CMSC 330: Organization of Programming Languages

Relationships between Functional, Imperative, and Object-Oriented Programming

OCaml Language Choices

- Implicit or explicit declarations?
  - Explicit – variables must be introduced with let before use
  - But you don't need to specify Tes
- Static or dynamic Tes?
  - Static – but you don't need to state Tes
  - OCaml does *Te inference* to figure out Tes for you
  - Good: less work to write programs
  - Bad: easier to make mistakes, harder to find errors
- Functional programming

What about Imperative Programming?

- In C or Java, we're used to doing things like:

```
int x = 3;
int y = 4;

void foo(void) {
  x = 42;
  y = x + 2;
}

int bar(void) {
  return x + y;
}
```

  - Can we model this without imperative constructs?
    - Imperative = able to *change* values in memory

Idea: "Thread" State through Fns

```
Te state = (char * int) list
let read (s:state) (x:char):int = List.assoc x s
let write (s:state) (x:char) (i:int):state =
  let s' = List.remove_assoc x s in
  (x,i)::s'

let foo (s0:state):state =  (* could change to state*unit *)
  let s1 = write s0 'x' 42 in
  let s2 = write s1 'y' ((read s1 'x') + 2) in
  s2

let bar (s0:state):(state*int) =
  (s0, (read s0 'x') + (read s0 'y'))
```


This Can Actually be a Good Idea

- The Haskell language is *purely functional*
  - No way to write to memory, ever
- But, you can play the trick we just saw
  - In Haskell, something that behaves like the state Te is a *monad*
  - Used for a bunch of different things
    - And there's some interesting theory to go with it
- OCaml is only *mostly functional*
  - It does actually have imperative constructs

Monad: make the state implicit

- Signature, and example implementation

```
module Te MONAD =
  sig
    Te 'a monad
    val return : 'a -> 'a monad
    val bind : 'a monad -> ('a -> 'b monad) -> 'b monad
  end

Module Monad : MONAD =
  struct
    Te 'a monad = state -> 'a * state
    let return x = function s -> (x,s)
    let bind m f = function s ->        (* sequencing *)
      let (v,s') = m s in f v s'
  end
```

State Monad: adding read/write

```
module Te STATE_MONAD =
  sig
    Te 'a monad
    Te state = (char*int) list
    val return : 'a -> 'a monad
    val bind : 'a monad -> ('a -> 'b monad) -> 'b monad
    val readm : char -> int monad
    val writem : char -> int -> unit monad
    val run : state -> 'a monad -> 'a
  end

let m : int monad =
  bind (writem 'x' 42)  (function _ ->
  bind (readm 'x')      (function x ->
  bind (writem 'y' x+2) (function _ ->
       (readm y);;

run [] m
```

x := 42;
y := x+2;
return y

Implementing the State Monad

```
Module StateMonad =
  struct
    Te 'a monad = state -> 'a * state
    let return (x:'a) : 'a monad = function s -> (x,s)
    let bind (m:'a monad) (f:'a -> 'b monad):'b monad =
      function s -> let (v,s') = m s in f v s'
    let run (s:state) (m:'a monad): 'a = m s
    let readm (x:char) : int monad =
      function s -> ((read s x),s)
    let writem (x:char) (i:int): unit monad =
      function s -> let s' = write s x i in ((),s')
  end
```

In Haskell, this monad is built in. The run function is called implicitly: the program is effectively of Te 'a monad And Haskell itself passes in the empty state to start it going

Object-Oriented Languages

- We've seen how we could model state without imperative features
- What about object-oriented constructs?
  - Can we encode them?

The Java Stack Class

```
public class Stack<T> {
  private class Entry {
    T elt; Entry next;
    Entry(T e, Entry n) { elt = e; next = n; }
  };
  private Entry theStack;
  public void push(T e) {
    theStack = new Entry(e, theStack);
  }
  public T pop() {
    if (theStack == null) throw new NoSuchElementException();
    T temp = theStack.elt;
    theStack = theStack.next;
    return temp;
  }
}
```


Writing a "Stack" in OCaml: Take 1

```
Module type STACK =
  sig
    type 'a stack
    val new_stack : unit -> 'a stack
    val push : 'a stack -> 'a -> unit
    val pop : 'a stack -> 'a
  end

module Stack : STACK =
  struct
    type 'a stack = 'a list ref
    let new_stack () = ref []
    let push s x = s := (x::!s)
    let pop s = match !s with
      [] -> failwith "Empty stack"
    | (h::t) -> s := t; h
  end
```


Writing a "Stack" in OCaml: Take 2

```
type 'a stack = { push: 'a -> unit; pop: unit -> 'a }
let new_stack () : 'a stack =
  let this = ref [] in
    let pushF x = this := (x::!this) in
    let popF () = match !this with
        [] -> failwith "Empty stack"
      | (h::t) -> this := t; h
    in
    {push = pushF, pop = popF}
```

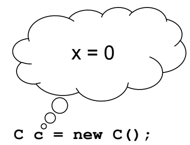
```
# let s = new_stack ();;
val s : '_a stack = { push = <fun>; pop = <fun> }
# s.push 3;;
- : unit = ()
# s.pop ();;
- : int = 3
```

Relating Objects and Closures

- An object...
  - Is a collection of fields (data)
  - ...and methods (code)
  - When a method is invoked, it is passed an implicit *this* parameter it can use to access fields
- A closure...
  - Is a pointer to an environment (data)
  - ...and a function body (code)
  - When a closure is invoked, it is passed its environment to access free variables

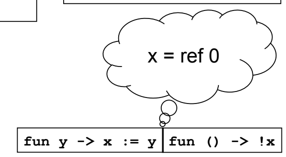
Relating Objects and Closures (cont'd)

```
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) )
```




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


Encoding Objects with Lambda

- We can apply this transformation in general

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

  - becomes

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

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

But …

- How is the encoding affected if a class refers to itself?

```
Class IntList {
  int x;
  IntList next;
  …
  void append(IntList x) {
    if (next == null) then
      next = x;
    else
      next.append(x);
  }
}
```

- *Important: how the class is represented must appear in its type, but then this violates abstraction. Try encoding the above to see why.*

Recall a Useful Higher-Order Function

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

- Can we encode this in Java?

A Map Method for Stack

- To write a map method, we need some way of passing a function into another function
  - We can do that with an object with a known method

```
public interface Function<A,B> {
  A eval(B arg);
}
```

A Map Method for Stack, cont'd

- Here are two classes which both implement this Function interface:

```
class AddOne implements Function<Integer,Integer> {
  Integer eval(Integer arg) {
    return new Integer(arg.intValue() + 1);
  }
}
```

```
class MultTwo implements Function<Integer,Integer> {
  Integer eval(Integer arg) {
    return new Integer(arg.intValue() * 2);
  }
}
```


A Map Method for Stack, cont'd

```
class Entry<S> {
    S elt; Entry<S> next;
    Entry(S x, Entry<S> n) { elt = x; next = n; }
    <U> Entry<U> map(Function<S,U> f) {
      if (next == null) return new Entry<U>(f.eval(elt), null);
      else return new Entry<U>(f.eval(elt), next.map(f));
    }
  }

public class Stack<T> {
  private Entry<T> theStack;
  ...
  <S> Stack<S> map(Function<T,S> f) {
    Stack<S> s = new Stack<S>();
    s.theStack = theStack.map(f);
    return s;
  }
}
```


A Map Method for Stack, con't.

- Then to apply the function, we just do

```
Stack<Integer> s = ...;
Stack<Integer> t = s.map(new AddOne());
Stack<Integer> u = s.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

Relating Closures and Objects

Note: We're being lazy and using Object instead of generics, but obviously you could switch to obtain better type safety

```
let app f x = f x
```

a = 3

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

```
let add a b = a + b;;
let f = add 3;;
app f 4;;
```

```
interface F {
  Object eval(Object y);
}
class C {
   static Object app(F f, Object x) {
     return f.eval(x);
   }
}
```

```
class G implements F {
  int a;
  G(int a) { this.a = a; }
  Object eval(Object y) {
    return
      new Integer(a +
           ((Integer) y).intValue());
  }
}
```

```
F adder = new G(3);
C.app(adder, 4);
```

a = 3

Encoding Lambda with Objects

- We can apply this transformation in general

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

  - becomes

```
interface F { Object eval(Object x); }
class G implements F {
  Object eval(Object x) { /* body of fn */ }
}
class C {
  T h(F f, ...) {
    ...f.eval(y)...
  }
}
```

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

Code as Data

- The key insight in all of these examples is to treat *code* as if it were *data*
  - Higher-order functions allow code to be passed around the program
  - As does object-oriented programming
- This is a powerful programming technique
  - And it can solve a number of problems quite elegantly
- Closures and objects are related
  - Both of them allow data to be associated with higher-order code as its passed around (but we can even get by without this)