11 Subtyping and Static Information Loss

11.1 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?

11.2 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
```