

# Comp 311

# Functional Programming

Nick Vrvilo, Two Sigma Investments  
Robert “Corky” Cartwright, Rice University

October 3, 2019

# Announcements

- Homework 2 is due two weeks from today
- Assignment description PDF on Piazza
- No provided “skeleton” code
- Simple interface (compilation/linking) check provided

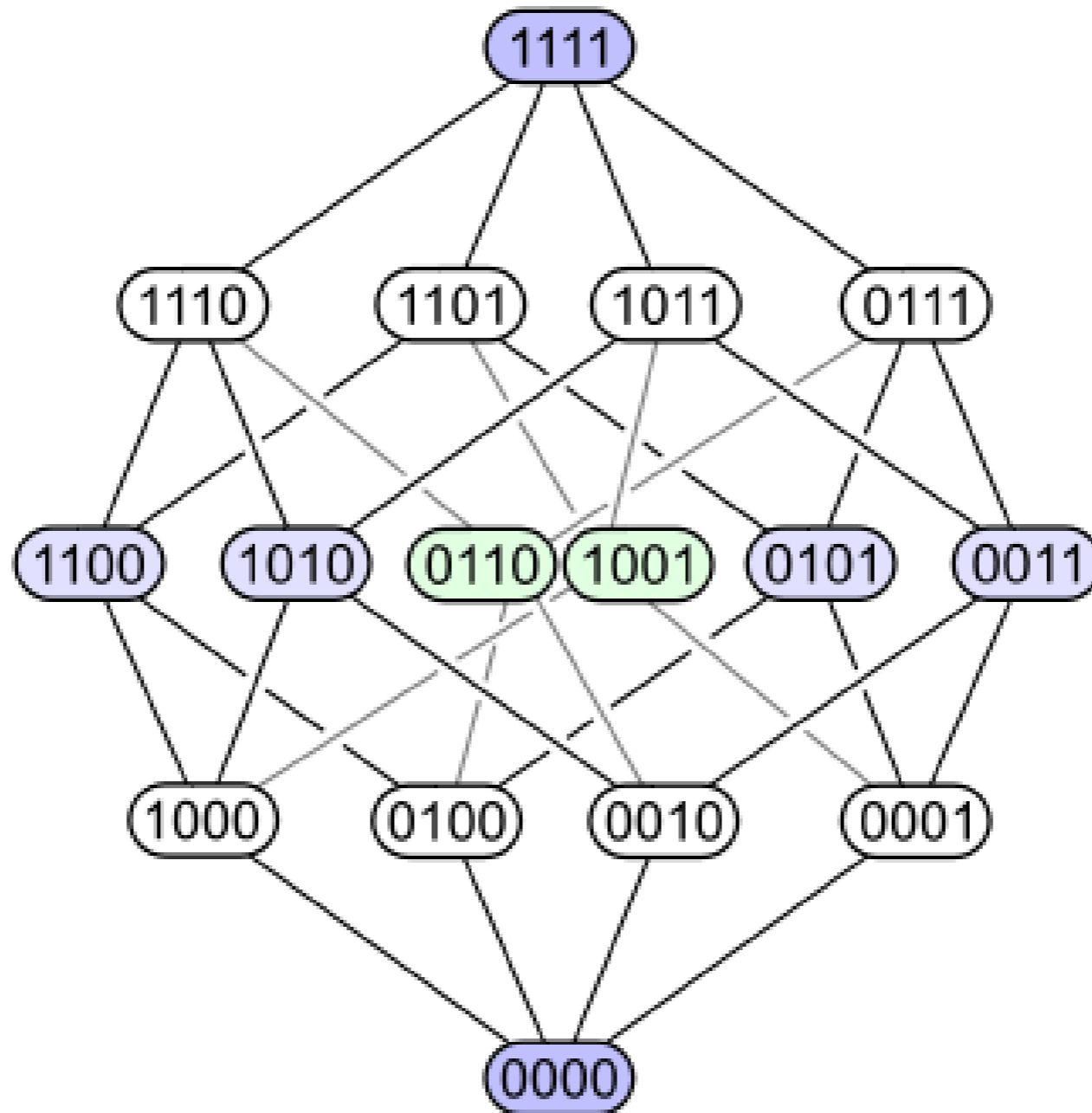
# Scala Type Hierarchy

# Type Hierarchies

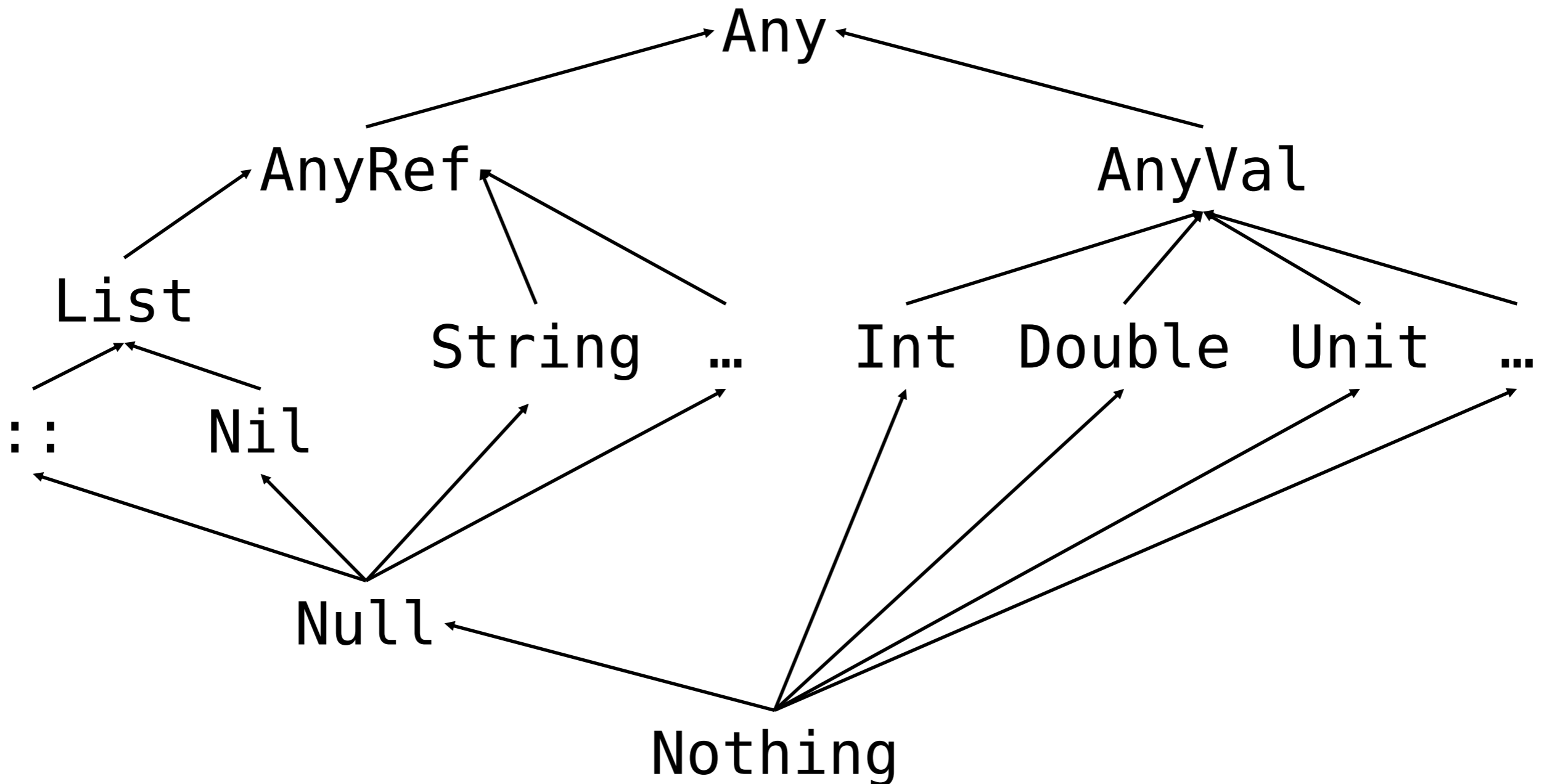
Inheritance (subclass / superclass relationships) form a *complete lattice* in the Scala type system:

- Each pair of classes has exactly one:
  - *Least upper-bound*
  - *Greatest lower-bound*
- The same applies to all value types

# Hasse Diagrams



# Scala Type Lattice



# Parametric Polymorphism (Parametric/Generic Types)

# Parametric Types

- We have defined two forms of lists: lists of ints and lists of shapes
- Many computations useful for one are useful for the other:
  - Map, reduce, filter, etc.
- It would be better to define lists and their operations once for all of these cases



# Parametric Types

- Higher-order functions take functions as arguments and return functions as results
- Likewise, *parametric types*, a.k.a., a *generic types*, takes types as arguments and return types as results

# Parametric Lists

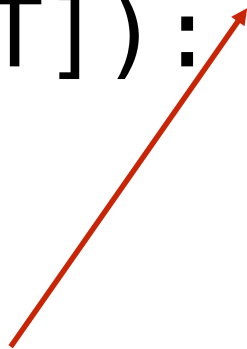
- Every application of this parametric type to an argument yields a new type:

```
abstract class List[T] {  
    def ++(ys: List[T]): List[T]  
}
```

# Parametric Lists

- Every application of this parametric type to an argument yields a new type:

```
abstract class List[T <: Any] {  
  def ++(ys: List[T]): List[T]  
}
```



- We augment the declarations of type parameters to permit an upper bound on all instantiations of a parameter
- By default, the bound is *Any*

# Syntax of Parametric Class

## Definitions

```
<modifiers> class C[T1 <: N, ..., TN <:
N] extends N {
    <ordinary class body>
}
```

- We denote type parameters as T1, T2, etc.
- We denote all other types with N, M, etc.

# Syntax of Parametric Class

## Definitions

```
<modifiers> class C[T1 <: N, ..., TN <:
N] extends N {
    <ordinary class body>
}
```

- Declared type parameters T1, ..., TN are in scope throughout the entire class definition, including:
  - The bounds of type parameters
  - The `extends` clause
- Object definitions must not be parametric

# Parametric Lists

- Every application of this parametric type yields a new type:

```
List[Int]  
List[String]  
List[List[Double]]  
etc.
```

# Parametric Lists

- Every application (a.k.a., *instantiation*) of this parametric type yields a new type:

```
abstract class List[T] {  
  def ++(ys: List[T]): List[T]  
}
```



Note that our parametric type can be instantiated with type parameters, including its own!

# Parametric Lists

```
case class Empty[S]() extends List[S] {  
  def ++(ys: List[S]) = ys  
}
```

```
case class Cons[T](head: T, tail: List[T]) extends List[T] {  
  def ++(ys: List[T]) = Cons[T](head, tail ++ ys)  
}
```

Our definition requires a separate type `Empty[S]` for every instantiation of `S`. Thus we must define `Empty` as a class rather than an object.



# Covariance

- Can one instantiation of a parametric type be a subtype of another?
- Currently our rules allow this only in the reflexive case:

`List[Int] <: List[Int] in E`

# Covariance

- It would be useful to allow some instantiations to be subtypes of another
- For example, we would like it to be the case that:

```
List[Int] <: List[Any]
```

# Covariance

- In general, we say that a parametric type  $C$  is covariant with respect to its type parameter  $S$  if:

$$S <: T \text{ in } E$$

implies

$$C[S] <: C[T] \text{ in } E$$

- We must be careful that such relationships do not break the soundness of our type system

# Covariance

- For a parametric type such as:

```
abstract class List[T <: Any] {  
    def ++(ys: List[T]): List[T]  
}
```

- And types  $S$  and  $T$ , such that  $S <: T$  in some environment  $E$ :
  - What must we check about the body of class `List` to allow for `List[S] <: List[T]` in  $E$ ?

# Covariance

- Consider instantiations for types `String` and `Any`:

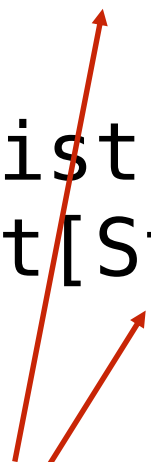
```
abstract class List[Any] {  
  def ++(ys: List[Any]): List[Any]  
}  
abstract class List[String] {  
  def ++(ys: List[String]): List[String]  
}
```

# Covariance

- If these were ordinary classes connected by an `extends` class:
  - We would need to ensure that the overriding definition of `++` in class `List[String]` was compatible with the overridden definition in `List[Any]`

# Covariance

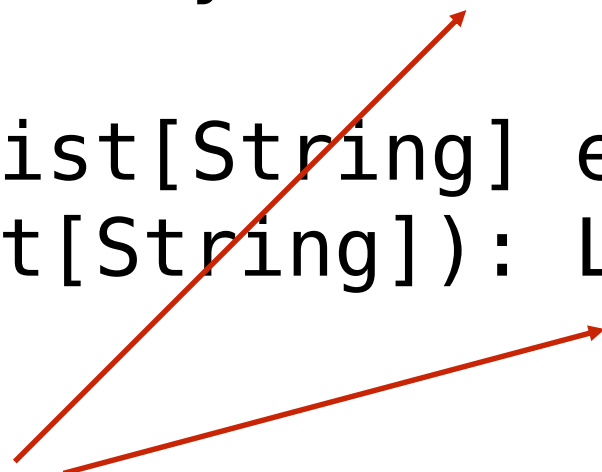
```
abstract class List[Any] {  
  def ++(ys: List[Any]): List[Any]  
}  
abstract class List[String] extends List[Any] {  
  def ++(ys: List[String]): List[String]  
}
```



But if `List[String] <: List[Any]` in E  
then this is not a valid override

# Covariance

```
abstract class List[Any] {  
  def ++(ys: List[Any]): List[Any]  
}  
abstract class List[String] extends List[Any] {  
  def ++(ys: List[String]): List[String]  
}
```



On the other hand, the return types  
are not problematic



# Covariance

- From our example, we can glean the following rule:
  - We allow a parametric class  $C$  to be covariant with respect to a type parameter  $T$  so long as  $T$  does not appear in the types of the method parameters of  $C$

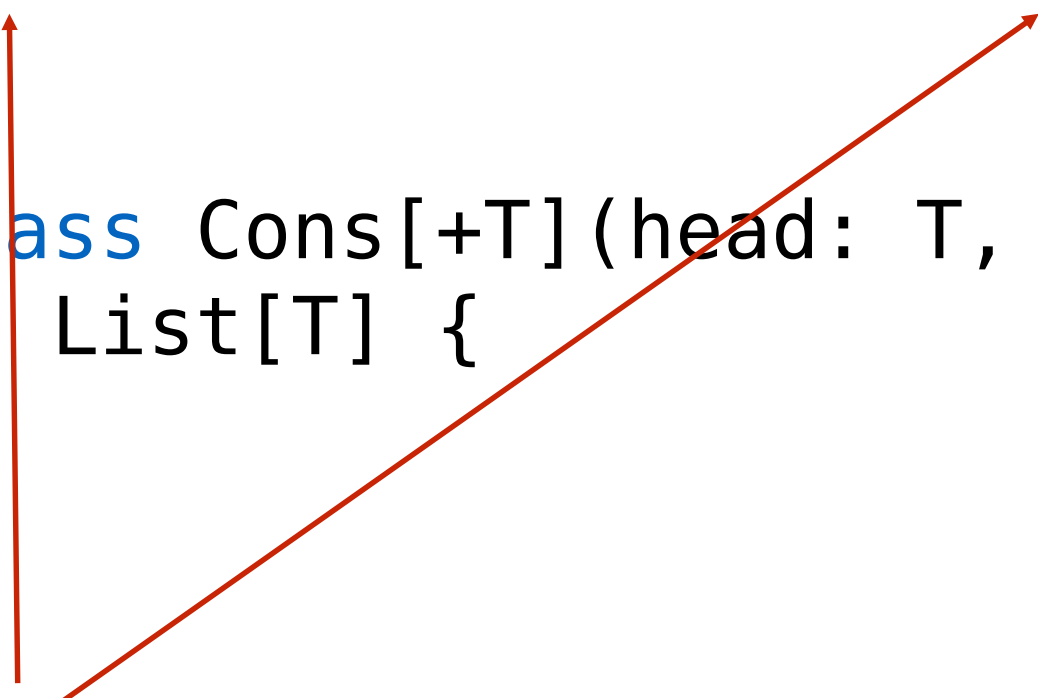
# Covariance

```
abstract class List[+T] {}
```

- We stipulate that a parametric type is covariant in a parameter  $T$  by prefixing a  $+$  at the definition of  $T$
- (We will return to our definition of `append` later)

# Covariance

```
case object Empty extends List[Nothing] {  
}  
  
case class Cons[+T](head: T, tail: List[T])  
extends List[T] {  
}
```



*Now we can define Empty as an object that extends the bottom of the List types*

# Covariance and Append

- The problem with our original declaration of `append` was that it was not general enough:
  - There is no reason to require that we always append lists of identical type
  - Really, we can append a `List[S]` for any supertype of our `List[T]`
  - The result will be of type `List[S]`

# Lower Bounds on Type Parameters

- Thus far, we have allowed type parameters to include upper bounds:

$$T <: S$$

- They can also include lower bounds:

$$T >: U$$

- Or they can include both:

$$T >: S <: U$$

# Parametric Functions

- Just as we can add type parameters to a class definition, we can also add them to a function definition
- The type parameters are in scope in the header and body of the function

# Covariance and Append

```
abstract class List[+T] {  
  def ++[S >: T](ys: List[S]): List[S]  
}  
  
case object Empty extends List[Nothing] {  
  def ++[S](ys: List[S]) = ys  
}  
  
case class Cons[+T](head: T, tail: List[T])  
extends List[T] {  
  def ++[S >: T](ys: List[S]) = Cons(head, tail ++ ys)  
}
```

# Map Revisited

```
abstract class List[+T] {  
  ...  
  def map[U](f: T => U): List[U]  
}
```

*Why is this occurrence of T acceptable?*





# We Consider Specific Instantiations

```
abstract class List[Any] {  
  ...  
  def map[U](f: Any => U): List[U]  
}  
abstract class List[String] {  
  ...  
  def map[U](f: String => U): List[U]  
}
```

*Then List[String] is an acceptable subtype of List[Any]  
provided that  $(String \Rightarrow U) \supseteq (Any \Rightarrow U)$   
which requires that  $String \leq Any$ .*

# Generalizing Our Rules

- In our example, type parameter `T` occurs as the parameter of an arrow type:
  - `(String => U) >: (Any => U)` in `E` provided:
    - `String <: Any` in `E`
    - `U <: U` in `E`
  - So subtype `List[String] <: List[Any]` is permitted

# Contravariance

- In general, we say that a parametric type  $C$  is contravariant with respect to its type parameter  $S$  if:

$$S <: T \text{ in } E$$

implies

$$C[T] <: C[S] \text{ in } E$$

- We must be careful that such relationships do not break the soundness of our type system

# Contravariance

- Syntactically, contravariant type parameter declarations are annotated with a minus sign:

```
case class F[-A,+B]
```

# Annotating Polarity

```
abstract class List[+T] {  
  def ++[S- >: T+](ys: List[S-]): List[S+]  
  def map[U-](f: T+ => U-): List[U+]  
}
```

# We Generalize Our Rules for Checking Variance As Follows

- Covariant type parameters (declared with +) are allowed to occur only in positive locations
- Type parameters with no annotation are allowed to be used in all locations
- Contravariant type parameters are allowed to occur only in negative locations

# An Example of How We Might Use Contravariant Type Parameters

```
abstract class Function1[-S,+T] {  
    def apply(x:S): T  
}
```

# Map Revisited

```
case object Empty extends List[Nothing] {  
  ...  
  def map[U](f: Nothing => U) = Empty  
}
```



# Map Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  ...
  def map[U](f: T => U) =
    Cons(f(head), tail.map(f))
}
```

# Syntactic Sugar: Currying

- Scala provides special syntax for defining a function that immediately returns another function:

```
def f(x0: T0, ..., xN: TN) = (y0: U0, ..., yM: UM) => expr
```

can be rewritten as:

```
def f(x0: T0, ..., xN: TN)(y0: U0, ..., yM: UM) = expr
```

- Defining a function in this way is called “currying”, after the computer scientist Haskell Curry

# Folding Revisited

```
abstract class List[+T] {  
  ...  
  def foldLeft[S](x: S)(f: (S, T) => S): S  
  def foldRight[S](x: S)(f: (T, S) => S): S  
}
```

*Note that these functions are curried*



# Folding Revisited

```
case object Empty extends List[Nothing] {  
  ...  
  def foldLeft[S](x: S)(f: (S, T) => S) = x  
  def foldRight[S](x: S)(f: (T, S) => S) = x  
}
```

# Folding Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  ...
  def foldLeft[S](x: S)(f: (S, T) => S) =
    tail.foldLeft(f(x, head))(f)

  def foldRight[S](x: S)(f: (T, S) => S) =
    f(head, tail.foldRight(x)(f))
}
}
```

Note that *foldLeft* is tail-recursive, but *foldRight* is not; therefore, *foldLeft* is usually preferred.

# Fold Revisited

```
def foldLeft[S >: T](x: S)(f: (S, S) => S) =  
  tail.foldLeft(f(x, head), f)
```

```
Cons(1, Cons(2, Cons(3, Empty))).foldLeft(0)(_+_ ) ↪  
Cons(2, Cons(3, Empty)).foldLeft(0 + 1, _+_ ) ↪  
Cons(2, Cons(3, Empty)).foldLeft(1, _+_ ) ↪  
Cons(3, Empty).foldLeft(1 + 2, _+_ ) ↪  
Cons(3, Empty).foldLeft(3, _+_ ) ↪  
Empty.foldLeft(3 + 3, _+_ ) ↪  
Empty.foldLeft(6, _+_ ) ↪  
6
```

# Folding Revisited

```
def foldRight[S >: T](x: S)(f: (S, S) => S) =  
  f(tail.foldRight(x, f), head)
```

```
Cons(1, Cons(2, Cons(3, Empty))).foldRight(0) (_+_ ) ⇨  
Cons(2, Cons(3, Empty)).foldRight(0, _+_ ) + 1 ⇨  
Cons(3, Empty).foldLeft(0, _+_ ) + 2 + 1 ⇨  
Empty.foldLeft(0, _+_ ) + 3 + 2 + 1 ⇨  
0 + 3 + 2 + 1 ⇨  
6
```

# Reduce Revisited

```
abstract class List[+T] {  
  ...  
  def reduce[S >: T](f: (S, S) => S): S  
}
```

*We can elide a zero element for the reduction  
provided that the list is non-empty*



# Reduce Revisited

```
case object Empty extends List[Nothing] {  
  ...  
  def reduce[S](f: (S, S) => S) =  
    throw new UnsupportedOperationException  
}
```

# Reduce Revisited

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  ...
  def reduce[S >: T](f: (S, S) => S) =
    tail.foldLeft[S](head)(f)
}
```



*We explicitly instantiate the type parameter to foldLeft. Without this, type inference will instantiate the type parameter based on the static type of head (which is T) and then signal an error that f is not of type (T, T) => T.*

# Forall and Exists

```
abstract class List[+T] {  
  ...  
  def forall(p: T => Boolean) =  
    map(p).foldLeft(true, _&&_)  
  
  def exists(p: T => Boolean) =  
    map(p).foldLeft(false, _||_)  
}
```

# Length

```
abstract class List[+T] { ...  
  def length: Int  
}  
case object Empty extends List[Nothing] { ...  
  def length = 0  
}  
case class Cons[+T](head: T, tail: List[T])  
extends List[T] { ...  
  def length = map((_:T) => 1).reduce(_+_)  
}
```

*In what real contexts could we justify this definition of length?*

# Pointwise Addition

```
def pointwiseAdd(xs: List[Int], ys: List[Int]): List[Int] = {  
  require (xs.length == ys.length)  
  
  (xs, ys) match {  
    case (Empty, Empty) => Empty  
    case (Cons(x1, xs1), Cons(y1, ys1)) =>  
      Cons(x1 + y1, pointwiseAdd(xs1, ys1))  
  }  
}
```

# Generalizing to ZipWith

```
// in class List:  
def zipWith[U,V](f: (T, U) => V)(that: List[U]): List[V] = {  
  require (this.length == that.length)  
  
  (this, that) match {  
    case (Empty, Empty) => Empty  
    case (Cons(x1,xs1), Cons(y1,ys1)) =>  
      Cons(f(x1,y1), xs1.zipWith(f)(ys1))  
  }  
}
```

# Defining The Zip Function

```
// in class List:  
def zip[U](that: List[U]) = zipWith((_ , _ : U))(that)
```

# Defining Flatten

```
def flatten[S](xs: List[List[S]]) = {  
  xs.foldLeft(Empty)(_ ++ _)  
}
```

*Because the specific type of List needed,  
we define as a top level function*



# Defining FlatMap

```
abstract class List[+T] {  
  ...  
  def flatMap[S](f: T => List[S]) =  
    flatten(this.map(f))  
}  
}
```

*In contrast to flatten, our flatMap function  
can be defined on arbitrary lists*

# Defining FlatMap

- These definitions suggest that flatMap is the best thought of as the more primitive notion
- We can define flatMap as a method on lists directly and then define flatten in terms of it

# Defining FlatMap

```
abstract class List[+T] { ...  
  def flatMap[S](f: Nothing => List[S]): List[S]  
}
```

```
case object Empty extends List[Nothing] { ...  
  def flatMap[S](f: Nothing => List[S]) = Empty  
}
```

```
case class Cons[+T](head: T, tail: List[T])  
extends List[T] { ...  
  def flatMap[S](f: T => List[S]) =  
    f(head) ++ tail.flatMap(f)  
}
```

# Defining Filter

```
abstract class List[+T] {  
  ...  
  def filter[U](p: T => Boolean): List[T]  
}
```

# Defining Filter

```
case object Empty extends List[Nothing] {  
  ...  
  def filter[U](p: T => Boolean) = Empty  
}
```

# Defining Filter

```
case class Cons[+T](head: T, tail: List[T])
extends List[T] {
  ...
  def filter[U](p: T => Boolean) = {
    if (p(head)) Cons(head, tail.filter(p))
    else tail.filter(p)
  }
}
```

# For Expressions

- As with all expressions, for expressions reduce to a value
- The value reduced to is a collection
- The type of collection produced depends on the types of collections iterated over
- Each iteration produces a value to include in the resulting collection

# Many Maps and Filters Can Be Expressed Using For Expressions

```
for (x <- xs) yield square(x) + 1
```



# Many Maps and Filters Can Be Expressed Using For Expressions

```
for (x <- xs) yield square(x) + 1
```

*We call this a generator*

# Many Maps and Filters Can Be Expressed Using For Expressions

`for` *clauses* `yield` *body*

# Many Maps and Filters Can Be Expressed Using For Expressions

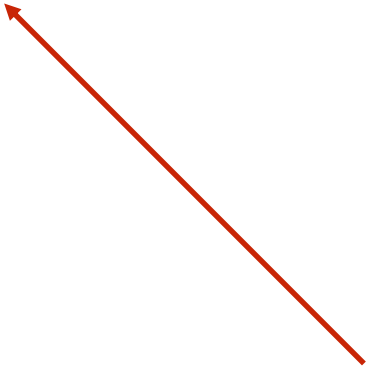
```
for (i <- 1 to 10) yield square(i) + 1
```



*Includes 10  
(closed interval)*

# Many Maps and Filters Can Be Expressed Using For Expressions

```
for (i <- 0 until 10) yield square(i) + 1
```

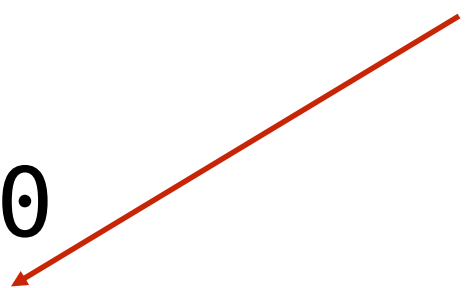


*Does not include 10  
(half-open interval)*

# Many Maps and Filters Can Be Expressed Using For Expressions

```
for {  
  i <- 0 until 10  
  if i % 2 == 0  
} yield square(i) + 1
```

*Predicate filter  
for even numbers*



*Use curly braces in place of parents to  
allow for multiple expression clauses*



# Many Maps and Filters Can Be Expressed Using For Expressions

```
for (i <- 0 until 10 by 2)  
  yield square(i) + 1
```

*Specifying a “step” for the range is another way to get the even numbers*

# Many Maps and Filters Can Be Expressed Using For Expressions

```
// BAD FORM  
for (i <- 0 until xs.length)  
  yield square(xs.nth(i)) + 1
```

# Many Maps and Filters Can Be Expressed Using For Expressions

```
// Write this instead  
for (x <- xs)  
  yield square(x) + 1
```



# Takeaways

- Variance:
  - Consumed values are **contra**-variant (e.g., function arguments)
  - Produced values are **co**-variant (e.g., function return values)
- Scala's *for*-comprehensions are a concise short-hand for composing monadic operations: *flatMap*, *map*, *filter*