Lecture 11

1 Variance and method arguments

Earlier, we wrote the simplest container we could imagine that only stored a single value. The following container slightly more sophisticated, because it can store two values:

```
class Two[+T](private val x: T, private val y: T) {
  def get1(): T = x
  def get2(): T = y
}
```

This container uses a covariance annotation, so when `A <: B`, we have `Two[A] <:Two[B]`. Recall that the intuition for subtyping is that `Two[A]` can be used wherever a `Two[B]` is expected. Now, suppose we modify the class to add functional update methods:

```
class Two[+T](private val x: T, private val y: T) {
  ...
  def update1(newX: T): Two[T] = new Two(newX, y)
  def update2(newY: T): Two[T] = new Two(x, newY)
}
```

This is a natural operation on containers and we should be able to write it. But, let's ensure that even with this operation, `Two[A]` can be used whenever a `Two[B]` is expected.

Concretely, consider the following two classes:

```
class Dog {
  def makeSound(): String = "woof"
}

class Poodle extends Dog {
  def bite(): String = "nip"
}
```

We should be able to use a `Two[Poodle]` in all contexts where a `Two[Dog]` is expected.

Consider the following function, which simply updates a container to store a `Dog`:

```
def store[A <: Two[Dog]](x: A): A = x.update1(new Dog)
```

However, it uses bounded-quantification to any subtype of `Two[Dog]`. Therefore, we can apply it to a `Poodle` container and know that the result is still a `Poodle` container. Of course, this isn't true, since the function stores a `Dog`. Therefore, the following code goes wrong:

```
val twoPoodles = new Two[Poodle](new Poodle(), new Poodle())
val aPoodle = storeDog(twoPoodles).get1() // Produces a Dog, not a Poodle!
aPoodle.bite() // Crash! Dogs don't have a .bite() method.
```

Naturally, Scala (and Java) won't allow this to happen. In general, when a class has a covariant type parameter `T`, its methods can produce values of type `T`, but methods' arguments cannot consume values of type T (because of the error described above). Therefore, the type-checker will not allow us to write the update methods, because they use a covariant type parameter as the type of a method argument.

A solution The problem with the code above is that when a Two[A] is treated as a Two[B], we can update the container to store any B-typed value, which then breaks code that expects A-typed values. Instead, we need to ensure that we can only store subtypes of A in the container. We can express this constraint using bounded-quantification:

```
class Two[+T](private val x: T, private val y: T) {
  ...
  def update1[S >: T](newX: T): Two[S] = new Two(newX, y)
  def update2[S >: T](newY: T): Two[S] = new Two(x, newY)
}
```

This version of the class does type-check.

If we have a Poodle-container:

```
val x = new Two[Poodle](new Poodle, new Poodle)
```

And we update one of the poodles:

```
x.update1[Poodle](new Poodle)
```

The type-parameter S is bound to the type Poodle, so the return type is Two[Poodle]. However, we can also update the the container with a Dog:

```
x.update1[Dog](new Dog)
```

However, in this case, the return type is Two[Dog] (although one of the dogs is a poodle).

Appending Lists A similar problem arises with list concatenation. Since lists are covariant, they cannot have an append method with this type:

```
sealed trait List[A] {
  def append[A](other: List[A]): List[A] = ...
  ...
}
```

However, we can use the same trick we used above to write an append method with the following type:

```
sealed trait List[A] {
  def append[B :> A](other: List[B]): List[B] = lst match {
    case Nil => Nil
    case hd :: tl => hd :: tl.append(other)
  }
  ...
}
```

2 A Model of Type-Checking

To understand generics, we've had to develop an intuition for how the Scala type-checker works. A deep understanding of the Scala (or Java) type system is beyond the scope of this class. However, it is important to have a reasonably accurate mental model of type-checking to understand and debug type-errors.

```
sealed trait Expr {

  override def toString(): String = this match {
    case EInt(n) => n.toString
    case EBool(b) => b.toString
    case EAdd(e1, e2) => s"$e1 + $e2"
    case ELT(e1, e2) => s"$e1 < $e2"
    case EIf(e1, e2, e3) => s"if ($e1) { $e2 } else { $e3 }"
    case EVal(x, e1, e2) => s"val $x = $e1\n$e2"
    case EId(x) => x
  }
}

case class EInt(n: Int) extends Expr
case class EBool(b: Boolean) extends Expr
case class EAdd(e1: Expr, e2: Expr) extends Expr
case class ELT(e1: Expr, e2: Expr) extends Expr
case class EIf(e1: Expr, e2: Expr, e3: Expr) extends Expr
case class EVal(x: String, e1: Expr, e2: Expr) extends Expr
case class EId(x: String) extends Expr
```

Figure 22.1: A subset of Scala expressions

Figure 22.1 is a data structure that represents a fragment of Scala expressions, including numbers, boolean, addition, the less-than operator, if-expressions, and identifiers (bound with `val`). Since the only values in this fragment are integers and booleans, we can represent types as follows:

```
sealed trait Type
case object TInt extends Type
case object TBool extends Type
```

Ignoring identifiers, we can write a simple recursive function to type-check programs in this fragment, as shown in fig. 22.2.

Type-checking identifiers is a little trickier. When we see an identifier, there is way to determine its type, unless we remember the type of the expression it was bound to. Therefore, we need an auxiliary parameter, known as the environment, to "remember" the type of identifiers so that we can recall them later (fig. 22.3).

The actual Scala type-checker is vastly more complicated. But, it follows this basic design.

```
object TypeError extends RuntimeException("Type error")

object TypeChecker {

  def tc(expr: Expr): Type = expr match {
    case EInt(_) => TInt
    case EBool(_) => TBool
    case EAdd(e1, e2) => (tc(e1), tc(e2)) match {
      case (TInt, TInt) => TInt
      case _ => throw TypeError
    }
    case ELT(e1, e2) => (tc(e1), tc(e2)) match {
      case (TInt, TInt) => TBool
      case _ => throw TypeError
    }
    case EIf(e1, e2, e3) => (tc(e1), tc(e2), tc(e3)) match {
      case (TBool, t1, t2) if (t1 == t2) => t1
      case _ => throw TypeError
    }
  }
}
```

Figure 22.2: Type-checking, excluding identifiers

```
object TypeChecker {
  def tc(env: Map[String, Type], expr: Expr): Type = expr match {
    case EInt(_) => TInt
    case EBool(_) => TBool
    case EAdd(e1, e2) => (tc(env, e1), tc(env, e2)) match {
      case (TInt, TInt) => TInt
      case _ => throw TypeError
    }
    case ELT(e1, e2) => (tc(env, e1), tc(env, e2)) match {
      case (TInt, TInt) => TBool
      case _ => throw TypeError
    }
    case EIf(e1, e2, e3) => (tc(env, e1), tc(env, e2), tc(env, e3)) match {
      case (TBool, t1, t2) if (t1 == t2) => t1
      case _ => throw TypeError
    }
    case EVal(x, e1, e2) => tc(env + (x -> tc(env, e1)), e2)
    case EId(x) => env.getOrElse(x, throw TypeError)
  }
}
```

Figure 22.3: Type-checking identifiers.