CS 3110

Lecture 24: Dynamic Dispatch

Prof. Clarkson

Spring 2015

Today's music: "The Core" by Eric Clapton

Review

Current topic: functional vs. object-oriented programming

Today:

- Continue encoding objects in OCaml
- The core of OOP
  - dynamic dispatch
  - sigma calculus

Review: key features of OOP

1. Encapsulation
2. Subtyping
3. Inheritance
4. Dynamic dispatch

Review: Counters

```
class Counter {
    protected int x = 0;
    public int get() { return x; }
    public void inc() { x++; }
}
```

Review: Objects

- Type of *object* is record of functions

```
type counter = {
   get : unit -> int;
   inc : unit -> unit;
}
```

- Let-binding hides internal state (with *closure*)

```
let x = ref 0 in {
   get = (fun () -> !x);
   inc = (fun () -> x := !x+1);
}
```

Review: Classes

- *Representation type* for internal state:

```
type counter_rep = {
    x : int ref;
}
```

- *Class* is a function from representation type to object:

```
let counter_class (r:counter_rep) = {
    get = (fun () -> !(r.x));
    inc = (fun () -> (r.x := !(r.x) + 1));
}
```

- *Constructor* uses class function to make a new object:

```
let new_counter () =
    let r = {x = ref 0} in
    counter_class r
```

Review: Inheritance

- Subclass creates an object of the superclass with the same internal state as its own
  - Bind resulting *parent object* to **`super`**
- Subclass creates a new object with same internal state
- Subclass copies (*inherits*) any implementations it wants from superclass

4. DYNAMIC DISPATCH

This

```
class SetCounter {
    protected int x = 0;
    public int get() { return x; }
    public void set(int i) { x = i; }
    public void inc() {
        this.set(this.get() + 1);
    }
}
```

`This`

- Enables methods to invoke other methods of same object
- (and more...)

- How to implement in OCaml?
  - objects are already parameterized on internal state
  - now, also parameterize object on...itself!
  - much like `let rec` in PS3, employ *backpatching*

Implementing `this`: Idea

- Create an object initially filled out with all dummy methods
  - It will become **`this`**
- Pass that object into the class function
- Imperatively update the methods in object with the right code
  - That code can use **`this`**

Implementing this: Code

```
type set_counter = {
  get : (unit -> int)  ref;
  set : (int  -> unit) ref;
  inc : (unit -> unit) ref;
}

let set_counter_class (r : counter_rep)
(this : set_counter) =
  ...

let new_set_counter () =
  ...
```

Implementing `this`: Code

```
let new_set_counter () =
  let r = {x = ref 0} in
  let obj = {
      get = ref (fun () -> 0);
      set = ref (fun n -> ());
      inc = ref (fun () -> ());
    } in
  set_counter_class r obj
```

Implementing `this`: Code

```
let set_counter_class
(r : counter_rep)
(this : set_counter) =
  this.get := (fun () -> !(r.x));
  this.set := (fun n -> r.x := n);
  this.inc := (fun () ->
    let n = !(this.get)()
    in !(this.set) (n+1));
  this
```

`This`

- Enables methods to invoke other methods of same object: check.
- **(and more...)**

Question #1

What is printed: A or B?

```
class C {
   void m() { this.m2(); }
   void m2() { System.out.println("A"); }
}
class D extends C {
   void m2() { System.out.println("B"); }
}
C c = new D();
c.m();
```

Question #1

What is printed: A or B?

```
class C {
   void m() { this.m2(); }
   void m2() { System.out.println("A"); }
}
class D extends C {
   void m2() { System.out.println("B"); }
}
C c = new D();
c.m();
```

Question #2

```
class SetCounter {
    protected int x = 0;
    public int get() { return x; }
    public void set(int i) { x = i; }
    public void inc() {
        this.set(this.get() + 1);
    }
}
class InstrCounter extends SetCounter {
    protected int a = 0;
    public int accesses() { return a; }
    public void set(int i) {
        a++;
        super.set(i);
    }
}
```

Does calling **`inc`** on an **`InstrCounter`** update **`a`?**

A: Yes, B: No

Question #2

```
class SetCounter {
    protected int x = 0;
    public int get() { return x; }
    public void set(int i) { x = i; }
    public void inc() {
        this.set(this.get() + 1);
    }
}
class InstrCounter extends SetCounter {
    protected int a = 0;
    public int accesses() { return a; }
    public void set(int i) {
        a++;
        super.set(i);
    }
}
```

Does calling `inc` on an `InstrCounter` update `a`?

A: Yes, B: No

Semantics of dynamic dispatch

`e1.m(e2) --> e1'.m(e2)`
    if **`e1 --> e1'`**

`v1.m(e2) --> v1.m(e2')`
    if **`e2 --> e2'`**

`v1.m(v2) --> e{v2/x}{v1/this}`
    if class of **`v1`** is **`C`** and **`m(x) { e }`** is defined in **`C`**
    (otherwise search upward through class hierarchy for definition of **`m`**)
    `v1` is the *receiving object* of the method call

I'm simplifying; the Java semantics takes 30 pages. #serious

`Counter` example

```
class SetCounter {
    protected int x = 0;
    public int get() { return x; }
    public void set(int i) { x = i; }
    public void inc() {
        this.set(this.get() + 1);
    }
}
class InstrCounter extends SetCounter {
    protected int a = 0;
    public int accesses() { return a; }
    public void set(int i) {
        a++;
        super.set(i);
    }
}
```

Does calling **`inc`** on an **`InstrCounter`** update **a?**

YES!

Counter example

```
class SetCounter {
    protected int x = 0;
    public int get() { return x; }
    pu
    pu

    }
}
class
    protected int a = 0;
    public int a  esses() { return a; }
    public void set(int i) {
        a++;
        super.set(i);
    }
}
```

When superclass method invokes another method of same object that is overridden in subclass, **need to use the subclass's code**, not the superclass's

Does calling **`inc`** on an **`InstrCounter`** update **a?**

YES!

Implementation of overriding: idea

- Create an object initially filled out with all dummy methods
  - It will become **`this`**
- Starting from the top of the class hierarchy and moving downward:
  - Pass that object into the class function that constructs the object
  - Make a copy of the object to preserve the superclass's methods
  - Imperatively update the methods in the object with the right code
    - Subclasses can thus override methods in superclasses
    - That code can use **`this`** and **`super`**

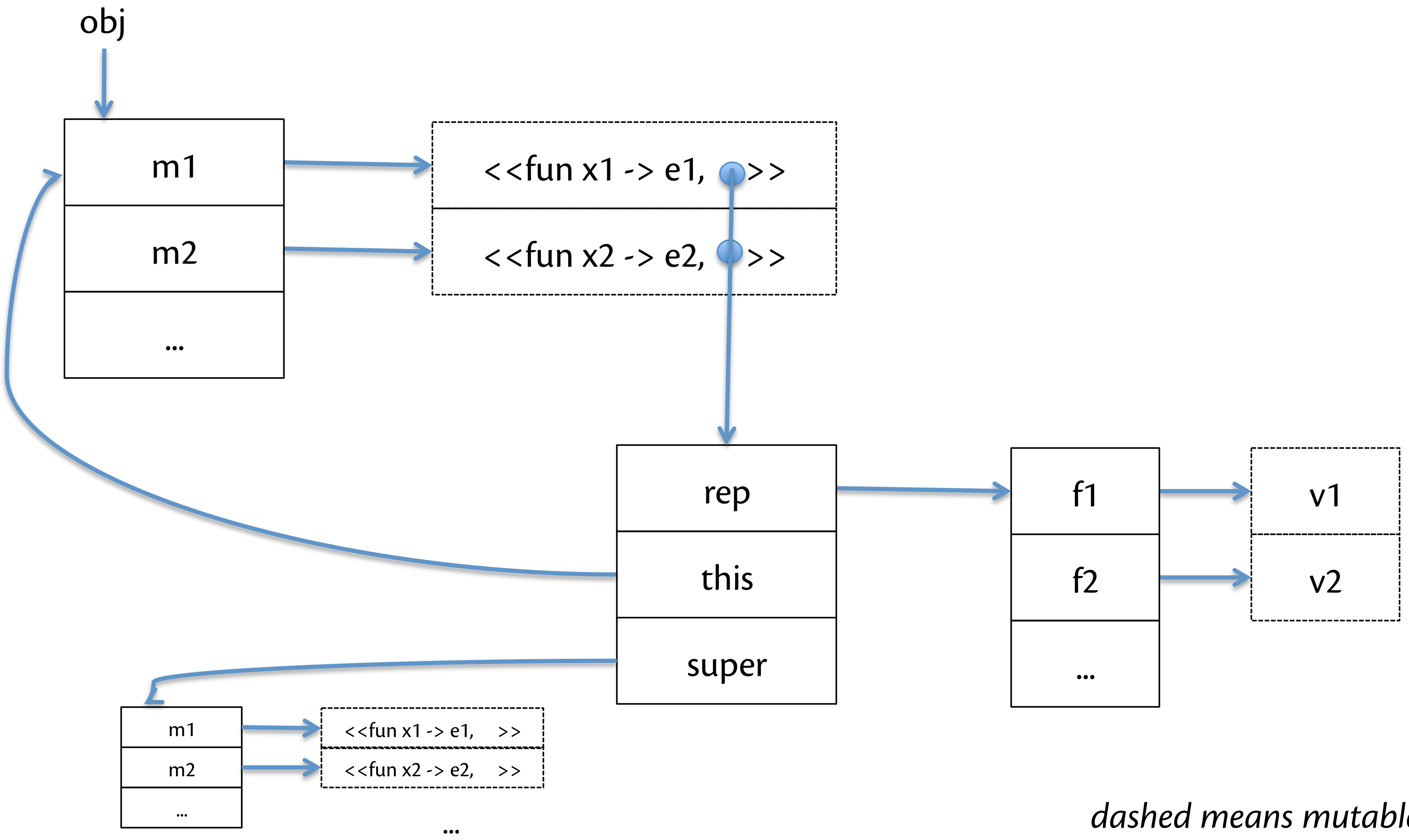
Implementation of overriding: code

```
let new_instr_counter () =
  let r = { x=ref 0; a=ref 0 } in
  let obj = {
        get = ref (fun () -> 0);
        set = ref (fun n -> ());
        inc = ref (fun () -> ());
        accesses = ref (fun () -> 0);
    } in
  instr_counter_class r obj
```

Implementation of overriding: code

```
let instr_counter_class
(r : instr_counter_rep) (this : instr_counter) =
  let super =
    let sc = set_counter_class
       (... r) (... this) in
    {get = !(sc.get);
     set = !(sc.set);
     inc = !(sc.inc);}
  in
    this.get := super.get;
    this.set :=
       (fun n -> r.a := !(r.a) + 1; super.set n);
    this.inc := super.inc;
    this.accesses := (fun () -> !(r.a));
    this
```

Object encoding in one picture



dashed means mutable

Recap

1. **Encapsulation:**
   - Internal state created by passing class function a record
   - Record never visible to outside world
   - Closures are a key part of that implementation
2. **Subtyping:**
   - Insert explicit coercion functions to upcast from subtype to supertype
   - Wouldn't be needed if OCaml's type system were a little richer
3. **Inheritance:**
   - Class function copies implementation of method from one object to another
   - Could implement more efficiently by having one method table per class instead of per object
4. **Dynamic dispatch:**
   - Object is parameterized on itself
   - Tricky to implement in OCaml

What is an object?

An object is a record of mutable functions, and is parameterized on both internal state and itself.
(in our encoding, anyway)

- We implemented (encoded) objects in OCaml
- "Just because you've implemented something doesn't mean you understand it." – Brian Cantwell Smith

I hope:

- Now you understand objects a little better
- Now you appreciate the power and complexity of OOP a lot better
- (go to CS 4120 for the full enchilada)

The core of OOP

- Just as FP has a core calculus inside it
  - lambda calculus
- OOP has a core calculus
  - sigma calculus

Sigma calculus

Syntax:

```
e ::=
   | x                          variable
   | e.l                        method invocation
   | e1.l := $x.e2              method update
   | [l1 = $x1.e1;              object
       ...;
      ln = $xn.en]
```

Methods

```
$x . e
```

- the body is `e`
- the bound variable `x` is the receiving object
- note: no other arguments (not needed!)

Semantics

```
e.l --> e'.l
   if e --> e'

v.l --> e{v/x}
   if v=[...; l = $x.e; ...]
```

values are objects

```
e1.l := $x.e2 --> e1'.l := $x.e2
   if e1 --> e1'

v.l := $y.e' --> v'
   if v=[...; l = $x.e; ...]
   and v'=[...; l = $y.e'; ...]
   and v and v' are the same except for l
```

Everything else is a luxury

- **Fields** are syntactic sugar for methods that ignore their receiving object
- **Integers** can be coded up as objects
- Other **data types** can be coded up as objects
- **Classes** are just objects with a method named new that constructs an object whose methods are copied over from the class
- Even **lambda calculus** can be encoded…

Encoding lambda in sigma

Ideas:

- A function is an object with a method **`eval`** and a field **`arg`**
- The field is filled in at the time of function application
- The method causes the function to be applied
  (i.e., beta reduction)

```
T : lambda_expr -> sigma_expr
T(x)       = x
T(e1 e2) = (T(e1).arg := T(e2)).eval
T(\x.e)   = [ arg = [];
                eval = $x . T(e){x.arg/x} ]
```

Closures vs. Objects

- We encoded objects in OCaml
  - closures (i.e., first-class functions) were an essential part of that encoding
- We encoded lambda calculus in sigma calculus
  - objects are an essential part of that encoding
  - (And you saw in 2110 that inner classes (like adapters for GUI buttons) capture variables from an outer scope)
- So closures can be implemented with objects
  - All of FP can be done in OOP
- And objects can be implemented with closures
  - All of OOP can be done in FP

Zen Koan

- The venerable master Zardoz was walking with a student, Zed. Hoping to prompt the master into a discussion, Zed said "Master, I have heard that objects are a very good thing - is this true?" Zardoz looked pityingly at the student and replied, "Foolish pupil - objects are merely a pitiable substitute for closures."
- Chastised, Zed took leave from the master and retreated into a quiet cell in the basement of Gates Hall, intent on studying closures. Zed carefully read the 3110 course notes, and implemented an OOP language using OCaml and closures. Zed learned much, and looked forward to informing the master of this progress.
- On the next walk with Zardoz, Zed attempted to impress the master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a pitiable substitute for closures." Zardoz responded by hitting Zed with a stick, saying "When will you learn? Closures are merely a pitiable substitute for objects."
- At that moment, Zed became enlightened.