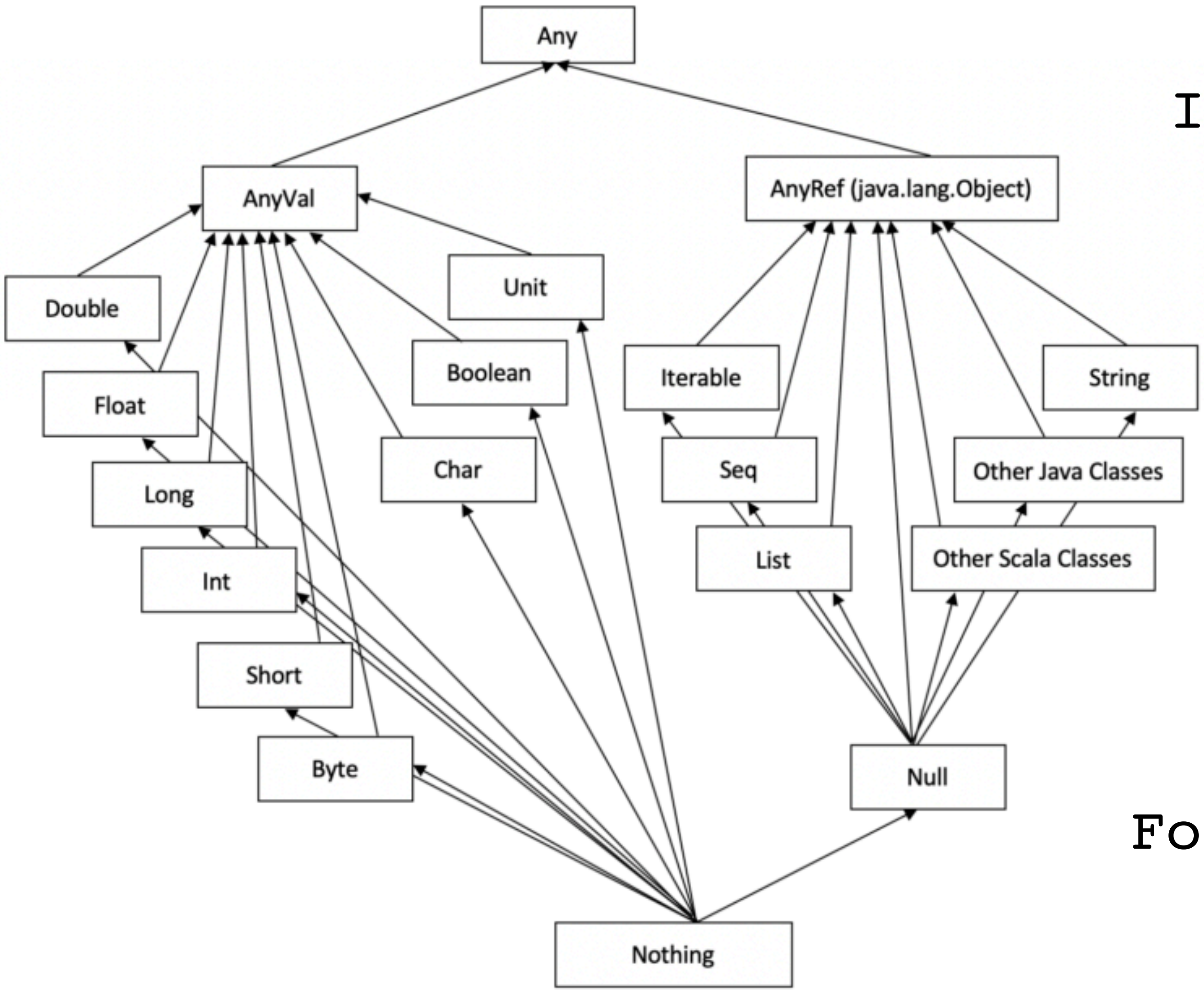
Advanced Subtyping

Nancy Hitschfeld
Matías Toro

Subtyping



`Int <: AnyVal`

`Int <: Any`

`Null <: String`

`String <: AnyRef`

`Foo <: AnyRef`

Nominal Subtyping

```
class A
trait T
class B extends A with T
class C extends B
```

C <: B

B <: A

B <: T

C <: A

C <: T

```
trait T1
trait T2 extends T1
trait T3 extends T1
trait T4 extends T2 with T3
```

T4 <: T2

T4 <: T3

T3 <: T1

T2 <: T1

T4 <: T1

Subtyping and Liskov

```
class A{
  def foo = 1
}
class B extends A{
  def bar = 2
}
```

```
def foo(a: A) = a.foo
foo(new B())
```

Ok, because B <: A

```
def foo(a: B) = a.bar
foo(new A())
```

Wrong, because B $\not<:$ A

Substructural Subtyping

In Scala, functions are *first-class citizen*.

They can be stored in variables and passed as arguments to other functions.

```
def id(x: Int): Int = x
def cube(x: Int): Int = x * x * x
def factorial(x: Int): Int = …
```

```
def sum(f: (Int) => Int, a: Int, b: Int): Int =
  if (a > b) 0
  else f(a) + sum(f, a + 1, b)
```

```
sum(id, 1, 10) // suma normal
sum(cube, 1, 10) // suma de cubos
sum(factorial, 1, 10) // suma de factoriales
```

Substructural Subtyping

In Scala, functions are *first-class citizen*.

They can be stored in variables and passed as arguments to other functions.

What about subtyping between function types?

```
(Any) => Animal ?? (Animal) => Animal

(Animal) => Cat ?? (Animal) => Animal

(Any) => Cat ?? (Animal) => Animal
```

Substructural Subtyping

Let’s consider the following example

```
def foo(f: (Animal) => Animal) = {
  f(new Dog("Bond")).talk()
}
def present(a: Animal): Animal = {
  a.talk();a
}
def doMiau(d: Cat): Cat = {
  d.meow();d
}
def objectify(a: Animal): Object = {
  new Object()
}
def doPrint(o: Object): Dog = {
  print(o.toString);
  new Dog("Bond")
}
```

```
foo(present)
```

```
foo(doMiau)
```

```
foo(objectify)
```

```
foo(doPrint)
```

Substructural Subtyping

Let’s consider the following example

```
def foo(f: (Animal) => Animal) = {
  f(new Dog("Bond")).talk()
}
def present(a: Animal): Animal = {
  a.talk();a
}
def doMiau(d: Cat): Cat = {
  d.meow();d
}
def objectify(a: Animal): Object = {
  new Object()
}
def doPrint(o: Object): Dog = {
  print(o.toString);
  new Dog("Bond")
}
```

```
foo(present)

-> present(new Dog(“Bond")).talk()

//dog = new Dog(“Bond")

-> present(dog).talk()

-> {dog.talk();dog}.talk()

-> {dog}.talk() // “guau!"

-> dog.talk()

// "guau!"
```

Substructural Subtyping

Let's consider the following example

```
def foo(f: (Animal) => Animal) = {
  f(new Dog("Bond")).talk()
}
def present(a: Animal): Animal = {
  a.talk();a
}
def doMiau(d: Cat): Cat = {
  d.meow();d
}
def objectify(a: Animal): Object = {
  new Object()
}
def doPrint(o: Object): Dog = {
  print(o.toString);
  new Dog("Bond")
}
```

```
foo(doMiau)

-> doMiau(new Dog("Bond")).talk()

//dog = new Dog("Bond")

-> doMiau(dog).talk()

-> {dog.meow();dog}.talk()

//error!
```

Substructural Subtyping

Let’s consider the following example

```
def foo(f: (Animal) => Animal) = {
  f(new Dog("Bond")).talk()
}
def present(a: Animal): Animal = {
  a.talk();a
}
def doMiau(d: Cat): Cat = {
  d.meow();d
}
def objectify(a: Animal): Object = {
  new Object()
}
def doPrint(o: Object): Dog = {
  print(o.toString);
  new Dog("Bond")
}
```

```
foo(objectify)

-> objectify(new Dog("Bond")).talk()

//dog = new Dog("Bond")

-> objectify(dog).talk()

-> {new Object()}.talk()

-> new Object().talk()

//error!
```

Substructural Subtyping

Let’s consider the following example

```
def foo(f: (Animal) => Animal) = {
  f(new Dog("Bond")).talk()
}
def present(a: Animal): Animal = {
  a.talk();a
}
def doMiau(d: Cat): Cat = {
  d.meow();d
}
def objectify(a: Animal): Object = {
  new Object()
}
def doPrint(o: Object): Dog = {
  print(o.toString);
  new Dog("Bond")
}
```

```
foo(doPrint)

-> doPrint(new Dog("Bond")).talk()

//dog = new Dog("Bond")

-> doPrint(dog).talk()

-> {print(dog.toString);

    new Dog("Bond")}.talk()

-> // Dog("Bond")

-> {new Dog("Bond")}.talk()

-> new Dog("Bond").talk()

//guau!
```

Substructural Subtyping

Let’s consider the following example

```
def foo(f: (Animal) => Animal) = {
  f(new Dog("Bond")).talk()
}
def present(a: Animal): Animal = {
  a.talk();a
}
def doMiau(d: Cat): Cat = {
  d.meow();d
}
def objectify(a: Animal): Object = {
  new Object()
}
def doPrint(o: Object): Dog = {
  print(o.toString);
  new Dog("Bond")
}
```

foo(present) ✓

foo(doMiau) ✗

foo(objectify) ✗

foo(doPrint) ✓

Substructural Subtyping

In Scala, functions are *first-class citizen*.

They can be stored in variables and passed as arguments to other functions.

Functions are **contravariant** in the **domain**, and **covariant** in the **codomain**.

$$(T_1 \Rightarrow T_2 <: T_3 \Rightarrow T_4) \iff T_3 <: T_1 \land T_2 <: T_4$$

```
(Any) => Animal <: (Animal) => Animal
(Animal) => Cat <: (Animal) => Animal
(Any) => Cat <: (Animal) => Animal
```

Substructural Subtyping

Tuples are **covariants!**

$$(T_1, T_2) <: (T_3, T_4) \iff T_1 <: T_3 \land T_2 <: T_4$$

```
(Animal, Animal) <: (Any,Animal)

(Animal, Cat) <: (Animal, Animal)

(Animal, Cat) <: (Any, Animal)
```

Subtyping and Generics

What about subtyping?

`List[Dog]`

`List[Cat]`

vs.

`List[Animal]`

We'll come back to this

Java Arrays

Long long time ago in **Java**, before generics, there were polymorphic arrays (using Scala like syntax: this doesn't compile in Scala!!)

```
class A{
    def main(args: Array[String]): Unit = {
        val array1: TreeNode[] = {t1, t2};
        val array2: TreeNode[] = new TreeNode[2]
        array2[0] = t1;
        array2[1] = t2;
    }
}
```

Array

Java Arrays

What about subtyping?

Given A <: B, if T[A] <: T[B] then T is **covariant** in its parameter

Given A <: B, if T[B] <: T[A] then T is **contravariant** in its parameter

Given A <: B, if T[B] is not related to T[A] then T is **invariant** in its parameter

```
Integer[] <: Number[] <: Object[]
```

Java Arrays are **covariant**!!

Java Arrays

Java Arrays are covariant.

`Dog[] <: Animal[]`   `Cat[] <: Animal[]`

```
def present(as: Animal[]){
    …
}

val dogs: Dog[] = {new Dog("Alex"), new Dog("Bond")}
present(dogs)
dogs[0].bark()
```

Super nice, right?

Arrays

It comes at a price…

```
Dog[] <: Animal[]          Cat[] <: Animal[]
```

```
def present(as: Animal[]){
    as[0] = new Cat("SneakAttack")
}

val dogs: Dog[] = {new Dog("Alex"), new Dog("Bond")}
present(dogs)
dogs[0].bark()
```

Exception in thread "main" java.lang.ArrayStoreException: java.lang.String

Generics and subtyping

What about subtyping for inmutable lists?

`List[Dog]`

`List[Cat]`

vs.

`List[Animal]`

Generics and subtyping

They can be covariant.

Shiba <: Dog <: Animal

```
def sort(dogs: List[Dog]): List[Dog] = {
  dogs
}
val shibas: List[Shiba] =
         List(new Shiba("Alex"), new Shiba(“Bond"))
val dogs: List[Dog] = sort(shibas)
dogs(0).bark()
```

A Shiba knows how to bark (Liskov)

Generics and subtyping

They cannot be contravariant.

Shiba <: Dog <: Animal

```
def sort (shibas: List[Shiba]): List[Shiba] = {
  shibas
}
val dogs: List[Dog] =
          List(new Dog("Alex"), new Dog("Bond"))
val shibas: List[Shiba] = sort(dogs)
shibas(0).wow()
```

A dog doesn't know how to *wow*

Generics and subtyping

What about subtyping for mutable lists?

`ListBuffer[Dog]`

`ListBuffer[Cat]`

vs.

`ListBuffer[Animal]`

Generics and subtyping

They cannot be covariant.

```
def present(as: ListBuffer[Animal]) {
  as(0) = new Cat("Backstab")
}
val dogs: ListBuffer[Dog] =
    ListBuffer(new Dog("Alex"), new Dog("Bond"))
present(dogs)
dogs(0).bark()
```

We would get an error here.

Generics and subtyping

They cannot be contravariant.

Shiba <: Dog <: Animal

```
def sort(shibas: ListBuffer[Shiba]): ListBuffer[Shiba] = {
  shibas
}
val dogs: ListBuffer[Dog] =
      ListBuffer(new Dog("Alex"), new Dog("Bond"))
val shibas: ListBuffer[Shiba] = sort(dogs)
shibas(0).wow()
```

A dog does not know how to *wow*

Generics and subtyping

What about subtyping?

`ListBuffer[Dog]`

`ListBuffer[Cat]`

vs.

`ListBuffer[Animal]`

We can only get stuff from covariant list, and
we can only put stuff in contravariant lists.

Generics are **invariant**!

Generics are invariant by default

But the following program does not compile :(

```
class Cell[T](var element: T) {
  def getContent(): T = {element}
  def setContent(e: T) = {element = e}
}
```

```
def present(as: Cell[Animal]) {
  as.getContent().talk()
}
val cell = new Cell[Dog](new Dog("Alex"))
present(cell)
```

Type mismatch.
Required: Cell[Animal]
Found: Cell[Dog]

Bounds to the rescue (again!)

Upper and Lower bounds

```
def present[T<:Animal](as: Cell[T]){…}
def present[T>:Animal](as: Cell[T]){…}
```

```
def present[T<:Animal](as: Cell[T]) {
  as.getContent().talk()
}
val cell = new Cell[Dog](new Dog("Alex"))
present[Dog](cell)
```

More bounds!

```
def present[T <: Animal with Foo](…){…}
```

```
def present[T <: Animal with Foo with Bar](…){…}
```

Get-put principle

- Use covariance for methods which return a generic type
- Use contravariance for methods which take a generic type
- Use invariance for methods which both accept and return a generic type

```
def getPut[T<:Animal](cell: Cell[T]) = {
  val a: Animal = cell.getContent()
  cell.setContent(new Dog("Bond"))
}
val cell: Cell[Dog] = new Cell();
cell.setContent(new Dog("Alexander"))
getPut(cell)
```

It doesn't compile
(The cell may be
Cell[Cat])

DCC
CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Get-put principle

- Use covariance for methods which return a generic type
- Use contravariance for methods which take a generic type
- Use invariance for methods which both accept and return a generic type

```
def getPut[T>:Animal](cell: Cell[T]) = {
  val a: Animal = cell.getContent()
  cell.setContent(new Dog("Bond"))
}
val cell: Cell[Dog] = new Cell();
cell.setContent(new Dog("Alexander"))
getPut(cell)
```

It doesn't compile (The cell may be Cell[Any])

Advanced control of variance

Scala supports declaration-site variance.

```
class Cell[+A] // A covariant class
class Cell[-A] // A contravariant class
class Cell[A] // An invariant class
```

Advanced control of variance

Scala supports declaration-site variance.

```
Cell[Cat] <: Cell[Animal]
```

```
class Cell[+T](var element: T) {
  def getContent(): T = {element}
  def setContent(e: T) ={ … }
}
```

Covariant type T occurs in contravariant position

```
class Cell[-T](var element: T) {
  def getContent(): T = {element}
  def setContent(e: T) ={ … }
}
```

Contravariant type T occurs in covariant position

```
Cell[Animal] <: Cell[Cat]
```

License