Pathless Scala: A Calculus for the Rest of Scala

Guillaume Martres
EPFL
Lausanne, Switzerland
guillaume.martres@epfl.ch

Abstract
Recent work on the DOT calculus successfully put core aspects of Scala on a sound foundation, but subtyping in DOT is structural and therefore not easily amenable to studying the parts of Scala that are deeply tied to its nominal subtyping system. On the other hand, the Featherweight Java calculus has proven to be a great basis for studying many aspects of Java and Java-like languages. Continuing this tradition, we present Pathless Scala: an extension of Featherweight Generic Java that closely models multiple inheritance and intersection types as they exist in the Scala language today. We define the semantics of Pathless Scala by erasing it to a simpler calculus in a way that closely models how Scala is compiled to Java bytecode in practice. More than a one-off, we believe that this calculus could be extended to describe many more features of Scala, although reconciling it with DOT remains an open problem.

CCS Concepts: • Software and its engineering → Formal language definitions; Inheritance; Polymorphism.

Keywords: Trait, Featherweight, Erasure, Dotty, Java

1 Introduction
Formalizing a programming language lets us reason about the behavior of programs in the language by developing its metatheory but it also means that the implementation strategies used by compilers can themselves be formalized. While the DOT calculus [Amin et al. 2016] has been very useful as a reasoning tool for various aspects of the Scala type system, it is not really suitable for answering questions such as "How do I compile this Scala program to Java bytecode?" ¹.

To answer this question our main source of inspiration will be [Igarashi et al. 2001] which defines two calculi: Featherweight Java (FJ) which models single-class inheritance and Featherweight Generic Java (FGJ) which adds type parameters to the language, and then proceeds to define a way to compile FGJ to FJ via erasure.

We define Pathless Scala (PS) as an extension of FGJ without casts (which we choose to not study), adding multiple inheritance via traits and intersection types in the style of DOT. Unlike DOT and as its name indicate, PS lacks type members and therefore path-dependent types.

Real Scala compilers erase traits to Java interfaces, but FJ does not model interfaces so cannot be directly used as a target for our erasure. Instead our target calculus is a fragment of FJ&λ[Bettini et al. 2018] which extends FJ with interfaces. FJ&λ also supports intersections and lambdas, but because these features are not present in Java bytecode, they are not useful for our purpose and we do not use them in our erasure mapping.

2 Syntax

\[
\begin{align*}
C, D, E & \quad \text{class name} \\
X, Y, Z & \quad \text{type variable} \\
N, P, Q & \quad \text{non-variable} \\
S, T, U & \quad \text{type} \\
L & \quad \text{class declaration} \\
H & \quad \text{abstract method} \\
x, y, z & \quad \text{expression} \\
e & \quad \text{variable} \\
f, g & \quad \text{method call} \\
e.m & \quad \text{object} \\
\end{align*}
\]

\[\text{Syntax of Pathless Scala}\]

¹The answer to this question matters even when compiling Scala to a different backend such as JavaScript, because alternative backends strive to preserve the semantics of the JVM to ease cross-compilation [Doeraene 2018, § 2.1]
To ease comparison between PS and FGJ, we reuse as much as possible the conventions of FGJ: our metavariables (Figure 1) with the same name have similar meanings, and our typing and erasure rules reuse the name and format of existing rules in FGJ when appropriate. Our notations are essentially the same: we write "⊂" as a shorthand for "extends", an overline represents a possibly empty list or set, • is the empty list or set and a comma allows concatenating one or more element to the front of a list. We also make use of wavy underlines to denote an optional part of a rule (if multiple parts of a rule are marked optional, it means they must all be present or all be absent, no in-between).

Just like in FGJ, a PS program consists of a class table (which maps a class name to a class declaration) followed by an expression. The declaration of a class defines its name, abstract methods are allowed in traits but not in proper classes). Proper classes also define a list of term parameters which serve both as constructor parameters and getters. Our syntax is faithful to Scala, except that we omit the val keyword in front of constructor parameters which we normally need to generate getters. We informally define the mapping between well-formed cast-less FGJ programs and PS programs with an example. The FGJ class declaration:

```
class A<T> < B<T> {  
D y;  
A(C x, D y) {  
  super(x);  
  this.y = y;  
}  
< ⊂ T> C foo(C z) { return new B<S>(z).bar<C>(y).f; }  
}
```

can be translated into:

```
class A[T](x: C, y: D) < B[T](x) {  
def foo[S <: T](z: C) = new B[S](z).bar[C](y).f  
}
```

We do not need to support translating more complex constructors because of the restrictions imposed on well-formed classes by FGJ.

For convenience, we define auxiliary functions returning the parents and the method declarations of applied class types:

\[
\text{parents}(\text{Object}) = \bullet
\]

\[
\text{mdecls}(\text{Object}) = \bullet
\]

\[
\begin{align*}
\text{parents}(C[\bar{X} <: N]) & \triangleq \bar{P} \{ \bar{H}; \bar{M} \} \\
\text{mdecls}(C[\bar{T}]) & = [\bar{T}/\bar{X}]\{\bar{H}; \bar{M}\}
\end{align*}
\]

As an example, using the definition of A above we have:

\[
\text{parents}(A[\text{Object}]) = B[\text{Object}] \\
\text{mdecls}(A[\text{Object}]) = x : C, y : D
\]

The set of base types of \(N\) is the transitive closure of \(\text{parents}(N)\) plus \(N\) itself.

## 3 Multiple Inheritance in Scala

The main difference between trait inheritance in Scala and interface inheritance in Java is that the order in which parent traits are inherited matter. In particular, Scala defines a canonical order of the base types of a class called its linearization.

### 3.1 \(\mathcal{L}(N)\): The Base Types of \(N\) in Linearization Order

[Odersky and Zenger 2005] defines linearization for class types \(C\), but it is useful to generalize it to non-variable types \(N\):

\[
\mathcal{L}(N) = N, \mathcal{L}(N_n) \circlearrowleft \ldots \circlearrowleft \mathcal{L}(N_1)
\]

Where \(\circlearrowleft\) denotes concatenation with elements on the right replacing identical elements of the left operand. It is illegal to inherit the same class twice if it is applied to different type arguments and so \(\circlearrowleft\) is undefined in that case:

\[
\bullet \circlearrowleft \bar{N} = \bar{N}
\]

\[
N_0 \in \bar{N}_r
\]

\[
(N_0, \bar{N}_l) \circlearrowleft \bar{N}_r = \bar{N}_l \circlearrowleft \bar{N}_r
\]

\[
N_0 = C_0[\ldots] \quad \bar{N}_r = C_r[\ldots] \quad C_0 \not\in C_r
\]

\[
(N_0, \bar{N}_l) \circlearrowleft \bar{N}_r = N_0, (\bar{N}_l \circlearrowleft \bar{N}_r)
\]

We will use linearization to determine which base type of \(N\) contains the implementation of \(m\) that will be called at runtime which we dub the implementer of \(m\) in \(N\).

### 3.2 \(\text{mimpl}(m, N)\): The Implementer of \(m\) in \(N\)

The following class table is legal:

```
class One; class Two
trait Base { def foo(): Object }
trait Sub1 < Base ( def foo(): Object = new One )
trait Sub2 < Base ( def foo(): Object = new Two )
class A < Base, Sub1, Sub2
```

\(^2\text{In real Scala this is in fact possible with variant type parameters.}\)
However, the equivalent class table in Java (using `interface` instead of `trait`) would be illegal: both Sub1 and Sub2 contain a concrete implementation of `foo` and neither trait overrides the other. By contrast, in Scala this typechecks ³ and (new A).`foo()` will evaluate to new Two() because Sub2 precedes Sub1 in the linearization of A.

In general, concrete methods override abstract methods in both Java and Scala, but if we compare a concrete method \( M \) defined in \( C \) with another concrete method \( M' \) defined in \( D \) then:

- In Java, \( M \) overrides \( M' \) if \( D \) is a base type of \( C \).
- In Scala, \( M \) overrides \( M' \) in \( N \) if \( C \) precedes \( D \) in \( \mathcal{L}(N) \).

Since a type \( P \) will always appear before its parent in any linearization involving \( P \), this generalizes the Java rule.

From this it follows that \( \text{mimpl}(m, N) \) must be the first type in \( \mathcal{L}(N) \) containing a concrete declaration of \( m \). In the example above we had \( \mathcal{L}(A) = A, \text{Sub2}, \text{Sub1}, \text{Object} \) and so we get \( \text{mimpl}(\text{foo}, A) = \text{Sub2} \) as expected.

For a class \( C \) to be well-formed, it is not enough for \( \text{mimpl} \) to be defined for all its members, we must also check that the selected implementations are valid overrides.

### 3.3 isValid(m, C): The Implementation of m in C Is a Valid Override of All Declarations of m in the Base Types of C

Like in Java, for an override to be valid its type must match the type of all the overridden methods, meaning the type and term parameters must be identical (up to alpha-renaming which we do not model) and the result type is allowed to vary covariantly. Additionally, the override must not be accidental, a concept specific to Scala.

```scala
class C[X <: N] ... N_i = mimpl(m, C[X])
\forall n \in \mathcal{L}(C[X]), \text{override}_{C[N]}(m, N_i, n)
and noAccidentalOverride(m, N_i, n)
```

#### 3.3.1 override₃(m, N_i, N_o): the Type of m in N_i Overrides the Type of m in N_o in the Type Environment \( \Delta \).

The following class table is illegal:

```scala
class One; class Two
trait L { def foo(): One = new One }
trait R { def foo(): Two = new Two }
class LR <= L, R ()
```

Even though \( \text{mimpl}(\text{foo}, \text{LR}) = \text{R} \), the override is invalid because Two is not a subtype of One. This is enforced by:

```
def m[Y <: P](x: T): T_i = ... \in \text{mdecls}(N_i)
def m[Y <: P](x: T): T_o \equiv ... \in \text{mdecls}(N_o)

\text{override}_\Delta(m, N_i, N_o)
```

Interestingly, our definition of override is more expressive than the one in FGJ as it takes a type environment \( \Delta \) representing the type parameters of the class. This is needed to be able to typecheck:

```scala
class X
class Base { def foo(): X = ... }
class S <: X <: Base { def foo(): S = ... }
```

The corresponding Java code is valid and yet Sub is not well-formed in FGJ because the type parameter \( S <: X \) is not in the type environment when the override check is done [Igarashi et al. 2001, Figure 6].

#### 3.3.2 noAccidentalOverride(m, N_i, N_o): m in N_i Does Not Accidentally Override m in N_o.

The following class table is not well-formed in Scala:

```scala
class One; class Two
trait Base { def foo(): Object }
trait Sub1 <: Base { def foo(): Object = ... }
trait Unrelated { def foo(): Object }
trait Sub2 <: Unrelated { def foo(): Object = ... }
class A <: Object, Sub1, Sub2
```

Although we have \( \text{mimpl}(\text{foo}, A) = \text{Sub2} \) and \( \text{override}(m, \text{Sub2}, \text{Sub1}) \), the compiler complains ⁴:

```
method foo in trait Sub2 cannot override a concrete member without a third member that's overridden by both (this rule is designed to prevent "accidental overrides")
```

In other words, when \( N_i \) overrides a concrete member \( m \) defined in \( N_o \), we must ensure that \( N_i \) and \( N_o \) have a common base type which also declares \( m \):

```scala
isConcrete(m, N_o) \rightarrow N_c = \mathcal{L}(N_i) \cap \mathcal{L}(N_o)
\exists n \in N_c . def m ... \in \text{mdecls}(n)
```

#### 4.1 Subtyping and Well-Formedness

Most of the subtyping and well-formedness rules (Figures 2 and 3) are straightforward adaptations of the FGJ rules with S-CLASS and WF-CLASS generalized to handle traits. The

---

³To be precise, foo in Sub2 needs to be declared with the `override` keyword for A to compile, but we do not model this in our calculus: when translating code from PS into real Scala, `override` should be added everywhere it is legal to do so [Odersky et al. 2021, § 5.2.3].

⁴after adding `override` to the definition of foo in Sub2
Δ ⊢ S <: S [S-REFL]  Δ ⊢ X <: Δ(X) [S-VAR]  
Δ ⊢ S <: T  Δ ⊢ T <: U  
Δ ⊢ S <: U  [S-TRANS]  
P ∈ parents(N)  
Δ ⊢ N <: P  [S-CLASS]  
Δ ⊢ T <: T1 & T2 <: T  [S-AND1]  
Δ ⊢ T1 & T2 <: T  [S-AND2]  

Figure 2. Subtyping

Δ ⊢ Object ok [WF-OBJECT]  
Δ ⊢ X ok [WF-VAR]  

\[
\begin{align*}
\text{class} & \quad \Delta \vdash \text{trait } C[X <: N] \ldots \quad \Delta \vdash \overline{T} ok \quad \Delta \vdash \overline{T} <: [\overline{T}/X]N \\
& \quad \Delta \vdash C[\overline{T}] ok
\end{align*}
\]  
[WF-CLASS]  
Δ ⊢ T1 ok  Δ ⊢ T2 ok  [WF-AND]  

Figure 3. Well-formed types

rules for intersections however are lifted from the DOT calculus [Rompf and Amin 2016, Figure 1]. The subtyping relationship defined by these rules induces a partial order in which \(T_1 \& T_2\) is the greatest lower bound of \(T_1\) and \(T_2\).

Note that intersections in Scala are first-class types: they can appear in any position and members of an intersection can be arbitrary types, by contrast in Java the operands of the intersection cannot be type variables and the intersection itself can only appear in casts and upper-bounds of type parameters.

4.2 Typing Expressions

The typing rules for expressions (Figure 4) are also very close to the FGJ rules, but the helper functions \(\text{getters}\) (which corresponds to \text{fields} in FGJ) and \text{mtype} are different. Both of these functions take a subscript representing the type environment.

4.2.1 \(\text{getters}_\Delta(T)\): The List of Getters Accessible From a Value of Type \(T\). We can read the getters of a class directly from its definition (we do not need to recurse on its parent class because \text{GT-CLASS} ensures that the class parameters of a well-formed class includes the class parameters of its parent):

\[
\begin{align*}
\text{getters}_\Delta(X) &= \Delta(X) \\
\text{getters}_\Delta(\text{Object}) &= \bullet
\end{align*}
\]

\[
\begin{align*}
\text{class } C[X <: N](f : U) \ldots \\
\text{getters}_\Delta(C[\overline{T}]) &= [\overline{T}/X]f : U
\end{align*}
\]

In an intersection, the getters will usually be defined only on one side, but if they happen to be defined on both sides we can safely take the union without worrying about the same getter appearing with different types since \text{GT-CLASS} also ensures that we cannot inherit from multiple unrelated classes and that getters in sub-classes and super-classes have matching types:
What is the type of `x.foo()`? In Java (and FJ&\lambda) this would be an error, even though one can override both of these methods at once via covariant overriding. The problem is that there is no Java type representing the greatest lower bound of `A` and `B`, whereas as we’ve seen above in Scala this is simply `A & B`. This means we can define:

\[
\begin{align*}
\text{mtype}\_\Delta(m, T_1) &= \{ Y < : P \rightarrow S \rightarrow U_1 \\
\text{mtype}\_\Delta(m, T_2) &= \{ Y < : P \rightarrow S \rightarrow U_2 \\
\Delta_m &= \Delta_Y < : P \\
U_{12} &= \begin{cases} U_1 & \text{if } \Delta_m \vdash U_1 < : U_2 \\
U_2 & \text{if } \Delta_m \vdash U_2 < : U_1 \\
U_1 \cup U_2 & \text{otherwise} \end{cases}
\end{align*}
\]

(we could also write `U_{12} = U_1 \cup U_2` if we took care to always normalize types when comparing them for equality). The rules for the remaining cases are then unsurprising:

\[
\text{mtype}\_\Delta(m, X) = \text{mtype}\_\Delta(m, \Delta(X))
\]

\[
\text{def } m[Y < : P](x : T) : T \equiv m \in \text{mdecls}(N)
\]

\[
\text{mtype}\_\Delta(m, N) = \{ Y < : P \rightarrow \overline{S} \rightarrow T \}
\]

\[
\text{parents}(N) = N_1, ..., N_n \quad \text{def } m ... \notin \text{mdecls}(N)
\]

\[
\text{mtype}\_\Delta(m, N) = \text{mtype}\_\Delta(m, N_1 & ... & N_n)
\]

\[
\text{mtype}\_\Delta(m, T_1) \text{ defined } \quad \text{mtype}\_\Delta(m, T_2) \text{ undefined}
\]

\[
\text{mtype}\_\Delta(m, T_1 \& T_2) = \text{mtype}\_\Delta(m, T_1)
\]

\[
\text{mtype}\_\Delta(m, T_1) \text{ undefined } \quad \text{mtype}\_\Delta(m, T_2) \text{ defined}
\]

\[
\text{mtype}\_\Delta(m, T_1 \& T_2) = \text{mtype}\_\Delta(m, T_2)
\]

4.3 Typing Declarations

Figure 5 lists the typing rules for declarations. Unlike FGJ override validation happens when typing the class (using the isValid judgment we defined in the previous section) and not the method since we need to check the validity of overrides defined in parents too. Unlike Scala, we do not perform override validation in traits since these checks are redundant with the ones done on the classes extending those traits, although in practice it’s of course better to find out about errors at the definition site rather than at the use site.

4.3.1 Checking for Abstract Methods in Classes. One might assume that a method is abstract in a class if there are no concrete implementation of this method among its base types. However, both Java and Scala 3 allow "re-abstacting" a method, for example in:

trait Base { def foo(): Object = ... }
trait Sub < Base { def foo(): Object }

class A < Object, Sub
class B < Object, Base, Sub

`A` and `B` have the same linearization so we’d expect them to be equivalent, but in fact an inherited method is considered abstract in a class if it is abstract among all the parents of this class, so `A` is not well-formed since it only inherits an abstract `foo` from `Sub`. Although this concept exists in Java, it is not modeled in FJ&\lambda which does not allow an abstract method to override a concrete one. To represent this we define the mutually recursive `mnames_{abs}(N)` and `mnames_{con}(N)` to be the sets of names of respectively abstract and concrete members of `N`:

\[
\text{mdecls}(N) = \text{def } m_{abs} ... ; \text{def } m_{con} ... = ...
\]

\[
\text{parents}(N) = m_{con} \cup (m_{names}_{con}(P) \setminus m_{abs})
\]

\[
\text{mnames}_{con}(N) = m_{abs} \cup (m_{names}_{abs}(P) \setminus m_{names}_{con}(P))
\]

GT-CLASS then takes care of checking that `mnames_{A}` is empty for proper classes. For convenience we also define the set of all method names in `N` as:

\[
\text{mnames}(N) = m_{names_{abs}(N)} \cup m_{names_{con}(N)}
\]

5 Erasure

Our target calculus is FJ&\lambda without lambdas or intersections, we name the resulting fragment Featherweight Java with Default methods (FJD)\(^6\).\(^5\)

5.1 Type Erasure

Given a type environment `\Delta`, we write `|T|_\Delta` for the type erasure of `T` which is defined in FGJ as:

\[
\frac{\text{def } m[Y < : P](x : T) : T \equiv m \in \text{mdecls}(N)}{|X|_\Delta = |\Delta(X)|_\Delta}
\]

\[
|C[\ldots]|_\Delta = C
\]

In general, we strive to have erasure preserve as much of the structure of the original program as possible to keep the translation simple and to allow interoperability between programs written in the source and target language. In particular, the mapping above preserves subtyping in FGJ: if

\(^5\)see the definition of `m` in [Bettini et al. 2018, p. 15]

\(^6\)FJ was already taken by Featherweight Java with inner classes [Igarashi and Pierce 2002].
\[
\begin{align*}
\Delta, \Gamma & \vdash x : T & \text{[GT-VAR]} \\
\Delta, \Gamma & \vdash e_0 : T_0 & \text{getters}_\Delta(T_0) = f : T \\
\Delta, \Gamma & \vdash e_0.f_j : T_j & \text{[GT-FIELD]} \\
\Delta & \vdash V \text{ ok} & \Delta & \vdash V <: [V/Y]P & \Delta, \Gamma & \vdash z : S & \Delta & \vdash S <: [V/Y]U & \text{[GT-INVK]} \\
\Delta, \Gamma & \vdash e_0, m[V][\bar{z}] : [V/Y]U_0 \\
\Delta & \vdash N \text{ ok} & \text{getters}_\Delta(N) = f : T & \Delta, \Gamma & \vdash \bar{S} : T & \Delta & \vdash S <: T & \text{[T-NEW]}
\end{align*}
\]

Figure 4. Syntax Directed Typing Rules

\[
\begin{align*}
\Delta & = X <: N, Y <: P & \Delta & \vdash T, T, P \text{ ok} & \Delta & \vdash x : T, \text{this : } C[X] & \vdash e_0 : S & \Delta & \vdash S <: T & \text{[GT-METHOD]}
\end{align*}
\]

\[
\text{def } m[Y <: P](x : T) : T = e_0 \text{ OK IN } C[X <: N]
\]

If \( \text{mimpl}(m, C[X]) \) is defined then \( \text{isValid}(m, C) \) must also be defined \( \text{mnames}_{abs}(C) = \bullet \) [GT-CLASS]

\[
\text{class } C[X <: N](g : U, f : T) \sim P(g), Q \{ M \} \text{ OK}
\]

\[
\begin{align*}
\Delta & \vdash S <: FGD T & \text{then } |S|_\Delta <: FGD |T|_\Delta \text{ (Lemma A.3.5}_{FGD} \) which reduces the amount of casts that need to be inserted when erasing expressions to a minimum (Theorem 4.5.3}_{FGD}).
\end{align*}
\]

Unfortunately, no matter how we erase intersection types, we cannot preserve subtyping in general because although \( T_1 \& T_2 \) is the greatest lower bound of \( T_1 \) and \( T_2 \), there might not exist a specific type in FJD representing the greatest lower bound of \( |T_1|_\Lambda \) and \( |T_2|_\Lambda \). Nevertheless, since we’re trying to preserve as much structure as possible, it seems logical to define:

\[
|T_1 \& T_2|_\Lambda = \text{erasedGlb}(|T_1|_\Lambda, |T_2|_\Lambda)
\]

where \( \text{erasedGlb} \) always returns one of its arguments. In fact this is what both Java and Scala do, but they differ on the implementation of \( \text{erasedGlb} \):

- Java simply defines \( \text{erasedGlb}(T_1, T_2) = T_1 \) [Gosling et al. 2015, § 4.6]. This means the user can tweak the erasure by reordering types which can be useful for evolving code in a binary-compatible way.

\( \text{erasedGlb}(T_1, T_2) = T_1 \) [Gosling et al. 2015, § 4.6]. This means the user can tweak the erasure by reordering types which can be useful for evolving code in a binary-compatible way.

\( \text{erasedGlb}(T_1, T_2) = T_1 \) [Gosling et al. 2015, § 4.6]. This means the user can tweak the erasure by reordering types which can be useful for evolving code in a binary-compatible way.

\( \text{erasedGlb}(T_1, T_2) = T_1 \) [Gosling et al. 2015, § 4.6]. This means the user can tweak the erasure by reordering types which can be useful for evolving code in a binary-compatible way.
\[ |x|_{\Delta, \Gamma} = x \quad [\text{E-VAR}] \]
\[
\Delta; \Gamma \vdash e_0 : T_0 \quad |T_0|_{\Delta} = C \quad [\text{E-FIELD}] \]
\[
\Delta; \Gamma \vdash e_0 : T_0 \quad \text{erasedReceiver}_\Delta(m, T_0) = C \quad m \text{type}_{\text{FJD}}(m_C, C) = \overline{U} \rightarrow U_0 \quad e'_i = |e|_{\Delta, \Gamma}^{U_i} \quad [\text{E-INVK}] \]
\[
\Delta; \Gamma \vdash e_0 : T_0 \quad |e_0.m[\overline{V}](\overline{e})|_{\Delta, \Gamma} = |e_0|_{\Delta, \Gamma}^{C.m_C}(e'_i) \quad [\text{E-NEW}] \]
\end{align*}

\textbf{Figure 7. Expression Erasure}

\begin{equation}
\Gamma = x : T, \text{this} : C[\overline{X}] \quad \Delta = X <: N, Y <: P \quad [\text{E-METHOD}] \end{equation}
\[
\Delta = X <: N \quad m \text{type}(\overline{V} : P)(x : T) : T_0 = e_0|_{\overline{X} <: N, C} = |T_0|_{\Delta} m_C(x) \{ \text{return } |e_0|_{\Delta, \Gamma}^{T_i} \} \]
\end{equation}

\begin{equation}
K = C[|U|_{\Delta} g, |T|_{\Delta} f](\text{super}(\overline{g}) ; \text{this} ; \overline{f}) ; \} \quad M = |M|_{\Delta} \cup \{ \text{bridges}(m, C) | \forall m \in \text{mnames}(C) \} \quad [\text{E-CLASS}] \end{equation}
\[
\text{class } C[|X <: N](g : U, f : T) < P(\overline{g}), \overline{Q} \{ \overline{M} \}] = \text{class } C < |P|_{\Delta}, \overline{Q}\{ |T|_{\Delta} f ; K ; M \} \]
\end{equation}
\[
\Delta = X <: N \quad M' = |M|_{\Delta, C} \quad [\text{E-TRAIT}] \]

\textbf{Figure 8. Class Table Erasure}

- On the other hand, Scala 2 defines \texttt{erasedGlb} to prefer subtypes over supertypes (thus actually returning the greatest lower bound of the erased types) and proper classes over traits (because both casting and method call are usually faster on classes than on interfaces [Click and Rose 2002; Shipilev 2020]). Unfortunately, completely specifying the behavior of Scala 2 here is extremely hard because it inadvertently depends on implementation details of the compiler.

- Scala 3 preserves the two properties from Scala 2 mentioned above and additionally ensures that erase preserves commutativity of intersection (\(|T_1 \cap T_2|_{\Delta} = |T_2 \cap T_1|_{\Delta}\)) by applying a tie-break based on the lexicographical order of the names of the compared types. The following pseudo-code accurately specifies its behavior:

```scala
1 def erasedGlb(tp1: Type, tp2: Type): Type =
2   if tp1.isProperClass && !tp2.isProperClass then
3     return tp1
4   if tp2.isProperClass && !tp1.isProperClass then
5     return tp2
6   if tp1 <: tp2 then return tp1
7   if tp2 <: tp1 then return tp2
8   if tp1.name < tp2.name then tp1 else tp2
```

The Scala 3 algorithm preserves most interesting properties of intersections but has one non-obvious shortcoming: it does not preserve associativity, consider:

\begin{verbatim}
trait X; trait Y; trait Z extends X
\end{verbatim}

Then \(|X \& Y \& Z| = Z \& (X \& (Y \& Z)) = X\). The problem is that while the lexicographic ordering by itself is total, it is applied inconsistently because \textit{incomparability of subtyping is not transitive}: in our example neither \(X <: Y\) nor \(Y <: X\) making \(X \& \) and \(Y \& \) incomparable, but even though \(Y \& \) and \(Z\) are also incomparable it is not true that \(X \& \) and \(Z\) are incomparable.

To rectify this we propose \textit{ordering classes by the number of base types they have}. In other words, we replace the subtyping checks on lines 6 and 7 in the listing above by:

```scala
val relativeLength = L(tp1).length - L(tp2).length
if relativeLength > 0 then return tp1
```

\footnote{Since this change would break binary compatibility, it will have to wait until the next major version of Scala.}
if relativeLength < 0 then return tp2

This still means we prefer subtypes over supertypes since a subclass necessarily has more base types than any of its parent, but incomparability is now transitive which is enough to make erasedGlb itself transitive.

In the rest of this section, we will assume erasedGlb prefers classes over traits as well as subtypes over supertypes but otherwise will stay independent of any particular implementation.

5.2 Expression Erasure

Because type erasure does not preserve subtyping we might need to insert casts both on prefixes of calls as well as on method arguments. To keep the typing rules in Figure 7 readable, we delegate casting $|e|_{\Delta \Gamma}$ to $T$ to an auxiliary judgment $|e|^T_{\Delta \Gamma}$ which is mutually recursive with the main judgment:

$$
\begin{align*}
|e|^T_{\Delta \Gamma} &= \\
&= \begin{cases} 
  e' & \text{if } S <\!:\!FJD T \\
  (T)e' & \text{otherwise}
\end{cases}
\end{align*}
$$

Casting the prefix of a getter call to the appropriate type is easy: we know that erasedGlb will always return the most specific class type in an intersection and that traits do not contain getters, therefore if $\text{getters}_A(T_0) = f : T$ then $\text{fields}_{FJD}([T_0]_A) = f : [T]_A$ and E-FIELD is straightforward, but finding the right cast for the receiver of a method call is more involved.

5.2.1 $\text{erasedReceiver}_{\Delta}(m, N)$: The First Erased Parent Type Where $m$ Is Defined.

Given $x : L & R$ and the class table:

trait L { def l(): Object }
trait R { def r(): Object }

Then the type of $|x|_{\Delta \Gamma}$ will be either $L$ or $R$ (depending on the definition of erasedGlb), but that means that one of $x. l()$ and $x. r()$ will require casting the receiver, therefore E-INVK relies on the following auxiliary function:

$$
\text{erasedReceiver}_{\Delta}(m, X) = \text{erasedReceiver}_{\Delta}(m, \Delta(X))
$$

\text{erasedReceiver}_{\Delta}(m, C[\ldots]) = C

\text{erasedReceiver}_{\Delta}(m, T_1 & T_2) =

\begin{align*}
\text{erasedReceiver}_{\Delta}(m, T_1) & \text{ if } \text{mtype}_{\Delta}(m, T_1) \text{ is defined} \\
\text{erasedReceiver}_{\Delta}(m, T_2) & \text{ otherwise}
\end{align*}

Additionally, erasure does not preserve method names: $m$ is erased to $m_C$ where $C$ is the type of the receiver, this is justified in the following section.

5.3 Class Table Erasure

Given the class table:

trait X; class Y extends X
trait L[T] { def foo(): T }

trait R[T <: X] { def foo(): T }
class A <: Object, L[Y], R[Y] {
  def foo(): Y = new Y
}

One might hope we could erase it just by erasing each type and expression appearing in it:

interface L { Object foo() }
interface R { X foo() }
class A <: Object, L, R {
  Y foo() { return new Y(); }
}

But that would be incorrect: a method in FJD must have exactly the same type as the methods it overrides (just like in Java bytecode). Compilers normally handle this by generating synthetic bridge methods [Bracha et al. 2003]:

interface L { Object foo() }
interface R { X foo() }
class A <: Object, L, R {
  Y foo() { return overload of foo returning Y(); }
  X foo() { return overload of foo returning Y(); }
}

Notice that the types of the new methods added in $A$ match the types of the overridden methods in $L$ and $R$ and simply forward to the actual implementation of Foo in $A$, thus restoring the semantics present in the source program. But we cannot directly reuse this technique since our target calculus does not support overloading, faced with the same problem FGJ adopted the following strategy:

In [Generic Java], the actual erasure is somewhat more complex, involving the introduction of bridge methods [...] instead, the rule E-METHOD merges two methods into one by inline-expanding the body of the actual method into the body of the bridge method.

But this works because FGJ only supports single-class inheritance, whereas in the example above we need two bridges in $A$ corresponding to the two traits containing an overridden foo. Like FGJ, we shy away from introducing overloading in our target calculus and instead employ the following scheme:

- When erasing a call to $m$, we replace it by a call to $m_C$ where $C$ is the erased receiver of $m$ (see the previous section).
- When erasing the declaration of $m$ in $C$, we rename it to $m_C$.
- When erasing a class $C$, we add enough bridge methods so that erased calls to $m$ always end up being forwarded to the implementer of $m$ in $C$.

For our example this means we get:

interface L { Object fooL() }
interface R { X fooR() }
class A <: Object, L, R {
  Y fooY { return new Y(); }
}
This scheme wouldn’t be practical in a real compiler since it would make it much harder for Java and Scala code to interoperate, but as a model we believe it’s close enough to the real thing to be useful. The exact rules are described in Figure 8 which makes use of the following judgments:

\[ \text{mtype}_{\text{FD}}(m_E, E) = \overline{T} \rightarrow T_0 \]
\[ \text{mtype}_{\text{FD}}(m_D, D) = U \rightarrow U_0 \]
\[ x_i = \text{this}.\overline{m}_D(\overline{e}) \quad e_i = \begin{cases} x_i & \text{if } T_i = U_i \\ (U_i)x_i & \text{otherwise} \end{cases} \]
\[ \text{bridge}(m_E, m_D) = T_0 \text{mtype}_{\text{FD}}(\overline{T} x) \{ \text{return } e_0; \} \]
\[ \text{minimpl}(m, N) = D[\overline{T}] \]
\[ \overline{E}(... \subseteq \{ n \in \mathcal{L}(N) \setminus D[\overline{T}] \mid \text{def } m \ldots \in \text{mdecls}(n) \} \]
\[ \text{bridges}(m, N) = \text{bridge}(m_E, m_D) \]

Note that this definition of bridges can generate unnecessary bridges since it does not take into account that a parent class might already have defined an equivalent bridge.

### 6 Related Work

#### 6.1 Multiple Inheritance and the Diamond Problem

What should happen when multiple matching methods from unrelated classes are inherited? There is no standard solution here but languages usually pick one of the following approaches:

- In Java and C++ with virtual inheritance, the class definition is considered invalid and an error is emitted.
- In C++ with non-virtual inheritance, the ambiguity resolution is delayed until the method call site, where the user can upcast the receiver to manually resolve the ambiguity. See [Waterson et al. 2006] for a precise treatment of inheritance in C++ including a soundness proof but make sure to prepare a pot of coffee first.
- Several languages like Scala will attempt to determine a linearization order for the parent classes and use that to resolve the ambiguity. The C3 linearization algorithm [Barrett et al. 1996] originally defined for Dylan is especially popular, being notably used by Python and Raku. This form of linearization is guaranteed to be monotonic: two classes will always appear in the same order in any given linearization, this isn’t true in Scala when traits are involved which lets us define class hierarchies more freely at the cost of making linearization harder to reason about.

#### 6.2 Related Calculi

**Featherweight Java** was first extended with interfaces and intersection types faithful to Java semantics in FJ&\& [Bettini et al. 2018]. The semantics of intersection types were then generalized beyond what Java supports in FJ&\& [Dezani-Ciancaglini et al. 2019] to allow intersections in any position (like Scala) and not just as target of casts, finally [Dezani-Ciancaglini et al. 2020] showed how to erase FJ&\& into FJ&\&. Pathless Scala can be seen as a generalization of FJ&\&\&, but we found it easier to extend FGJ with traits and intersections rather than to extend FJ&\&\& with polymorphism and generalize its interfaces to traits. However, FJ&\&\& stripped of intersections and lambdas makes for a great target calculus as it closely models most of the important aspects of Java bytecode, although we would really need to extend it with overloading to describe Scala’s erasure faithfully.

**Featherweight Scala** (FS) [Cremet et al. 2006] is not an extension of Featherweight Java despite its name: it does have nominal classes (including inner classes) but uses type members and path-dependent types rather than type parameters and is therefore more closely related to DOT. FS also includes multiple inheritance via traits, but it does not precisely model the overriding rules of Scala like we do in Section 3.

DOT was first described in [Amin et al. 2012] but wasn’t proved sound until [Amin et al. 2016], although this version of DOT lacked union and intersection types. A soundness proof for DOT with intersections was then presented in [Rompf and Amin 2016], and unions finally made a comeback in [Giarrusso et al. 2020]. DOT has also been extended in multiple ways to bring it closer to Scala [Kabir and Lhoták 2018; Rapoport 2019; Stucki and Giarrusso 2021] but the gap between the two remains large.

### 7 Conclusion and Future Work

We have presented Pathless Scala, a convenient calculus for formalizing the semantics and compilation schemes of parts of Scala which we found to be understudied. In particular, we believe its important to specify language features and their erasure together rather than leaving the latter as an implementation detail. They inevitably leak to the user (e.g., via Java reflection) and interoperability (of Scala 2 code with Scala 3 code, or of Scala code with Java code) requires the same type to be erased in the same way by multiple different compilers. We know from having had to reverse-engineer how Scala 2 erasure works that this can end up being much harder than it needs to be. Therefore, we are particularly interested in extending Pathless Scala to cover other aspects of Scala with non-trivial erasures such as union types or polymorphic function types. Eventually, this could serve as a basis for a more precise version of the Scala Language Specification [Odersky et al. 2021].

In this work we’ve focused on erasing Scala types into "bytecode Java" types, but in practice we also need to worry about erasing Scala types into "source Java" types: the bytecode format defines a Signature attribute [Lindholm et al. 2015, § 4.7.8] which lets us specify a polymorphic Java
method signature that will be ignored by the JVM at runtime but used by the Java compiler for typechecking, thus improving the interoperability between Scala and Java. It would be useful to specify an erasure from PS into full FJ&\lambda as a way to model this process. The Java compiler will also use this attribute if it is available to compute the erased signature it will emit when invoking the method, therefore we should also define an erasure of FJ&\lambda into FJD based on the semantics of Java erasure and verify that the composition of these two mapping are equivalents to the erasure mapping of PS into FJD to avoid issues such as


We did not define evaluation semantics for PS, instead we described erasure rules to a simpler calculus known to be sound. For the sake of rigor, it would be good to follow the FGJ model: give evaluation rules to our calculus independent of its erasure, prove soundness, and show that directly evaluating a PS program is equivalent to erasing and then evaluating it. Given that our calculus intentionally excludes the hard parts of DOT, we believe that the existing proofs given in the FJ paper can be extended in a straightforward way to achieve this, but we have not completed this work yet.

Of course, eventually we should also strive to reconcile Pathless Scala and DOT, but that is likely to be a much longer-term project given how difficult it has been to extend the meta-theory of DOT so far, meanwhile the rest of Scala awaits us!

References


