OCaml Language Choices

• Implicit or explicit declarations?
  – Explicit – variables must be introduced with `let` before use
  – But you don’t need to specify types

• Static or dynamic types?
  – Static – but you don’t need to state types
  – OCaml does type inference to figure out types 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:

```ocaml
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

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

Idea: “Thread” State through Fns

```ocaml
let x = 3;
let y = 4;
function foo() {
  x = 42;
y = x + 2;
}

function bar() {
  return x + y;
}
```

Monad: make the state implicit

• Signature, and example implementation

```ocaml
module Te MONAD =
  signature {
    type 'a monad
    val return : 'a -> 'a monad
    val bind : 'a monad -> ('a -> 'b monad) -> 'b monad
  }

module Monad : MONAD =
  struct
    let monad = state -> 'a * state
    let return x = function x -> (x, x)
    let bind m f = function x -> (* sequencing *)
      (let (v, s') = m x in if v = s')
  end
```
State Monad: adding read/write

Module Te STATE_MONAD =
  sig
    'a monad
    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

Implementing the State Monad

Module StateMonad =
  struct
    '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 -> let s' = read s x in ((),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
    '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

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

Encoding Objects with Lambda

- We can apply this transformation in general
  - class C { f1 ... fn; m1 ... mn; }
  - becomes
    ```java
    let make () =
        let f1 = ... in
        ...
        let fn = ... in
        { fun ... , (* body of ml *)
          ...
          fun ..., (* body of mn *) }
    ```
    - make () is like the constructor
    - the closure environment contains the fields

Recall a Useful Higher-Order Function

- 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
    ```java
    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:

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

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

A Map Method for Stack, con’t.

- Then to apply the function, we just do

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

Relating Closures and Objects

- 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

Encoding Lambda with Objects

- We can apply this transformation in general

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

  - becomes

  ```java
  interface F { Object eval(Object x); }
  class G implements F {
      Object eval(Object x) { /* body of fn */ }
  }
  ```

  - `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)