Scala Days

Polymorphic Function Types in Scala 3

Guillaume Martres

Arteris IP / Scala Center

About me

- 2016-2022: PhD at **EPFL**
- 2022-2023: **Scala Center**
- Since last week: **Arteris IP** + 20% time with the **Scala Center**
  - Arteris develops *interconnects* for System-on-Chips.
  - Hiring Scala engineers (in Nice, France) to research Domain Specific Languages for hardware design!

Table of Contents

1. Methods versus Functions
2. Handling polymorphism
3. Compiler Implementation
4. Detailed Examples
5. The (Possible) Future

Table of Contents

1. Methods versus Functions
2. Handling polymorphism
3. Compiler Implementation
4. Detailed Examples
5. The (Possible) Future

What is a method?

A **method** is a member of a scope (class, object, ...) declared using def:

```
def conv(x: Int): String = x.toString
```

What is a function?

A **value** is an instance of a type. The type determines how we can use the value.

What is a function?

A **value** is an instance of a type. The type determines how we can use the value.

In particular, a **function value** is an instance of a **function type**, for example:

```
val f: Int => String = ...
```

What is a function?

A **value** is an instance of a type. The type determines how we can use the value.

In particular, a **function value** is an instance of a **function type**, for example:

```
val f: Int => String = ...
```

The type `Int => String` is a short-hand for `scala.Function1[Int, String]`:

```
trait Function1[-T, +R]:
  def apply(x: T): R
```

What is a function?

A **value** is an instance of a type. The type determines how we can use the value.

In particular, a **function value** is an instance of a **function type**, for example:

```
val f: Int => String = ...
```

The type `Int => String` is a short-hand for `scala.Function1[Int, String]`:

```
trait Function1[-T, +R]:
  def apply(x: T): R
```

If `f` is a value, then `f(1)` expands to `f.apply(1)`

What is a lambda?

A **lambda** is a convenient way to create an instance of a function type:

```
(x: Int) => x + 1
```

is equivalent to:

```
new Function1[Int, Int]:
  def apply(x: Int): Int = x + 1
```

What is a lambda?

A **lambda** is a convenient way to create an instance of a function type:

```
(x: Int) => x + 1
```

is equivalent to:

```
new Function1[Int, Int]:
  def apply(x: Int): Int = x + 1
```

... which itself expands to:

```
class anon() extends Function1[Int, Int]:
  def apply(x: Int): Int = x + 1
new anon()
```

Method references

A reference to a method is not a value, but it can be automatically converted into one:

```
def inc(x: Int) = x + 1
List(1,2,3).map(inc)
```

Method references

A reference to a method is not a value, but it can be automatically converted into one:

```scala
def inc(x: Int) = x + 1
List(1,2,3).map(x => inc(x))
```

Method references

A reference to a method is not a value, but it can be automatically converted into one:

```
def inc(x: Int) = x + 1
List(1,2,3).map(x => inc(x))
```

This process is called eta-expansion.

Table of Contents

1. Methods versus Functions

2. Handling polymorphism

3. Compiler Implementation

4. Detailed Examples

5. The (Possible) Future

The missing square

	Method	**Function**

	`def m(x: Int): List[Int] =`
	`  List(x)`
		`val f: Int => List[Int] =`
		`  x => List(x)`

The missing square

	Method	**Function**

Monomorphic	`def m(x: Int): List[Int] =`
	`  List(x)`
		`val f: Int => List[Int] =`
		`  x => List(x)`

The missing square

	Method	**Function**

Monomorphic	`def m(x: Int): List[Int] =`
	`List(x)`
		`val f: Int => List[Int] =`
		`x => List(x)`

Polymorphic	`def m[T](x: T): List[T] =`
	`List(x)`
		❓

Inventing polymorphic functions

```
def m[T](x: T): List[T] = List(x)
```

```
val f: ... = ... List(x) ...
```

Inventing polymorphic functions

```scala
def m[T](x: T): List[T] = List(x)
```

```scala
val f: ... = ... List(x) ...
```

Inventing polymorphic functions

```scala
def m[T](x: T): List[T] = List(x)
```

```scala
val f: ... = ... List(x) ...
```

The type of `f` needs to have a polymorphic method `apply` as a member so we can call:

```scala
f[Int](1) == List(1)
```

Inventing polymorphic functions

```
def m[T](x: T): List[T] = List(x)
```

```
val f: ... = ... List(x) ...
```

The type of f needs to have a polymorphic method apply as a member so we can call:

```
f.apply[Int](1) == List(1)
```

Manual encoding

```
trait MkList:
  def apply[T](x: T): List[T]

val f: MkList = new MkList:
  def apply[T](x: T): List[T] = List(x)
```

This works, but it requires creating a new trait each time we need a polymorphic function with different parameters.

What if we could use a lambda?

```
def m[T](x: T): List[T] = List[T](x)

val f: T => List[T] =
   (x: T) => List[T](x)
```

What if we could use a lambda?

```
def m[T](x: T): List[T] = List[T](x)

val f: T => List[T] =
  (x: T) => List[T](x)
```

What if we could use a lambda?

```
def m[T](x: T): List[T] = List[T](x)

val f: [T] => T => List[T] =
  [T] => (x: T) => List[T](x)
```

What if we could use a lambda?

```
def m[T](x: T): List[T] = List[T](x)

val f: [T] => T => List[T] =
  [T] => (x: T) => List[T](x)
```

`f` is a **polymorphic function value** with a **polymorphic function type**!

Example usecase

In Scala 3, all tuples extend `scala.Tuple` which defines:

```scala
def map[F[_]](f: [T] => T => F[T]): Map[this.type, F]
```

Example usecase

In Scala 3, all tuples extend `scala.Tuple` which defines:

```
def map[F[_]](f: [T] => T => F[T]): Map[this.type, F]
```

```
val x: (Int, String) = (1, "")
val y: (List[Int], List[String]) =
  x.map([T] => (x: T) => List(x))
```

The Function Zoo

	From Term	From Type
To Term	`val f: Int => Int`	`val f: [T] => T => List[T]`

The Function Zoo

	From Term	**From Type**
To Term	`val f: Int => Int`	`val f: [T] => T => List[T]`
To Type		`type F[T] = List[T]`

The Function Zoo

	From Term	**From Type**
To Term	`val f: Int => Int`	`val f: [T] => T => List[T]`

To Type		`type F[T] = List[T]`
		`type F = [T] =>> List[T]`

Table of Contents

1. Methods versus Functions
2. Handling polymorphism
3. Compiler Implementation
4. Detailed Examples
5. The (Possible) Future

How do we implement this?

Source code	Desugared form
`Int => List[Int]`	`Function1[Int, List[Int]]`
`[T] => T => List[T]`	🤔

First attempt

```
val f: [T] => T => List[T] =
  [T] => (x: T) => List[T](x)
```

First attempt

```scala
val f: [T] => T => List[T] =
  [T] => (x: T) => List[T](x)
```

```scala
trait PolyFunction1[-Param[_], +Result[_]]:
  def apply[T](x: Param[T]): Result[T]

val f = new PolyFunction1[[X] =>> X, List]:
  def apply[T](x: T): List[T] = List[T](x)
```

First attempt

```
val f: [T] => T => List[T] =
  [T] => (x: T) => List[T](x)
```

```
trait PolyFunction1[-Param[_], +Result[_]]:
  def apply[T](x: Param[T]): Result[T]

val f = new PolyFunction1[[X] =>> X, List]:
  def apply[T](x: T): List[T] = List[T](x)
```

What if we want to **bound** T?

First attempt

```
val f: [T] => T => List[T] =
  [T] => (x: T) => List[T](x)
```

```
trait PolyFunction1[-Param[_], +Result[_]]:
  def apply[T](x: Param[T]): Result[T]

val f = new PolyFunction1[[X] =>> X, List]:
  def apply[T](x: T): List[T] = List[T](x)
```

What if we want to **bound** T? E.g. `[T <: AnyRef] => T => List[T]`

Getting complicated!

```scala
trait PolyFunction1[
    -Bound[_],
    -Param[x <: Bound[x]],
    +Result[x <: Bound[x]]
]:
  def apply[T <: Bound[T]](x: Param[T]): Result[T]
```

Getting complicated!

```
trait PolyFunction1[
    -Bound[_],
    -Param[x <: Bound[x]],
    +Result[x <: Bound[x]]
]:
  def apply[T <: Bound[T]](x: Param[T]): Result[T]
```

What about multiple type parameters? Multiple term parameters?

Taking a step back

Can we use **structural typing** to avoid having to define all these traits?

Taking a step back

Can we use **structural typing** to avoid having to define all these traits?

```
val s: scala.Selectable { def foo(): Int } = ...
val x: Int = s.foo()
```

Putting it all together

Source code	Desugared form
`Int => List[Int]`	`Function1[Int, List[Int]]`

`[T] => T => List[T]`	`scala.PolyFunction {`
	`  def apply[T](x: T): List[T]`
	`}`

`scala.PolyFunction` is an empty trait which is allowed to have a polymorphic apply refinement.

Putting it all together

Source code	Desugared form
`Int => List[Int]`	`Function1[Int, List[Int]]`

`[T <: Int] => T => List[T]`	`scala.PolyFunction {`
	`  def apply[T <: Int](x: T): List[T]`
	`}`

`scala.PolyFunction` is an empty trait which is allowed to have a polymorphic apply refinement.

Type erasure (1/2)

When compiling to Java bytecode, we need to **erase** type parameters:

```
// Scala
trait Function1[-T, +R]:
  def apply(x: T): R

val f: String => List[String] = …
f("").head
```

```
// Java bytecode
interface Function1:
  def apply(x: Object): Object

val f: Function1 = …
f.apply("").asInstanceOf[List]
 .head.asInstanceOf[String]
```

Type erasure (2/2)

```scala
val g: [T] => (x: T) => List[T] =
  [T] => (x: T) => List(x)
g[String]("").head
```

We could use any compilation scheme we want, but if we want to be efficient, we need a class with an apply method!

Type erasure (2/2)

```
val g: [T] => (x: T) => List[T] =
  [T] => (x: T) => List(x)
g[String]("").head
```

We could use any compilation scheme we want, but if we want to be efficient, we need a class with an apply method!

```
val g: Function1 =
  (x: Object) => List.apply(x)
g.apply("").asInstanceOf[List].head.asInstanceOf[String]
```

Type erasure (2/2)

```
val g: [T] => (x: T) => List[T] =
  [T] => (x: T) => List(x)
g[String]("").head
```

We could use any compilation scheme we want, but if we want to be efficient, we need a class with an apply method!

```
val g: Function1 =
  (x: Object) => List.apply(x)
g.apply("").asInstanceOf[List].head.asInstanceOf[String]
```

We erase a polymorphic function with N term arguments like a monomorphic function with N term arguments.

Table of Contents

1. Methods versus Functions
2. Handling polymorphism
3. Compiler Implementation
4. Detailed Examples
5. The (Possible) Future

Example 1: Generic programming

```
trait Order[A]:
  def lessOrEqual(x: A, y: A): Boolean
```

Example 1: Generic programming

```
trait Order[A]:
  def lessOrEqual(x: A, y: A): Boolean
```

```
given Order[Int] with
  def lessOrEqual(x: Int, y: Int) = x <= y
given Order[String] with
  def lessOrEqual(x: String, y: String) = x <= y
```

Example 1: Generic programming

```
trait Order[A]:
  def lessOrEqual(x: A, y: A): Boolean
```

```
case class Foo(a: Int, b: String)

given Order[Foo] with
  def lessOrEqual(x: Foo, y: Foo) =
```

Example 1: Generic programming

```
trait Order[A]:
  def lessOrEqual(x: A, y: A): Boolean
```

```
case class Foo(a: Int, b: String)

given Order[Foo] with
  def lessOrEqual(x: Foo, y: Foo) =
    summon[Order[Int]].lessOrEqual(x.a, y.a)
    && summon[Order[String]].lessOrEqual(x.b, y.b)
```

Example 1: Generic programming

```
trait Order[A]:
  def lessOrEqual(x: A, y: A): Boolean
```

```
case class Foo(a: Int, b: String)

given Order[Foo] with
  def lessOrEqual(x: Foo, y: Foo) =
    val inst = summon[ProductInstances[Order, Foo]]
    inst.foldLeft2(x, y)(true)(


    )
```

Example 1: Generic programming

```
trait Order[A]:
  def lessOrEqual(x: A, y: A): Boolean
```

```
case class Foo(a: Int, b: String)

given Order[Foo] with
  def lessOrEqual(x: Foo, y: Foo) =
    val inst = summon[ProductInstances[Order, Foo]]
    inst.foldLeft2(x, y)(true)(
      [T] => (acc: Boolean, order: Order[T], x1: T, y1: T) =>
        acc && order.lessOrEqual(x1, y1)
    )
```

Example 2: Preserving type information

```
enum SList:
  case SNil
  case SCons(head: String, tail: SList)

  def foldRight[B](z: B)(op: (String, B) => B): B = ...
```

Example 2: Preserving type information

```
enum SList:
  case SNil
  case SCons(head: String, tail: SList)

  def foldRight[B](z: B)(op: (String, B) => B): B = ...


  def appended(elem: Int): SList =
    val newTail: SList = SCons(elem, SNil)
    foldRight(newTail)(SCons(_, _))
```

Example 2: Preserving type information

```
import scala.compiletime.ops.int.*

enum SList[N <: Int]:
  case SNil extends SList[0]
  case SCons[M <: Int](head: String, tail: SList[M]) extends SList[M+1]

  def foldRight[B](z: B)(op: (String, B) => B): B = ...


  def appended(elem: Int): SList[N+1] =
    val newTail: SList[1] = SCons(elem, SNil)
    foldRight(newTail)(SCons(_, _))
```

Example 2: Preserving type information

```
import scala.compiletime.ops.int.*

enum SList[N <: Int]:
  case SNil extends SList[0]
  case SCons[M <: Int](head: String, tail: SList[M]) extends SList[M+1]

  def foldRight[B](z: B)(op: (String, B) => B): B = ...
  def foldRightN[B[_ <: Int]](z: B[0])
    (op: [M <: Int] => (String, B[M]) => B[M+1]): B[N] = ...
  def appended(elem: Int): SList[N+1] =
    val newTail: SList[1] = SCons(elem, SNil)
    foldRightN[[X] =>> SList[X+1]](newTail)([M <: Int] => SCons(_, _))
```

Example 3: Encapsulation

```
trait Base[A]:
  extension (x: A) def base: A
```

```
def test[A](a: A)(using Base[A]) =
  a.base
```

Example 3: Encapsulation

```
trait Base[A]:
  extension (x: A) def base: A
```

```
trait Derived[A] extends Base[A]:
  extension (x: A) def dangerous: A
  /** `f` is allowed to call `base`
   *  but not `dangerous` on its input. */
  def compute(f: A => A): A
```

```
def test[A](a: A)(using d: Derived[A]) =
  d.compute(a => a.dangerous)
```

Example 3: Encapsulation

```
trait Base[A]:
  extension (x: A) def base: A
```

```
trait Derived[A] extends Base[A]:
  extension (x: A) def dangerous: A
  /** `f` is allowed to call `base`
   *  but not `dangerous` on its input. */
  def compute(f: A => A): A
  def computeSafe(f:         ): A
```

```
def test[A](a: A)(using d: Derived[A]) =
  d.compute(a => a.dangerous)
```

Example 3: Encapsulation

```
trait Base[A]:
  extension (x: A) def base: A
```

```
trait Derived[A] extends Base[A]:
  extension (x: A) def dangerous: A
  /** `f` is allowed to call `base`
   *  but not `dangerous` on its input. */
  def compute(f: A => A): A
  def computeSafe(f: Any => A): A
```

```
def test[A](a: A)(using d: Derived[A]) =
  d.compute(a => a.dangerous)
```

Example 3: Encapsulation

```
trait Base[A]:
  extension (x: A) def base: A
```

```
trait Derived[A] extends Base[A]:
  extension (x: A) def dangerous: A
  /** `f` is allowed to call `base`
   *  but not `dangerous` on its input. */
  def compute(f: A => A): A
  def computeSafe(f: [T] => T => T): A
```

```
def test[A](a: A)(using d: Derived[A]) =
  d.compute(a => a.dangerous)
```

Example 3: Encapsulation

```
trait Base[A]:
  extension (x: A) def base: A
```

```
trait Derived[A] extends Base[A]:
  extension (x: A) def dangerous: A
  /** `f` is allowed to call `base`
   *  but not `dangerous` on its input. */
  def compute(f: A => A): A
  def computeSafe(f: [T] => T => T): A
```

```
def test[A](a: A)(using d: Derived[A]) =
  d.compute(a => a.dangerous)
  d.computeSafe(a => a.base)
```

Example 3: Encapsulation

```
trait Base[A]:
  extension (x: A) def base: A
```

```
trait Derived[A] extends Base[A]:
  extension (x: A) def dangerous: A
  /** `f` is allowed to call `base`
   *  but not `dangerous` on its input. */
  def compute(f: A => A): A
  def computeSafe(f: [T] => T => Base[T] ?=> T): A
```

```
def test[A](a: A)(using d: Derived[A]) =
  d.compute(a => a.dangerous)
  d.computeSafe(a => a.base)
```

Example 3: Encapsulation

This technique is used in **cats-effect** to keep Async#cont safe, see https://typelevel.org/cats-effect/docs/typeclasses/async.

Table of Contents

1. Methods versus Functions

2. Handling polymorphism

3. Compiler Implementation

4. Detailed Examples

5. The (Possible) Future

Idea 1: Polymorphic eta-expansion (1/2)

SIP-49: Polymorphic Eta-Expansion (by Quentin Bernet)

SIP stands for Scala Improvement Process.

Idea 1: Polymorphic eta-expansion (1/2)

SIP-49: Polymorphic Eta-Expansion (by Quentin Bernet)

SIP stands for Scala Improvement Process. Roughly, the process is:

1. Have an idea! 💡

Idea 1: Polymorphic eta-expansion (1/2)

SIP-49: Polymorphic Eta-Expansion (by Quentin Bernet)

SIP stands for Scala Improvement Process. Roughly, the process is:

1. Have an idea! 💡
2. Discuss it on contributors.scala-lang.org

Idea 1: Polymorphic eta-expansion (1/2)

SIP-49: Polymorphic Eta-Expansion (by Quentin Bernet)

SIP stands for Scala Improvement Process. Roughly, the process is:

1. Have an idea! 💡
2. Discuss it on contributors.scala-lang.org
3. Write a SIP proposal with a formal specification.

Idea 1: Polymorphic eta-expansion (1/2)

SIP-49: Polymorphic Eta-Expansion (by Quentin Bernet)

SIP stands for Scala Improvement Process. Roughly, the process is:

1. Have an idea! 💡
2. Discuss it on contributors.scala-lang.org
3. Write a SIP proposal with a formal specification.
4. The SIP committee members vote on accepting it as **experimental**.

Idea 1: Polymorphic eta-expansion (1/2)

SIP-49: Polymorphic Eta-Expansion (by Quentin Bernet)

SIP stands for Scala Improvement Process. Roughly, the process is:

1. Have an idea! 💡
2. Discuss it on contributors.scala-lang.org
3. Write a SIP proposal with a formal specification.
4. The SIP committee members vote on accepting it as **experimental**.
5. Implement the SIP in the compiler as experimental, gather feedback.

Idea 1: Polymorphic eta-expansion (1/2)

SIP-49: Polymorphic Eta-Expansion (by Quentin Bernet)

SIP stands for Scala Improvement Process. Roughly, the process is:

1. Have an idea! 💡
2. Discuss it on contributors.scala-lang.org
3. Write a SIP proposal with a formal specification.
4. The SIP committee members vote on accepting it as **experimental**.
5. Implement the SIP in the compiler as experimental, gather feedback.
6. The SIP committee members vote on accepting it as **stable**.

Idea 1: Polymorphic eta-expansion (1/2)

SIP-49: Polymorphic Eta-Expansion (by Quentin Bernet)

SIP stands for Scala Improvement Process. Roughly, the process is:

1. Have an idea! 💡
2. Discuss it on contributors.scala-lang.org
3. Write a SIP proposal with a formal specification.
4. The SIP committee members vote on accepting it as **experimental**.
5. Implement the SIP in the compiler as experimental, gather feedback.
6. The SIP committee members vote on accepting it as **stable**.
7. The feature is marked as stable in the compiler.

Idea 1: Polymorphic eta-expansion (2/2)

Adapt **polymorphic method references** by eta-expansion:

```
def singleton[T](x: T): List[T] = List(x)
(1, "").map(singleton)
```

Idea 1: Polymorphic eta-expansion (2/2)

Adapt **polymorphic method references** by eta-expansion:

```
def singleton[T](x: T): List[T] = List(x)
(1, "").map([T] => (x: T) => singleton[T](x))
```

What about regular lambdas?

If we only adapt method references, this will work:

```
(1, "").map(List.apply)
```

… but this won’t work:

```
(1, "").map(List(_))
```

What about regular lambdas?

If we only adapt method references, this will work:

```
(1, "").map(List.apply)
```

... but this won’t work:

```
(1, "").map(List(_))

(1, "").map(_.toString)
```

Idea 2: type parameter clause inference

Instead, we could adapt regular lambdas into polymorphic lambdas by type parameter clause inference combined with the usual type inference.

```
val f: [T] => T => String =
              x => x.toString
```

Idea 2: type parameter clause inference

Instead, we could adapt regular lambdas into polymorphic lambdas by type parameter clause inference combined with the usual type inference.

```
val f: [T] => T => String =
       [T] => x => x.toString
```

Idea 2: type parameter clause inference

Instead, we could adapt regular lambdas into polymorphic lambdas by type parameter clause inference combined with the usual type inference.

```
val f: [T] => T => String =
       [T] => (x: T) => x.toString
```

Idea 2: type parameter clause inference

Regular eta-expansion can also be combined with type parameter clause inference and type inference.

```
def singleton[T](x: T): List[T] = List(x)

val f: [T] => T => List[T] =
  singleton
```

Idea 2: type parameter clause inference

Regular eta-expansion can also be combined with type parameter clause inference and type inference.

```
def singleton[T](x: T): List[T] = List(x)

val f: [T] => T => List[T] =
  x => singleton(x)
```

Idea 2: type parameter clause inference

Regular eta-expansion can also be combined with type parameter clause inference and type inference.

```
def singleton[T](x: T): List[T] = List(x)

val f: [T] => T => List[T] =
  [T] => x => singleton(x)
```

Idea 2: type parameter clause inference

Regular eta-expansion can also be combined with type parameter clause inference and type inference.

```
def singleton[T](x: T): List[T] = List(x)

val f: [T] => T => List[T] =
  [T] => (x: T) => singleton(x)
```

Idea 2: type parameter clause inference

Regular eta-expansion can also be combined with type parameter clause inference and type inference.

```
def singleton[T](x: T): List[T] = List(x)

val f: [T] => T => List[T] =
  [T] => (x: T) => singleton[T](x)
```

```
val g: [T, S] => (T, S) => List[(T, S)] =
   singleton
```

Idea 2: type parameter clause inference

Regular eta-expansion can also be combined with type parameter clause inference and type inference.

```
def singleton[T](x: T): List[T] = List(x)

val f: [T] => T => List[T] =
  [T] => (x: T) => singleton[T](x)
```

```
val g: [T, S] => (T, S) => List[(T, S)] =
  [T, S] => (x: (T, S)) => singleton[(T, S)](x)
```

Idea 3: Better subtyping

For regular function types:

```
Any => Int    <:   Any =>
```

Idea 3: Better subtyping

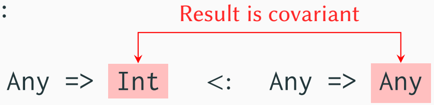
For regular function types:

Result is covariant

```
Any => Int   <:   Any => Any
```

Idea 3: Better subtyping

For regular function types:



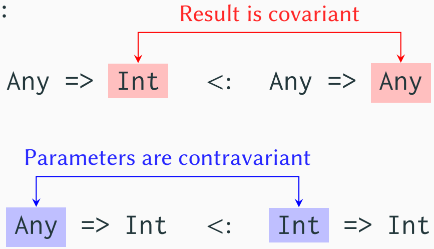
Result is covariant

```
Any => Int    <:   Any => Any
```

```
Any => Int    <:         => Int
```

Idea 3: Better subtyping

For regular function types:



Result is covariant

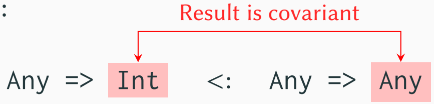
`Any => Int <: Any => Any`

Parameters are contravariant

`Any => Int <: Int => Int`

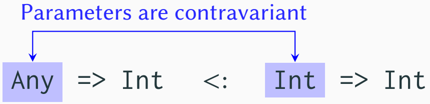
Idea 3: Better subtyping

For regular function types:



Result is covariant

```
Any => Int   <:   Any => Any
```



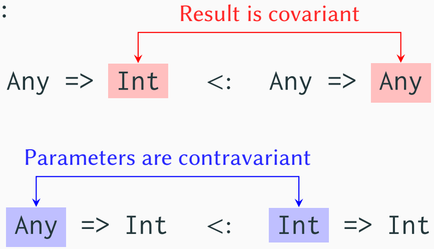
Parameters are contravariant

```
Any => Int   <:   Int => Int
```

For polymorphic function types everything is invariant currently, but ideally:

Idea 3: Better subtyping

For regular function types:



Result is covariant

```
Any => Int   <:   Any => Any
```

Parameters are contravariant

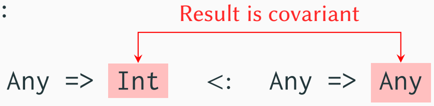
```
Any => Int   <:   Int => Int
```

For polymorphic function types everything is invariant currently, but ideally:

```
[T <: Any] => Seq[T] => Option[T] <: [T <:     ] =>           =>     
```

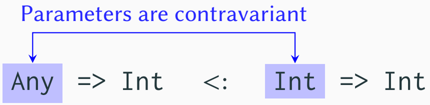
Idea 3: Better subtyping

For regular function types:



Result is covariant

```
Any => Int   <:   Any => Any
```



Parameters are contravariant

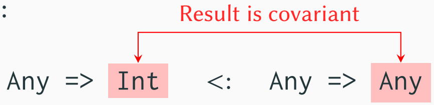
```
Any => Int   <:   Int => Int
```

For polymorphic function types everything is invariant currently, but ideally:

```
[T <: Any] => Seq[T] => Option[T] <: [T <:     ] =>           => 
```

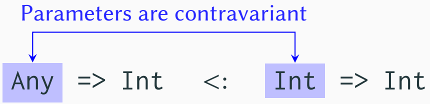
Idea 3: Better subtyping

For regular function types:



Result is covariant

`Any => Int   <:   Any => Any`



Parameters are contravariant

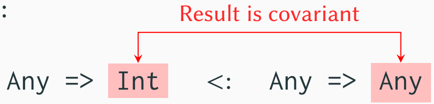
`Any => Int   <:   Int => Int`

For polymorphic function types everything is invariant currently, but ideally:

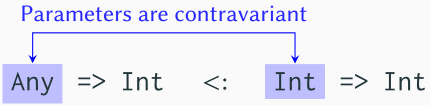
```
[T <: Any ] => Seq[T] => Option[T] <: [T <:     ] =>          => Any
```

Idea 3: Better subtyping

For regular function types:


Result is covariant

```
Any => Int   <:   Any => Any
```


Parameters are contravariant

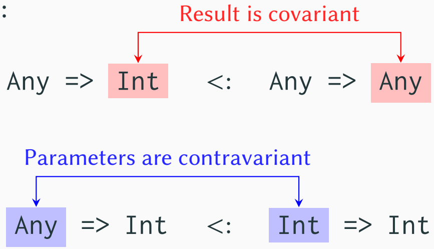
```
Any => Int   <:   Int => Int
```

For polymorphic function types everything is invariant currently, but ideally:

```
[T <: Any ] => Seq[T] => Option[T] <: [T <:     ] => List[T] => Any
```

Idea 3: Better subtyping

For regular function types:



Result is covariant

```
Any => Int   <:   Any => Any
```

Parameters are contravariant

```
Any => Int   <:   Int => Int
```

For polymorphic function types everything is invariant currently, but ideally:

```
[T <: Any] => Seq[T] => Option[T] <: [T <: Int] => List[T] => Any
```

Thank you!

Come to the Scala Spree this Friday! https://github.com/scalacenter/sprees

Resources:

- Slides for this talk: http://guillaume.martres.me/talks/scaladays23-madrid.pdf
- Scala 3 Compiler Academy on Youtube.
- `#scala-contributors` on the Scala Discord.