Secrets of the Scala Type System

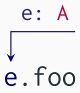
Guillaume Martres - Scala Center

December 22, 2022

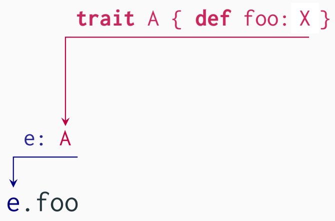
Member lookup

```
e.foo
```

Member lookup



Member lookup

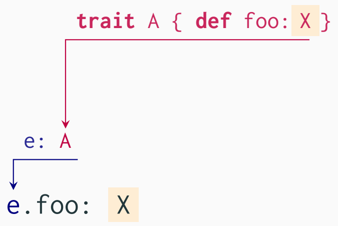


```
trait A { def foo: X }
```

```
e: A
```

```
e.foo
```

Member lookup

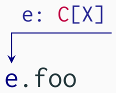


```
trait A { def foo: X }
```

```
e: A
```

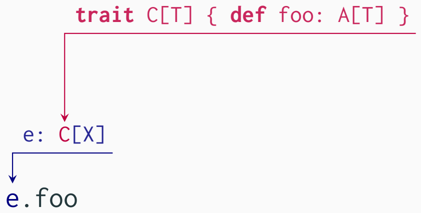
```
e.foo: X
```

Member lookup - Type argument in prefix



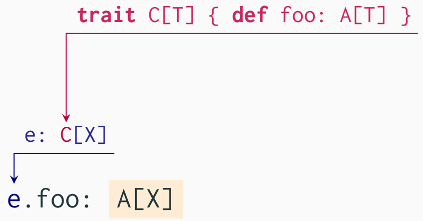
```
 e: C[X]
e.foo
```

Member lookup - Type argument in prefix



```
trait C[T] { def foo: A[T] }
e: C[X]
e.foo
```

Member lookup - Type argument in prefix



```
trait C[T] { def foo: A[T] }
```

```
e: C[X]
```

```
e.foo: A[X]
```

- T is **substituted** by X in the type of foo.

Intersection types

```
e: A & B
```

Intersection types

$$\begin{gathered}\texttt{e: A } \mathbin{\&} \texttt{ B} \\ \Big\Updownarrow \\ \texttt{e: A} \quad \textbf{and} \quad \texttt{e: B}\end{gathered}$$

Intersection types

$$\texttt{e: A}\ \&\ \texttt{B}$$
$$\Updownarrow$$
$$\texttt{e: A}\quad \textbf{and}\quad \texttt{e: B}$$

- For example,

```
class AB extends A, B
val ab: A & B = new AB
```

Intersection types

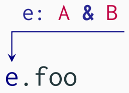
$$\texttt{e: A \& B} \iff \texttt{e: A} \;\; \textbf{and} \;\; \texttt{e: B}$$

- For example,

```
class AB extends A, B
val ab: A & B = new AB
```

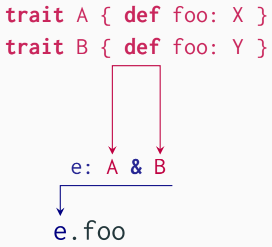
- In Scala 2, `A with B` is used instead (I'll explain the difference later)

Intersection types - Member lookup



```
 e: A & B
e.foo
```

Intersection types - Member lookup



```
trait A { def foo: X }
trait B { def foo: Y }

e: A & B

e.foo
```

Intersection types - Member lookup

```
trait A { def foo: X }
trait B { def foo: Y }
```

```
e: A & B
```

```
e.foo: X & Y
```

Intersection types - Example

This means the following code compiles:

```scala
trait A { def foo: Int }
trait B { def foo: String }

// 🤔
def test(x: A & B): Int & String = x.foo
```

Intersection types - Example

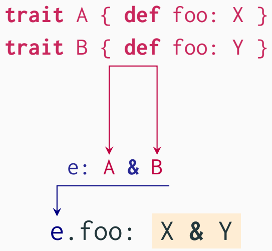
This means the following code compiles:

```
trait A { def foo: Int }
trait B { def foo: String }

// 🤔
def test(x: A & B): Int & String = x.foo
```

This looks weird, but is not a problem because there is no value of type `A & B`.

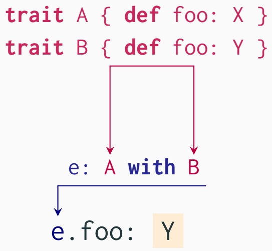
Scala 3 vs Scala 2 - Intersection types

Scala 3



```scala
trait A { def foo: X }
trait B { def foo: Y }

e: A & B

e.foo: X & Y
```

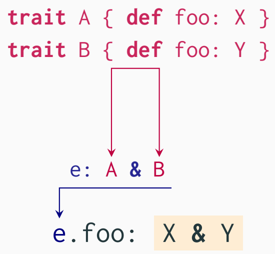
Scala 2



```scala
trait A { def foo: X }
trait B { def foo: Y }

e: A with B

e.foo: Y
```

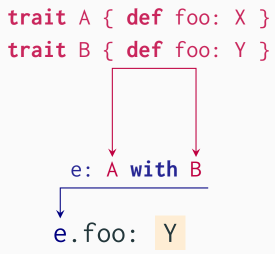
Scala 3 vs Scala 2 - Intersection types

Scala 3



```
trait A { def foo: X }
trait B { def foo: Y }

    e: A & B

e.foo: X & Y
```

Scala 2



```
trait A { def foo: X }
trait B { def foo: Y }

    e: A with B

e.foo: Y
```

- **with** is not *commutative*: `A with B` is different from `B with A`.

Scala 3 vs Scala 2 - Inheritance

On the other hand, these mean the same thing:

Scala 3

```
class AB extends A, B
```

Scala 2

```
class AB extends A with B
```

Scala 3 vs Scala 2 - Inheritance

On the other hand, these mean the same thing:

Scala 3

```scala
class AB extends A, B
```

Scala 2

```scala
class AB extends A with B
```

- … so why don't we write `class AB extends A & B`?

Scala 3 vs Scala 2 - Inheritance

On the other hand, these mean the same thing:

Scala 3

```scala
class AB extends A, B
```

Scala 2

```scala
class AB extends A with B
```

- … so why don't we write `class AB extends A & B`?
- Because inheritance is not always commutative!

Linearization: when inheritance order matters

```
trait Base:
  def print(): Unit

trait L extends Base:
  override def print(): Unit = println("L")

trait R extends Base:
  override def print(): Unit = println("R")

class LR extends L, R

(new LR).print()
```

Linearization: when inheritance order matters

```scala
// $\mathcal{L}$(Base) = Base, AnyRef
trait Base:
  def print(): Unit

trait L extends Base:
  override def print(): Unit = println("L")

trait R extends Base:
  override def print(): Unit = println("R")

class LR extends L, R

(new LR).print()
```

Linearization: when inheritance order matters

```
// ℒ(Base) = Base, AnyRef
trait Base:
  def print(): Unit
// ℒ(L) = L, ℒ(Base)
trait L extends Base:
  override def print(): Unit = println("L")

trait R extends Base:
  override def print(): Unit = println("R")

class LR extends L, R

(new LR).print()
```

Linearization: when inheritance order matters

```
// 𝓛(Base) = Base, AnyRef
trait Base:
  def print(): Unit
// 𝓛(L) = L, 𝓛(Base)
trait L extends Base:
  override def print(): Unit = println("L")
// 𝓛(R) = R, 𝓛(Base)
trait R extends Base:
  override def print(): Unit = println("R")

class LR extends L, R

(new LR).print()
```

Linearization: when inheritance order matters

```scala
// $\mathcal{L}(\text{Base}) = \text{Base}, \text{AnyRef}$
trait Base:
  def print(): Unit
// $\mathcal{L}(\text{L}) = \text{L}, \mathcal{L}(\text{Base})$
trait L extends Base:
  override def print(): Unit = println("L")
// $\mathcal{L}(\text{R}) = \text{R}, \mathcal{L}(\text{Base})$
trait R extends Base:
  override def print(): Unit = println("R")
// $\mathcal{L}(\text{LR}) = \text{LR}, \mathcal{L}(\text{R}) \vec{+} \mathcal{L}(\text{L})$
class LR extends L, R

(new LR).print()
```

Linearization: when inheritance order matters

```
// $\mathcal{L}(\mathrm{Base}) = \mathrm{Base}, \mathrm{AnyRef}$
trait Base:
  def print(): Unit
// $\mathcal{L}(\mathrm{L}) = \mathrm{L}, \ \mathcal{L}(\mathrm{Base})$
trait L extends Base:
  override def print(): Unit = println("L")
// $\mathcal{L}(\mathrm{R}) = \mathrm{R}, \ \mathcal{L}(\mathrm{Base})$
trait R extends Base:
  override def print(): Unit = println("R")
// $\mathcal{L}(\mathrm{LR}) = \mathrm{LR}, \ \mathcal{L}(\mathrm{R}) \overrightarrow{+} \mathcal{L}(\mathrm{L}) = \mathrm{LR}, \mathrm{R}, \mathrm{L}, \mathrm{Base}, \mathrm{AnyRef}$
class LR extends L, R

(new LR).print()
```

Linearization: when inheritance order matters

```
// ℒ(Base) = Base, AnyRef
trait Base:
  def print(): Unit
// ℒ(L) = L, ℒ(Base)
trait L extends Base:
  override def print(): Unit = println("L")
// ℒ(R) = R, ℒ(Base)
trait R extends Base:
  override def print(): Unit = println("R")
// ℒ(LR) = LR, ℒ(R) +⃗ ℒ(L) = LR, R, L, Base, AnyRef
class LR extends L, R

(new LR).print() // “R”
```

Linearization: more complex example

Exercise: What does this print? :)

```scala
// L and R as before
trait LR extends L, R
trait RL extends R, L
class LRRL extends LR, RL

(new LRRL).print()
```

Union types

```
e: A | B
```

Union types

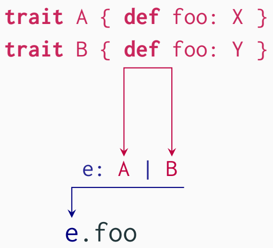
$$\begin{gathered}\texttt{e: A | B} \\ \Big\Updownarrow \\ \texttt{e: A} \quad \textbf{or} \quad \texttt{e: B}\end{gathered}$$

Union types

$$\begin{tikzcd} \texttt{e: A | B} \arrow[d, Leftrightarrow] \\ \texttt{e: A} \ \textbf{or}\ \texttt{e: B} \end{tikzcd}$$

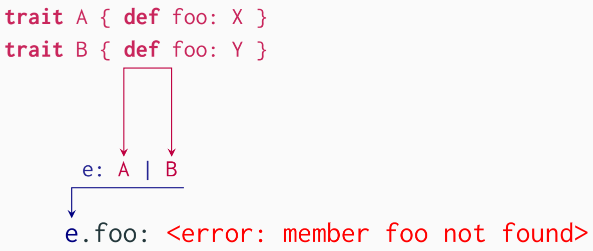
```
val x: Int | String = if cond then 1 else "hello"
```

Union types - Member lookup



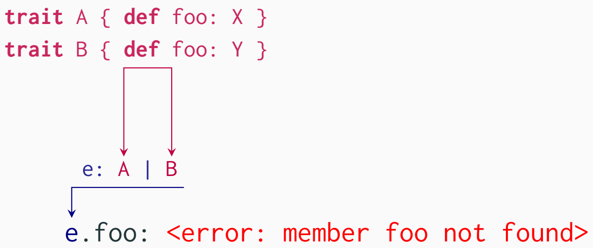
```
trait A { def foo: X }
trait B { def foo: Y }

    e: A | B

   e.foo
```

Union types - Member lookup



```
trait A { def foo: X }
trait B { def foo: Y }

   e: A | B

  e.foo: <error: member foo not found>
```

Union types - Member lookup

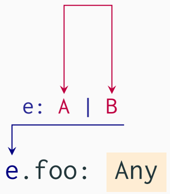


```
trait A { def foo: X }
trait B { def foo: Y }

    e: A | B

   e.foo: <error: member foo not found>
```

- The members of A | B are the members of the *common base classes* of A and B.

Union types - Member lookup

```
trait Base { def foo: Any }
trait A extends Base { def foo: X }
trait B extends Base { def foo: Y }
```



- The members of A | B are the members of the *common base classes* of A and B.

Wildcards

```
e: C[?]
```

Wildcards

```
e: C[?]
```

$$\Updownarrow$$

There exists a type T such that

```
e: C[T]
```

Wildcards

`e: C[?]`

$$\Updownarrow$$

There exists a type T such that

`e: C[T]`

```
val a: Array[?] = Array[String]()
```

Wildcards

`e: C[? <: Hi]`

$$\Updownarrow$$

There exists a type `T` such that

`T <: Hi` and `e: C[T]`

```
val a: Array[?] = Array[String]()
val b: Array[? <: AnyRef] = Array[String]()
```

Wildcards

`e: C[? >: Lo <: Hi]`

$$\Updownarrow$$

There exists a type T such that

`T >: Lo` and `T <: Hi` and `e: C[T]`

```scala
val a: Array[?] = Array[String]()
val b: Array[? <: AnyRef] = Array[String]()
val c: Array[? >: String] = Array[String]()
```

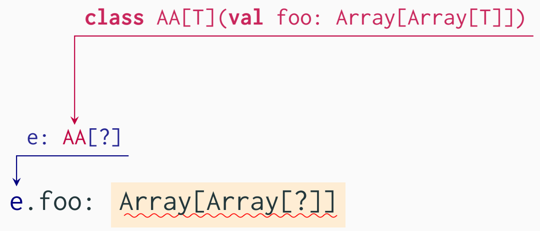
Wildcards - Member lookup

$$\begin{tikzcd} \texttt{e: AA[?]} \arrow[d] \\ \texttt{e.foo} \end{tikzcd}$$

Wildcards - Member lookup

```
class AA[T](val foo: Array[Array[T]])

e: AA[?]

e.foo
```

Wildcards - Member lookup



```
class AA[T](val foo: Array[Array[T]])

e: AA[?]

e.foo: Array[Array[?]]
```

- Type parameters cannot be directly substituted by wildcards!

Wildcards - Member lookup - Substitution counter example

```scala
class AA[T](val foo: Array[Array[T]])
val a: AA[Int] = new AA(Array(Array(1)))
```

Wildcards - Member lookup - Substitution counter example

```
class AA[T](val foo: Array[Array[T]])
val a: AA[Int] = new AA(Array(Array(1)))

val e: AA[?] = a
```

Wildcards - Member lookup - Substitution counter example

```
class AA[T](val foo: Array[Array[T]])
val a: AA[Int] = new AA(Array(Array(1)))

val e: AA[?] = a
val x: Array[Array[?]] = e.foo // Should be an error!
```

Wildcards - Member lookup - Substitution counter example

```
class AA[T](val foo: Array[Array[T]])
val a: AA[Int] = new AA(Array(Array(1)))

val e: AA[?] = a
val x: Array[Array[?]] = e.foo // Should be an error!

x(0) = Array[String]("")
```

Wildcards - Member lookup - Substitution counter example

```
class AA[T](val foo: Array[Array[T]])
val a: AA[Int] = new AA(Array(Array(1)))

val e: AA[?] = a
val x: Array[Array[?]] = e.foo // Should be an error!

x(0) = Array[String]("")

a.foo(0)(0): Int // runtime crash (ClassCastException) if no error!
```

Wildcards - Member lookup - Substitution counter example

```
class AA[T](val foo: Array[Array[T]])
val a: AA[Int] = new AA(Array(Array(1)))

val e: AA[?] = a
val x: Array[Array[?]] = e.foo // Should be an error!

x(0) = Array[String]("")

a.foo(0)(0): Int // runtime crash (ClassCastException) if no error!
```

`e.foo` can instead be typed as `Array[? <: Array[?]]`

Type members

Type Member T

```
class A {type T; def foo: T = ...}
```

Type Parameter T

```
class A[T] { def foo: T = ... }
```

Type members

Type Member T

```
class A {type T; def foo: T = ...}
val x: A { type T = Int } = ...
```

Type Parameter T

```
class A[T] { def foo: T = ... }
val x: A[Int] = ...
```

Type members

Type Member T

```scala
class A {type T; def foo: T = ...}
val x: A { type T = Int } = ...
val y: A = x
```

Type Parameter T

```scala
class A[T] { def foo: T = ... }
val x: A[Int] = ...
val y: A[?] = x
```

Type members

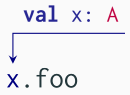
Type Member T

```
class A {type T; def foo: T = ...}
val x: A { type T = Int } = ...
val y: A = x

val z: y.T = y.foo
```

Type Parameter T

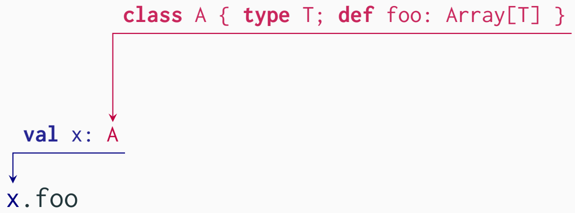
```
class A[T] { def foo: T = ... }
val x: A[Int] = ...
val y: A[?] = x

val z: Any = y.foo
```

Type members - Member lookup



```
 val x: A
x.foo
```

Type members - Member lookup

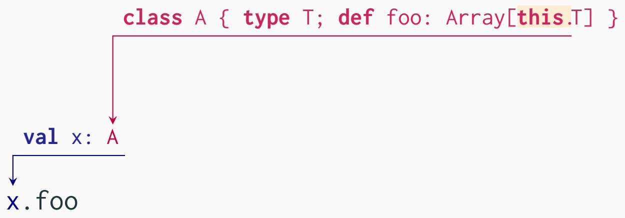


```
class A { type T; def foo: Array[T] }
```

```
val x: A
```

```
x.foo
```

Type members - Member lookup

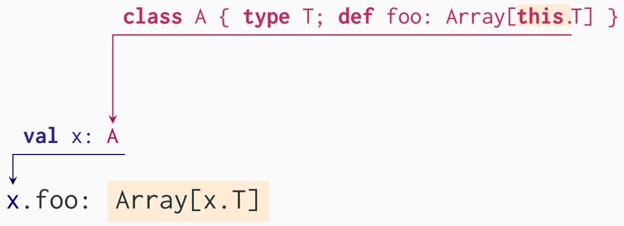


```
class A { type T; def foo: Array[this.T] }
```

```
val x: A
```

```
x.foo
```

Type members - Member lookup



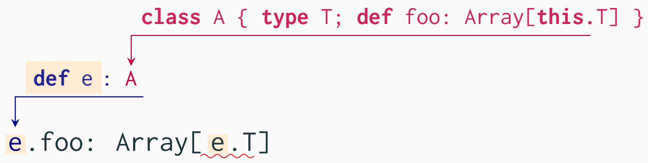
```
class A { type T; def foo: Array[this.T] }
```

```
val x: A
```

```
x.foo: Array[x.T]
```

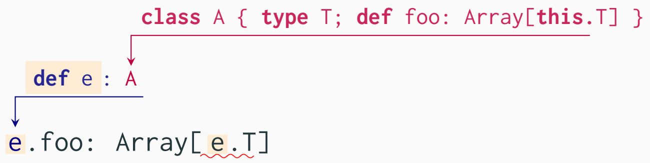
- this is **substituted** by x in the type of foo.

Type members - Member lookup - Skolemization



```
class A { type T; def foo: Array[this.T] }

def e : A

e.foo: Array[e.T]
```

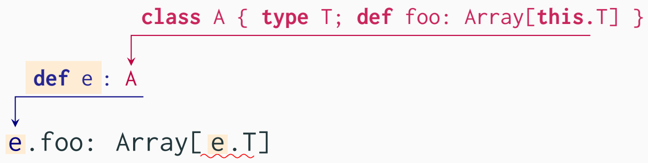
Type members - Member lookup - Skolemization



```
class A { type T; def foo: Array[this.T] }
def e : A
e.foo: Array[e.T]
```

- e is not a **val**, so e.T is not a valid type

Type members - Member lookup - Skolemization

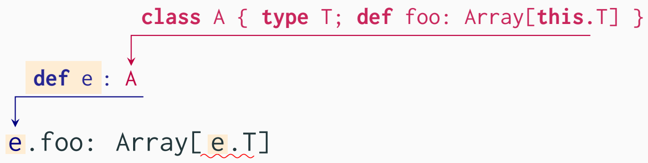


```
class A { type T; def foo: Array[this.T] }

def e : A

e.foo: Array[e.T]
```

- e is not a **val**, so `e.T` is not a valid type
- We can rewrite `e.foo` as:

```
{
  val tmp = e
  tmp.foo
}
```

Type members - Member lookup - Skolemization

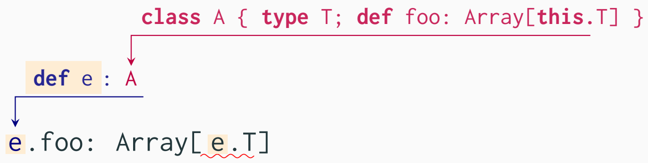


```
class A { type T; def foo: Array[this.T] }

def e : A

e.foo: Array[e.T]
```

- e is not a **val**, so `e.T` is not a valid type
- We can rewrite `e.foo` as:

```
{
  val tmp : A = e
  tmp.foo
}
```

Type members - Member lookup - Skolemization

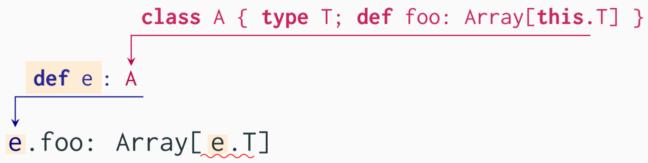


```
class A { type T; def foo: Array[this.T] }
def e : A
e.foo: Array[e.T]
```

- e is not a **val**, so `e.T` is not a valid type
- We can rewrite `e.foo` as:

```
{
  val tmp : A = e
  tmp.foo : Array[tmp.T]
}
```

Type members - Member lookup - Skolemization



```
class A { type T; def foo: Array[this.T] }
def e : A
e.foo: Array[e.T]
```

- e is not a **val**, so e.T is not a valid type
- We can rewrite e.foo as:

```
{
  val tmp : A = e
  tmp.foo : Array[tmp.T]
} : Array[?]
```

The wildcard trick

```
class AA[T](val foo: Array[Array[T]])

val x: AA[?]

x.foo
```

The wildcard trick

```scala
class AA[T](val foo: Array[Array[T]])
```

```scala
val x: AA[x.T]
```

```scala
x.foo
```

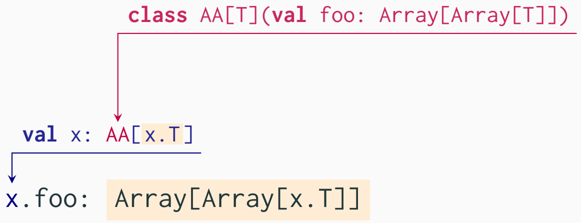
The wildcard trick

```
class AA[T](val foo: Array[Array[T]])
```

```
val x: AA[x.T]
```

```
x.foo: Array[Array[x.T]]
```

The wildcard trick



```
class AA[T](val foo: Array[Array[T]])

val x: AA[x.T]

x.foo: Array[Array[x.T]]
```

- If the prefix isn’t a **`val`**, make up a temporary one like in the previous slide

The wildcard trick - Example

```
import scala.collection.mutable.ListBuffer

val x: ListBuffer[?] = ListBuffer("a", "b")

x.append(x.apply(0)) // ListBuffer(“a”, “b”, “a”)
```

Thank you!

Resources:

- Slides for this talk: http://guillaume.martres.me/talks/romandie22-12.pdf
- The Scala 3 language reference: docs.scala-lang.org/scala3/reference
- A previous talk: Scala 3, Type Inference and You! on Youtube.
- Scala 3 Compiler Academy on Youtube.
- `#scala-contributors` on the Scala Discord.
- My thesis: guillaume.martres.me/thesis.pdf