: `x`  

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

- **Syntax**
  - `fun x1 ... xn -> e`

- **Evaluation**
  - An anonymous function is an expression
  - In fact, *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**
  - `(fun x1 ... xn -> e) : (t1 -> ... -> tn -> u)`
  - when `e : u` under assumptions `x1 : t1`, ..., `xn : tn`.
    - (Same rule as `let f x1 ... xn = e`
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

- Example:
  - $(\text{fun } x \rightarrow x+x) \ 1 \Rightarrow 1+1 \Rightarrow 2$
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:
  - $(\text{fun } x \rightarrow x+x) \ 1 : \text{int}$
  - since $(\text{fun } x \rightarrow x+x) : \text{int} \rightarrow \text{int}$ and $1 : \text{int}$
Quiz 1: What does this evaluate to?

let y = (fun x -> x+1) 2 in
(fun z -> z-2) y

A. Error
B. 2
C. 1
D. 0
Quiz 1: What does this evaluate to?

\[
\begin{align*}
\text{let } y &= (\text{fun } x \rightarrow x+1) \ 2 \ \text{in} \\
&\quad (\text{fun } z \rightarrow z-2) \ y
\end{align*}
\]

A. Error
B. 2
C. 1
D. 0
Quiz 2: What is this expression’s type?

(fun x y -> x) 2 3

A. Type error
B. int
C. int -> int -> int
D. 'a -> 'b -> 'a
Quiz 2: What is this expression’s type?

(fun x y -> x) 2 3

A. Type error
B. int
C. int -> int -> int
D. 'a -> 'b -> 'a
Functions and Binding

- 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))
Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
  g 1
A. Error
B. 2
C. 1
D. 0
```
Quiz 3: What does this evaluate to?

```ml
let f = fun x -> 0 in
let g = f in
  g 1
```

A. Error
B. 2
C. 1
D. 0
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
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?

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

('a -> 'b) -> 'a list -> 'b list
The Fold Function

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

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

- `a = “accumulator”`
- Usually called `fold left` to remind us that `f` takes the accumulator as its first argument

- What's the type of `fold`?
  
  `= ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
Example

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

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

We just built the sum function!
```
Another Example

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

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

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

Let’s build the reverse function with fold!

```ocaml
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** $f$ $v$ $[v_1; v_2; \ldots; v_n]$
  
  $$= \text{fold } f \ (f \ v \ v_1) \ [v_2; \ldots; v_n]$$

  $$= \text{fold } f \ (f (f \ v \ v_1) \ v_2) \ [\ldots; v_n]$$

  $$= \ldots$$

  $$= f \ (f \ (f \ (f \ v \ v_1) \ v_2) \ \ldots) \ v_n$$

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

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 4: What does this evaluate to?

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

A. [ 1.0; 2.0; 3.0 ]
B. [ 4.0; 8.0; 12.0 ]
C. Error -- the *. function takes floats, not ints
D. [4; 8; 12 ]
Quiz 5: What does this evaluate to?

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

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

```haskell
fold (fun a y -> y::a) [] [3;4;2]
```

A.  [ 9 ]  
B.  [ 3;4;2 ]  
C.  [ 2;4;3 ]  
D.  Error
Quiz 6: What does this evaluate to?

```haskell
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
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
Combining map and fold

- Idea: map a list to another list, and then fold over it to compute the final result
  - Basis of the famous “map/reduce” framework from Google, since these operations can be parallelized

```plaintext
let countone l =
  fold (fun a h -> if h=1 then a+1 else a) 0 l

let countones ss =
  let counts = map countone ss in
  fold (fun a c -> a+c) 0 counts

countones [[1;0;1]; [0;0]; [1;1]] = 4
countones [[1;0]; []; [0;0]; [1]] = 2
```
fold_right

- Right-to-left version of fold:

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

- Left-to-right version used so far:

```ocaml
let rec fold f a l = match l with
    [] -> a
  | (h::t) -> fold f (f a h) t
```
Left-to-right vs. right-to-left

$$\text{fold } f \ v \ [v_1; v_2; \ldots; v_n] =$$
$$f \ (f \ (f \ (f \ v \ v_1) \ v_2) \ \ldots) \ v_n$$

$$\text{fold}_\text{right} \ f \ [v_1; v_2; \ldots; v_n] \ v =$$
$$f \ (f \ (f \ (f \ v_n \ v) \ \ldots) \ v_2) \ v_1$$

$$\text{fold} \ (\text{fun} \ x \ y \ \rightarrow \ x - y) \ 0 \ [1;2;3] = -6$$

since $$((0-1)-2)-3) = -6$$

$$\text{fold}_\text{right} \ (\text{fun} \ x \ y \ \rightarrow \ x - y) \ [1;2;3] \ 0 = 2$$

since $$1-(2-(3-0))) = 2$$
When to use one or the other?

- Many problems lend themselves to `fold_right`
- But it does present a performance disadvantage
  - The recursion builds of a deep stack: One stack frame for each recursive call of `fold_right`
- An optimization called `tail recursion` permits optimizing `fold` so that it uses no stack at all
  - We will see how this works in a later lecture!