Type & polymorphism basics

«Previous Next»

This lesson covers:

What are static types? Why are they useful?

According to Pierce: “A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute.”

Types allow you to denote function domain & codomains. For example, from mathematics, we are used to seeing:

```
f: R -> N
```

this tells us that function “f” maps values from the set of real numbers to values of the set of natural numbers.

In the abstract, this is exactly what *concrete* types are. Type systems give us some more powerful ways to express these sets.

Given these annotations, the compiler can now *statically* (at compile time) verify that the program is *sound*. That is, compilation will fail if values (at runtime) will not comply to the constraints imposed by the program.

Generally speaking, the typechecker can only guarantee that *unsound* programs do not compile. It cannot guarantee that every sound program *will* compile.

With increasing expressiveness in type systems, we can produce more reliable code because it allows us to prove invariants about our program before it even runs (modulo bugs in the types themselves, of course!). Academia is pushing the limits of expressiveness very hard, including value-dependent types!

Note that all type information is removed at compile time. It is no longer needed. This is called erasure.

Types in Scala

Scala’s powerful type system allows for very rich expression. Some of its chief features are:

- **parametric polymorphism** roughly, generic programming
- **(local) type inference** roughly, why you needn’t say `val i: Int = 12: Int`
- **existential quantification** roughly, defining something *for some* unnamed type
- **views** we’ll learn these next week; roughly, “castability” of values of one type to another

Parametric polymorphism

Polymorphism is used in order to write generic code (for values of different types) without compromising static typing richness.

For example, without parametric polymorphism, a generic list data structure would always look like this (and indeed it did look like this in Java prior to generics):

```
scala> 2 :: 1 :: "bar" :: "foo" :: Nil
res5: List[Any] = List(2, 1, bar, foo)
```

Now we cannot recover any type information about the individual members.

```
scala> res5.head
res6: Any = 2
```

And so our application would devolve into a series of casts (“asInstanceOf[]”) and we would lack type safety (because these are all dynamic).

Polymorphism is achieved through specifying *type variables*.

```
scala> def drop1[A](l: List[A]) = l.tail
drop1: [A](l: List[A])List[A]

scala> drop1(List(1,2,3))
res1: List[Int] = List(2, 3)
```

Scala has rank-1 polymorphism

Roughly, this means that there are some type concepts you'd like to express in Scala that are "too generic" for the compiler to understand. Suppose you had some function

```
def toList[A](a: A) = List(a)
```

which you wished to use generically:

```
def foo[A, B](f: A => List[A], b: B) = f(b)
```

This does not compile, because all type variables have to be fixed at the invocation site. Even if you "nail down" type `B`,

```
def foo[A](f: A => List[A], i: Int) = f(i)
```

…you get a type mismatch.

Type inference

A traditional objection to static typing is that it has much syntactic overhead. Scala alleviates this by providing *type inference*.

The classic method for type inference in functional programming languages is *Hindley-Milner*, and it was first employed in ML.

Scala's type inference system works a little differently, but it's similar in spirit: infer constraints, and attempt to unify a type.

In Scala, for example, you cannot do the following:

```
scala> { x => x }
<console>:7: error: missing parameter type
       { x => x }
```

Whereas in OCaml, you can:

```
# fun x -> x;;
- : 'a -> 'a = <fun>
```

In scala all type inference is *local*. Scala considers one expression at a time. For example:

```
scala> def id[T](x: T) = x
id: [T](x: T)T

scala> val x = id(322)
x: Int = 322

scala> val x = id("hey")
x: java.lang.String = hey

scala> val x = id(Array(1,2,3,4))
x: Array[Int] = Array(1, 2, 3, 4)
```

Types are now preserved, The Scala compiler infers the type parameter for us. Note also how we did not have to specify the return type explicitly.

Variance

Scala's type system has to account for class hierarchies together with polymorphism. Class hierarchies allow the expression of subtype relationships. A central question that comes up when mixing OO with polymorphism is: if T' is a subclass of T, is Container[T'] considered a subclass of Container[T]? Variance annotations allow you to express the following relationships between class hierarchies & polymorphic types:

	Meaning	Scala notation
covariant	C[T'] is a subclass of C[T]	[+T]
contravariant	C[T] is a subclass of C[T']	[-T]
invariant	C[T] and C[T'] are not related	[T]

The subtype relationship really means: for a given type T, if T' is a subtype, can you substitute it?

```
scala> class Covariant[+A]
defined class Covariant

scala> val cv: Covariant[AnyRef] = new Covariant[String]
cv: Covariant[AnyRef] = Covariant@4035acf6

scala> val cv: Covariant[String] = new Covariant[AnyRef]
<console>:6: error: type mismatch;
 found   : Covariant[AnyRef]
 required: Covariant[String]
       val cv: Covariant[String] = new Covariant[AnyRef]
                                   ^
```

```
scala> class Contravariant[-A]
defined class Contravariant

scala> val cv: Contravariant[String] = new Contravariant[AnyRef]
cv: Contravariant[AnyRef] = Contravariant@49fa7ba

scala> val fail: Contravariant[AnyRef] = new Contravariant[String]
<console>:6: error: type mismatch;
 found   : Contravariant[String]
 required: Contravariant[AnyRef]
       val fail: Contravariant[AnyRef] = new Contravariant[String]
                                         ^
```

Contravariance seems strange. When is it used? Somewhat surprising!

```
trait Function1 [-T1, +R] extends AnyRef
```

If you think about this from the point of view of substitution, it makes a lot of sense. Let's first define a simple class hierarchy:

```
scala> class Animal { val sound = "rustle" }
defined class Animal

scala> class Bird extends Animal { override val sound = "call" }
defined class Bird

scala> class Chicken extends Bird { override val sound = "cluck" }
defined class Chicken
```

Suppose you need a function that takes a `Bird` param:

```
scala> val getTweet: (Bird => String) = // TODO
```

The standard animal library has a function that does what you want, but it takes an `Animal` parameter instead. In most situations, if you say "I need a ___, I have a subclass of ___", you're OK. But function parameters are contravariant. If you need a function that takes a `Bird` and you have a function that takes an `Chicken`, that function would choke on a `Duck`. But a function that takes an `Animal` is OK:

```
scala> val getTweet: (Bird => String) = ((a: Animal) => a.sound )
getTweet: Bird => String = <function1>
```

A function's return value type is covariant. If you need a function that returns a `Bird` but have a function that returns a `Chicken`, that's great.

```
scala> val hatch: (() => Bird) = (() => new Chicken )
hatch: () => Bird = <function0>
```

Bounds

Scala allows you to restrict polymorphic variables using *bounds*. These bounds express subtype relationships.

```
scala> def cacophony[T](things: Seq[T]) = things map (_.sound)
<console>:7: error: value sound is not a member of type parameter T
       def cacophony[T](things: Seq[T]) = things map (_.sound)
                                                      ^

scala> def biophony[T <: Animal](things: Seq[T]) = things map (_.sound)
biophony: [T <: Animal](things: Seq[T])Seq[java.lang.String]

scala> biophony(Seq(new Chicken, new Bird))
res5: Seq[java.lang.String] = List(cluck, call)
```

Lower type bounds are also supported; they come in handy with contravariance and clever covariance. `List[+T]` is covariant; a list of Birds is a list of Animals. `List` defines an operator `::(elem T)` that returns a new `List` with `elem` prepended. The new `List` has the same type as the original:

```
scala> val flock = List(new Bird, new Bird)
flock: List[Bird] = List(Bird@7e1ec70e, Bird@169ea8d2)

scala> new Chicken :: flock
res53: List[Bird] = List(Chicken@56fbda05, Bird@7e1ec70e, Bird@169ea8d2)
```

`List` *also* defines `::[B >: T](x: B)` which returns a `List[B]`. Notice the `B >: T`. That specifies type `B` as a superclass of `T`. That lets us do the right thing when prepending an `Animal` to a `List[Bird]`:

```
scala> new Animal :: flock
res59: List[Animal] = List(Animal@11f8d3a8, Bird@7e1ec70e, Bird@169ea8d2)
```

Note that the return type is `List[Animal]`.

Quantification

Sometimes you do not care to be able to name a type variable, for example:

```
scala> def count[A](l: List[A]) = l.size
count: [A](List[A])Int
```

Instead you can use “wildcards”:

```
scala> def count(l: List[_]) = l.size
count: (List[_])Int
```

This is shorthand for:

```
scala> def count(l: List[T forSome { type T }]) = l.size
count: (List[T forSome { type T }])Int
```

Note that quantification can get tricky:

```
scala> def drop1(l: List[_]) = l.tail
drop1: (List[_])List[Any]
```

Suddenly we lost type information! To see what’s going on, revert to the heavy-handed syntax:

```
scala> def drop1(l: List[T forSome { type T }]) = l.tail
drop1: (List[T forSome { type T }])List[T forSome { type T }]
```

We can’t say anything about T because the type does not allow it.

You may also apply bounds to wildcard type variables:

```
scala> def hashcodes(l: Seq[_ <: AnyRef]) = l map (_.hashCode)
hashcodes: (Seq[_ <: AnyRef])Seq[Int]
```

```
scala> hashcodes(Seq(1,2,3))
<console>:7: error: type mismatch;
 found   : Int(1)
 required: AnyRef
Note: primitive types are not implicitly converted to AnyRef.
You can safely force boxing by casting x.asInstanceOf[AnyRef].
       hashcodes(Seq(1,2,3))
                     ^

scala> hashcodes(Seq("one", "two", "three"))
res1: Seq[Int] = List(110182, 115276, 110339486)
```

See Also Existential types in Scala by D. R. MacIver

Built at @twitter by @stevej, @marius, and @lahosken with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman, and @mccv; Russian translation by appigram; Chinese simple translation by jasonqu; Korean translation by enshahar;