9 Static Versus Dynamic Types

Subtpying is a great way to enable client-side reuse, requiring a client to write a single function that can work with argument of several types, namely all the subtypes of the type that has been declared. But subtyping brings a whole range of issues to the fore. Not the least of which the need to distinguish between *static types* (or compile-time types) and *dynamic types* (or run-time types).

Types are associated with identifiers in your program — such identifiers include field declarations such as `x` in `val x : T= ...` and parameter names in methods, such as `y` in `def m (y:T):U = ...`. The type of those identifiers describe the kind of values they can be bound to. (Once you have the static type of the identifiers of your program, you can determine what is the static type of every expression in your program. For instance if m is a method in some class T declared as

`def` m `(x:Int):Int = ...`

then an expression such as `exp.`m`(10)` has static type `Int` if `exp` has static type T, since m is a method declared to return a value of type `Int` if given a value of type `Int`.)

The *static type* of an identifier is the type that the identifier is declared to have in the source code. For instance, if we declare `val x:Point = ...`, then identifier `x` has static type `Point`.

In contrast, the *dynamic type* of an identifier only makes sense when the program is executing and that identifier has actually been bound to a value, and is the actual type of the value that the identifier is bound to. (Note that an identifier may be bound to different values at different points in a program execution — for instance, the parameter to a method will be bound to different values when the method is called with different arguments at different points in the program execution.) The best way to think about it is to use the following execution model for Scala: during program execution, doing a `new C(...)` creates a structure (just like a *struct* in Scheme) whose fields are the field of class `C` along with an extra field holding the name of the class `C`, which is of course the actual type of the newly-created structure. When that structure is bound to an identifier, the dynamic type of that identifier is the type in the structure. (Just like for static types, every expression in a program, during execution, has a dynamic type, which is the type of the value that the expression evaluates to during execution.)

To show the difference between static types and dynamic types, recall the `rotateAroundPoint()` method for points, declared to have signature

```
def rotateAroundPoint (p:Point, a:angle, c:Point):Point
```

The static type of both `p` and `q` is `Point`. The static return type of the method is also `Point`.

In contrast, the dynamic type of `p` and `q` only make sense when the method is called at execution. For instance, writing

```
val p1:Point = Point.cartesian(0,1)
val cq1:CPoint = CPoint.cartesian(10,20,Color.red())
val result:Point = rotatePointAround(p1,math.Pi/2,cq1)
```

has the following effect: `p1` is bound to an actual point (intuitively a structure with fields containing coordinates 0 and 1), `cq1` is bound to an actual color pointed (intuitively a structure with fields containing coordinates 10 and 20 and a color field containing *red*), and the call to `rotateAroundPoint()` ensure that in the body of `rotateAroundPoint()` parameter `p` is bound to the actual point $(0, 1)$ — and thus the dynamic type of `p` is a `Point` — while parameter `q` is bound to the actual color point $(10, 20, \textit{red})$ — and thus the dynamic type of `q` is a `CPoint`. Here, the static and dynamic types of `p` in `rotateAroundPoint()` agree, but the static and dynamic types of `q` in `rotateAroundPoint()` differ.

The static type is what's used by Scala to do type checking. That makes sense, because type checking occurs before the program is run, and because the dynamic type of an identifier depends on the execution of the program, we do not know the dynamic type of identifiers before program execution. Thus, the only thing that the type checker can work with is the static type of identifiers. What the type checker does, then is whenever the code says that there is a method call such as `p.foo()`, it checks the static type of `p`, and asks whether the static type of `p` has a method `foo()`; if yes, checking continues, if not, a type-checking error is reported saying that `foo()` is undefined in the static type of `p`.

(There is another restriction on the type checker, which is more pragmatic: you want it to be fast. People don't like when type checking or compiling in general takes too long. So to make sure that type checking can be done quickly, when type checking a method call, *the type checker only uses the signature of the method* — that is, its declared parameter types, and its declared return type. It does not actually look inside the method to see the kind of calls that are being made. It relies on the fact that it has type checked that method already to make sure it is guaranteed to be safe. This is going to be important later.)

We could imagine that the dynamic type of identifiers could be "guessed" or somewhat derived by the type checker, but there are deep reasons why that cannot be done. To give you an idea of the difficulties involved, consider that the actual type of values may and in fact often will depend on some *a priori* unpredictable value. For instance, suppose that `x` is an integer derived from something the user input to the program, then the following code

```
if (x>0)
  Point.cartesian(1.0,2.0)
else
  CPoint.cartesian(10.0,20.0,Color.blue())
```

creates either an actual `Point` or an actual `CPoint` depending on the value of `x` — whether the result has dynamic type `Point` or `CPoint` therefore depends on user input, which is definitely unknown before the program executes.

So the type checker uses the static types to do its job. Why do we even care about dynamic types? Because that's what gives OO languages much of their expressive power.

9.1 Dynamic Dispatch

Consider the following function

```
def printPoint (p:Point):Unit =
  println(p.toString())
```

Clearly, because `printPoint()` expects a `Point`, we should be able to give it a `CPoint`, since `CPoint` is a subtype of `Point` — we will see below exactly how we can reason about those kind of situations formally, but for now, let's rely on our intuition.

So the following two calls should be acceptable:

```
printPoint(Point.cartesian(1.0,2.0))
printPoint(CPoint.cartesian(10.0,20.0,Color.red())
```

What gets printed? When you call

```
printPoint(Point.cartesian(1.0,2.0)),
```

this eventually invokes method `toString()` — which `toString()` method is invoked? The one defined for the actual instance that is passed to `printPoint()` — the `toString()` method in `Point`. So this prints `cart(1.0,2.0)`. By way of contrast, when you call

```
printPoint(CPoint.cartesian(10.0,20.0,Color.red())),
```

this eventually invokes the `toString()` method defined for the actual instance that is passed to `printPoint()` — the `toString()` method in `CPoint`. So this prints `cart(10.0,20.0,red)`.

In Scala (and in Java, and in several modern object-oriented languages), when you invoke a method on an instance, then the method that gets called is the method defined in the actual type of the instance. So if you write `p.toString()`, the method `toString()` that is called is the one defined in the *dynamic type* of `p`, since the dynamic type is the one corresponding to the actual type of the value carried by `p`. This is called *dynamic dispatch* — the method called (or dispatched) is the one in the dynamic type of the value on which you invoke the method.

(In some other languages, the method called is the one defined in the static type of the value on which the method is invoked. That's the default behavior in C++, for instance. That's called, naturally enough, *static dispatch*.)

Dynamic dispatch is pretty powerful, because it makes it easy to adjust the behavior of objects by simply giving different definitions for a given method. It is one of the hallmarks of object-oriented programming, and one of the reason it took off when it did. But it's also a software-engineering nightmare. Why?

Consider the definition `printPoint()` above. Someone looking at the code might think that the call to `toString()` is to the method defined in `Point`, but we know that's not true. The methods invoked depend on the dynamic type of the value passed to `printPoint()`, which can be of any subtype of `Point`. But that means that someone can define a subtype of `Point` that happens to do *anything* in its `toString()`, and the function `printPoint()` will happily call that method. Unless one knows exactly what the possible subtypes of `Point` exist and what their methods are doing, it is essentially impossible to predict just what `printPoint()` does. This gets worse when developing a library and offer something like `printPoint()`, because then you cannot guarantee anything to users about what the function can do.[16]

So what do we have? The type checker uses static types to do its check, while the execution engine uses dynamic types to determine exactly what method to call during executon. Recall that the purpose of the type system is to ensure safety, namely, that there is never an attempt during execution to invoke a method that does not exist. Whether there is an attempt to invoke a method that does not exist depends on the dynamic type (since that's what the execution engine uses to find the method to execute), but the type checker only has access to the static types. It turns out that in order for the type checker to never accept a program that is not safe, it suffices for the type checker to guarantee the following invariant: that during execution, the dynamic type of an identifier is always a subtype of its static type. (Why is this enough?) This forces the type checker to reject some programs when it cannot guarantee that this invariant will hold.

And understanding subtyping is in part understanding why the type checker rejects some programs but not others. We will see that next time.

[16]One way around this is to restrict the extent to which `Point` can be subtyped. Languages have ways to do that.