2 ADTs and Algebraic Specifications

One of the first questions we are faced with when trying to design a piece of software is "What is the data that the program must manipulate? What are the objects? For example, when designing a game or a simulation, likely data includes artifacts such as spaceships, planets, and stars. For an animation package, it may include pictures, characters, backgrounds. These are all fairly concrete data. Other forms of data are more abstract — for instance, trajectories for animated characters, behaviors for interactive animations, or strategies for computer-controlled game characters.

For the purpose of our discussion today, let's focus on a very simple kind of ADT, namely points in two dimensions, or points in the plane. (If you want, think of them as two-dimensional coordinates.)

So how do we go about thinking about points? When designing software, it's best to keep an open mind as to the exact form the software will take. We can simply assume that a point is an object without committing to anything else. In particular, we will not want to commit to a particular way of representing a point. Let's embody this into a principle.

> **The Principle of Least Commitment**: Don't commit yourself any more or sooner than necessary.

So points are objects of some kind. It does not really matter what objects are. What matters about objects is how they behave. So let's ask the question: how should points behave?

That's a vague question. Let's refine it somewhat, and look for something specific. Behaviors are induced when objects are acted upon. So how do we want to act on points? In other words, what operations do we want to support on points?

Here are some sensible operations on points: creating new points, moving a point, finding the coordinate of a point, computing the distance between two points, rotating a point about the origin, determining if a point is the origin, plotting a point on a canvas on the screen. You should be easily able to think of others.

Operations on points naturally divide into three kinds of operations. The first kind of operations are those that create a point, which we will call *creators* or sometimes constructors. The second kind of operations are those that extract information from point or transform a point, which we will call *accessors*. When an accessor returns true or false we often call the accessor a *predicate*. The final kind of operations are those that do not extract information or create new points, but rather effect a change in the real world, such as plotting a point on the screen and therefore physically changing the presentation of the screen, which we will call *effectors*. Effectors are those operations that actually *do* something that you as a human being sitting at the computer notice. Printing out information on the screen for you to read is an effector operation.

For now, we will keep the design exceedingly small and simple. In particular, we'll leave out some behaviors and operations that we will need later. But that's okay. Part of the point of object-oriented design and programming is that it makes it easy to add behaviors to objects in the future.

I will also most likely make mistakes, something voluntarily, sometimes not. Between that and needing to add operations later on, we will certainly need to revise our design in the future. Revising a design is must less expensive that revising a program, because a program accumulates all sort of cruft that a design does not have, including implementation choices. Moreover, a good design is much easier to turn into correct code than a bad design. Worse, a buggy design makes it impossible to write correct code. Thus, it pays to get the design right.

The concept of data with an associated set of operations is important enough that we'll give it a name: an *abstract data type* (ADT for short):

> **Abstract Data Type**: An abstract data type is a set of data and a set of operations that can be performed on the data, along with a description of what those operations do.

Let's consider an ADT for points. First, the operations. What operations should we want to support on points? Of course, we need to create points — we want creators. There are several choices possible, and here are mine. I want a creator `cartesian` that creates a point from its cartesian coordinates — that is, distance from the y-axis, and distance from the x-axis. Another creator I want is `polar`, which creates a point from its polar coordinates — that is, its distance from the origin, and the vector from the origin to the point makes with the vector of the positive x-axis.

Being able to create points is the first step. We also want to extract information from points — we want accessors. I will consider operations to extract the x- and y-coordinates of a point, as well as extracting its distance from the origin and the angle it makes with the positive x-axis. I will also have operations to compute the distance between two points, to move a point by a given distance in both the x and y direction, an operation to add two points (in the sense of vector addition), and an operation to rotate a point about the origin. The two predicates I will consider are to check if two points are the same (i.e. describe the same position in the plane), and whether a point is the origin.

2.1 Signature of the Point ADT

An ADT is made up of a signature and a specification. A *signature* consists of the names of the operations together with their type.

For the Point ADT, a reasonable signature would be as follows:

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

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

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

(This ADT assumes that we have types `Float` for floating-point numbers, and `Boolean` for Booleans.)

Notice that the creators all return a `Point` object, as expected, while all the operations take a `Point` object as the first (sometimes only) argument.

A signature describes one aspect of the interface, namely, what shape the operations have, that is, what they expect as arguments and what they return as a result. The signature gives no clue as to how those operations are meant to behave, however. We need to remedy that situation. This is the second part of the definition of ADT.

2.2 Specification of the Point ADT

A *specification* (or spec, for short) is a kind of guarantee (or contract) between clients and implementors.

Clients

- depend on the behavior guaranteed by the spec, and
- promise not to depend on any behavior not guaranteed by the spec.

Implementors

- guarantee that a provided abstraction behaves as specified by the spec, and
- do not guarantee any behavior not covered by the spec.

It is hard to specify how objects behave. Usually, this is done in English, in an informal way. But the resulting specification is often incomplete, incorrect, ambiguous, or confusing. You'll see examples of those very often.

Let me introduce a *formal* way of specifying behavior, as a set of algebraic equations that the operations of the interface must obey. Because of that, we will call it an *algebraic specification*. (There are other ways of specifying behavior, which we may talk about later in the course.)

The basic rule is that there is an equation for every combination of operation and creator, describing how that operation works for an object created using that creator. Convince yourself that once you know that, you can then tell what the result of any operation is on any object of the ADT, since any objcet of the ADT can only have been created using a combination of creators and operations.

Let's do this by looking at every operation one after the other, and for every operation, seeing how to behaves when applied to objects created by the various creators.

Specifying `xCoord` and `yCoord` is pretty straightforward, if you remember your high-school trigonometry:

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

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

Similarly, `distanceFromOrigin` and `angleWithXAxis` are easy:

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

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

Second, the equation for `angleWithXAxis` applied to `cartesian` point is a *conditional* equation, because the result depends on whether the point lies on the y-axis or not.

Distance is easy for cartesian points, not so easy for polar points, and the easiest way to do that is to convert the polar coordinates to the cartesian ones. Note that we have four cases to consider, because there are four different ways of constructing two points:

$$\begin{aligned}
\texttt{distance(cartesian(}x\texttt{,}y\texttt{),cartesian(}x'\texttt{,}y'\texttt{))} &= \sqrt{(x-x')^2+(y-y')^2} \\
\texttt{distance(cartesian(}x\texttt{,}y\texttt{),polar(}r\texttt{,}\theta\texttt{))} &= \sqrt{(x-r\cos\theta)^2+(y-r\sin\theta)^2} \\
\texttt{distance(polar(}r\texttt{,}\theta\texttt{),cartesian(}x\texttt{,}y\texttt{))} &= \sqrt{(x-r\cos\theta)^2+(y-r\sin\theta)^2} \\
\texttt{distance(polar(}r\texttt{,}\theta\texttt{),polar(}r'\texttt{,}\theta'\texttt{))} &= \sqrt{(r\cos\theta-r'\cos\theta')^2+(r\sin\theta-r'\sin\theta')^2}
\end{aligned} \tag{1}$$

This is a bit painful, I admit. There is, in this particular instance, a way to simplify the specification. You could replace the four equations above by the single equation:

$$\texttt{distance(}p\texttt{,}q\texttt{)} = \sqrt{(\texttt{xCoord(}p\texttt{)} - \texttt{xCoord(}q\texttt{)})^2 + (\texttt{yCoord(}p\texttt{)} - \texttt{yCoord(}q\texttt{)})^2} \tag{2}$$

Exercise: *Show that the single equation* (2) *implies the four equations* (1) *in the presence of the rest of the specification. That is, show that any of the equations in* (1) *can be derived from* (2) *and the other equations of the specification — specifically, the equations for* `xCoord` *and* `yCoord`*. For example:*

$$\begin{aligned}
&\texttt{distance(cartesian(}x\texttt{,}y\texttt{),cartesian(}x'\texttt{,}y'\texttt{))} \\
&\quad = \sqrt{\begin{aligned}&(\texttt{xCoord(cartesian(}x\texttt{,}y\texttt{))} - \texttt{xCoord(cartesian(}x'\texttt{,}y'\texttt{))})^2 \\ &\quad + (\texttt{yCoord(cartesian(}x\texttt{,}y\texttt{))} - \texttt{yCoord(cartesian(}x'\texttt{,}y'\texttt{))})^2\end{aligned}} \\
&\quad = \sqrt{(x-x')^2+(y-y')^2}
\end{aligned}$$

where the first equality holds by equation (2)*, taking p to be* `cartesian(`x`,`y`)` *and q to be* `cartesian(`x'`,`y'`)`*, and the second equality holds by the equation for* `xCoord`*.*

Operation `move` is straightforward:

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

Let's look at the first equation and see what it means. You should read it as: if you `move` a point created as `cartesian(`x`,`y`)` by some dx and dy, then the result is that same as if you had called `cartesian(`$x + dx$`,`$y + dy$`)` directly. Makes sense? That's how you're supposed to read those equations.

Operation `add` is equivalently easy, if painful, to specify directly — because, again, there are four cases to consider.

$$\begin{aligned}
\texttt{add(cartesian(}x\texttt{,}y\texttt{),cartesian(}x'\texttt{,}y'\texttt{))} &= \texttt{cartesian(}x+x'\texttt{,}y+y'\texttt{)} \\
\texttt{add(cartesian(}x\texttt{,}y\texttt{),polar(}r\texttt{,}\theta\texttt{))} &= \texttt{cartesian(}x+r\cos\theta\texttt{,}y+r\sin\theta\texttt{)} \\
\texttt{add(polar(}r\texttt{,}\theta\texttt{),cartesian(}x\texttt{,}y\texttt{))} &= \texttt{cartesian(}x+r\cos\theta\texttt{,}y+r\sin\theta\texttt{)} \\
\texttt{add(polar(}r\texttt{,}\theta\texttt{),polar(}r'\texttt{,}\theta'\texttt{))} &= \texttt{cartesian(}r\cos\theta+r'\cos\theta'\texttt{,}r\sin\theta+r'\sin\theta'\texttt{)}
\end{aligned} \tag{3}$$

Again, we could have replaced those four equations by the single equation:

$$\texttt{add(}p\texttt{,}q\texttt{)} = \texttt{cartesian(xCoord(}p\texttt{)+xCoord(}q\texttt{),yCoord(}p\texttt{)+yCoord(}q\texttt{))} \tag{4}$$

or by the following single equation:

$$\texttt{add(}p\texttt{,}q\texttt{)} = \texttt{move(}p\texttt{,xCoord(}q\texttt{),yCoord(}q\texttt{))} \tag{5}$$

Exercise: *Show that either of* (4) *or* (5) *implies all the equations in* (3), *in the presence of the specification for the other operations.*

Rotate is more interesting, because there is an easy way to rotate polar points, and rotation for cartesian points is more intricate:

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

We are left with the two predicates, both straightforward.

$$\texttt{isEqual(cartesian(}x\texttt{,}y\texttt{),cartesian(}x'\texttt{,}y'\texttt{))} = \begin{cases} \mathit{true} & \text{if } x = x' \text{ and } y = y' \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{isEqual(cartesian(}x\texttt{,}y\texttt{),polar(}r\texttt{,}\theta\texttt{))} = \begin{cases} \mathit{true} & \text{if } x = r\cos\theta \text{ and } y = r\sin\theta \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{isEqual(polar(}r\texttt{,}\theta\texttt{),cartesian(}x\texttt{,}y\texttt{))} = \begin{cases} \mathit{true} & \text{if } x = r\cos\theta \text{ and } y = r\sin\theta \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\texttt{isEqual(polar(}r\texttt{,}\theta\texttt{),polar(}r'\texttt{,}\theta'\texttt{))} = \begin{cases} \mathit{true} & \text{if } r = r' \text{ and } \theta \equiv \theta' \\ \mathit{false} & \text{otherwise} \end{cases}$$

(Note that the last equation uses $\equiv$ to compare angles, since angles that differ by multiples of 2π are considered the same angle. So $\theta \equiv \theta'$ holds when there exists an integer n such that $\theta - \theta' = 2n\pi$.)

Exercise: *Come up with a single equation that can replace the four equations for* `isEqual`*, and show that that single equation in fact implies the four equations for* `isEqual`.

Finally, `isOrigin`:

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

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

These equations seem only to specify the behavior of the accessors, but in fact, they describe the interaction between the accessors and the creators, and thereby implicitly also specify the behavior of the creators.

Let me emphasize again: the above specification *describes* how the operations behave, what they do. But I have not said how points are implemented. And I don't care right now. I do not care how the operations do what they claim to do, I am just describing how I want them to behave. Figuring out how to implement objects, how to go from a description of their behavior to actual code that implements that behavior, is a decision that we will resist making for as long as possible, per the Principle of Least Commitment.

Despite this lack of description of how objects are implemented, the specification lets us predict the result of any sequence of creations and operations. For instance, suppose that I wanted to know the distance between the result of rotating the point $(3, 4)$ in cartesian coordinates by $\pi/2$ about the origin and the result of moving point $(1, \pi/4)$ in polar coordinates upwards by 7 and leftwards by 11—in other words, I want to know the result of:

$$\texttt{distance(rotate(cartesian(3,4),}\pi/2\texttt{),move(polar(1,}\pi/4\texttt{),7,}-11\texttt{))}$$

Well, I can use the equations in the specification to replace equals by equals and simplify the above expression, just like you would do in algebra. This is where the name algebraic specification comes from, by the way — it lets us reason algebraically, that is, by simplification. Here is one possible derivation. See if you can spot the equations I used at every step:

$$\begin{aligned}
&\texttt{distance(rotate(cartesian(3,4),}\pi/2\texttt{),move(polar(1,}\pi/4\texttt{),7,}-11\texttt{))}\\
&\quad= \texttt{distance(rotate(cartesian(3,4),}\pi/2\texttt{),cartesian(}1\cos(\pi/4)+7\texttt{,}1\sin(\pi/4)-11\texttt{))}\\
&\quad= \texttt{distance(rotate(cartesian(3,4),}\pi/2\texttt{),cartesian(}(1/\sqrt{2})+7\texttt{,}(1/\sqrt{2})-11\texttt{))}\\
&\quad= \texttt{distance(cartesian(}3\cos(\pi/2)-4\sin(\pi/2)\texttt{,}3\sin(\pi/2)+4\cos(\pi/2)\texttt{),}\\
&\qquad\qquad\texttt{cartesian(}(1/\sqrt{2})+7\texttt{,}(1/\sqrt{2})-11\texttt{))}\\
&\quad= \texttt{distance(cartesian(}-4\texttt{,}3\texttt{),cartesian(}(1/\sqrt{2})+7\texttt{,}(1/\sqrt{2})-11\texttt{))}\\
&\quad= \sqrt{(-4-(1/\sqrt{2})-7)^2+(3-(1/\sqrt{2})+11)^2}\\
&\quad= \sqrt{(-11-(1/\sqrt{2}))^2+(14-(1/\sqrt{2}))^2}\\
&\quad= \sqrt{318-3\sqrt{2}}\\
&\quad\approx 17.7131973
\end{aligned}$$

And the point of this is: we can compute the result *without having ever said a word about how points are implemented*!