Variance

Principles of Functional Programming

Variance

You have seen the the previous session that some types should be covariant whereas others should not.

Roughly speaking, a type that accepts mutations of its elements should not be covariant.

But immutable types can be covariant, if some conditions on methods are met.

Definition of Variance

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

In general, there are *three* possible relationships between `C[A]` and `C[B]`:

`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*

Definition of Variance

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

In general, there are *three* possible relationships between `C[A]` and `C[B]`:

`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*

Exercise

Assume the following type hierarchy and two function types:

```
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?

- ○ `FtoO <: AtoF`
- ○ `AtoF <: FtoO`
- ○ A and B are unrelated.

Exercise

Assume the following type hierarchy and two function types:

```
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?

- ○ `FtoO <: AtoF`
- ○ `AtoF <: FtoO`
- ○ A and B 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 revised definition of the `Function1` trait:

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

Variance Checks

We have seen in the array example that the combination of covariance with certain operations is unsound.

In this case the problematic operation was the update operation on an array.

If we turn `Array` into a class, and `update` into a method, it would look like this:

```
class Array[+T]:
  def update(x: T) = ...
```

The problematic combination is

- the covariant type parameter `T`
- which appears in parameter position of the method `update`.

Variance Checks (2)

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.

Variance and Lists

Let's get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.

Variance and Lists

Let's get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.

Here are the essential modifications:

```
trait List[+T]
  ...
object Empty extends List[Nothing]
  ...
```

Idealized Lists

Here a definition of lists that implements all the cases we have seen so far:

```
trait List[+T]:

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

  override def toString =
    def recur(prefix: String, xs: List[T]): String = xs match
      case x :: xs1 => s"$prefix$x${recur(", ", xs1)}"
      case Nil => ")"
    recur("List(", this)
```

Idealized Lists(2)

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

extension [T](x: T) def :: (xs: List[T]): List[T] = ::(x, xs)

object List:
  def apply() = Nil
  def apply[T](x: T) = x :: Nil
  def apply[T](x1: T, x2: T) = x1 :: x2 :: Nil
  ...
```

(We'll see later how to do with just a single `apply` method using a *vararg* parameter.)

Making Classes Covariant

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

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!

Exercise

Why does the following code not type-check?

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

Possible answers:

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

Exercise

Why does the following code not type-check?

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

Possible answers:

- `prepend` turns `List` into a mutable class.
- `prepend` fails variance checking.
- `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(Apple)
           ^ type mismatch
           required: Apple
           found   : Orange
```

So, `List[Apple]` cannot be a subtype of `List[Fruit]`.

Lower Bounds

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

Q: How can we make it variance-correct?

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*.

Exercise

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:

0	does not type check
0	`List[Apple]`
0	`List[Orange]`
0	`List[Fruit]`
0	`List[Any]`

Exercise

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:

0	does not type check
0	`List[Apple]`
0	`List[Orange]`
0	`List[Fruit]`
0	`List[Any]`

Extension Methods

The need for a lower bound was essentially to decouple the new parameter of the class and the parameter of the newly created object. Using an extension method such as in `::` above, sidesteps the problem and is often simpler:

```
extension [T](x: T):
  def :: (xs: List[T]): List[T] = ::(x, xs)
```