OCaml Closures

CPSC 310 - Programming Languages

Returning Functions

- In OCaml you can pass functions as arguments
- and you can return functions as results

```
let choose_fn b =
  let f1 x = x + 1 in
  let f2 x = x * 1 in
  if b then f1 else f2
```

Multi-argument Functions

- Consider rewriting the previous code

```
let choose_fn b =
  if b then (fun x -> x + 1) else (fun x -> x * 1)
```

- Another version

```
let choose_fn b =
  (fun x -> if b then x + 1 else x * 1)
```

- As a multi-argument function

```
let choose_fn b x =
  if b then x + 1 else x * 1
```

Currying

- A way to make a multiple argument function is to encode it as a function that takes a single argument and returns a function that takes the rest.
- This encoding is called currying the function
  - Named after the logician Haskell Curry
  - Discovered by Schönfinkel and Frege

Curried Functions in OCaml

- The OCaml syntax defaults to currying, for example

  ```
  let add x y = x + y
  ```

  is equivalent to

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

Syntax Conventions for Currying

- OCaml conventions for currying
  - `->` is right associative, so `int -> int -> int` is the same as `int -> (int -> int)`
  - function application is left associative, so `add 1 1` is the same as `(add 1) 1`

Partial Application

- We could use tuples to encode support for multiple arguments

```
let add (x, y) = x + y  (* int * int -> int *)
let add x y = x + y     (* int -> int -> int *)
```

- The curried version supports partial application which allows us to provide some arguments now and the rest later

```
let add x y = x + y  (* int -> int -> int *)
let addOne = add 1   (* int -> int *)
addOne 1             (* int *)
```

Currying is Standard in OCaml

- Almost all function are curried
  - Standard library `map`, `fold`, etc
- The OCaml implementation makes currying efficient
  - an inefficient implementation would do a lot of useless allocation and destruction of closures

Closure Example

```
let f x =
  let g = fun y -> x + y in
  g
```

```
f 10 = (fun y -> x + y) 10
     = (fun y -> 10 + y)
```

Another Closure Example

```
let x = 1 in
let f = fun y -> x in
let x  = 2 in
f 0
```

What does this expression evaluate to?

Scope

- **Dynamic scope**: the body of a function is evaluated in the current dynamic environment at the time the function is called, not the old dynamic environment that existed at the time the function was defined.
- **Static (or Lexical) scope**: the body of a function is evaluated in the old dynamic environment that existed at the time the function was defined, not the current environment when the function is called.

Closures Implement Static Scope

- An environment is a mapping from variable names to values
- A closure is a pair `(f, e)` consisting of function code `f` and an environment `e`
- When a closure is invoked, `f` is evaluated using `e` to look up variable bindings.

Another Closure Example

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

(mult_sum (3, 4)) 5

<function> = fun w -> w * z
<environment> = x = 3, y = 4, z = 7
```

Aside: Higher-Order Functions in C

```
// define type int_fun which takes an int and returns an in
typedef int (*int_fun) (int);

void app(int_fun f, int *a, int n) {
    for (int 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};
    app(add_one, a, 3);
}
```

Aside: Higher-Order Functions in C

- C does not support closures
  - Since nested functions are not allowed
  - Unbound symbols are in the global scope

```
int y = 1;
void app(int(*f) (int), n) {
    return f(n);
}
int add_y(int x) {
    return x + y;
}
int main() {
    app(add_y, 2);
}
```

Aside: Higher-Order Functions in C

- C cannot access non-local variables in C
- OCaml code

```
let add x y = x + y
```

- Equivalent C code is illegal

```
int (*add(int x)) (int) {
  return add_y;
}
int add_y(int y) {
  return x + y; // x is undefined
}
```