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

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

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

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

\[
\text{class C} \{ \ f1 \ldots \ fn; \ ml \ldots \ mn; \ \}
\]

• becomes

\[
\begin{align*}
\text{let make()} &= \\
&\text{let } f1 = \ldots \\
&\ldots \\
&\text{and } fn = \ldots \text{ in} \\
&(\ \text{fun} \ldots, (* \text{body of } ml *) \\
&\ldots \\
&\text{fun} \ldots, (* \text{body of } mn *)
\end{align*}
\]

• make () is like the constructor

• The closure environment contains the fields
Quiz 1: Is Circle Encoded Correctly?

```java
class Circle {
    float r = 0;
    void set_r (float t) { r = t; }
    float get_r () { return r; }
    float area(){
        return 3.14 * r * r;
    }
}
```

C c = new Circle();
c.set_r(1.0);
float y = c.get_r();
c.area();

A. True
B. False

```javascript
let make () =
    let r = 0.0 in
    ((fun y -> let r = y in ()),
     (fun () -> r),
     fun ()-> r *. r *. 3.14)
```

let (set_r, get_r, area) =
    make ();;
set_r 1.0;;
let y = get_r();;
area();;
Quiz 1: Is Circle Encoded Correctly?

```java
class Circle {
    float r = 0;
    void set_r (float t) { r = t; }
    float get_r () { return r; }
    float area(){
        return 3.14 * r * r;
    }
}
```

C c = new Circle();
c.set_r(1.0);
float y = c.get_r();
c.area();

A. True
B. False

```javascript
let make () =
    let r = ref 0.0 in
    ((fun y -> let r := y in ()),
     (fun () -> !r),
     fun ()=> !r *. !r *. 3.14
)

let (set_r, get_r, area) = make ();
set_r 1.0;;
let y = get_r();
Area();
```
Relating Closures to Objects

× A closure is like an object with a designated `eval()` method
  • The type of `eval` corresponds to the type of the closure’s function, `T -> U`

```java
interface Func<T,U> {
    U eval(T x);
}
class G implements Func<T,U> {
    U eval(T x) { /* body of fn */ }
}
```

× Environment is stored as field(s) of `G`
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;;

ew Add1().eval(2);
new Add1().eval(3)
Quiz 2: What does this evaluate to?

```java
interface IntIntFun {
    Integer eval(Integer x);
}

class Foo implements IntIntFun {
    Integer eval(Integer x) {
        return x * 2;
    }
}

new Foo(5);
```

A. 5
B. 10
C. 6
D. None of the above
Quiz 2: What does this evaluate to?

```java
interface IntIntFun {
    Integer eval(Integer x);
}

class Foo implements IntIntFun {
    Integer eval(Integer x) {
        return x * 2;
    }
}

new Foo(5);
```

A. 5  
B. 10  
C. 6  
D. None of the above (should be called `new Foo().eval(5)`)
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());
Quiz 3: What does this evaluate to?

```java
interface IntIntFun {
    Integer eval(Integer x);
}
class Foo implements IntIntFun {
    Integer eval(Integer x) {
        return x * 2;
    }
}
interface IntIntFunFun {
    Integer eval(IntIntFun x);
}
class AppToFive implements IntIntFunFun {
    Integer eval(IntIntFun f) {
        return f.eval(5);
    }
}
new AppToFive().eval(new Foo());
```

A. 5  
B. 10  
C. 6  
D. Error
Quiz 3: What does this evaluate to?

```java
interface IntIntFun {
    Integer eval(Integer x);
}
class Foo implements IntIntFun {
    Integer eval(Integer x) {
        return x * 2;
    }
}
interface IntIntFunFun {
    Integer eval(IntIntFun x);
}
class AppToFive implements IntIntFunFun {
    Integer eval(IntIntFun f) {
        return f.eval(5);
    }
}
```

```java
new AppToFive().eval(new Foo());
```

A. 5  
B. 10  
C. 6  
D. Error
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

```java
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> eval(Int x) {
        return new Add().eval(x);
    }
}
```

```kotlin
fun b -> a + b
let add1 = add 1;;
add1 4;;
```

```clojure
(let [add a b = a + b]
  (def add1 (fn [b] (+ a b))
  (add1 4))

(let [add a b = a + b]
  (let [add1 = (add 1)]
    (add1 4))
```

```python
a = 1
Func<Int,Int> add1 = new Add().eval(1);
add1.eval(4);
```
Encoding Closures with Objects

We can apply this transformation in general

\[
\begin{align*}
\text{(fun x -> (* body of fn *))} & \quad \text{...} \\
\text{let h f ... = ...f y...} & \quad \text{...}
\end{align*}
\]

- \text{becomes}

```java
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)...
    }
}
```

- \text{F is the interface of a closure’s function}
- \text{G represents the particular function}
Quiz 4: Are these two versions equivalent?

A. True
B. False
Quiz 4: Are these two versions equivalent?

let mult x y = x * y
let f = mult 2 in
f 3;;

A. True
B. False

interface IntIntFun {
   Integer eval(Integer x);
}
class Mult implements IntIntFun {
   private int x;
   Mult(int x) { this.x = x }
   Integer eval(Integer y) {
      return x * y;
   }
}
Mult f = new Mult(2);
f.eval(3);
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 results collected in a list

- Can we encode this in Java?
  - Using object-oriented programming
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);
    }
}
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

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

(can make this polymorphic but will keep it simpler for now)
A Map Method for Lists (cont.)

Examples

- Two classes which both implement Function interface

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

```java
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

```java
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

```ocaml
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

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

- A classes which implements IntBinFunction interface

```java
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:

```java
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

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

• Write this

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

- Closures and objects are related
  - Both of them allow
    - Data to be associated with higher-order code
    - Passing 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

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