Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert "Corky" Cartwright, Rice University

September 27, 2018

Announcements

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

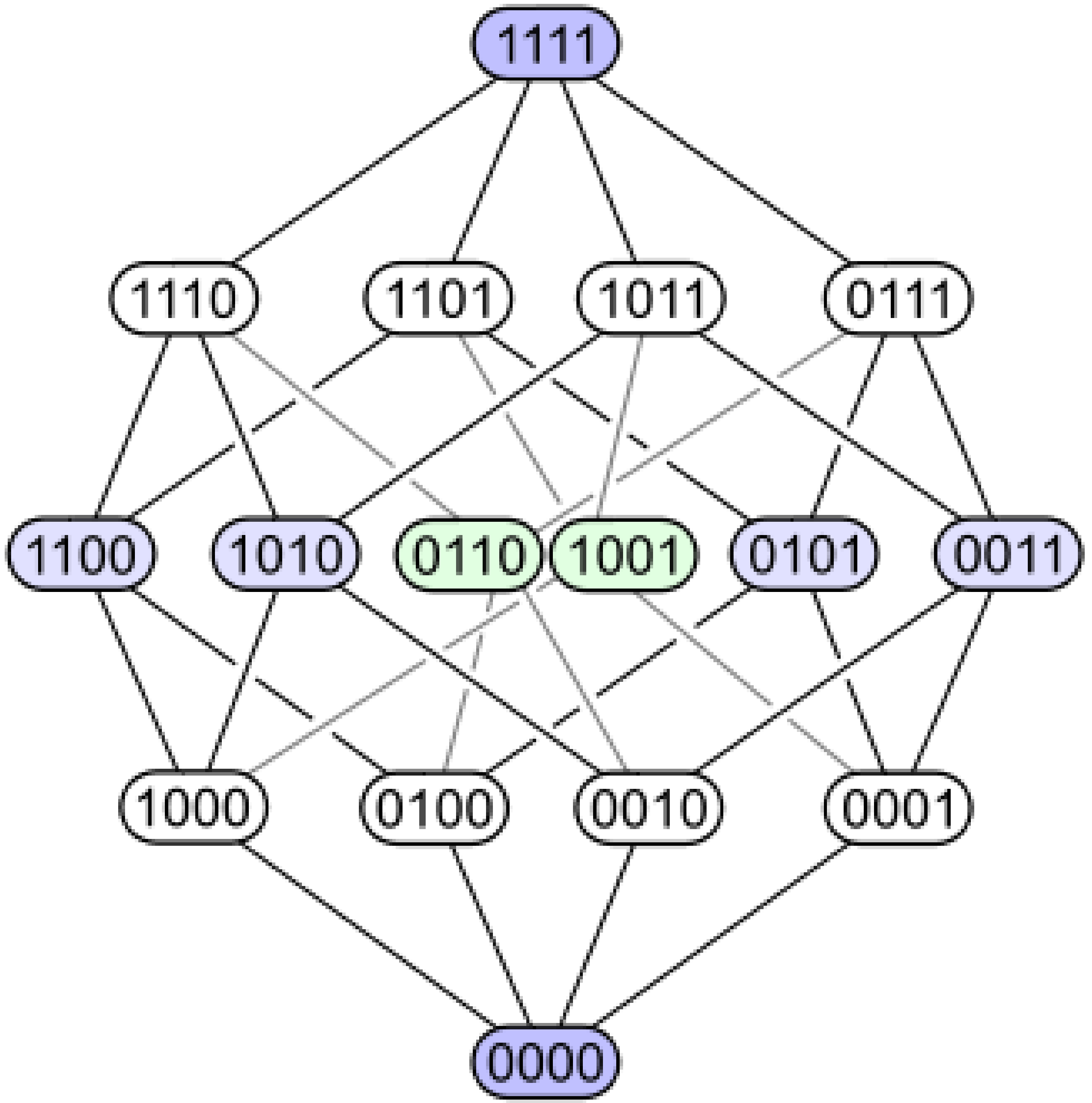
Scala Type Hierarchy

Type Hierarchies

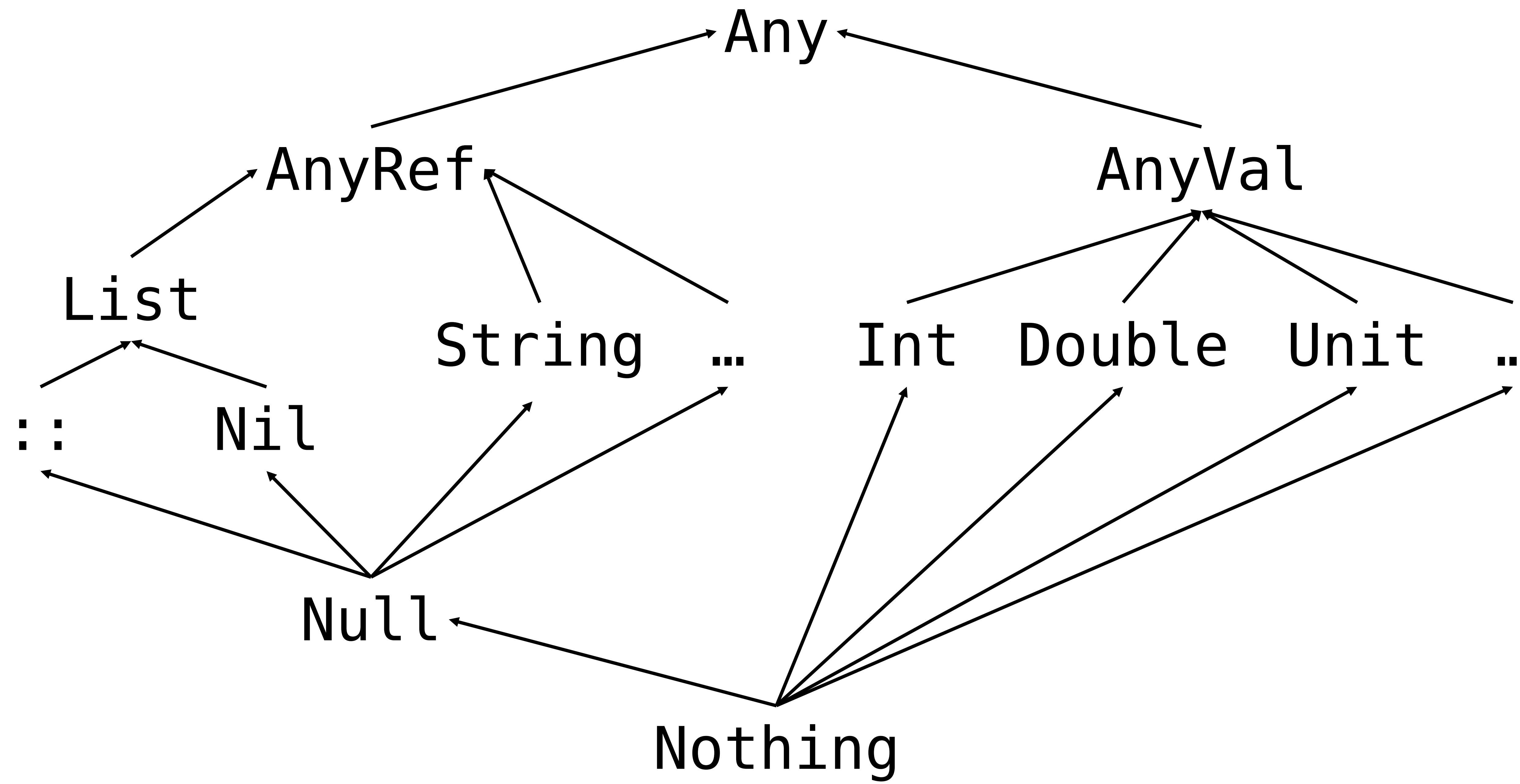
Inheritance (subclass / superclass relationships) form a *complete lattice* in the Scala type system:

- 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 in E
```

implies

```
C[S] <: C[T] 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

```scala
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]
}
```