Programmation Appliquée en Scala

Types and Type Directed Programming

Prof. Nastaran Fatemi
Christopher Meier

Printemps 2025

In this course

- Intersection and Union types
- Variance
- Type Bounds

More safety with Union and Intersection types

In Scala 3, the type system was enhanced with two new features:

- **Intersection types**: `(A & B)`
- **Union types**: `(A | B)`

These feautures allow developers to precisely describe values that conform to multiple types simultaneously, or one of several possible types.

In the following slides, we discover them in more details.

Intersection Types

Used on types, the **&** operator creates an intersection type.

The type **`S & T`** represents **values that are of the type S and T at the same time**.

The members of an intersection type `A & B` are all the members of A and all the members of B.

Intersection types are commutative: `A & B` is the same type as `B & A`.

Intersection Types : Example

Example: Take the traits `Resettable`, and `Growable[A]`. Here we want the function `f` to use the methods defined in both of these traits:

```scala
trait Resettable:
  def reset(): Unit

trait Growable[A]:
  def add(a: A): Unit

//We define f with the parameter `x` of type `Resettable & Growable[String]`
//x is therefore required to be both a `Resettable` and a `Growable[String]`.
def f(x: Resettable & Growable[String]): Unit =
  x.reset()
  x.add("first")
```

Intersection Types : Example (ctd.)

Intersection types can be useful to **describe requirements structurally**.

- That is, in our example `f`, we directly express that we are happy with any value for `x` as long as it's a subtype of both Resettable and Growable.
- We did not have to create a nominal helper trait like the following:

```
trait Both[A] extends Resettable, Growable[A]
def f(x: Both[String]): Unit
```

Note that there is also an important difference between these two alternatives:

- With the intersection type, any value that happens to be both `Resettable` and `Growable[String]` can be passed to `f`.
- With the explicit extension type, only values that explicitly extend the trait `Both[String]` can be passed.

Union Types

Used on types, the **|** operator creates a so-called union type.

The type **A | B** represents **values that are either of the type A or of the type B**.

A union type `A | B` includes all values of both types.

Like intersection types, union types are commutative: `A | B` is equivalent to `B | A`.

Dually to intersection types, a union type is a **supertype of all combinations of its constituent types**.

- For example, `A | B` is a supertype of both `A` and `B`.
- `A | B` is not just a supertype of both `A` and `B`, but their nearest common supertype, or *least upper bound*.

A mathemnatical lattice

The addition of union and intersection types to Scala 3 ensures that Scala's type system forms a mathematical lattice.

A lattice is a partial order in which any two types have both a unique least upper bound, or LUB, and a unique greatest lower bound.

- The **least upper bound** of any two types is their **union**
- The **greatest lower bound** is their **intersection**.

Least upper and greatest lower bounds

Take the following example hierarchy:

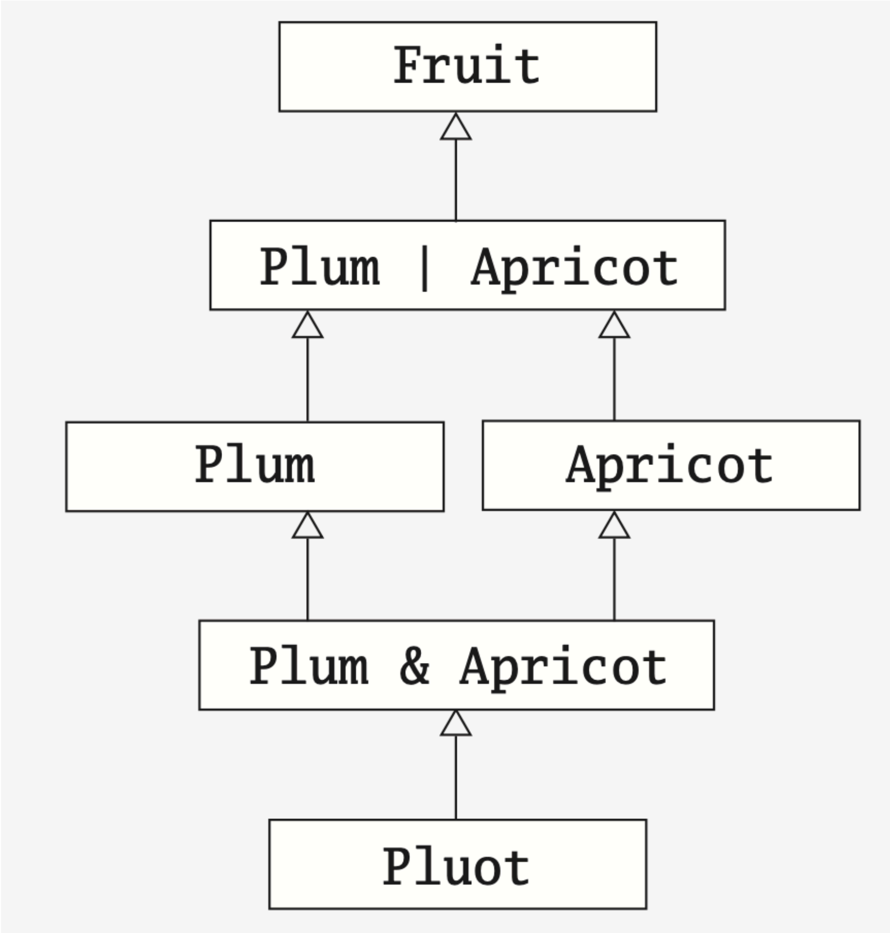
```scala
trait Fruit
trait Plum extends Fruit
trait Apricot extends Fruit
trait Pluot extends Plum, Apricot
```

The least upper bound of `Plum` and `Apricot`:

- `Plum | Apricot`

The greatest lower bound of `Plum` and `Apricot`:

- `Plum & Apricot`



Implication of Union Types on type inference

Whereas in Scala 2, the type inference algorithm had to settle on an approximation of the least upper bound of some pairs of types, Scala 3 can simply form a union of those types.

Example:

```
val plumOrApricot: Plum | Apricot = new Plum {}
// This compiles fine, because Plum | Apricot <: Fruit
val fruit: Fruit = plumOrApricot
// But you cannot use a Fruit where Plum | Apricot is needed
scala> val doesNotCompile: Plum | Apricot = fruit
```

Type inference on Union Types : Example

To access members of a union type, you must perform a pattern match to determine the actual class of the value at runtime.

Example: here, the `help` method accepts a parameter named `id` of the union type `Username | Password`, that can be either a `Username` or a `Password`:

```
case class Username(name: String)
case class Password(hash: Hash)

def help(id: Username | Password) =
  val user = id match
    case Username(name) => lookupName(name)
    case Password(hash) => lookupPassword(hash)
  // more code here ...
```

Type inference on Union Types : Example (ctd.)

This code is a flexible and type-safe solution. If you attempt to pass in a type other than a `Username` or `Password`, the compiler flags it as an error:

```
help("hi")   // error: Found: ("hi" : String) Required: Username | Password
```

You'll also get an error if you attempt to add a case to the match expression that doesn't match the `Username` or `Password` types:

```
case 1.0 => ???   // ERROR: this line won't compile
```

Alternative to Union types

As shown, union types can be used to represent alternatives of several different types, without requiring those types to be part of a custom-crafted class hierarchy, or requiring explicit wrapping. Without union types, it would require pre-planning of the class hierarchy.

Example:

```scala
trait UsernameOrPassword
case class Username(name: String) extends UsernameOrPassword
case class Password(hash: Hash) extends UsernameOrPassword
def help(id: UsernameOrPassword) = ...
```

– Pre-planning does not scale very well, as requirements of API users might not be foreseeable. Also, cluttering the type hierarchy with marker traits like `UsernameOrPassword` makes the code more difficult to read.

Union and Intersection Types : Conclusion

As a a conclusion, Union and Intersection Types simplify code, improve pattern matching, and reduce boilerplate in many use cases like function parameters, type constraints, and more.

Quiz

What is the inferred type of `guess`?

```
def guess(l1: List[Int], n:Int) = n match
    case 0 => l1.foldLeft(List[Int]())((x, y) => x :+ y)
    case x if x > 0 => l1.map(_ * 2.0).reduceLeft((y, z) => y * n - z)
    case _ => Nil
```

- A `Any`
- B `List[Int] | Double`
- C `List[Any]`
- D `List[Nothing] | Any`
- E Some other type

Polymorphism

There are two principal forms of polymorphism:

- Subtyping
- Generics

In this part, we will look at their interactions.

We will particularly study:

- Type bound
- Variance

Example of `IntSet` Class

We use the example of an `IntSet` class, which implements a *set of Integers* as a binary tree, with two possible types of trees:

- A tree for the *empty set* : **Empty**
- A tree consisting of *an integer and two sub-trees* : **NonEmpty**

```
abstract class IntSet:
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean

Object Empty extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
```

Example of `IntSet` Class (ctd.)

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet:

  def contains(x: Int): Boolean =
    if x < elem then left.contains(x)
    else if x > elem then right.contains(x)
    else true

  def incl(x: Int): IntSet =
    if x < elem then NonEmpty(elem, left.incl(x), right)
    else if x > elem then NonEmpty(elem, left, right.incl(x))
  else this

end NonEmpty
```

Quiz

Consider the method `assertAllPos` which

- takes an `IntSet` (`Empty` sets or `NonEmpty` sets)
- returns the `NonEmpty` set if all its elements are positive, returns `Empty` if the input is `Empty`.
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`?

```
def assertAllPos(s: IntSet): ?
```

- A `NonEmpty`
- B `IntSet`
- C `Nothing`
- D Some other type

Type Bounds

To express that `assertAllPos` takes

- `Empty` sets to `Empty` sets, and
- `NonEmpty` sets to `NonEmpty` sets:

```
def assertAllPos[S <: IntSet](r: S): S = ...
```

Here, “`<: IntSet`” is an **upper bound** of the type parameter `S`:

It means that `S` can be instantiated only to types that conform to `IntSet`.

Generally, the notation

- `S <: T` means: *S is a subtype of T*, and
- `S >: T` means: *S is a supertype of T*, or *T is a subtype of S*.

Lower Bounds

You can also use a lower bound for a type variable.

example

```
[S >: NonEmpty]
```

introduces a type parameter `S` that can range only over **supertypes** of `NonEmpty`.

So `S` could be one of `NonEmpty`, `IntSet`, `AnyRef`, or `Any`.

We will see later where lower bounds are useful.

Mixed Bounds

Finally, it is also possible to mix a lower bound with an upper bound.

For instance,

```
[S >: NonEmpty <: IntSet]
```

would restrict `S` any type on the interval between `NonEmpty` and `IntSet`.

Covariance

There’s another interaction between subtyping and type parameters we need to consider. Given:

```
NonEmpty <: IntSet
```

is

```
List[NonEmpty] <: List[IntSet]    ?
```

Intuitively, this makes sense: A list of non-empty sets is a special case of a list of arbitrary sets.

We call types for which this relationship holds **covariant** because their subtyping relationship varies with the type parameter.

Does covariance make sense for all types, not just for `List`?

Arrays

For perspective, let's look at arrays in Java (and C#).

Reminder:

- An array of `T` elements is written `T[]` in Java.
- In Scala we use parameterized type syntax `Array[T]` to refer to the same type.

Arrays in Java are covariant, so one would have:

```
NonEmpty[] <: IntSet[]
```

Array Typing Problem

But covariant array typing causes problems.

To see why, consider the Java code below.

```java
NonEmpty[] a = new NonEmpty[]{
  new NonEmpty(1, new Empty(), new Empty())};
IntSet[] b = a;
b[0] = new Empty();
NonEmpty s = a[0];
```

It looks like we assigned in the last line an `Empty` set to a variable of type `NonEmpty`!

What went wrong?

The Liskov Substitution Principle

The following principle, stated by Barbara Liskov, tells us when a type can be a subtype of another.

> If `A <: B`, then everything one can to do with a value of type `B` one should also be able to do with a value of type `A`.

The actual definition Liskov used is a bit more formal. It says:

Let $q(x)$ be a property provable about objects x of type B. Then $q(y)$ should be provable for objects y of type A where $A <: B$.

Exercise

The problematic array example would be written as follows in Scala:

```scala
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty, Empty))
val b: Array[IntSet] = a
b(0) = Empty
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

- A A type error in line 1
- B A type error in line 2
- C A type error in line 3
- D A type error in line 4
- E A program that compiles and throws an exception at run-time
- F A program that compiles and runs without exception

Variance in Scala

Say `C[T]` is a parameterized type and `A`, `B` are types such that **`A <: B`**.

The possible relationships between `C[A]` and `C[B]` are:

- `C[A] <: C[B]` => C is *covariant*
- `C[A] >: C[B]` => C is *contravariant*
- Neither `C[A]` nor `C[B]` is a subtype of the other
  => C is *nonvariant*

Scala lets you declare the variance of a type by annotating the type parameter:

```
class C[+A] { ... }   // C is covariant
class C[-A] { ... }   // C is contravariant
class C[A]  { ... }   // C is nonvariant
```

Répond à la ***question 4*** *sur* *https://etc.ch/u5aY*

Quiz

Assume the following type hierarchy and two function types:

```scala
trait Fruit
class Apple extends Fruit
class Orange extends Fruit

type FtoO = Fruit => Orange
type AtoF = Apple => Fruit
```

According to the *Liskov Substitution Principle*, which of the following should be true?

- A FtoO <: AtoF
- B AtoF <: FtoO
- C FtoO and AtoF are unrelated.

Typing Rules for Functions

Generally, we have the following rule for subtyping between function types:

If `A2 <: A1` and `B1 <: B2`, then

```
A1 => B1  <:  A2 => B2
```

So functions are *contravariant* in their argument type(s) and *covariant* in their result type.

This leads to the following definition of the `Function1` trait:

```
package scala
  trait Function1[-T, +U]:
    def apply(x: T): U
```

Variance Checks

The Scala compiler will check that there are no problematic combinations when compiling a class with variance annotations.

Roughly,

- *covariant* type parameters can only appear in method results.
- *contravariant* type parameters can only appear in method parameters.
- *invariant* type parameters can appear anywhere.

The precise rules are a bit more involved, fortunately the Scala compiler performs them for us.

Variance-Checking the Function Trait

Let's have a look again at Function1:

```
trait Function1[-T, +U]:
  def apply(x: T): U
```

Here,

- T is contravariant and appears only as a method parameter type
- U is covariant and appears only as a method result type

So the method is checks out OK.

Making Classes Covariant

Sometimes, we have to put in a bit of work to make a class covariant.

Consider the trait `List` whose definition is simplified as follows:

```
trait List[+T]:

  def isEmpty = this match
    case Nil => true
    case => false

  override def toString = ...

case class ::[+T](head: T, tail: List[T]) extends List[T]
case object Nil extends List[Nothing]
```

Making Classes Covariant (ctd.)

Consider adding a `prepend` method to `List` which prepends a given element, yielding a new list.

A first implementation of `prepend` could look like this:

```
trait List[+T]:
  def prepend(elem: T): List[T] = ::(elem, this)
```

But that does not work!

Quiz

Why does the following code not type-check?

```
trait List[+T]:
  def prepend(elem: T): List[T] = ::(elem, this)
```

Possible answers:

- A prepend turns `List` into a mutable class.
- B prepend fails variance checking.
- C prepend's right-hand side contains a type error.

Prepend Violates LSP

Indeed, the compiler is right to throw out `List` with `prepend`, because it violates the Liskov Substitution Principle:

Here's something one can do with a list `xs` of type `List[Fruit]`:

```
xs.prepend(Orange)
```

But the same operation on a list `ys` of type `List[Apple]` would lead to a type error:

```
ys.prepend(Orange)
      ^ type mismatch
      required: Apple
      found   : Orange
```

Lower Bounds

But `prepend` is a natural method to have on immutable lists!

Q: How can we make it variance-correct?

We can use a *lower bound*:

```
def prepend [U >: T] (elem: U): List[U] = ::(elem, this)
```

This passes variance checks, because:

- *covariant* type parameters may appear in *lower bounds* of method type parameters
- *contravariant* type parameters may appear in *upper bounds*.

Répond à la ***question 6*** *sur https://etc.ch/u5aY*

Quiz

Assume `prepend` in trait `List` is implemented like this:

```
def prepend [U >: T] (elem: U): List[U] = ::(elem, this)
```

What is the result type of this function:

```
def f(xs: List[Apple], x: Orange) = xs.prepend(x)   ?
```

Possible answers:

- A does not type check
- B `List[Apple]`
- C `List[Orange]`
- D `List[Fruit]`
- E `List[Any]`

References

- Coursera course “Functional Programming Principles in Scala”, Odersky, M.
- Odersky, M., Spoon, L., & Venners, B. (2021). Programming in Scala (5th ed.). Artima Press. ISBN: 978-0-989-77450-1.
- Scala documentation on https://docs.scala-lang.org/