CMSC 330: Organization of Programming Languages

Objects and Functional Programming

and

Parameter Passing

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

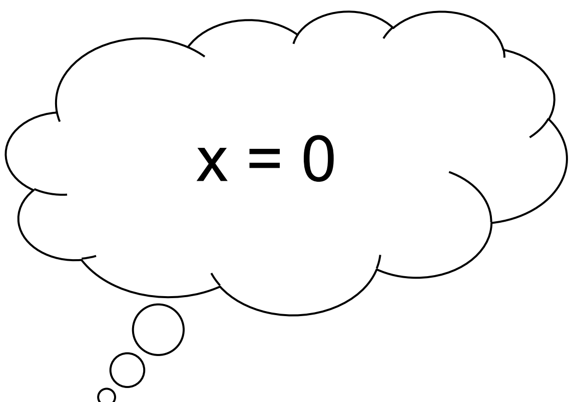
Relating Objects and Closures

- 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

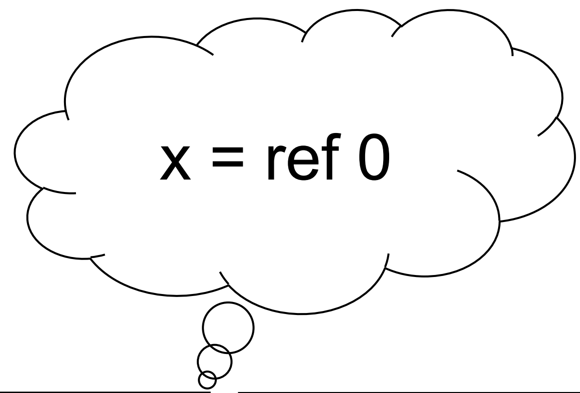
Relating Objects and Closures (cont.)

```
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 Functions

- We can apply this transformation in general

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

  - becomes

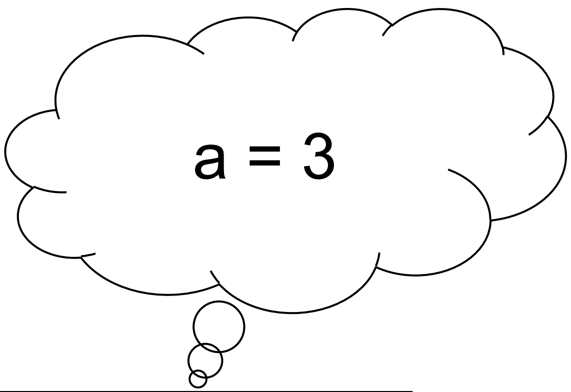
```
let make () =
 let f1 = ...
 ...
 and fn = ... in
 ( fun ... , (* body of m1 *)
   ...
   fun ...,   (* body of mn *)
 )
```

Tuple containing closures

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

Relating Closures and Objects

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



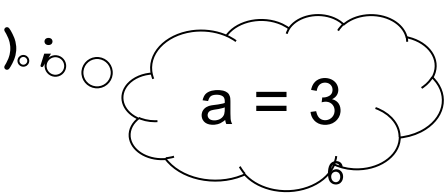

```
fun b -> a + b
```

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

```
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);
  }
}
```

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



Encoding Functions 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 {
  Typ h(F f, ...) {
      ...f.eval(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)

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

An Integer List Abstraction in Java

```
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);
  }

}
```

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 resulting modified list
- Can we encode this in Java?
  - Using object oriented programming

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

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

A Map Method for Lists (cont.)

- Examples
  - Two classes which both implement Function interface

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

```
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

```
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 fold

```
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

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

A Fold Method for Lists (cont.)

- Examples
  - A classes which implements IntBinFunction interface

```
class Sum implements IntBinFunction {
  Integer eval(Integer arg1, Integer arg2) {
    return new Integer(arg1 + arg2);
  }
}
```

The New 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:

```
MyList l = ...;
int s = l.fold (new AddOne(), 0);
```

- The result is that s contains the sum of the elements in l

Parameter passing

- Semantics for how arguments are passed to functions, and what happens to them while they're there
- Three main styles
  - Call by value (most common)
  - Call by reference (next most)
  - Call by name (least common)
    - Also called lazy evaluation

Parameter Passing in OCaml

- Quiz: What value is bound to z?

```
let add x y = x + y

let z = add 3 4
```

7

```
let add x y = x + y

let z = add (add 3 1) (add 4 1)
```

9

```
let r = ref 0
let add x y = (!r) + x + y
let set_r () = r := 3; 1

let z = add (set_r ()) 2
```

6

Actuals evaluated before call

Call-by-Value

- In call-by-value (*cbv*), arguments to functions are fully evaluated before the function is invoked
  - Also in OCaml, in let x = e1 in e2, the expression e1 is fully evaluated before e2 is evaluated
- C, C++, and Java also use call-by-value

```
int r = 0;

int add(int x, int y) { return r + x + y; }

int set_r(void) {
  r = 3;
  return 1;
}

add(set_r(), 2);
```

Call-by-Value in Imperative Languages

- In C, C++, and Java, call-by-value has another feature
  - What does this program print? 0

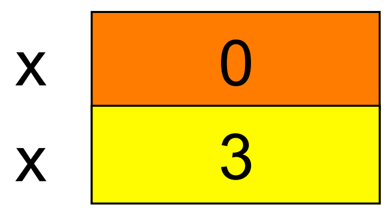
```
void f(int x) {
  x = 3;
}

int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

  - Cbv protects function arguments against modifications

Call-by-Value (cont.)

- Actual parameter is copied to stack location of formal parameter

```
void f(int x) {
  x = 3;
}
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```

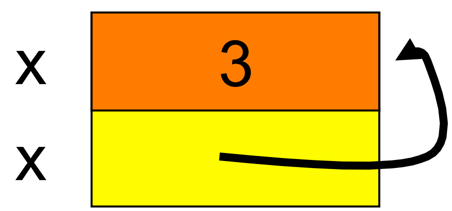


Call-by-Reference

- Alternative idea
  - Implicitly pass a pointer or reference to actual parameter
  - If the function writes to it the actual parameter is modified

```
void f(int x) {
  x = 3;
}
```

```
int main() {
  int x = 0;
  f(x);
  printf("%d\n", x);
}
```



Call-by-Reference (cont.)

- Advantages
  - Avoid copying entire argument to called function
    - More efficient when passing large (multi-word) arguments
    - Can do this without explicit pointer manipulation
- Disadvantages
  - More work to pass non-variable arguments
    - Examples: constant, function result
  - May be hard to tell if function modifies arguments
  - Introduces aliasing

Aliasing

- We say that two names are aliased if they refer to the same object in memory
  - C examples (this is what makes optimizing C hard)

```
int x;
int *p, *q; /*Note that C uses pointers to
                simulate call by reference */
p = &x;   /* *p and x are aliased */
q = p;    /* *q, *p, and x are aliased */
```

```
struct list { int x; struct list *next; }
struct list *p, *q;
...
q = p;    /* *q and *p are aliased */
          /* so are p->x and q->x */
          /* and p->next->x and q->next->x... */
```

Call-by-Reference (cont.)

- Call-by-reference is still around (e.g., C++)

```
int x = 0;                        // C++
void f(int& y) { y = 1; }         // y = reference var
f(x); printf("%d\n", x);          // prints 1
f(2);                             // error
```

- Seems to be less popular in newer languages
  - Older languages still use it
    - Examples: Fortran, Ada, C with pointers
  - Possible efficiency gains not worth the confusion
  - The “hardware” is basically call-by-value
    - Although call by reference is not hard to implement and there may be some support for it

Call-by-Value Discussion

- Cbv is standard for languages with side effects
  - When we have side effects, we need to know the order in which things are evaluated
    - Otherwise programs have unpredictable behavior
  - Call-by-value specifies the order at function calls
  - Call-by-reference can sometimes give different results
- Differences blurred for languages like Java
  - Language is call by value
  - But most parameters are object references anyway
    - Still have aliasing, parameter modifications at object level

Call-by-Name

- Call-by-name (cbn)
  - First described in description of Algol (1960)
  - Generalization of Lambda expressions (to be discussed later)
  - Idea: In a function:

    Let add x y = x+y
    add (a*b) (c*d)

Example:
add (a*b) (c*d) =
(a*b) + (c*d) ← executed function

    Then each use of x and y in the function definition is just a literal substitution of the actual arguments, (a*b) and (c*d), respectively
  - Usage: It's the core evaluation strategy in Haskell

Call-by-Name (cont.)

- In call-by-name (*cbn*), arguments to functions are evaluated at the last possible moment, just before they're needed

```
let add x y = x + y

let z = add (add 3 1)  (add 4 1)
```

OCaml; cbv; arguments evaluated here

Haskell; cbn; arguments evaluated here

```
add x y = x + y

z = add (add 3 1) (add 4 1)
```

Call-by-Name (cont.)

- What would be an example where this difference matters?

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

OCaml; cbv; infinite recursion at call

```
cond p x y = if p then x else y
loop n = loop n
z = cond True 42 (loop 0)
```

Haskell; cbn; never evaluated because parameter is never used

Call by Name Examples

- P(x) {x = x + x;}

  What is: Y = 2;
  P(Y); ← becomes Y = Y+Y = 4
  write(Y)

- F(m) {m = m + 1; return m;}

  What is: int A[10];
  m = 1;
  P(A[F(m)])

becomes P(A[F(m)]) → A[F(m)] = A[F(m)]+A[F(m)]
→ A[m++] = A[m++]+A[m++]
→ A[2] = A[3]+A[4]

Call by Name Anomalies

- Write a function to exchange values of X and Y
- Usual way - swap(x,y) { t=x; x=y; y=t; }
  - Cannot do it with call by name!
- Reason
  - Cannot handle both of following
    - swap(A[m], m)
    - swap(m, A[m])
  - One of these must fail
    - swap(A[m], m) → t = A[m] ; A[m] = m; m = t;
    - swap(m, A[m]) → t = m ; m = A[m]; A[m] = t;   // fails!

Two Cool Things to Do with CBN

- CBN is also called lazy evaluation
  - CBV is also known as eager evaluation

- Build control structures with functions

```
let cond p x y = if p then x else y
```

- Build “infinite” data structures

```
integers n = n::(integers (n+1))
take 10 (integers 0)  (* infinite loop in cbv *)
```

Simulating CBN with CBV

- Thunk
  - A function with no arguments
- Algorithm
  1. Replace arguments $a_1 \ldots a_k$ by thunks $t_1 \ldots t_k$
     - When called, t_i evaluates and returns a_i
  2. Within body of the function
     - Replace formal argument with thunk invocations

```
let add1 x = x + 1 in add1 (2+3)
```

↓

```
let add1 x = x() + 1 in add1 (fun () -> (2+3))
```

Simulating CBN with CBV (cont.)

```
let cond p x y = if p then x else y
let rec loop n = loop n
let z = cond true 42 (loop 0)
```

- becomes...

Get 1st arg

Return 2nd arg

Never invoked

```
let cond p x y = if (p ()) then (x ()) else (y ())
let rec loop n = loop n  (* didn't transform... *)
let z = cond (fun () -> true)
             (fun () -> 42)
             (fun () -> loop 0)
```

Parameters are now thunks

Three-Way Comparison

- Consider the following program under the three calling conventions
  - For each, determine i's value and which a[i] (if any) is modified

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

Example: Call-by-Value

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]	f	g
1	0	1	2		
				1	1
				5	2

Example: Call-by-Reference

```
int i = 1;

void p(int f, int g) {
  g++;
  f = 5 * i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i/g	a[0]	a[1]/f	a[2]
1	0	1	2
2		10	
2		10	

Example: Call-by-Name

```
int i = 1;

void p(int f, int g) {
  g++;          i++;
  f = 5 * i;    a[i] = 5*i;
}

int main() {
  int a[] = {0, 1, 2};
  p(a[i], i);
  printf("%d %d %d %d\n",
    i, a[0], a[1], a[2]);
}
```

i	a[0]	a[1]	a[2]
1	0	1	2
2			10
2			10

The expression a[i] isn't evaluated until needed, in this case after i has changed.

Other Calling Mechanisms

- Call-by-result
  - Actual argument passed by reference, but not initialized
  - Written to in function body (and since passed by reference, affects actual argument)
- Call-by-value-result
  - Actual argument copied in on call (like cbv)
  - Mutated within function, but does not affect actual yet
  - At end of function body, copied back out to actual
- These calling mechanisms didn't really catch on
  - They can be confusing in cases
  - Recent languages don’t use them

CBV versus CBN

- CBN is flexible- strictly more programs terminate
  - E.g., where we might have an infinite loop with cbv, we might avoid it with cbn by waiting to evaluate
- Order of evaluation is really hard to see in CBN
  - Call-by-name doesn't mix well with side effects (assignments, print statements, etc.)
- Call-by-name is more expensive since
  - Functions have to be passed around
  - If you use a parameter twice in a function body, its thunk (the unevaluated argument) will be called twice
    - Haskell actually uses *call-by-need* (each formal parameter is evaluated only once, where it's first used in a function)