CMSC 330: Organization of Programming Languages

Objects and Functional Programming

OOP vs. FP

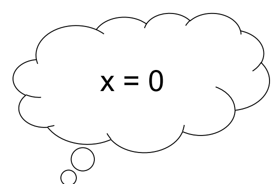
- Object-oriented programming (OOP)
  - Computation as interactions between objects
  - Objects encapsulate state, which is usually mutable
    - Accessed / modified via object's public methods
- Functional programming (FP)
  - Computation as evaluation of functions
    - Mutable data used to improve efficiency
  - Higher-order functions implemented as closures
    - Closure = function + environment

Relating Objects to Closures

- An object...
  - Is a collection of fields (data)
  - ...and methods (code)
  - When a method is invoked
    - Method has implicit this parameter that can be used to access fields of object
- A closure...
  - Is a pointer to an environment (data)
  - ...and a function body (code)
  - When a closure is invoked
    - Function has implicit environment that can be used to access variables

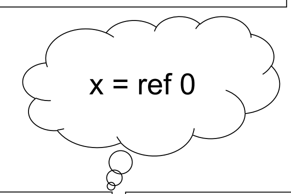
Relating Objects to Closures (cont.)

```
class C {
  int x = 0;
  void set_x(int y) { x = y; }
  int get_x() { return x; }
}
```




```
C c = new C();
c.set_x(3);
int y = c.get_x();
```

```
let make () =
 let x = ref 0 in
   ( (fun y -> x := y),
     (fun () -> !x) )
```




```
fun y -> x := y
```

```
fun () -> !x
```

```
let (set, get) = make ();;
set 3;;
let y = get ();;
```

Encoding Objects with Closures

- We can apply this transformation in general

```
class C { f1 ... fn; m1 ... mn; }
```

  - becomes

```
let make () =
 let f1 = ...
 ...
 and fn = ... in
 ( fun ... , (* body of m1 *)
   ...
   fun ...,  (* body of mn *)
 )
```

Tuple containing closures

  - make ( ) is like the constructor
  - The closure environment contains the fields

Relating Closures to Objects

```
let add1 x = x + 1
```

```
interface IntIntFun {
  Integer eval(Integer x);
}
class Add1 implements IntIntFun {
  Integer eval(Integer x) {
    return x + 1;
  }
}
```

```
add1 2;;
add1 3;;
```

```
new Add1().eval(2);
new Add1().eval(3)
```


Relating Closures to Objects

```
let app_to_1 f = f 1
```

```
interface IntIntFunFun {
  Integer eval(IntIntFun x);
}
class AppToOne
  implements IntIntFunFun {
    Integer eval(IntIntFun f) {
      return f.eval(1);
    }
}
```

```
app_to_1 add1;;
```

```
new AppToOne().eval(new Add1());
```


Relating Closures to Objects

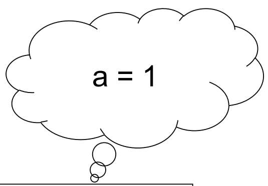
```
interface Func<T,U> {
  U eval(T x);
}
class Add1 implements Func<Integer,Integer> {
  public Integer eval(Integer x) {
    return x + 1;
  }
}
class AppToOne
  implements Func<Func<Integer,Integer>,Integer> {
    public Integer eval(Func<Integer,Integer> f) {
      return f.eval(1);
  }
}
```

```
app_to_1 add1;;
```

```
new AppToOne().eval(new Add1());
```

Relating Closures to Objects

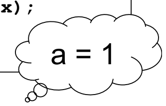
```
let add a b = a + b;;
```




```
fun b -> a + b
```

```
let add1 = add 1;;
add1 4;;
```

```
class Add
  implements Func<Int,Func<Int,Int>> {
    private static class AddClosure
      implements Func<Int,Int> {
        private final Int a;
        AddClosure(Int a) {
          this.a = a;
        }
        Integer eval(Int b) {
          return a + b;
        }
    }
    Func<Int,Int> eval(Int x) {
      return new AddClosure(x);
    }
}
```




```
Func<Int,Int> add1 = new Add().eval(1);
add1.eval(4);
```


Encoding Closures with Objects

- We can apply this transformation in general

```
...(fun x -> (* body of fn *)) ...
let h f ... = ...f y...
```

  - becomes

```
interface F<T,U> { U eval(T x); }
class G implements F<T,U> {
  U eval(T x) { /* body of fn */ }
}
class C {
  Typ1 h(F<Typ2,Typ3> f, ...) {
     ...f.eval(y)...
  }
}
```

  - F is the interface to the callback
  - G represents the particular function

Code as Data

- Closures and objects are related
  - Both of them allow
    - Data to be associated with higher-order code
    - Pass code around the program
- The key insight in all of these examples
  - Treat code as if it were data
    - Allowing code to be passed around the program
    - And invoked where it is needed (as callback)
- Approach depends on programming language
  - Higher-order functions (OCaml, Ruby, Lisp)
  - Function pointers (C, C++)
  - Objects with known methods (Java)

Code as Data (cont.)

- This is a powerful programming technique
  - Solves a number of problems quite elegantly
    - Create new control structures (e.g., Ruby iterators)
    - Add operations to data structures (e.g., visitor pattern)
    - Event-driven programming (e.g., observer pattern)
  - Keeps code separate
    - Clean division between higher & lower-level code
  - Promotes code reuse
    - Lower-level code supports different callbacks

An Integer List Abstraction in Java

```
public class MyList {

  private class ConsNode {
    int head; MyList tail;
    ConsNode (int h, MyList l) { head = h; tail = l; }
  }

  private ConsNode contents;

  public MyList () {
    contents = null;
  }

  public MyList (int h, MyList l) {
    contents = new ConsNode (h, l);
  }

  public MyList cons (int h) {
    return (new MyList (h, this));
  }

  public int hd () {
    return contents.head;
  }

  public MyList tl () {
    return contents.tail;
  }

  public boolean isNull () {
    return (contents == null);
  }

}
```


Recall a Useful Higher-Order Function

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

- Map applies an arbitrary function f
  - To each element of a list
  - And returns the resulting modified list
- Can we encode this in Java?
  - Using object oriented programming

A Map Method for Lists in Java

- Problem – Write a map method in Java
  - Must pass a function into another function
- Solution
  - Can be done using an object with a known method
  - Use interface to specify what method must be present

```
public interface IntFunction {
  int eval(int arg);
}
```


A Map Method for Lists (cont.)

- Examples
  - Two classes which both implement Function interface

```
class AddOne implements IntFunction {
  int eval (int arg) {
    return (arg + 1);
  }
}
```

```
class MultTwo implements IntFunction {
  int eval(int arg) {
    return (arg * 2);
  }
}
```

The New List Class

```
class MyList {
  …
  public MyList map (IntFunction f) {
    if (this.isNull()) return this;
    else return (this.tl()).map(f).cons (f.eval (this.hd()));
  }
}
```


Applying Map To Lists

- Then to apply the function, we just do

```
MyList l = ...;
MyList l1 = l.map(new AddOne());
MyList l2 = l.map(new MultTwo());
```

  - We make a new object
    - That has a method that performs the function we want
  - This is sometimes called a callback
    - Because map "calls back" to the object passed into it
  - But it's really just a higher-order function
    - Written more awkwardly

We Can Do This for Fold Also!

- Recall fold

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

  - Fold accumulates a value (in a) as it traverses a list
  - f is used to determine how to "fold" the head of a list into a
- This can be done in Java using an approach similar to map!

A Fold Method for Lists in Java

- Problem – Write a fold method in Java
  - Must pass a function into another function
- Solution
  - Can be done using an object with a known method
  - Use interface to specify what method must be present

```
public interface IntBinFunction {
  Integer eval(Integer arg1, Integer arg2);
}
```

A Fold Method for Lists (cont.)

- Examples
  - A classes which implements IntBinFunction interface

```
class Sum implements IntBinFunction {
  Integer eval(Integer arg1, Integer arg2) {
    return new Integer(arg1 + arg2);
  }
}
```

- Note: this is not curried
  - How might you make it so?

The New List Class

```
class MyList {
  …
  public MyList map (IntFunction f) {
    if (this.isNull()) return this;
    else return (this.tl()).map(f).cons (f.eval (this.hd()));
  }

  public int fold (IntBinFunction f, int a) {
    if (this.isNull()) return a;
    else return (this.tl()).fold(f, f.eval(a, this.hd()));
  }
}
```


Applying Fold to Lists

- To apply the fold function, we just do this:

```
MyList l = ...;
int s = l.fold (new Sum(), 0);
```

- The result is that s contains the sum of the elements in l

Java 8 eases the syntax

- Java 8 allows you to make objects that act as functions, more easily
  - Instead of this

```
MyList l = ...;
MyList l1 = l.map(new AddOne());
MyList l2 = l.map(new MultTwo());
```

  - Write this

```
MyList l = ...;
MyList l1 = l.map((x) -> x + 1);
MyList l2 = l.map((y) -> y * 2);
```