4 Object-Oriented ADT Implementations

Last time, we talked about implementing ADTs, and we have looked at sample implementations in Scheme (as well as in ACL2).

Those implementations, while they do provide all the operations in the signature and satisfy the specification, are not perfect, however. The main problem is that they expose the representation of the elements of the ADT. This makes it too easy for clients to rely on representation details, such as the "bad client" we saw last time. A client relying on implementation details means that we, as implementers of the ADT, cannot just change the implementation without breaking client code.

Many languages have facilities to prevent the exposure of data representation, thereby ensuring that clients cannot take advantage of it.

One approach, called *signature ascription*, is to "wall off" the data representation and the operations that operate on it, and by only exposing the operations themselves.

Another approach, which is similar in its results but philosophically different, is to push the operations inside the data representation (and hide that data representatino). Because the operations live inside the data representation, they know the data representation (which they need in order to actually implement their required functionality), but clients cannot access that representation. This approach is the basis for object orientation.

We will be using Scala as an implementation language for this course. Scala is similar to Java, and you can think of it as the next step in the evolution of Java. Some remarks on the language: Scala is a class-based object-oriented language. "Object-oriented" here means that every value in the language is an object. "Class-based" means that objects are created from classes. A class is essentially a template, a description of how objects of the class are created.

It helps to think of a program has having two parts. One part, the part that correspond to the source code, is static. (Static means non-moving, which we take to mean non-executing). It represents what information about the program we have before anything executes. In Scala, the only thing we know before a program executes are what classes are defined. Thus, the classes are static. Classes exist, in some sense, even before programs start executing. The other part is the dynamic part, which corresponds to program execution. During execution, instances of the classes, that is, objects, get created, updated, destroyed. Thus, objects are dynamic.

You should probably be aware that not every object-oriented language is class-based. Self,

for example, has no concept of class, but still has objects. Scala is also statically typed — types are associated with variables, and before a program is run those types are checked, to make sure that values of the right kind are stored in variables, or passed to methods. Not every object-oriented language is statically typed. Smalltalk, for example, checks types dynamically, like Scheme does. We'll talk more about types next lecture.

4.1 Object-Oriented Signatures and Specifications

To help us devise implementation for ADTs in object-oriented languages, we consider a slightly different way of writing signatures and specifications. Recall the POINT ADT signature from last time (except I've replaced all `Float` with `Double` for convenience):

```
CREATORS
  cartesian : (Double, Double) -> Point
  polar :     (Double, Double) -> Point

OPERATIONS
  xCoord :              (Point) -> Double
  yCoord :              (Point) -> Double
  angleWithXAxis :      (Point) -> Double
  distanceFromOrigin :  (Point) -> Double
  distance :            (Point, Point) -> Double
  move :                (Point, Double, Double) -> Point
  add :                 (Point, Point) -> Point
  rotate :              (Point, Double) -> Point

  isEqual :             (Point, Point) -> Boolean
  isOrigin :            (Point) -> Boolean
```

An object-oriented signature, to a first approximation, consider that operations (but not the creators), which must take at least one argument of the type of the ADT[1] take that value on which they act as an *implicit* argument, as opposed to an *explicit* argument that appears in the argument list. Implicit arguments are meant to capture the idea that the operations live inside an object (an element of the ADT) and therefore have access to that element as an implicit argument. For instance, if `p` and `q` are `Point`s, while we would write `add(p,q)` to add `p` and `q` in a conventional language, in an object-oriented setting we would call the `add` operation inside `p`, usually written `p.add(q)`, and `p` here is considered the implicit argument to `add`, while `q` is an explicit argument.

Thus, here is the object-oriented signature for the POINT ADT:

[1] if not, such an operation probably has no business being part of the ADT.

```
CREATORS
  cartesian : (Double, Double) -> Point
  polar :     (Double, Double) -> Point

OPERATIONS
  xCoord :             () -> Double
  yCoord :             () -> Double
  angleWithXAxis :     () -> Double
  distanceFromOrigin : () -> Double
  distance :           (Point) -> Double
  move :               (Double, Double) -> Point
  add :                (Point) -> Point
  rotate :             (Double) -> Point

  isEqual :            (Point) -> Boolean
  isOrigin :           () -> Boolean
```

Let's change how those operations are used, to understand exactly where the implicit argument to operations is coming from. Creators are invoked as before, e.g., `cartesian(10,20)`. Operations, on the other hand, are invoked on an expression yielding a `Point` value, e.g., `p.xCoord()`, where `p` is a `Point` value, or `p.rotate(2.0).move(3,4)`, again where `p` is a `Point` value.

We can easily adapt the specification of points to this new way of invoking operations — here is the specification from Lecture 2, adapted to the signature above, and presented in such a way that we have exactly two equations per operation, once for each creator.

$$\texttt{cartesian}(x,y)\texttt{.xCoord()} = x$$
$$\texttt{polar}(r,\theta)\texttt{.xCoord()} = r\cos\theta$$

$$\texttt{cartesian}(x,y)\texttt{.yCoord()} = y$$
$$\texttt{polar}(r,\theta)\texttt{.yCoord()} = r\sin\theta$$

$$\texttt{cartesian}(x,y)\texttt{.distanceFromOrigin()} = \sqrt{x^2+y^2}$$
$$\texttt{polar}(r,\theta)\texttt{.distanceFromOrigin()} = r$$

$$\texttt{cartesian}(x,y)\texttt{.angleWithXAxis()} = \begin{cases} \tan^{-1}(y/x) & \text{if } x \neq 0 \\ \pi/2 & \text{if } y \geq 0 \text{ and } x = 0 \\ -\pi/2 & \text{if } y < 0 \text{ and } x = 0 \end{cases}$$
$$\texttt{polar}(r,\theta)\texttt{.angleWithXAxis()} = \theta$$

$$\texttt{cartesian}(x,y)\texttt{.distance}(q) = \sqrt{(x - q\texttt{.xCoord()})^2 + (y - q\texttt{.yCoord()})^2}$$

$$\texttt{polar}(r,\theta)\texttt{.distance}(q) = \sqrt{(p\texttt{.xCoord()} - q\texttt{.xCoord()})^2 + (p\texttt{.yCoord()} - q\texttt{.yCoord()})^2}$$

$$\texttt{cartesian}(x,y)\texttt{.move}(dx,dy) = \texttt{cartesian}(x+dx, y+dy)$$
$$\texttt{polar}(r,\theta)\texttt{.move}(dx,dy) = \texttt{cartesian}(r\cos\theta + dx, r\sin\theta + dy)$$

$$\texttt{cartesian}(x,y)\texttt{.add}(q) = \texttt{cartesian}(x + q\texttt{.xCoord()}, y + q\texttt{.yCoord()})$$
$$\texttt{polar}(r,\theta)\texttt{.add}(q) = \texttt{cartesian}(r\cos\theta + q\texttt{.xCoord()}, r\sin\theta + q\texttt{.yCoord()})$$

$$\texttt{cartesian}(x,y)\texttt{.rotate}(\rho) = \texttt{cartesian}(x\cos\rho - y\sin\rho, x\sin\rho + y\cos\rho)$$
$$\texttt{polar}(r,\theta)\texttt{.rotate}(\rho) = \texttt{polar}(r, \theta+\rho)$$

$$\texttt{cartesian}(x,y)\texttt{.isEqual}(q) = \begin{cases} \textit{true} & \text{if } x = q\texttt{.xCoord()} \text{ and } y = q\texttt{.yCoord()} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{polar}(r,\theta)\texttt{.isEqual}(q) = \begin{cases} \textit{true} & \text{if } r = q\texttt{.distanceFromOrigin()} \text{ and} \\ & \quad \theta \equiv q\texttt{.angleWithXAxis()} \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{cartesian}(x,y)\texttt{.isOrigin()} = \begin{cases} \textit{true} & \text{if } x = 0 \text{ and } y = 0 \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\texttt{polar}(r,\theta)\texttt{.isOrigin()} = \begin{cases} \textit{true} & \text{if } r = 0 \\ \textit{false} & \text{otherwise} \end{cases}$$

(Where $\equiv$ for angles is defined in Lecture 2.)

4.2 Implementation in Scala

Let's implement the above signature, then. The one decision we have to make, at this point, is how to represent points. The same decision was had to make when we were thinking of implementing points in Scheme last time. Just like in that case, we have two natural representation for points — as a pair of cartesian coordinates, or as a triple where the first element of the triple is a flag indicating whether the next two elements are the coordinates of the point in cartesian coordinates or in polar coordinates. Let's do the first representation first. We define a class `Point` to hold the representation of points. The definition of the class specifies the values that must be supplied to the class to construct an instance of the class:

```
class Point (xpos : Double, ypos : Double) {

  // OPERATIONS
```

```
  ...

  // CANONICAL METHODS

  ...

}
```

To construct an instance of a point, we will use an expression `new Point(10.2,20.4)` or somesuch, where `xpos` will be bound to `10.2` in the newly created instance, and `ypos` will be bound to `20.4` in the newly created instance. Both `xpos` and `ypos` are available as fields in the instance created. (They are not accessible from outside the instance, though — they are *private*.)

Let's fill in the body of this class.

There are several rules that we will follow when writing classes in this course. Four, to be precise, to be introduced throughout this example. Here is the first one:

(1) The only methods in the class that we should be able to invoke are those corresponding to the operations in the signature, as well as the canonical methods.

Canonical methods will be defined next lecture. Now, the class can define other methods, we just have to make sure they are not accessible from outside the class.

Scala lets us restrict accessibility to methods (and to fields) using the `private` keyword. (Much more can and will be said about `private`.) By default, methods and fields without a qualifier are public.

We will not use fields much in this course. (Most of the time, they will be hidden as arguments to the class, as we did above for `xpos` and `ypos`.) When we do use fields, though, they will always be private.

(2) All fields are private.

Fields are not part of the signature, so the spirit of rule (1) says that they should indeed be private. This is not a big restrictions, as we can always use methods (if the signature tells us to) to read and update fields. Mostly, this is to make sure that the rest of the code does not depend on there being a particular field in the object, so that we can, for instance, change the representation of an ADT without worrying about breaking code elsewhere in our program.

Let's implement the operations:

```
class Point (xpos : Double, ypos : Double) {
```

```
  // OPERATIONS

  def xCoord ():Double = xpos

  def yCoord ():Double = ypos

  def distanceFromOrigin ():Double = math.sqrt(xpos*xpos+ypos*ypos)

  def angleWithXAxis ():Double = math.atan2(ypos,xpos)

  def distance (q:Point):Double = math.sqrt(math.pow(xpos-q.xCoord(),2)+
                                            math.pow(ypos-q.yCoord(),2))

  def move (dx:Double, dy:Double):Point = new Point(xpos+dx,ypos+dy)

  def add (q:Point):Point = this.move(q.xCoord(),q.yCoord())

  def rotate (t:Double):Point =
     new Point(xpos*math.cos(t)-ypos*math.sin(t),
               xpos*math.sin(t)+ypos*math.cos(t))

  def isEqual (q:Point):Boolean = (xpos == q.xCoord()) &&
                                  (ypos == q.yCoord())

  def isOrigin ():Boolean = (xpos == 0) && (ypos == 0)

  // CANONICAL METHODS

  ...
}
```

This is all completely straightforward. A method is defined using

`def` *name* `(`*argname*`:`*argtype*`,...):`*resulttype* `=` *body*

Here, *body* is an expression that returns a value of type *resulttype*. We can use the keyword `this` to stand for the implicit argument in any method body, as in Java. I will often use `this` explicitly to emphasize when I'm invoking a method on the same object.

Classes can define not only methods but fields as well. In the above class, `xpos` and `ypos` are fields, albeit implicit fields, appearing only in the signature of the class. Explicit fields can be defined using

`val` *name* `:` *type* `=` *initvalue*

or

`var` *name* `:` *type* `=` *initvalue*

A `val` field is a field that cannot be updated, while a `var` field can be updated. For us, for the time being, fields are never updated. This is important enough that I will make it a rule that we will only break towards the end of the course:

(3) Fields, once initialized, are never updated.

In combination with rule (2) that makes every field private, this makes every instance of the class *immutable*—once created, it cannot be changed. Immutable instances have a host of advantages: the code is easier to reason about, it is easier to replace the code or debug it, etc. As we will see when we look at mutation, understanding what actually happens when a field is updated can get very tricky when a program uses all the features of Scala. Because of this, and other reasons that we will return to in the course of the semester, we will restrict our attention to immutable instances.

Okay, so we have class `Point`, that lets us create a representation of a point in cartesian coordinates. The only thing missing are the creators. Now, we cannot put the creators within the representation of points, because, intuitively, when we invoke the creators, we may not have any point around. So what do we do with them?

Really, we would like to define two functions `cartesian` and `polar` that live outside the the `Point` class, and that look like:

```
def cartesian (x:Double,y:Double):Point =
   new Point(x,y)

def polar (r:Double,theta:Double):Point =
   if (r<0)
     throw new Error("r negative")
   else
     new Point(r*math.cos(theta),r*math.sin(theta))
```

But Scala doesn't let us define "free-floating" functions like that. They need to live inside something. It seems silly to define a class *just* to have those two functions live inside it, so we'll use a special kind of class called a *singleton class* — also known as a *module* — that is, a class that has only one instance, and that instance is created automatically for you when the program starts. Here is a possible definition:

```
Object Creators {

  def cartesian (x:Double,y:Double):Point =
     new Point(x,y)
```

```
   def polar (r:Double,theta:Double):Point =
      if (r<0)
        throw new Error("r negative")
      else
        new Point(r*math.cos(theta),r*math.sin(theta))
}
```

This creates an instance (called `Creators`) of the class `Creators`, and doesn't let you define new instances of that class. To invoke methods in the `Creators` module, you would call them like you would any other methods, that is, as `Creators.cartesian(10.2,20.4)`, or `Creators.polar(10.0,math.Pi/2)`.

Calling the module `Creators` is not very mnemonic, especially if we have other ADTs around with their own creators. So we shall define a module called `Point` in which the creators for the POINT ADT live — that the module has the same name as the class used for representation of a point is something that Scala allows.[2]

This gives us our fourth rule:

(4) Creators live in a module (singleton class) of the same name as the ADT.

Here is the code for the `Point` module and the `Point` class, which we can put in a file `Point.scala`:

```
object Point {

  def cartesian (x:Double,y:Double):Point =
     new Point(x,y)

  def polar (r:Double,theta:Double):Point =
     if (r<0)
       throw new Error("r negative")
     else
       new Point(r*math.cos(theta),r*math.sin(theta))
}

class Point (xpos : Double, ypos : Double) {

  // OPERATIONS

  def xCoord ():Double = xpos
```

[2]In that situation, the resulting module is sometimes called a *companion object to the class*. The details are actually not that important.

```
  def yCoord ():Double = ypos

  def distanceFromOrigin ():Double = math.sqrt(xpos*xpos+ypos*ypos)

  def angleWithXAxis ():Double = math.atan2(ypos,xpos)

  def distance (q:Point):Double = math.sqrt(math.pow(xpos-q.xCoord(),2)+
                                            math.pow(ypos-q.yCoord(),2))

  def move (dx:Double, dy:Double):Point = new Point(xpos+dx,ypos+dy)

  def add (q:Point):Point = this.move(q.xCoord(),q.yCoord())

  def rotate (t:Double):Point =
     new Point(xpos*math.cos(t)-ypos*math.sin(t),
               xpos*math.sin(t)+ypos*math.cos(t))

  def isEqual (q:Point):Boolean = (xpos == q.xCoord()) &&
                                  (ypos == q.yCoord())

  def isOrigin ():Boolean = (xpos == 0) && (ypos == 0)

  // CANONICAL METHODS

  ...

}
```

We are almost done. There's just one bit left to do, namely taking care of the last `...` at the bottom there, and how to run code.