10 Understanding Subtyping

10.1 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()
```

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 $\Uparrow^C_D$ 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 $\Uparrow^{\texttt{Point}}_{\texttt{CPoint}}$, and

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

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.[17] 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.

10.2 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)
```

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 $D \leq C$,

[17]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
```

there is a downcast function $\Downarrow^C_D$ 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:

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

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.