Secrets of the Scala Type System

Guillaume Martres - Scala Center

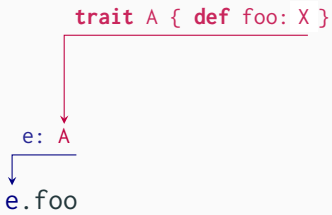
November 4, 2022

Member lookup

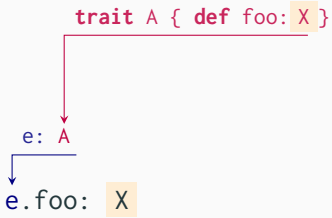
```
e.foo
```

Member lookup

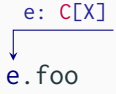
Member lookup



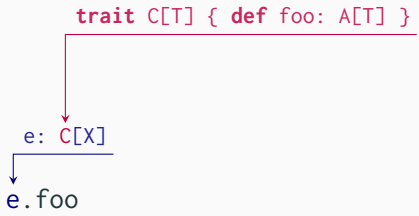
Member lookup



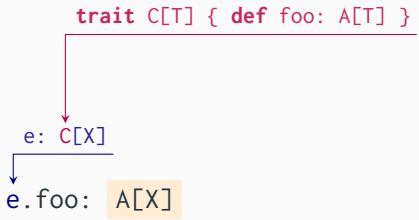
Member lookup - Type argument in prefix



Member lookup - Type argument in prefix



Member lookup - Type argument in prefix



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

Intersection types

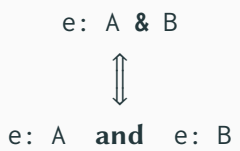
```
e: A & B
```

Intersection types

`e : A` **`&`** `B`

`e : A` **and** `e : B`

Intersection types



- For example,

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

Intersection types

`e: A & B`

`e: A` **and** `e: B`

- For example,

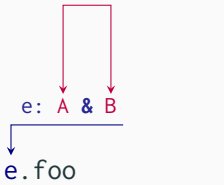
```
class AB extends A, B
val ab: A & B = new AB
```

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

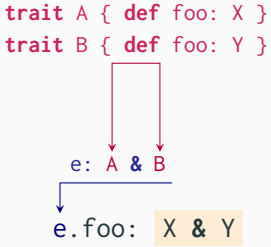
Intersection types - Member lookup

Intersection types - Member lookup

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

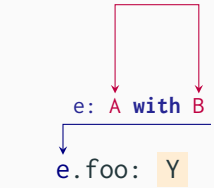


Intersection types - Member lookup

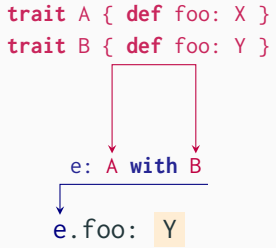


Scala 2 Compound type - Member lookup

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

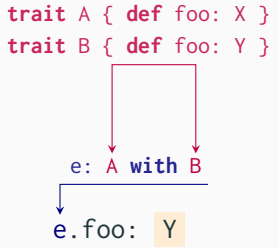


Scala 2 Compound type - Member lookup



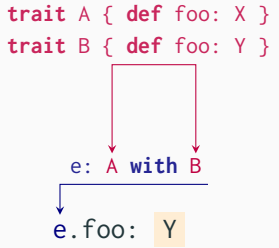
- **with** is not *commutative*: `A with B` is different from `B with A`.

Scala 2 Compound type - Member lookup



- **with** is not *commutative*: `A` **`with`** `B` is different from `B` **`with`** `A`.
- Why don't we write `class AB extends A & B`?

Scala 2 Compound type - Member lookup



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

```
// 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

```
// 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)
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()
```

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? :)

```
// 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

`e : A | B`

`e : A` **or** `e : B`

Union types

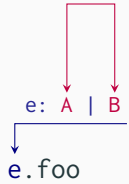
```
e: A | B
```

```
e: A  or  e: B
```

```
val x: List[Int | String] = List(1, "hello")
```

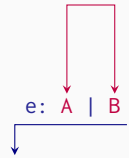
Union types - Member lookup

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



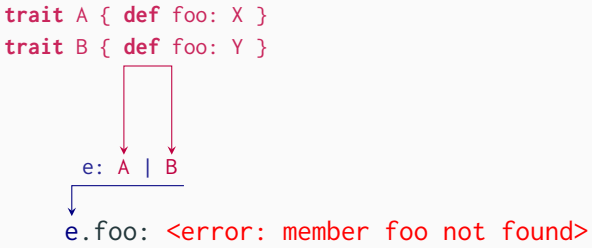
Union types - Member lookup

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




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

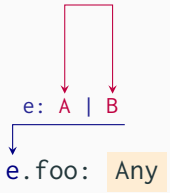
Union types - Member lookup



- 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[?]`

There exists a type `T` such that

`e: C[T]`

Wildcards

`e: C[?]`

There exists a type `T` such that

`e: C[T]`

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

Wildcards

`e: C[? <: Hi]`

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]`

There exists a type `T` such that

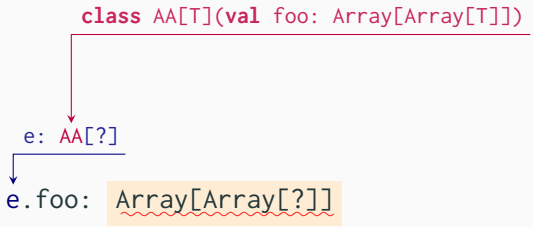
`T >: Lo` and `T <: Hi` and `e: C[T]`

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

Wildcards - Member lookup

Wildcards - Member lookup

Wildcards - Member lookup



- Type parameters cannot be directly substituted by wildcards!

Wildcards - Member lookup - Substitution counter example

```
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

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

Type Parameter T

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

Type members

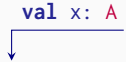
Type Member T

```
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
```

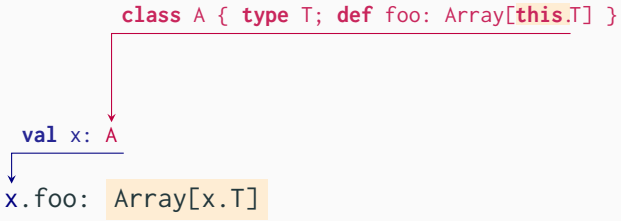
Type members - Member lookup



Type members - Member lookup

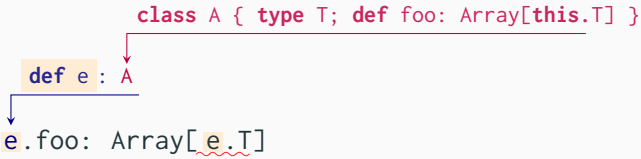
Type members - Member lookup

Type members - Member lookup

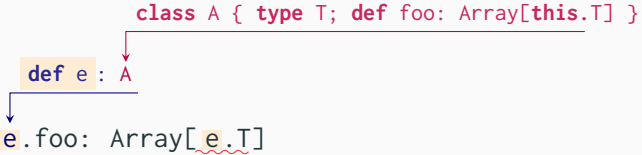


- `this` is **substituted** by `x` in the type of `foo`.

Type members - Member lookup - Skolemization

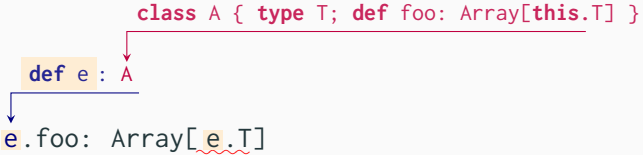


Type members - Member lookup - Skolemization



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

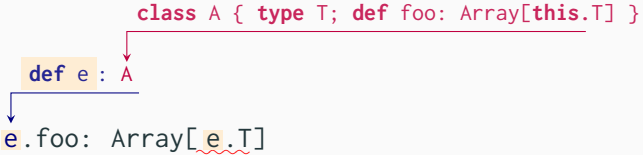
Type members - Member lookup - Skolemization



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

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

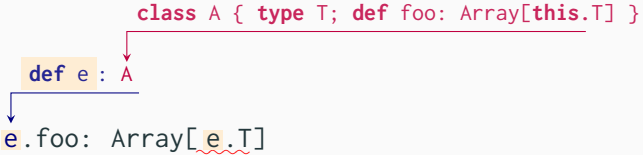
Type members - Member lookup - Skolemization



- 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
}
```

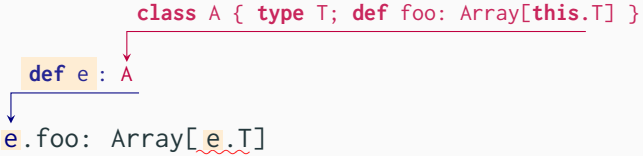
Type members - Member lookup - Skolemization



- 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

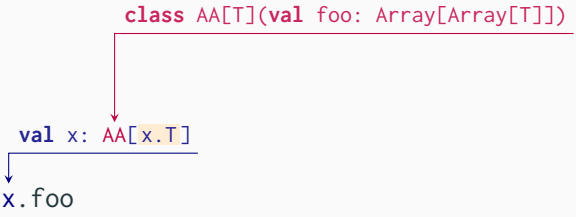


- e is not a **val**, so e.T is not a valid type
- We can rewrite e.foo as:

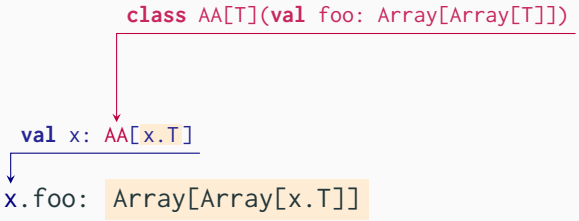
```
{
  val tmp : A = e
  tmp.foo : Array[tmp.T]
} : Array[?]
```

The wildcard trick

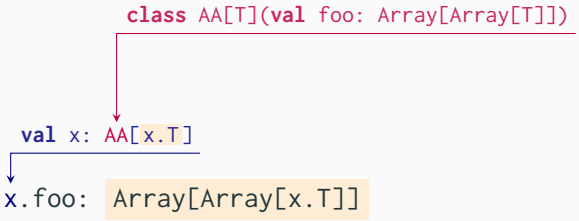
The wildcard trick



The wildcard trick



The wildcard trick



- 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”, “a”, “b”)
```

Thank you!

Resources:

- Slides for this talk: guillaume.martres.me/talks/scalaio22.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