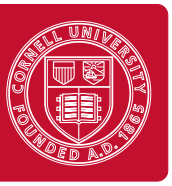
CS 4110 – Programming Languages and Logics
Lecture #22: Objects



Based on material by Stephen Chong, Greg Morrisett, Andrew Myers, George Necula, and Radu Rugina

1 Object-Oriented Design

In the last 25 years, *object-oriented languages* have become the dominant programming paradigm used in industry. An *object* consists of a set of operations coupled with a representation for (typically hidden) data. Objects can encapsulate almost any combination of operations and data, ranging from a single character up to an entire database. There is no standard definition of makes a language "object-oriented", but most languages support dynamic dispatch, encapsulation, subtyping, inheritance, and open recursion [Pierce, Chapter 18].

Dynamic dispatch Dynamic dispatch allows the code executed when a message is sent to an object—e.g., $o.m(x)$—to be determined by run-time values and not (just) by compile-time information such as types. As a result, different objects may respond to the same message in different ways. For example, consider the following Java program:

```
interface Shape { ... void draw() { ... } }
class Circle extends Shape { ... void draw() { ... } }
class Square extends Shape { ... void draw() { ... } }
...
Shape s = ...; //could be a circle a square, or something else.
s.draw();
```

Invoking `s.draw()` could run the code for any of the methods shown in the program (or for any other class that extends `Shape`).

In Java, all methods (except for static methods) are dispatched dynamically. In C++, only virtual members are dispatched dynamically. Note that dynamic dispatch is not the same as overloading, which is usually resolved using the static types of the arguments to the function being called.

Encapsulation Encapsulation allows an object to hide the representations of certain internal data structures. For example, Java programmers often keep instance variables private and write public methods for accessing and modifying the data stored in those variables.

```
class Circle extends Shape {
  private Point center;
  private int radius;
  ...
  public Point getX() { return center.x }
  public Point getY() { return center.y }
}
```

the coordinates of the center of the circle can only be obtained by invoking the `getX` and `getY` methods. The result is that all interactions with the object must be performed by invoking the methods exposed in its public interface and not by directly manipulating its instance variables.

Subtyping Another characteristic feature of object-oriented languages is subtyping. Subtyping fits naturally with object-oriented languages because (ignoring languages such as Java that allow certain objects to manipulate instance variables directly) the only way to interact with an object is to invoke a method. As a result, an object that supports the same methods as another object can be used wherever the second is expected. For example, if we write a method that takes an object of type `Shape` above as a parameter, it is safe to pass `Circle`, `Square`, or any other subtype of `Shape`, because they each support the methods listed in the `Shape` interface.

Inheritance To avoid writing the same code twice, it is often useful to be able to reuse the definition of one kind of object to define another kind of object. In class-based languages, inheritance is often supported through subclassing. For example, in the following Java program,

```
class A {
  public int f(...) { ... g(...) ... }
  public bool g(...) { ... }
}

class B extends A {
  public bool g(...) { ... }
}
...
new B.f(...)
```

`B` inherits the `f` method of its superclass `A`.

One way to implement inheritance is by duplicating code but this wastes space. Most languages introduce a level of indirection instead so that the code compiled for the object being inherited from can be used directly by the object doing the inheriting.

Note that inheritance is different than subtyping: subtyping is a relation on types while inheritance is a relation on implementations. These two notions are conflated in some languages like Java but kept separate in languages like C++ (which allows a "private base class") as well as in languages that are not based on classes.

Open recursion Finally, many object-oriented languages allow objects to invoke their own methods using the special keyword `this` (or `self`). Implementing `this` in the presence of inheritance requires deferring the binding of `this` until the object is actually created. We will see an example of this in the next section.

2 Object encodings

We've just surveyed the main features in object-oriented languages informally. How different are they to the features and type systems that we've already studied in this course? It turns out they are quite similar! Just using references and records, it is possible to encode the key features of object-oriented languages.

2.1 Simple Record Encoding

Let us start with a simple example, developing a representation for two-dimensional point objects using records and references. Records provide both dynamic lookup and subtyping: given a value v of some record type τ, the expression $v.f$ evaluates to a value that is determined by v not by τ—i.e., dynamic dispatch! Moreover, because the subtyping relation on record types allows extension, code that expects an object to have type τ can be used with a value of any subtype of τ.

Here is a simple example showing how we can encode records using objects. For concreteness, we use OCaml syntax rather than λ-calculus. The notation `(fun x -⟩ e)` denotes a λ-abstraction.

```
type pointRep = { x:int ref;
                  y:int ref }

type point = { movex:int -> unit;
               movey:int -> unit }

let pointClass : pointRep -> point =
  (fun (r:pointRep) ->
    { movex = (fun d -> r.x := !(r.x) + d);
      movey = (fun d -> r.y := !(r.x) + d) })

let newPoint : int -> int -> point =
  (fun (x:int) ->
    (fun (y:int) ->
      pointClass { x=ref x; y = ref y }))
```

The `pointRep` type defines the representation for the object's instance variables—a record with a mutable reference for each field. The `pointClass` function takes a record with this type and builds an object—a record with functions `movex` and `movey`, which translate the point horizontally and vertically. The constructor `newPoint` takes two integers, `x` and `y`, and uses `pointClass` to build an object whose fields are initialized to those coordinates.

2.2 Inheritance

Just as in standard object-oriented languages, we can extend our two-dimensional point with an extra coordinate by defining a subclass that inherits the methods of its superclass.

```
type point3D = { movex:int -> unit;
                 movey:int -> unit;
                 movez:int -> unit }

let point3DClass : point3DRep -> point3D =
  (fun (r:point3DRep) ->
    let super = pointClass r in
    { movex = super.movex;
      movey = super.movey;
      movez = (fun d -> r.z := !(r.x) + d) } )

let newPoint3D : int -> int -> int -> point3D =
  (fun (x:int) ->
```

```
    (fun (y:int) ->
      (fun (z:int) ->
        point3DClass { x=ref x; y = ref y; z = ref z })))
```

The most interesting part of this program is the `point3DClass` function. It takes an argument of type `point3DRep` and uses `pointClass` to build a `point` object `super`. It fills in the `movex` and `movey` methods for the object being constructed with the corresponding fields from `super`—i.e., it inherits those methods from the superclass—and defines the new method `movez` directly. Note that we can pass a record of type `point3DRep` to `pointClass` because `point3DRep` is a subtype of `pointRep`.

2.3 Self

Adding support for `self` is a bit trickier because we need `self` to be bound late. Here is an example that illustrates one possible implementation technique:

```
type altPointRep = { x:int ref;
                     y:int ref }

type altPoint = { movex:int -> unit;
                  movey:int -> unit;
                  move: int -> int -> unit }

let altPointClass : altPointRep -> altPoint ref -> altPoint =
  (fun (r:altPointRep) ->
    (fun (self:altPoint ref) ->
      { movex = (fun d -> r.x := !(r.x) + d);
        movey = (fun d -> r.y := !(r.y) + d);
        move  = (fun dx dy -> (!self.movex) dx; (!self.movey) dy) }))

let dummyAltPoint : altPoint =
  { movex = (fun d -> ());
    movey = (fun d -> ());
    move  = (fun dx dy -> ()) }

let newAltPoint : int -> int -> altPoint =
  (fun (x:int) ->
    (fun (y:int) ->
      let r = { x=ref x; y = ref y } in
      let cref = ref dummyAltPoint in
      cref := altPointClass r cref;
      !cref ))
```

For the sake of the example, we have added a method `move` that takes two integers and translates the point both horizontally and vertically. The implementation of `move` invokes the `movex` and `movey` methods from the current object—i.e., `self`.

To make `self` work as expected, we use a trick similar to the one we used to implement recursive definitions in our λ-calculus interpreter. Compared to our previous object encodings there are two key changes. First, the `newAltPointClass` now takes the `self` reference as an explicit parameter. This parameter is filled in with the actual object when it is constructed. Second, the

`newAltPoint` constructor "ties the recursive knot" by allocating a reference cell for the object—filled in initially with a dummy value—and then "back-patching" the reference with the actual object returned by the class.

There is a small problem with this encoding of self: the `self` parameter to `altPointClass` has type `altPoint ref` and references have an *invariant* subtyping rule. As a result, the type system will not allow us to pass a reference to an object generated by a subclass. However, as we do not assign to `self`, it would be safe to use a *covariant* subtyping rule. See Pierce, Chapter 18 for details on how this issue can be resolved.

2.4 Encapsulation

The simple object encoding we have developed in this section already gives us basic encapsulation. After we build an object, the instance variables are totally hidden—we can only manipulate them using object's methods. More complicated forms of abstraction and information hiding can be obtained using existential types. For the details of how records and existential types can be combined to encode objects: see "Comparing Object Encodings", by Bruce, Cardelli, and Pierce, Information and Computation 155(1/2):108–133, 1999.