CS51 **Spring 2024**

Code Review 8: Object-Oriented Programming

1 Terminology and Syntax

Objected-Oriented Programming (OOP) is a programming paradigm where we organize code based on **objects**, which contain **fields** and **methods**. **Please review Section 18.1 and 18.2 of the textbook and the notes in the Lab 16 solutions** for discussion about the motivation behind object-oriented programming.

Definition 1.1 (OOP Terminology)**. Objects** contain **data** and **code**. **Data** is stored using **instance variables,** and **methods** are code that provides the functionality of an object.

A **class** specifies how to create an object, and a new object is created by **instantiating** a class. The **class interface** specifies which values and methods an object provides.

When we use a method, we **invoke** the method.

The following table summarizes OOP syntax that you'll need for this course:

Concept	*Syntax*
Class interfaces	`class type` ⟨*interfacename*⟩ `= ...`
Class definition	`class` ⟨*classname*⟩ ⟨*args*⟩ `= ...`
Object definition	`object ... end`
Instance variables	`val (mutable)` ⟨*varname*⟩ `= ...`
Methods	`method` ⟨*methodname*⟩ ⟨*args*⟩ `= ...`
Instance variable update	`... <- ...`
Instantiating classes	`new` ⟨*classname*⟩ ⟨*args*⟩
Invoking methods	⟨*object*⟩`#`⟨*methodname*⟩ ⟨*args*⟩

Table 18.2: Syntactic extensions in OCaml supporting object-oriented programming.

Figure 1: OOP Syntax

Example 1.2 (Circle Class)**.** Let's define a class interface `display_elt`:

Listing 1: display_elt

```
type point = {x : int; y : int} ;;
class type display_elt =
  object
    method draw : unit
    method set_pos : point -> unit
    method get_pos : point
    method area : int
  end ;;
```

Here, we specify that classes satisfying the `display_elt` have three methods, with the types described above.

Here's how we can define a class satisfying the `display_elt` interface.

Listing 2: circle

```
class circle (p : point) (r : int) : display_elt =
  object
    val mutable pos = p
    method draw = G.fill_circle pos.x pos.y r
    method set_pos p = pos <- p
    method get_pos = pos
    method area = Float.pi * r * r
  end ;;
```

Class definitions can take in *arguments*, used to **instantiate** a class object. Note that the `pos` variable isn't accessible to users outside the class definition (however, we could make it accessible by including it in the class interface).

Here's, how we can create a new `circle` object.

Listing 3: Creating Objects

```
let c : display_elt = new circle {x = 50; y = 100} 20 ;;

c#set_pos {x = 100; y = 50} ;;

c#draw ;;
```

We can use the `new` keyword to create objects of a class. Note that class interface names can be used as types. We invoke methods by using the `#` symbol.

Remark 1.3 (Object Types vs Class Types)**.** In `OCaml`, when we create objects, **they are values**. In the example above, we created an object `c` that can be used like any other value in an expression. Since all **values have types**, **class objects have types**.

`OCaml` distinguishes between **class types** and **object types**. See the discussion here under "Class Types vs Just Types". Objects types refer to the types of the actual `OCaml` values when we create a class, while `class types` refer to the class interface. For instance, when we define the **class type** `display_elt`, we describe the methods provided by classes satisfying the interface. `OCaml` also implicitly creates an **object type** called `display_elt`.

2 Inheritance

Definition 2.1 (Class Inheritance)**.** A class can **inherit** behavior from another class.

The **subclass** inherits behavior from its **superclass**.

In `OCaml`, we can use the `inherit` keyword inside the subclass definition.

The `inherit` specification **works as if all the contents of the superclass definition were copied into the subclass definition**.

Example 2.2 (Rectangles)**.** We can define a class `rectangle`, which will be a subclass of `shape`.

Listing 4: Rectangle Class

```
class shape (p : point) : display_elt =
  object
   val mutable pos = p
   val mutable color = G.black
   method set_pos p = pos <- p
   method get_pos = pos
   method set_color c = color <- c
   method get_color = color
   method draw = failwtih "to be implemented in subclass"
   method area = failwith "to be implemented in subclass"
  end ;;


class rect (p : point) (w : int) (h : int) : display_elt =
  object (this)
    inherit shape p as super
    method! draw = G.fill_square pos.x pos.y w h
    method! area = w * h
  end ;;
```

Note that when we use `inherit`, we also inherit methods and values not specified in the `display_elt` class interface (i.e. instance variables `pos` and `color` and methods `set_color` and `get_color`.

We can invoke methods from the super class in the subclass definition by naming the superclass `super` (note we can use any name we want), and invoking methods by doing something like `super#method_name`.

We can also **bind the object itself to a variable**, as here `this` is a variable representing the object itself.

Subclasses can also **override** methods from the superclass using the `method!` syntax to indicate that the method is overridden.

We recommend you carefully review Labs 16 and 17 to make sure you're comfortable with the OOP syntax.

3 Subtyping

Definition 3.1 (Subtype)**.** We say that datatype α is a **subtype** of datatype β if a value of type α can be used in any context in which a value of type β is used. β is called a **supertype**.

Although the notion of subtyping can be generalized to any types, in this class, we will restrict our focus to object types or types that contain object types (e.g. `display_elt list`, `display_elt option`).

In terms of objects, we say that an object type o_1 is a **subtype** of object type o_2 if $o1$ has an at least as wide as an interface as o_2 (i.e. o_1 has all the same methods as o_2 and potentially more). **Note two methods are the same** if they have the **same name** and **same type**.

Example 3.2 (Drawable)**.** Suppose we defined the following class interface:

Listing 5: Drawable Class Interface

```
class type drawable =
  object
    method draw : unit
  end ;;
```

`display_elt` is a subtype of `drawable`, as `display_elt` has a wider interface.

Definition 3.3 (Coercion Operator)**.** The `OCaml` interpreter doesn't infer the subtype relation, meaning it cannot infer whether type α is a subtype of type β, which makes sense given that different systems can define the subtyping relation differently. `OCaml` essentially allows you, the programmer, to define what types are subtypes of other types.

In `OCaml`, we can say that an expression `e` of type `t1` can be used to a value of type `t2` using the `:>` operator. We write `e :> t2` or `e : t1 :> t2 .`

Example 3.4 (Subtyping in `OCaml`)**.** Suppose we define a function draw that takes in a `drawable` and simply draws that shape.

Listing 6: Draw Example

```
let draw_shape (d : drawable) : unit =
  d#draw ;;

let c : display_elt = new circle {x = 50; y = 100} 20 ;;

draw_shape (c : display_elt :> drawable) ;;

(* Could also do draw_shape (c :> drawable) ;; *)
```

Here, we tell `OCaml` that `display_elt` is a subtype of `drawable`, so `c` can be passed in to the `draw_shape` function.

4 Practice Problems

Problem 4.1 (Object Oriented Counters)**.**

1. See section 18.6 in the textbook

Problem 4.2 (Social Network)**.** "ConnectU" is a social media site for college students. Suppose we want to define a `user` class.

1. Define a class interface for a user `user` that includes the following methods:
   (a) Get a user's username
   (b) Get a user's student id (a string)
   (c) Add a friend to a user's list of friends; note that friendships are mutual, so if Alice is a friend of Bob, then Bob is a friend of Alice.
   (d) Get a user's friend list
   (e) Add a post (a string) to a user's list of posts
   (f) Remove a post
   (g) Retrieve all the user's posts
2. Define an implementation that satisfies the class interface which takes in a `username` and an `id`.
3. Define a class interface called `student` which has the same functionality as the `user` class but an additional method which outputs the school a student attends.
4. Define an implementation that satisfies the `student` interface which takes in a `username`, `id`, and the `school` the student attends.
5. Define a function `form_friend_group` that takes in a list of `users`, and for each user in the list, sets their friends to be everyone else in the list.
6. Define a list of four `students` and use `form_friend_group` on the `student` list.

Problem 4.3 (Polynomial)**.** In this problem, you will define classes that work with (univariate) polynomials.

A polynomial is an expression consisting of coefficients and variables.

We define a class interface `polynomial_type`.

Listing 7: polynomial_type

```
1
2 class type polynomial_type =
3   object
4     method get_coeffcients : float list (* int list returns the
          coefficients of the polynomial in order from lowest degree to
          highest degree. For example, [5, 3, 2] would represent the
          polynomial 5 + 3x + 2x^2. *)
5
6     method evalulate : float -> float (* evalulates a polynomial f(x)
          *)
```

```
7
8
9       method solve : float -> float list option (* solve c returns real
            solutions to x when a polynomial equals c. For example if a
            polynomial is 5 + 3x + 2x^2, polynomial_object#solve 10 returns
            a list of solutions to 5 + 3x + 2x^2 = 10  *)
10    end
```

Now we define the class definition for a `polynomial class`.

Listing 8: polynomial

```
1
2   class polynomial (coefficents : float list) : polynomial_type =
3     object
4
5       method get_coeffcients : float list
6
7       method evalulate (x : float) : float =
8         failwith "to be implemented"
9
10      method solve (c: float) : float list option =
11        None
12    end
```

1. Implement the `evaluate` method for the polynomial class. For example, for a polynomial $5 + 3x + 2x^2$ with coefficients $[5.0, 3.0, 2.0]$, the evaluate method called on 5 should return $5 + 3(5) + 2(5)^2 = 70$.

2. Implement a `linear_polynomial` class satisfying the `polynomial_type` class interface. A linear polynomial is an expression that has **at most two coefficients**. For example, $5x + 2$ (with coefficients as $[2.0, 5.0]$), $3x$ ($[0., 3.]$), and 11 ($[11.]$) are all polynomial functions. *Hint*: You may define a type so that users can only define at most two coefficients.

   For a constant polynomial c, `solve a` should return `[Float.infinity]` if $c = a$, `None` otherwise.

3. Implement a `quadratic_polynomial` class satisfying the `polynomial_type` class interface.

   Quadratic polynomials have at most three coefficients. To solve quadratic polynomials, we can use the quadratic formula.

   The solutions to a quadratic $ax^2 + bx + c = 0$ is given by the formula: $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

   Remember to properly handle cases when there are no real solutions or infinitely many solutions.