CS 3110

Lecture 23: Object Encoding

ob-ject: to feel distaste for something – *Webster's Dictionary*

Prof. Clarkson

Spring 2015

Today's music: "Beautiful Object" by Glass Candy

Review

Current topic: functional vs. object-oriented programming

- Last time: the expression problem; OOP vs. FP isn't only a matter of taste

Today:

- What is an object?
- Implement/encode objects in OCaml

Question #1: What is an object?

A. Objects are entities that combine state, behavior, and identity.
B. Objects have state and behavior.
C. Objects encapsulate data and operations.
D. An object is a data structure encapsulating some internal state and offering access to this state to clients with a collection of methods.
E. None of the above

Question #1: What is an object?

A. Objects are entities that combine state, behavior, and identity. **[Wikipedia]**
B. Objects have state and behavior. **[Oracle]**
C. Objects encapsulate data and operations. **[Carrano & Prichard]**
D. An object is a data structure encapsulating some internal state and offering access to this state to clients with a collection of methods. **[Pierce]**
E. None of the above

What are key features of OOP?

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

- (Classes?)
- …

1. Encapsulation

- Object has *internal state*
- Object's *methods* can inspect and modify that state
- Clients cannot directly access state except through methods

...how is this (un)like OCaml modules?

2. Subtyping

- *Type* of an object involves the names and types of its methods
- Object of type t can be used in place of an object of type t' if t is a *subtype* of t'
- Subtyping depends on names and types of methods

...how is this (un)like OCaml types?

3. Inheritance

- Objects *inherit* some of their behavior
- Associated with *classes*
  - templates from which objects can be constructed
- *Subclassing* derives new classes from old classes
  - add new methods
  - *override* implementations of old methods
  - inherit other old methods

...how is this (un)like OCaml modules?

4. Dynamic dispatch

- Some might argue this is the *defining* characteristic of objects
  - But it's the one you won't have heard about in 2110!
- Method that is invoked ("dispatched") on an object is determined at run-time ("dynamically") rather than at compile-time ("statically")
- Special keyword: `this` or `self`
  - Always in scope inside a method
  - Always bound to the receiving object of a method invocation

...how is this (un)like OCaml functions in a module?

Object encoding

- **Rest of this lecture:** encode objects in OCaml
- **Purpose:** *understand* OOP features better by approximating them in OCaml
- **Non-purpose:** *exactly* model Java objects in all their rich details
- **Non-purpose:** use the OCaml object system to mimic Java objects

Running example: counters

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

1. ENCAPSULATION

Objects as records

- A Java object is a collection of named values
- An OCaml record is also a collection of named values
- So we could try something like:

```
{ x   = 0;
  get = ...;
  set = ...;   }
```

- But that would fail to provide encapsulation of `x`

Encapsulation of private state

- Idea: use let-binding to hide the state

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

- Record exposes only the methods
- The private field is hidden by the let-binding
  - Really: a closure is created for each method that has the state in its environment

Object type

- Type of the object we just created:

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

- Note: **x** is not exposed in type

Method invocation

- Given an object:

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

- We can invoke methods with field accesses:

```
c.inc(); c.inc(); c.get()
```

- Note: the parens are the unit value

Functions with objects

- OCaml functions can manipulate objects:

```
let inc3 (c:counter) =
   c.inc(); c.inc(); c.inc()
```

- OCaml functions can construct new objects:

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

2. SUBTYPING

Subtype of Counter

```
class ResetCounter extends Counter {
    public void reset() { x = 0; }
}
```

Direct encoding of **ResetCounter**

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

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

we're duplicating code from **new_counter** :(

let's come back to that

Call function with a subtype

```
let rc = new_reset_counter()
inc3 rc (* won't work! wrong arg type *)

let counter__of__reset_counter
(rc : reset_counter) : counter = {
  get = rc.get;
  inc = rc.inc;
}
inc3 (counter__of__reset_counter rc)
```

Explicit coercion

- Use an explicit function call to *coerce* value of subtype into value of supertype
- Wouldn't be needed if OCaml supported *row polymorphism* on records
  - Basic idea: **`{x:int; y:int}`** can be used wherever **`{x:int}`** is expected
  - Problem: efficient implementation

3. INHERITANCE

Duplicated code

- **Problem:** duplicated code between objects
- **Solution:** classes
- What is a *class*?

  Data structure holding methods. Can be:

  - *instantiated* to yield a new object
  - *extended* to yield a new class
- We want to reuse method code when possible

  ...even if the representation of internal state changes

  ...let's parameterize on representation type

Refactor counter

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

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

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

What is a class?

- A function
  - from internal rep of object state
  - to record of methods, all of which use that shared state
- i.e., a way of generating related objects
- *Not* a type!
  - Many languages pun types and classes

Implementing inheritance: Idea

- 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

ResetCounter with inheritance

```
let reset_counter_class =
fun (r:counter_rep) ->
  let super = counter_class r in {
    get = super.get;
    inc = super.inc;
    reset = (fun () -> r.x := 0)
  }

let new_reset_counter () =
  let r = {x=ref 0} in
  reset_counter_class r
```

Implementing inheritance: Code

`reset_counter_class`

- first creates an object of the superclass with the same internal state as its own
- the resulting parent object is bound to **`super`**
- then creates a new object with same internal state
- copies (*inherits*) the implementations of **`get`** and **`inc`** from superclass
- provides its own implementation of new methods

Another subtype of Counter

```
class BackupCounter extends ResetCounter {
    protected int b = 0;
    public void backup() { b = x; }
    public void reset() { x = b; }
}
```

...adds method and a new field

...overrides one method

BackupCounter with inheritance

```
type backup_counter = {
  get : unit -> int;
  inc : unit -> unit;
  reset : unit -> unit;
  backup : unit -> unit
}

type backup_counter_rep = {
  x : int ref;
  b : int ref;
}
```

Class for BackupCounter

```
let backup_counter_class
(r : backup_counter_rep) =
  let super = reset_counter_class
  (counter_rep__of__backup_counter_rep r) in {
    get = super.get;
    inc = super.inc;
    reset = (fun () -> r.x := !(r.b));
    backup = (fun () -> r.b := !(r.x));
  }

let new_backup_counter () =
  let r = {x = ref 0; b = ref 0} in
  backup_counter_class r
```

Upcast

From subclass to superclass:

```
let counter_rep__of__backup_counter_rep
(r : backup_counter_rep) = {
  x = r.x;
}
```

Explicitly coerce representation, thereby forgetting about some fields

(to be continued)

4. DYNAMIC DISPATCH