Variance

Variance

- You have seen 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 are met
- Say C[T] is a parametrized 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

Based on the Liskov Substitution Principle, what is the relationship between the two types? – We check by replacing the type parameters from the second type in the first type; if they make sense, this means the types are covariant. So the answer is ***FtoO <: AtoF****.*

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 types and covariant in their result type
- This leads to the following revised definition of the Function1 trait:

```
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.

- 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 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)
- We can change that by making the list covariant
- Here are the essential modifications:

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

- The type list of Nothing really conveys the information that there's nothing in the list
- List of Nothing, on the one hand, says there's nothing in the lists, and on the other hand, ensures that that object is a subtype of any list type that the user might care to give

Idealized Lists

- Here is 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)

  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
```

Making Classes Covariant

- Consider adding a prepend method to List which prepends a given element, yielding a new list
- A fist implementation of prepend could be this:

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

But this doesn't work, because prepend fails variance checking (T is covariant and should not be the prepend's input parameter's type)

Prepend violates Liskov Substitution Principle

- Here is something one can do with a list xs of type List[Fruit]: ***xs.prepend(Orange)***
- But the same operation on a list of type List[Apple] would lead to a type error:
  ys.prepend(Orange) --> ***type mismatch***
  Required: Apple   ***Found: Orange***

Question: *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: *Having the* ***prepend*** *definition above, what is the result type of this function:*

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

The compiler will have to find a type U, which is a supertype of Apple and can take an Orange. The smallest such type is Fruit. The complier will instantiate my type U with fruit and that's the result type list of Fruit that I get back. So the answer is ***List[Fruit]****.*

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)