Objects vs. Functional Programming

Object-oriented programming (OOP)
- Computation as interactions between objects
- Objects encapsulate mutable data (state)
  - 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

An Integer “Stack” Abstraction in Java

```java
class Stack {
    class Node {
        Integer val; Node next;
        Node(Integer v, Node n) { val = v; next = n; }
    }
    private Node theStack;
    void push(Integer v) {
        theStack = new Node(v, theStack);
    }
    Integer pop() {
        if (theStack == null)
            throw new NoSuchElementException();
        Integer temp = theStack.val;
        theStack = theStack.next;
        return temp;
    }
}
```

A “Stack” Abstraction in OCaml

```ocaml
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
```

Another “Stack” Abstraction in OCaml

```ocaml
let new_stack () =
    let this = ref [] in
    let push x = this := (x::!this) and pop () = match !this with
        [] -> failwith "Empty stack"
      | (h::t) -> this := t; h in
    (push, pop)
```

Two OCaml Stack Implementations

1st implementation (OOP style)
- Based on modules
- Specifies methods for
  - Creating stack
  - Pushing value onto stack parameter
  - Popping value from stack parameter

2nd implementation (FP style)
- Based on closures
- Creating stack returns tuple containing
  - Closure for pushing value onto created stack
  - Closure for popping value from created stack
Relating Objects and Closures

- An object...
  - Is a collection of fields (data)
  - ...and methods (code)
  - When a method is invoked
    - Method has implicit 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

Encoding Objects with Functions

- 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 *)
  ...
  ... , (* body of mn *)
  )
  ```
  - `make()` is like the constructor
  - The closure environment contains the fields

A Map Method for Stack

- 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

Recall a Useful Higher-Order Function

- 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 Stack (cont.)

- Examples
  - Two classes which both implement `Function` interface

```java
class AddOne implements Function {
    Integer eval(Integer arg) {
        return new Integer(arg + 1);
    }
}

class MultTwo implements Function {
    Integer eval(Integer arg) {
        return new Integer(arg * 2);
    }
}
The New Stack Class

class Stack {
    class Node {
        Integer val; Node next;
        Node (Integer v, Node n) { val = v; next = n; }
        Entry map (Function f) {
            if (next == null)
              return new Node (f.eval(val), null);
            else return new Node (f.eval(val), next.map (f));
      }
    }
    Node theStack;
    Stack map (Function f) {
        Stack s = new Stack();
        Node (Integer v, Node n) { val = v; next = n; }
        Node (Integer v, Node n) {
            s.theStack = theStack.map (f);
            return s;
        }
    }
}

Applying Map To A Stack

- Then to apply the function, we just do

  Stack s = ...;
  Stack t = s.map (new AddOne());
  Stack 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

- Let app f x = f x
- Let f x = x
- Interface F {
  Integer eval (Integer y);
}
- Class C {
  Static integer app (F f, Integer x) {
    return f.eval (x);
  }
}
- Class G implements F {
  Integer a;
  G (Integer a) { this.a = a; }
  Integer eval (Integer y) {
    return new Integer (a + y);
  }
}
- Let add a b = a + b;
- Let f = add 3;
- Let g = new G (3);
- Let h = new H (4);
- Let f = add 3;
- Let g = new G (3);
- 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

Encoding Functions with Objects

- We can apply this transformation in general

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