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

So Far, only Functional Programming

- We haven’t given you any way so far to change something in memory in OCaml
  - All you can do is create new values from old
- This actually makes programming easier!
  - Don’t care whether data is shared in memory
    - Aliasing is irrelevant
  - Provides strong support for compositional reasoning and abstraction
    - Ex: Calling a function `f` with argument `x` always produces the same result

What about Imperative Programming?

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

```plaintext
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

```
type 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 s x 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 type 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

Imperative OCaml

- There are three basic operations on memory:
  - `ref : 'a -> 'a ref`
    - Allocate an updatable reference
  - `! : 'a ref -> 'a`
    - Read the value stored in reference
  - `:= : 'a ref -> 'a -> unit`
    - Write to a reference

```
let x = ref 3  (* x : int ref *)
let y = !x
x := 4
```

Comparison to L- and R-values

- Recall that in C/C++/Java, there’s a strong distinction between l- and r-values
  - An **r-value** refers to just a value, like an integer
  - An **l-value** refers to a location that can be written
- A variable’s meaning depends on where it appears
  - On the right-hand side, it’s an r-value, and it refers to the contents of the variable
  - On the left-hand side of an assignment, it’s an l-value, and it refers to the location the variable is stored in
L-Values and R-Values (cont’d)

- Notice that x, y, and 3 all have type `int`

```
int x, y;
x = 3;
y = x;
3 = x;
```

Comparison to OCaml

- In OCaml, an updatable location and the contents of the location have different types
  - The location has a `ref` type

```
let x = ref 0;;
let y = ref 0;;
x := 3;;  (* x : int ref *)
y := (!x);;
3 := x;;  (* 3 : int; error *)
```

Capturing a ref in a Closure

- We can use `ref` s to make things like counters that produce a fresh number “everywhere”

```
let next =
  let count = ref 0 in
  function () ->
    let temp = !count in
    count := (!count) + 1;
    temp;;
# next ();;
- : int = 0
# next ();;
- : int = 1
```

Semicolon Revisited; Side Effects

- Now that we can update memory, we have a real use for `;` and `(): unit`
  - `e1; e2` means evaluate `e1`, throw away the result, and then evaluate `e2`, and return the value of `e2`
  - `()` means “no interesting result here”
  - It’s only interesting to throw away values or use `()` if computation does something besides return a result

- A side effect is a visible state change
  - Modifying memory
  - Printing to output
  - Writing to disk
Grouping with `begin...end`

- If you’re not sure about the scoping rules, use `begin...end` to group together statements with semicolons

```ml
let x = ref 0
let f () =
  begin
    print_string "hello";
    x := (!x) + 1
  end
```

The Trade-Off of Side Effects

- Side effects are absolutely necessary
  - That’s usually why we run software! We want something to happen that we can observe

- They also make reasoning harder
  - Order of evaluation now matters
  - Calling the same function in different places may produce different results
  - Aliasing is an issue
    - If we call a function with refs `r1` and `r2`, it might do strange things if `r1` and `r2` are aliased

Object-Oriented Languages

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

The Stack Class

```ml
class Stack<Typ> {
  class Entry {
    Typ elt; Entry next;
    Entry(Typ e, Entry n) { elt = e; next = n; }
  }
  private Entry theStack;
  void push(Typ e) {
    theStack = new Entry(e, theStack);
  }
  Typ pop() {
    if (theStack == null) throw new NoSuchElementException();
    Typ temp = theStack.elt;
    theStack = theStack.next;
    return temp;
  }
}
```
Writing a “Stack” in OCaml: Take 1

```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
```

Writing a “Stack” in OCaml: Take 2

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

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 it can use to access variables

```ocaml
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) )
```

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

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 {
        Object eval(Object arg);
    }
    ```

Recall a Useful Higher-Order Function

- Can we encode this in Java?
  
  ```
  let rec map f = function
  [] -> []
  | (h::t) -> (f h)::(map f t)
  ```

A Map Method for Stack, cont’d

- Here are two classes which both implement this Function interface:
  
  ```
  class AddOne implements Function {
    Object eval(Object arg) {
        Integer old = (Integer) arg;
        return new Integer(old.intValue() + 1);
    }
  }
  ```

  ```
  class MultTwo implements Function {
    Object eval(Object arg) {
        Integer old = (Integer) arg;
        return new Integer(old.intValue() * 2);
    }
  }
  ```
**A Map Method for Stack, cont’d**

class Stack {
    class Entry {
        Object elt; Entry next;
        Entry(Object x, Entry n) { elt = x; next = n; }
        Entry map(Function f) {
            if (next == null) return new Entry(f.apply(elt), null);
            else return new Entry(f.apply(elt), next.map(f));
        }
    }
    Entry theStack;
    ...
    Stack map(Function f) {
        Stack s = new Stack();
        s.theStack = theStack.map(f);
        return s;
    }
}

**A Map Method for Stack, con't.**

- 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**

```plaintext
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;;;

**Encoding Lambda with Objects**

- We can apply this transformation in general

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

  - becomes

  ```plaintext
  interface F { Object eval(Object 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 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)