CMSC 330: Organization of Programming Languages

Functional Programming with OCaml

Reminders

• Homework 2 will be posted soon
Review

- Recursion is how all looping is done
- OCaml can easily pass and return functions

The Call Stack in C/Java/etc.

```c
void f(void) {
    int x;
    x = g(3);
}
int g(int x) {
    int y;
    y = h(x);
    return y;
}
int h (int z) {
    return z + 1;
}
int main(){
    f();
    return 0;
}
```
Nested Functions

- In OCaml, you can define functions anywhere
  – Even inside of other functions

```ocaml
let sum l =  
  fold ((fun (a, x) -> a + x), 0, l)
```

```ocaml
let pick_one n = 
  if n > 0 then (fun x -> x + 1)  
  else (fun x -> x - 1)  
(pick_one -5) 6 (* returns 5 *)
```

Nested Functions (cont’d)

- You can also use `let` to define functions inside of other functions

```ocaml
let sum l =  
  let add (a, x) = a + x in  
  fold (add, 0, l)
let pick_one n =  
  let add_one x = x + 1 in  
  let sub_one x = x - 1 in  
  if n > 0 then add_one else sub_one
```
How About This?

takes a number \( n \) and list \( l \) and adds \( n \) to every element in \( l \)

\[
\text{let addN } (n, l) = \\
\quad \text{let add } x = n + x \text{ in} \\
\quad \text{map (add, l)}
\]

---

\( \) (Equivalent to...)

\[
\text{let addN } (n, l) = \\
\quad \text{map ((fun } x \rightarrow n + x), l)
\]

---

Consider the Call Stack Again

\[
\text{let map } (f, n) = \text{match } n \text{ with} \\
\quad [] \rightarrow [] \\
\quad (h::t) \rightarrow (f h)::(\text{map } (f, t))
\]

\[
\text{let addN } (n, l) = \\
\quad \text{let add } x = n + x \text{ in} \\
\quad \text{map (add, l)}
\]

\[
\text{addN } (3, [1; 2; 3])
\]

\* Uh oh...how does \textit{add} know the value of \( n \)?

\quad The \textbf{wrong} answer for OCaml: it reads it off the stack

\quad \quad \textbullet \quad \text{The language could do this, but can be confusing (see above)}

\quad \textbullet \quad \text{OCaml uses static scoping like C, C++, Java, and Ruby}
Static Scoping

- In static or lexical scoping, (nonlocal) names refer to their nearest binding in the program text
  - Going from inner to outer scope
  - C example:

    ```c
    int x;
    void f() { x = 3; }
    void g() { char *x = "hello"; f(); }
    ```

    - In our example, add accesses addN's n

Returned Functions

- As we saw, in OCaml a function can return another function as a result
  - So consider the following example

    ```ocaml
    let addN n = (fun x -> x + n)
    (addN 3) 4 (* returns 7 *)
    ```

    - When the anonymous function is called, n isn’t even on the stack any more!
      - We need some way to keep n around after addN returns
Environments and Closures

- An *environment* is a mapping from variable names to values
  - Just like a stack frame

- A *closure* is a pair \((f, e)\) consisting of function code \(f\) and an environment \(e\)

- When you invoke a closure, \(f\) is evaluated using \(e\) to look up variable bindings

Example

```ocaml
let add x = (fun y -> x + y)
```

\((\text{add} 3) 4\) → \(<\text{closure}>\) 4 → 3 + 4 → 7
Another Example

```
let mult_sum (x, y) =
  let z = x + y in
  fun w -> w * z
```

```
(mult_sum (3, 4)) 5  -> <closure> 5  -> 5 * 7  -> 35
```

Yet Another Example

```
let twice (n, y) =
  let f x = x + n in
  f (f y)
```

```
twice (3, 4)  -> <closure> (<closure> 4)  -> <closure> 7  -> 10
```
Still Another Example

```plaintext
let add x = (fun y -> (fun z -> x + y + z))
```

```
(((add 1) 2) 3)  →  ((<closure> 2) 3)  →  (<closure> 3)  →  1+2+3
```

Currying

- We just saw another way for a function to take multiple arguments
  - The function consumes one argument at a time, creating closures until all the arguments are available

- This is called *currying* the function
  - Named after the logician Haskell B. Curry
  - But Schönfinkel and Frege discovered it
    - So it should probably be called Schönfinkelizing or Fregging
Curried Functions in OCaml

- OCaml has a really simple syntax for currying
  
  ```ocaml
  let add x y = x + y
  ```

  - This is identical to all of the following:
    
    ```ocaml
    let add = (fun x -> (fun y -> x + y))
    let add = (fun x y -> x + y)
    let add x = (fun y -> x+y)
    ```

- Thus:
  - `add` has type `int -> (int -> int)`
  - `add 3` has type `int -> int`

  ```text
  • The return of `add x` evaluated with `x = 3`
  • `add 3` is a function that adds 3 to its argument
  - `(add 3) 4 = 7`
  ```

- This works for any number of arguments

Curried Functions in OCaml (cont’d)

- Because currying is so common, OCaml uses the following conventions:
  - `->` associates to the right
    - Thus `int -> int -> int` is the same as
    - `int -> (int -> int)`

  - function application associates to the left
    - Thus `add 3 4` is the same as
    - `(add 3) 4`
Another Example of Currying

- A curried add function with three arguments:

  ```ml
  let add_th x y z = x + y + z
  ```

  - The same as

  ```ml
  let add_th x = (fun y -> (fun z -> x+y+z))
  ```

- Then...

  - `add_th` has type `int -> (int -> (int -> int))`
  - `add_th 4` has type `int -> (int -> int)`
  - `add_th 4 5` has type `int -> int`
  - `add_th 4 5 6` is 15

Currying and the map Function

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

- Examples

  ```ml
  let negate x = -x
  map negate [1; 2; 3] (* returns [-1; -2; -3 ] *)
  let negate_list = map negate
  negate_list [-1; -2; -3]
  let sum_pairs_list = map (fun (a, b) -> a + b)
  sum_pairs_list [(1, 2); (3, 4)] (* [3; 7] *)
  ```

- What's the type of this form of `map`?

  ```ml
  map : ('a -> 'b) -> 'a list -> 'b list
  ```
Currying and the fold Function

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

let add x y = x + y
fold add 0 [1; 2; 3]
let sum = fold add 0
sum [1; 2; 3]
let next n_ = n + 1
let length = fold next 0 (* warning: not polymorphic *)
length [4; 5; 6; 7]
```

• What's the type of this form of `fold`?
  
  `fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

Another Convention

• Since functions are curried, `function` can often be used instead of `match`
  – `function` declares an anonymous function of one argument
  – Instead of
  
  ```
  let rec sum l = match l with
    []  -> 0
  | (h::t) -> h + (sum t)
  ```

  – It could be written
  
  ```
  let rec sum = function
    []  -> 0
  | (h::t) -> h + (sum t)
  ```
Another Convention (cont’d)

Instead of

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

It could be written

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

Currying is Standard in OCaml

- Pretty much all functions are curried
  - Like the standard library `map`, `fold`, etc.
- OCaml plays a lot of tricks to avoid creating closures and to avoid allocating on the heap
  - It’s unnecessary much of the time, since functions are usually called with all arguments
Higher-Order Functions in C

- C has function pointers but no closures
  - (gcc has closures)

```c
typedef int (*int_func)(int);

void app(int_func f, int *a, int n) {
    int i;
    for (i = 0; i < n; i++)
        a[i] = f(a[i]);
}

int add_one(int x) { return x + 1; }

int main() {
    int a[] = {1, 2, 3, 4};
    app(add_one, a, 4);
}
```

Higher-Order Functions in Ruby

- Use `yield` within a method to call a code block argument

```ruby
def my_collect(a)
    b = Array.new(a.length)
    i = 0
    while i < a.length
        b[i] = yield(a[i])
        i = i + 1
    end
    return b
end

b = my_collect([1, 2, 3, 4, 5]) { |x| -x }
```
Higher-Order Functions in Java/C++

• An object in Java or C++ is kind of like a closure
  – it’s some data (like an environment)
  – along with some methods (i.e., function code)

• So objects can be used to simulate closures

• When we get to Java in the course, we’ll study how to implement some functional patterns in OO languages