21 Subtyping and Parameterized Types

Given a parameterized type `Foo[T]`, suppose we know that $\texttt{A} \leq \texttt{B}$ (`A` is a subtype of `B`), then what is the relationship between `Foo[A]` and `Foo[B]`? There are three possibilities:

(1) `Foo[A]` and `Foo[B]` are unrelated;

(2) $\texttt{Foo[A]} \leq \texttt{Foo[B]}$; or

(3) $\texttt{Foo[A]} \geq \texttt{Foo[B]}$.

If (1) holds, we say that `Foo` is *invariant* in `T`; if (2) holds, we say that `Foo` is *covariant* in `T`; and if (3) holds, we say that `Foo` is *contravariant* in `T`.[20] By "holds" here, I mean the following: which property can we assume and that does not lead to the type system accepting unsafe programs? (Remember, the whole goal of the type system is to rule out unsafe programs; subtyping is a way to allow more programs to type check, while still ruling out unsafe programs.)

Now, when you query people, and you ask, for instance, whether a `List[A]` should be a subtype of `List[B]` when `A` is a subtype of `B`, they think about it, and generally say yes, that should be the case: if you have a function that expects, say, a list of points, then it should work fine if you give it a list of color points. In other words, most people implicitly think of lists as being covariant.

So what happens in Scala? Consider our by now well-understood implementation of lists:

```
object List {

  def empty[A] ():List[A] = new ListEmpty[A]()

  def singleton[B] (i:B):List[B] = new ListSingleton[B](i)

  def merge[C] (L:List[C], M:List[C]):List[C] = new ListMerge[C](L,M)


  private class ListEmpty[T] () extends List[T] {
```

[20] if `Foo` has only one parameter, we usually simply say that `Foo` is invariant, covariant, or contravariant.

```
  def isEmpty ():Boolean = true
  def first ():T = throw new RuntimeException("empty().first()")
  def rest ():List[T] = throw new RuntimeException("empty().rest()")

  def length ():Int = 0

  override def hashCode ():Int = 41

  override def toString ():String = ""
}


private class ListSingleton[U] (i:U) extends List[U] {

  def isEmpty ():Boolean = false
  def first ():U = i
  def rest ():List[U] = List.empty()

  def length ():Int = 1

  override def hashCode ():Int = 41 + i.hashCode()

  override def toString ():String = " " + i.toString()
}


private class ListMerge[V] (L:List[V], M:List[V]) extends List[V] {

  def isEmpty ():Boolean =
    (L.isEmpty() && M.isEmpty())

  def first ():V =
    if (L.isEmpty())
      M.first()
    else
      L.first()

  def rest ():List[V] =
    if (L.isEmpty())
      M.rest()
    else
      List.merge(L.rest(),M)
```

```
    def length ():Int = L.length() + M.length()

    override def hashCode ():Int =
      41 * (
        41 + L.hashCode()
      ) + M.hashCode()

    override def toString ():String = L.toString() + M.toString()
  }
}


abstract class List[T] {

  def isEmpty ():Boolean
  def first ():T
  def rest ():List[T]
  def length ():Int
}
```

Let's write some code to see what happens if we try to subtype lists. Here are two functions that work on lists of points and color points:

```
def xCoords (l:List[Point]):List[Int] = {
  if (l.isEmpty())
    List.empty[Int]()
  else
    List.merge(List.singleton(l.first().xCoord()),xCoords(l.rest()))
}

def colors (l:List[CPoint]):List[String] = {
  if (l.isEmpty())
    List.empty[String]()
  else
    List.merge(List.singleton(l.first().color()),colors(l.rest()))
}
```

Function `xCoords()` takes a list of `Point`s and returns the list of all the x-coordinates of those points. Function `colors()` does something similar, taking a list of `CPoint`s and returning the list of all the colors of those points.

For all of the examples below, we shall use the following definitions:

```
  val p:Point = Point.cartesian(40,50)
  val q:Point = Point.cartesian(400,500)
  val l:List[Point] = List.merge(List.singleton(p),
                          List.merge(List.singleton(q),List.empty()))

  val cp:CPoint = CPoint.cartesian(10,20,Color.red())
  val cq:CPoint = CPoint.cartesian(100,200,Color.blue())
  val cl:List[CPoint] = List.merge(List.singleton(cp),
                           List.merge(List.singleton(cq),List.empty()))
```

Now, we can try:

```
scala> xCoords(l)
res0: List[Int] = 40 400

scala> xCoords(cl)
<console>:13: error: type mismatch;
 found   : List[CPoint]
 required: List[Point]
       xCoords(cl)
               ^
```

So we see that we cannot call `xCoords()` with a list of `CPoints`. In other words, by default, Scala considers parameterized types to be *invariant*. (So does Java, by the way.)

21.1 Covariance

Our intuition is that `List` should be covariant. It should be possible to call `xCoords()` passing in the list `cp` of `CPoints`.

You can tell Scala that you want `List` to be covariant by annotating the definition of `List`:

```
abstract class List[+T] {

  def isEmpty ():Boolean
  def first ():T
  def rest ():List[T]
  def length ():Int
}
```

Note that `+` before the type parameter — it says that `List` is now covariant in that parameter. Compiling the code does not cause a problem; Scala seems happy to consider `List` covariant. Let's try it out:

```
scala> xCoords(cl)
res0: List[Int] = 10 100
```

And everything works as expected.

So why the big deal? Why not make every parameterized type covariant by default. Because covariance is not always safe. In particular, mutation lets us write unsafe programs if we assume covariance.

Let me illustrate this. Suppose that we make `List` mutable by adding a method `update()` that lets us change the first element of a list. Here is the resulting code:

```
object List {

  def empty[A] ():List[A] = new ListEmpty[A]()

  def singleton[B] (i:B):List[B] = new ListSingleton[B](i)

  def merge[C] (L:List[C], M:List[C]):List[C] = new ListMerge[C](L,M)


  private class ListEmpty[T] () extends List[T] {

    def isEmpty ():Boolean = true
    def first ():T = throw new RuntimeException("empty().first()")
    def rest ():List[T] = throw new RuntimeException("empty().rest()")

    def length ():Int = 0

    def update (v:T):Unit =
      throw new RuntimeException("empty().update()")

    override def hashCode ():Int = 41

    override def toString ():String = ""
  }


  private class ListSingleton[U] (i:U) extends List[U] {

    private var value:U = i

    def isEmpty ():Boolean = false
    def first ():U = value
```

```
  def rest ():List[U] = List.empty()

  def length ():Int = 1

  def update (v:U):Unit = { value = v }

  override def hashCode ():Int = 41 + i.hashCode()

  override def toString ():String = " " + value.toString()
}


private class ListMerge[V] (L:List[V], M:List[V]) extends List[V] {

  def isEmpty ():Boolean =
    (L.isEmpty() && M.isEmpty())

  def first ():V =
    if (L.isEmpty())
      M.first()
    else
      L.first()

  def rest ():List[V] =
    if (L.isEmpty())
      M.rest()
    else
      List.merge(L.rest(),M)

  def length ():Int = L.length() + M.length()

  def update (v:V):Unit =
    if (L.isEmpty())
       M.update(v)
    else
       L.update(v)

  override def hashCode ():Int =
    41 * (
      41 + L.hashCode()
    ) + M.hashCode()
```

```
    override def toString ():String = L.toString() + M.toString()
  }
}


abstract class List[T] {

  def isEmpty ():Boolean
  def first ():T
  def rest ():List[T]
  def length ():Int

  def update (v:T):Unit   // mutates the first element of a list
}
```

This compiles, and works fine, but we had to make `List` invariant. If we add a `+` before the type parameter `T` to indicate that we want `List` to be covariant, then the compiler complains during type checking:

```
List.scala:73: error: covariant type T occurs in contravariant
position in type T of value v
  def update (v:T):Unit
```

So the type checker is unhappy if we try to make `List` covariant. Why is that? Assuming that the type checker is not unnecessarily conservative, then there must be a way to write an unsafe program if we assume covariance here.

Let's do that. Suppose that the type checker allowed us to say that `List` was covariant. Then we could write the following sequence (again, assuming the definitions for `p`,`q`,`l`,`cp`,`cq`,`cl` above), and it would type check:

```
  val cl2:List[Point] = cl
  cl2.update(Point.cartesian(99,99))
  colors(cl)
```

But it is easy to see that if we could execute the above sequence, then we would get a *method `color()` not found* error during execution of `colors()`: the first line makes `cl2` an alias for `cl`, which is a list of `CPoints` (that is, if you recall the sharing diagrams from last time, both `cl` and `cl2` are references to the same instance). But `cl2` has static type `List[Point]`, so we can call its `update()` method that expects a `Point`, and we can update `cl2` so that the first element is a `Point`. But because `cl2` and `cl` are aliases for the same underlying list, we've made the first element of `cl` a `Point`, so that `cl` is not a list of color points anymore, but actually has a `Point` in it. And when calling `colors(cl)`, the system would try to call `color()` on the first element of `cl`, which is a `Point`.

Looking at the sequence above, the second and third lines are pretty uncontroversial — we're calling methods on `cl2` and `cl` with an argument of the exact type each method expects. So the problem must come from the first line, where we have to use an upcast provided us by our assumption that `List` is covariant. The solution is to forbid this upcast, that is, forbid `List` from being covariant.

One might think that the problem has to do with the update, but it turns out to be more subtle than that. The problem is that `List[T]` has a method that expects an argument of type `T`. That's sufficient to cause problems, it turns out. Of course, an update is exactly a method of that sort, so that explains why updates are problematic. To show that this problem is indeed due to method of that sort, consider a different variant of `List`: instead of adding an `update()` method, we add a `find()` method that returns a Boolean indicating whether a particular element appears in the list.

```
object List {

  def empty[A] ():List[A] = new ListEmpty[A]()

  def singleton[B] (i:B):List[B] = new ListSingleton[B](i)

  def merge[C] (L:List[C], M:List[C]):List[C] = new ListMerge[C](L,M)


  private class ListEmpty[T] () extends List[T] {

    def isEmpty ():Boolean = true
    def first ():T = throw new RuntimeException("empty().first()")
    def rest ():List[T] = throw new RuntimeException("empty().rest()")

    def length ():Int = 0

    def find (v:T):Boolean =
      false

    override def hashCode ():Int = 41

    override def toString ():String = ""
  }


  private class ListSingleton[U] (i:U) extends List[U] {

    def isEmpty ():Boolean = false
```

```
  def first ():U = i
  def rest ():List[U] = List.empty()

  def length ():Int = 1

  def find (v:U):Boolean =
    ( v == value )

  override def hashCode ():Int = 41 + i.hashCode()

  override def toString ():String = " " + value.toString()
}


private class ListMerge[V] (L:List[V], M:List[V]) extends List[V] {

  def isEmpty ():Boolean =
    (L.isEmpty() && M.isEmpty())

  def first ():V =
    if (L.isEmpty())
      M.first()
    else
      L.first()

  def rest ():List[V] =
    if (L.isEmpty())
      M.rest()
    else
      List.merge(L.rest(),M)

  def length ():Int = L.length() + M.length()

  def find (v:V):Boolean =
    (L.find(v) || M.find(v))

  override def hashCode ():Int =
    41 * (
      41 + L.hashCode()
    ) + M.hashCode()

  override def toString ():String = L.toString() + M.toString()
```

```
  }
}


abstract class List[T] {

  def isEmpty ():Boolean
  def first ():T
  def rest ():List[T]
  def length ():Int

  def find (v:T):Boolean
}
```

Again, if we try to make List covariant, we get a complaint from the Scala compiler during type checking. Let me show you code that is indeed unsafe if we allowed List to be covariant.

The example is not as direct as with mutation above. We first need to consider the following class, which defines a subtype of List[CPoint]:

```
object BadListCP {

  def empty ():BadListCP = new ListEmpty()

  def cons (n:CPoint, L:BadListCP):BadListCP = new ListCons(n,L)

  private class ListEmpty () extends BadListCP {
    def isEmpty ():Boolean = true
    def first ():CPoint =
      throw new RuntimeException("empty().first()")
    def rest ():BadListCP =
      throw new RuntimeException("empty().rest()")
    def length ():Int = 0

    def find (f:CPoint):Boolean = false

    override def toString ():String = ""
  }

  private class ListCons (n:CPoint, L:BadListCP) extends BadListCP {
    def isEmpty ():Boolean = false
    def first ():CPoint = n
    def rest ():BadListCP = L
```

```
    def length ():Int = 1 + L.length()

    def find (f:CPoint):Boolean = {
      println("Trying to find value " + f + " with color " + f.color())
      (f == n) || L.find(f)
    }

    override def toString ():String = " " + n + L.toString()
  }
}

abstract class BadListCP extends List[CPoint] {

  def isEmpty ():Boolean
  def first ():CPoint
  def rest (): BadListCP
  def length ():Int
  def find (f:CPoint):Boolean
}
```

I'm using two different creators here, `empty()` and `cons()`, for simplicity, and to show that subtypes can have different creators. This is completely orthogonal to my point here about coming up with an unsafe program. The important thing is the definition of `find` in `BadListCP.ListCons`: it calls the `color()` method of the value `find()` is looking for. This is perfectly fine, because we know here that `find()` expects a `CPoint`. So what could go wrong?

Well, just like before, suppose that the type checker allowed us to say that `List` was covariant. This would mean, in particular, that `BadListCP` $\leq$ `List[CPoint]` $\leq$ `List[Point]`. We could then write the following sequence, and it would type check:

```
val blc:BadListCP = BadListCP.cons(cp,BadListCP.cons(cq,BadListCP.empty()))
val cl2:List[Point] = blc
cl2.find(Point.cartesian(10,20))
```

Let's see what would happen if we were to run this: we create a `BadListCP` `blc`, define an alias `cl2` for it (with static type `List[Point]`) via an upcast because `BadListCP` $\leq$ `List[Point]` by assumption, and then call `cl2.find()` passing in a `Point`. Because of dynamic dispatch, the `find()` method that gets called is the `find()` method in the dynamic type of `cl2`, which is `BadListCP`, so that the `find()` method that gets called is the one that tries to access the `color()` method of its argument, and this would cause a *method `color()` not found* error because `Point.cartesian(10,20)` does not produce an instance that has a `color()` method in it.

So to a first approximation, the problem is methods expecting an argument of the same type as the parameter of the type we are defining, and those methods prevent us from making the type covariant.

It turns out that there is a way to make a parameterized type covariant even though it has methods that expect a value of the type of the parameter. But we have to work a bit harder at it. The idea is to give a method like `find()` a type that prevents us from writing the kind of code in `BadListCP`. Here is the solution. Give `find()` the type:

```
  def find[U >: T] (f:U):Boolean
```

Intuitively, `find()` now can take as argument a value of any type that is a *supertype* of `T`, the parameter type. In particular, this means that the body of `find()` cannot take advantage of any method that `T` may have, since we are not guaranteed to give it a value of type `T`. Thus, in `BadListCP`, the method definition:

```
    def find[U >: CPoint] (f:U):Boolean = {
      println("Trying to find value " + f + " with color " + f.color())
      (f == n) || L.find(f)
    }
```

would fail type check, since `f` has some type that is a supertype of `CPoint`, so that the type checker cannot guarantee that `f` has a method named `color()`.

If we give `find()` the above type, we find that we can make `List` covariant — the code compiles, and executing it yields:

```
scala> xCoords(cl)
res0: List[Int] = 10 100

scala> cl.find(cp)
res1: Boolean = true

scala> cl.find(CPoint.cartesian(5,6,"red"))
res3: Boolean = false
```

Exercise: correct `update()` in the same way.

21.2 Contravariance

What about contravariance? Is there any time when contravariance holds? The answer is yes. You tell Scala to make a parameterized type contravariant by using a `-` annotation instead of a `+` annotation. Contravariance is rare, but it does occur. Here is an example:

```
class Writer[-T] {

  def write (v:T):Unit =
    println("Value = " + v.toString())

}
```

If we have a function that expects such a writer, and a value to write to the writer:

```
def sendToWriter (w:Writer[CPoint],v:CPoint) =
  w.write(v)
```

We can see how contravariance is useful. Because `CPoint` $\leq$ `Point`, by contravariance, `Writer[Point]` $\leq$ `Writer[CPoint]`. So we can call:

```
scala> sendToWriter(new Writer[CPoint], cp)
Value = (10,20)@red

scala> sendToWriter(new Writer[Point], cp)
Value = (10,20)@red
```

And in fact we can also write another subtype of `Writer[Point]`:

```
class FunkyPointWriter extends Writer[Point] {

  override def write (v:Point):Unit = {
    println("Writing point with x-coordinate " + v.xCoord())
    super.write(v)
  }
}
```

and using it:

```
scala> sendToWriter(new FunkyPointWriter, cp)
Writing point with x-coordinate 10
Value = (10,20)@red
```

21.3 Arrays in Java

Let's jump down to Java for this section. As I mentioned above, parameterized types in Java are invariant, and there is no way to make them either covariant or contravariant.

What about arrays, though? Arrays are clearly mutable in Java, and a few experiments indicate that arrays are covariant: we can pass an `Array[CPoint]` to a function expecting an `Array[Point]`, and the Java type checker will not complain. To illustrate, assume that we have Java implementations of `Point` and `CPoint`, and consider the following code:

```
public class Test1 {

  public static void showCollection (Point[] coll) {
    System.out.println("In showCollection()");
    for (Point p : coll)
      System.out.println(" element = " + p.xCoord() + " " + p.yCoord());
  }

  public static void main (String[] argv) {

    CPoint[] coll = new CPoint[2];

    coll[0] = new CPoint(10,10,"red");
    coll[1] = new CPoint(20,20,"blue");

    System.out.println("In main()");
    for (CPoint cp : coll)
      System.out.println(" element color = " + cp.color());

    showCollection(coll);

    System.out.println("In main()");
    for (CPoint cp : coll)
      System.out.println(" element color = " + cp.color());
  }
}
```

This code type checks and executes without problems: it is okay to pass the array of `CPoints` to `showCollection()`, because each `CPoint` has both a `xCoord()` and `yCoord()` method, and execution proceeds without encountering an undefined method:

```
In main()
 element color = red
 element color = blue
In showCollection()
 element = 10 10
 element = 20 20
In main()
```

```
element color = red
element color = blue
```

Now, we saw above that when we make `Lists` mutable and covariant, we introduce the possibility of letting the type checker accept unsafe programs. And I just said that in Java, arrays are both mutable and covariant. So what gives? If mutable lists cause a problem when they're covariant, then arrays should cause a problem too. And indeed, a variant of the example that I used for lists showcases this:

```
public class Test2 {

  public static void showCollection (Point[] coll) {
     System.out.println("In showCollection()");
     for (Point p : coll)
     System.out.println(" element = " + p.xCoord() + " " + p.yCoord());

     coll[0] = new Point(0,0);
  }

  public static void main (String[] argv) {

      CPoint[] coll = new CPoint[2];

      coll[0] = new CPoint(10,10,"red");
      coll[1] = new CPoint(20,20,"blue");

      System.out.println("In main()");
      for (CPoint cp : coll)
        System.out.println(" element color = " + cp.color());

      showCollection(coll);

      System.out.println("In main()");
      for (CPoint cp : coll)
        System.out.println(" element color = " + cp.color());
  }
}
```

`Test2` is very similar to `Test1`, except that function `showCollection()` now modifies the first element of the array, making it hold a new `Point`. The code type checks: By contravariance of arrays, because `CPoint` $\leq$ `Point`, the type system lets you pass a `CPoint[]` to a method expecting a `Point[]`. And because the array `coll` in `showCollection()` is declared to be an array of `Points` (its static type), the type system is quite happy to let you update the first element in the array into a different `Point`.

The problem is that passing an object (including an array) to a method in Java only passes a reference to that object. The object is not actually copied, as we saw when we saw the mutation model. So when function `showCollection()` updates the array through its argument `coll`, it ends up modifying the underlying array `coll` in function `main()`. But that means that when we come back from the `showCollection()` function, array `coll` is an array of `CPoint`s where the first element of the array is not a `CPoint` any longer, but rather a `Point`. And when we attempt to invoke method `color()` on that first element, Java would choke because that first element, being a plain `Point`, does not in fact implement the `color()` method. The program is unsafe, but the type system accepted it.

Java trades off this inadequacy of the type system by doing a runtime check at the statement that causes the problem: the update `coll[0] = new Point(0,0)`. Java catches the fact that you are attempting to modify an array by putting in an object that is not a subclass of the dynamic type of the data in the array, and throws an `ArrayStoreException`. Here, that's because we are trying to put a `Point` into an array with dynamic type `CPoint[]`:

```
In main()
 element color = red
 element color = blue
In showCollection()
 element = 10 10
 element = 20 20
Exception in thread "main" java.lang.ArrayStoreException: Point
      at Test2.showCollection(Test2.java:9)
      at Test2.main(Test2.java:23)
```

The point remains: the type system does not fully do its job, and has to deleguate to the runtime system the responsability of ensuring that the problem above does not occur. And that's a problem — recall that lecture we had about why it was a good idea to report errors early, such as when the program is being compiled as opposed to when it executes?