8 Understanding Subtyping

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).

Static and Dynamic 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.

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.

To show the difference between the two, 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 $(10, 20,$ *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.

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.[1]

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.

Upcasts. There are many ways to understand subtyping. The basic model that I will advocate is the following. First, start with the assumption that the type checker is only happy if static types agree, so that if an expression expects a value with static type T, the type checker will only be happy if that expression is given a value with static type T. Thus,

```
val x:T = <some-expr>
```

if okay if the static type of `<some-expr>` (that is the type that `<some-expr>` is declared to return — the result type of a method, say) is `T`, since `x` expects a `T`.

Similarly,

```
val x:T = <some-expr>
foo(x)
```

if okay if `foo()` has a parameter with static type `T`.

Let's define the following function on `Point`s:

```
def sum2 (pt:Point):Double =
  pt.xCoord() + pt.yCoord()
```

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

The type checker has no problems with the following code, in which all static types agree: `Point.cartesian()` returns a static `Point`, `p` expects a static `Point`, and `sum2()` expects a static `Point`. Running the code form the scala interactive loop (so that we see the results right away):

```
scala> {
         val p:Point = Point.cartesian(1.0,2.0)
         sum2(p)
       }
res1: Double = 3.0
```

So if the type checker is only happy when static types agree, where does subtyping come in? The type checker gets some help.

For every pair of subtypes $D \leq C$ in the program, think of there being a function $\Uparrow_D^C$ taking values of type D and giving back that same value but now looking like it has type C. This function is called an *upcast*, since it moves up the subtyping hierarchy by "transforming" Ds into Cs. (There is also such a thing as a downcast, to which we'll return below.)

When the type checker checks to see if type agrees and they don't, it sees whether or not it can insert an upcast (chosen from among the upcasts it has available — remember, every pair of subtypes yields an upcast) to make the types agree.

Consider the following example, a variant of the one above:

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         sum2(cp)
       }
res2: Double = 3.0
```

Here, `cp` is a `CPoint`, and `sum2()` expects a `Point` — that's a mismatch. Is there an upcast available? We know `Cpoint` is a subtype of `Point`, so there is an upcast $\Uparrow_{\texttt{CPoint}}^{\texttt{Point}}$, and

$$\texttt{sum2}(\Uparrow_{\texttt{CPoint}}^{\texttt{Point}}\texttt{cp})$$

has no type mismatch.

Similarly, in

```
scala> {
         val p:Point = CPoint.cartesian(1.0,2.0,Color.red())
         sum2(p)
       }
res3: Double = 3.0
```

the binding of `p` to a `CPoint` is a type mismatch — since `p` expects its value to be (statically) a `Point` but it is given a `CPoint`, but as we saw there is an upcast from `CPoint` to `Point` and the type checker is happy to insert it so that we have

$$\texttt{val p:Point = } \Uparrow^{\texttt{Point}}_{\texttt{CPoint}} \texttt{CPoint.cartesian(1.0,2.0,Color.red())}$$

which has no type mismatch.

There is no need for you to write upcasts explicitly.[2] The point is: upcasts are always safe to insert. An upcast can *never* make a safe program unsafe. (Why?) Therefore, if a program is safe after an upcast has been inserted, it would have been safe even without the upcast. That means that the type checker can actually take care of all the upcasts for you, since it never has to worry about screwing up and making a program that was safe the way you wrote it unsafe by adding the upcast.

Downcasts. Let's look at a different variant example. Consider the following function:

```
  def sum3 (cpt:CPoint):Double =
    cpt.xCoord() + cpt.yCoord() + cpt.color().code()
```

Again, if we call `sum3()` with a value of static type `CPoint`, the type checker is happy:

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         sum3(cp)
       }
res4: Double = 4.0
```

If we do not, however, it complains bitterly:

```
scala> {
         val p:Point = CPoint.cartesian(1.0,2.0,Color.red())
         sum3(p)
       }
<console>:9: error: type mismatch;
 found   : Point
 required: CPoint
       sum3(p)
```

[2] Although you can if you want. For any type C and D such that $D \leq C$, you can define an upcast easily. For `CPoint` and `Point`, for instance, you can define

```
def upCPointToPoint (cp:CPoint):Point = cp
```

As we see, the type checker did not help us: `sum3()` expected a value with static type `CPoint`, it received a value with static type `Point`. To convert one into the other, it would have needed a *downcast*. Again, just like for upcasts, for every pair of subtypes $D \leq C$, there is a downcast function $\Downarrow_D^C$ that takes values of type C and returns them unchanged but looking like values of type D.

The difference between downcasts and upcasts is that the type checker will never insert a downcast for you. That's because downcasts are not safe. In the above example, the downcast would be fine — why? Well, during execution, `p` gets an actual `CPoint` (the dynamic type of `p` during execution is indeed `CPoint`) — it just looks like a `Point` to the type system because the static type of `p` is `Point` (and note that the type checker will need to throw in an upcast to make the `CPoint` type agree with the `Point` type in the `val` line). Were that value `p` be passed to `sum3()`, it would be fine because `sum3()` will access the `xCoord()`, `yCoord()`, and `color()` methods of `p`, and it has all of those because its dynamic type is indeed `CPoint`. So we could expect that the system would insert a downcast like this:

$$\texttt{sum3}(\Downarrow_{\texttt{CPoint}}^{\texttt{Point}}\texttt{p})$$

The problem is that the system cannot guarantee that such a downcast is safe. Consider the following variant of the code above:

```
scala> {
         val p:Point = Point.cartesian(1.0,2.0)
         sum3(p)
       }
<console>:9: error: type mismatch;
 found   : Point
 required: CPoint
       sum3(p)
```

As far as the type system is concerned, when looking at `sum3(p)`, it sees that `p` has static type `Point`, and that `sum3()` expects a static type `CPoint`. Should it insert a downcast? No, because the dynamic type of `p` during execution is actually a `Point`, and were it passed to `sum3()` the method `color()` would not be defined. So that program's unsafe.

When type checking `sum3()`, the only information that the type checker looks at is the static type of the values it has to work with. It sees `p` with static type `Point` and `sum3()` expecting a value of static type `CPoint`. It cannot insert a downcast because just based on that information it doesn't know whether it is in the first case above (where the value `p` has dynamic type `CPoint`) or the second case above (where the value `p` has dynamic type `Point`). The first case would be fine, the second case would be disastrous. So it has to act conservatively, and never insert downcasts.

Bottom line: type checking requires static types to agree. upcasts can be used to bridge static types that do not agree, and they are inserted automatically for you by the type checker. Downcasts, however, are *never* inserted for you.

Information Loss. Consider the following two functions:

```
  def identityPoint (pt:Point):Point =
    pt

  def clonePoint (pt:Point):Point =
    Point.cartesian(pt.xCoord(),pt.yCoord())
```

First, note that `identityPoint()` just returns its argument, while `clonePoint()` constructs a new `Point` out of its argument. Second note that the two functions have exactly the same signature: they both take a `Point` argument, and return a `Point` result. Therefore, by the statement I made above regarding how the type checker works, the type checker will not be able to distinguish these two functions when reasoning about calls to those functions.

Let's make sure that we understand what happens when we try to use those functions. In this first example, all static types agree, so the type checker is happy.

```
scala> {
         val p:Point = Point.cartesian(1.0,2.0)
         val q:Point = identityPoint(p)
         sum2(q)
       }
res7: Double = 3.0
```

In this second example, the type checker is happy because it can insert an upcast when calling `identityPoint()`:

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         val q:Point = identityPoint(cp)
         sum2(q)
       }
res8: Double = 3.0
```

If we try to bind the result of `identityPoint()` to a `CPoint`, we fail as in this third example:

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         val cq:CPoint = identityPoint(cp)
         sum3(cq)
       }
<console>:10: error: type mismatch;
 found   : Point
 required: CPoint
       val cq:CPoint = identityPoint(cp)
```

There is an upcast inserted before the argument of `identityPoint`, but to take the result (which has static type `Point`) and bind it to `cq` which is declared to have static type `CPoint`, we need a downcast, and the type checker does not insert those for us. So we get a type-checking error. For exactly that same reason, the type checker gives an error if we also try to do that with `clonePoint()`:

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         val cq:CPoint = clonePoint(cp)
         sum3(cq)
       }
<console>:10: error: type mismatch;
 found   : Point
 required: CPoint
       val cq:CPoint = clonePoint(cp)
```

And rightly so — if this was not forbidden, we could pass `cq` to `sum3()` and we'd get a runtime error that the `color()` method does not exist in `cq`.

But there's something going on here. Looking at `identityPoint()`, it just returns its argument untouched. Meaning that if we construct a `CPoint` and pass it to `identityPoint`, the result will have dunamic type `CPoint`. In contrast, `clonePoint()` construct an actual `Point` from whatever its argument is, meaning that if we pass `clonePoint()` a colored point, the result will have dynamic type `Point`.

The types we have at this point, however, cannot pin down this difference between the behavior of `identityPoint()` and `clonePoint()`. Intuitively, the type of `identityPoint()` loses some information about what `identityPoint()` does, and that makes code such as the third example above fail, while we know full well the program is safe.

How can we recover from this information loss? We can use downcasts. Now, the type checker does not insert downcasts automatically for us — as we saw, they're not safe in general. Downcasts are only safe when the value they're downcasting has dynamic type that is a subtype of the type you're downcasting to. (Ouch, that's a mouthful.) And the thing is, we can insert downcasts by hand. We just need to define them first. Here's the downcast from `Point` to `CPoint`:

```
  def downPointToCPoint (pt:Point):CPoint =
    pt match {
      case cpt : CPoint => cpt
      case _ => throw new RuntimeException("conversion to CPoint failed")
    }
```

This function basically checks the dynamic type of its argument using `match` — if it is `CPoint`, then it returns that `CPoint`. If not, then it throws an exception indicating you've tried to downcast a `Point` to a `CPoint` when what you had was not really a `CPoint`.

And with this, we can now satisfy the type checker in our third example:

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         val q:Point = identityPoint(cp)
         sum3(downPointToCPoint(q))
       }
res12: Double = 4.0
```

The type checker is happy, and execution proceeds correctly.

What about using a downcast in the `clonePoint()` example? Something should go wrong, right, because we saw that the program is unsafe.

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         val q:Point = clonePoint(cp)
         sum3(downPointToCPoint(q))
       }
java.lang.RuntimeException: conversion to CPoint failed
at .downPointToCPoint(<console>:8)
  ...
```

Note that the program type checked — the type checker was happy with the downcast, since all the static types agreed — but the program failed during execution, with an exception from our downcast, stating that the downcast we're trying to perform is not possible.

What we've done, with downcast, is trade the niceness of getting compile-time errors (that is, the type checker complaining that something might be unsafe), for instead having run-time errors (the downcast failing with an exception because the program is unsafe).

Because of that, becaue they trade compile-time errors for run-time errors, downcasts are usually frowned upon. But they seem necessary in the cases like `identityPoint()` to recover information lost because the type system could not precisely enough describe the behavior of `identityPoint()`. Or could it?

Generic Methods. The problem is that the type of `identityPoint()` is the same as that of `clonePoint()`, even though the two functions behave quite differently. In particular, we know full well that whatever the dynamic type of the argument of `identityPoint()` is, the result will have that exact same dynamic type.

It turns out that by using a *generic method*, we can define a method a suitable type that says exactly that. Intuitively, we want the type of `identityPoint()` to say that it takes an argument of some type T and returns a result of that same exact type T, and does so for any type T that is a subtype of `Point`. Let's write that down:

```
  def identityPoint[T <: Point] (pt:T):T =
    pt
```

You can think of the `[T <: Point]` part as an extra parameter to the method, so that when you call it not only do you pass the regular arguments, but also another argument representing the type at which you want `identityPoint()` to work. Intuitively, `identityPoint[Point]()` takes a (static) `Point` as argument and returns a (static) `Point`, while `identityPoint[CPoint]` takes a (static) `CPoint` as argument and returns a (static) `CPoint`.

We can confirm that we can use this new function appropriately. First, the third example from before:

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         val cq:CPoint = identityPoint[CPoint](cp)
         sum3(cq)
       }
res14: Double = 4.0
```

and here's another example that shows that static types still need to agree:

```
scala> {
         val cp:CPoint = CPoint.cartesian(1.0,2.0,Color.red())
         val cq:CPoint = identityPoint[Point](cp)
         sum3(cq)
       }
<console>:10: error: type mismatch;
 found   : Point
 required: CPoint
       val cq:CPoint = identityPoint[Point](cp)
                                            ^
```

Again, since `identityPoint[Point]()` returns a value of static type `Point`, in order for that to match with the expected type `CPoint`, the type checker would need to introduce a downcast, which it will never do automatically, so we get a type error.

Two things. First off, you often do not have to specify the type argument when calling a generic function — that type can usually be inferred for you. So you can just write

```
val cq:CPoint = identityPoint(cp)
```

and the system will fill in the `[CPoint]` for you.

Second, if the bound on the type parameter in the generic method is `<: Any`, then it can simply be left out. For instance, here is a definition for a general identity function:

```
  def identity[T] (x:T):T = x
```

and some sample execution:

```
scala> {
         val i:Int = 100
         val j:Int = identity[Int](i)
         j
       }
res16: Int = 100
```