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

ACM Reference Format:
Guillaume Martres. 2021. Pathless Scala: A Calculus for the Rest of Scala. In *Proceedings of the 12th ACM SIGPLAN International Scala Symposium (SCALA '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486610.3486894

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

C, D, E	**class name**
X, Y, Z	**type variable**
$N, P, Q ::= C[\overline{T}]$	**non-variable type**
$S, T, U ::= X \mid N \mid T \,\&\, T$	**type**
$L ::=$	**class declaration**
$\quad \text{class}\, C[\overline{X <: N}](\overline{f : T}) \triangleleft N(\overline{f}), \overline{N}\, \{\overline{M}\}$	proper class
$\quad \text{trait}\, C[\overline{X <: N}] \triangleleft \overline{N}\, \{\overline{H}; \overline{M}\}$	trait
$H ::= \text{def}\, m[\overline{X <: N}](\overline{x : T}) : T$	**abstract method**
$M ::= H = e$	**concrete method**
f, g	**class parameter**
x, y, z	**variable**
$e ::=$	**expression**
$\quad x$	variable
$\quad e.f$	getter call
$\quad e.m[\overline{T}](\overline{e})$	method call
$\quad \text{new}\, C[\overline{T}](\overline{e})$	object

Figure 1. Syntax of Pathless Scala

SCALA '21, October 17, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9113-9/21/10…$15.00
https://doi.org/10.1145/3486610.3486894

[1] The 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 "$\triangleleft$" as a shorthand for "extends", an overline represents a possibly empty list or set, $\bullet$ 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, type parameters, parent types and member methods (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 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;
  }
  <S ◁ 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): 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:

$$\texttt{parents}(\text{Object}) = \bullet$$
$$\texttt{mdecls}(\text{Object}) = \bullet$$

$$\frac{\text{class}\, C \triangleleft P(\ldots)\,,\ \overline{Q}\ \{\overline{M}\}}{\begin{array}{c}\texttt{parents}(C[\overline{T}]) = [\overline{T/X}](P, \overline{Q}) \\ \texttt{mdecls}(C[\overline{T}]) = [\overline{T/X}]\overline{M}\end{array}}$$

$$\frac{\text{trait}\, C[\overline{X <: N}] \triangleleft \overline{P}\ \{\overline{H}; \overline{M}\}}{\begin{array}{c}\texttt{parents}(C[\overline{T}]) = \text{Object}, [\overline{T/X}]\overline{P} \\ \texttt{mdecls}(C[\overline{T}]) = [\overline{T/X}](\overline{H}, \overline{M})\end{array}}$$

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

$$\texttt{parents}(\texttt{A}[\text{Object}]) = \texttt{B}[\text{Object}]$$
$$\texttt{mdecls}(\texttt{A}[\text{Object}]) = \texttt{def foo}[\texttt{S} <: \text{Object}](\texttt{z} : \texttt{C}) : \texttt{C} = \ldots$$

The set of *base types* of N is the transitive closure of $\texttt{parents}(N)$ 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:

$$\frac{N_1, \ldots, N_n = \texttt{parents}(N)}{\mathcal{L}(N) = N, \mathcal{L}(N_n) \vec{+} \ldots \vec{+} \mathcal{L}(N_1)}$$

Where $\vec{+}$ 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 [2] and so $\vec{+}$ is undefined in that case:

$$\bullet \vec{+} \overline{N} = \overline{N}$$
$$\frac{N_0 \in \overline{N_r}}{(N_0, \overline{N_l}) \vec{+} \overline{N_r} = \overline{N_l} \vec{+} \overline{N_r}}$$
$$\frac{N_0 = C_0[\ldots] \qquad \overline{N_r} = \overline{C_r[\ldots]} \qquad C_0 \notin \overline{C_r}}{(N_0, \overline{N_l}) \vec{+} \overline{N_r} = N_0, (\overline{N_l} \vec{+} \overline{N_r})}$$

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 $\texttt{mimpl}(m, N)$: 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 ◁ Object, Sub1, Sub2
```

[2] 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 $\texttt{mimpl}(m, N)$ must be the first type in $\mathcal{L}(N)$ containing a concrete declaration of m. In the example above we had $\mathcal{L}(\texttt{A}) = \texttt{A}, \texttt{Sub2}, \texttt{Sub1}, \texttt{Object}$ and so we get $\texttt{mimpl}(\texttt{foo}, \texttt{A}) = \texttt{Sub2}$ as expected.

For a class C to be well-formed, it is not enough for 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.

$$\frac{\begin{array}{c}\text{class } C[\overline{X <: N}] \ldots \qquad N_i = \texttt{mimpl}(m, C[\overline{X}]) \\ \forall \mathbf{n} \in \mathcal{L}(C[\overline{X}]).\, \texttt{override}_{\overline{X<:N}}(m, N_i, \mathbf{n}) \text{ and} \\ \texttt{noAccidentalOverride}(m, N_i, \mathbf{n})\end{array}}{\texttt{isValid}(m, C)}$$

3.3.1 override$_\Delta(m, N_i, N_o)$: the Type of m in N_i Overrides the Type of m in N_o in the Type Environment Δ.

The following class table is illegal:

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

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

$$\frac{\begin{array}{c}\text{def } m[\overline{Y <: P}](\overline{x : T}) : T_i = \ldots \in \texttt{mdecls}(N_i) \\ \text{def } m[\overline{Y <: P}](\overline{x : T}) : T_o \mathrel{\tilde{=}} \ldots \in \texttt{mdecls}(N_o) \\ \Delta, \overline{Y <: P} \vdash T_i <: T_o\end{array}}{\texttt{override}_\Delta(m, N_i, N_o)}$$

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

```
class X
class Base { def foo(): X = ... }
class Sub[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:

```
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 $\texttt{mimpl}(\texttt{foo}, \texttt{A}) = \texttt{Sub2}$ and $\texttt{override}(m, \texttt{Sub2}, \texttt{Sub1})$, 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:

$$\frac{\texttt{isConcrete}(m, N_o) \qquad \overline{N_c} = \mathcal{L}(N_i) \cap \mathcal{L}(N_o) \qquad \exists \mathbf{n} \in N_c \,.\, \text{def } m \ldots \in \texttt{mdecls}(\mathbf{n})}{\texttt{noAccidentalOverride}(m, N_i, N_o)}$$

$$\frac{\texttt{isAbstract}(m, N_o)}{\texttt{noAccidentalOverride}(m, N_i, N_o)}$$

4 Typing

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

[3] To 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].

[4] after adding **override** to the definition of foo in Sub2

$$\Delta \vdash S <: S \text{ [S-REFL]} \qquad \Delta \vdash X <: \Delta(X) \text{ [S-VAR]}$$

$$\frac{\Delta \vdash S <: T \qquad \Delta \vdash T <: U}{\Delta \vdash S <: U} \text{ [S-TRANS]}$$

$$\frac{P \in \mathsf{parents}(N)}{\Delta \vdash N <: P} \text{ [S-CLASS]}$$

$$\frac{\Delta \vdash T_1 <: T}{\Delta \vdash T_1 \,\&\, T_2 <: T} \text{ [S-AND11]} \qquad \frac{\Delta \vdash T_2 <: T}{\Delta \vdash T_1 \,\&\, T_2 <: T} \text{ [S-AND12]}$$

$$\frac{\Delta \vdash T <: T_1 \quad \Delta \vdash T <: T_2}{\Delta \vdash T <: T_1 \,\&\, T_2} \text{ [S-AND2]}$$

Figure 2. Subtyping

$$\Delta \vdash \text{Object ok} \text{ [WF-OBJECT]}$$

$$\frac{X \in \mathsf{dom}(\Delta)}{\Delta \vdash X \,\mathsf{ok}} \text{ [WF-VAR]}$$

$$\frac{\left\{ \begin{array}{c} \text{class} \\ \text{trait} \end{array} \right\} C[\overline{X <: N}] \ldots \qquad \Delta \vdash \overline{T} \,\mathsf{ok} \qquad \Delta \vdash \overline{T} <: [\overline{T/X}]\overline{N}}{\Delta \vdash C[\overline{T}] \,\mathsf{ok}} \text{ [WF-CLASS]}$$

$$\frac{\Delta \vdash T_1 \,\mathsf{ok} \qquad \Delta \vdash T_2 \,\mathsf{ok}}{\Delta \vdash T_1 \,\&\, T_2 \,\mathsf{ok}} \text{ [WF-AND]}$$

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 `getters` (which corresponds to `fields` in FGJ) and `mtype` are different. Both of these functions take a subscript representing the type environment.

4.2.1 $\mathtt{getters}_\Delta(T)$: 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 GT-CLASS ensures that the class parameters of a well-formed class includes the class parameters of its parent):

$$\mathtt{getters}_\Delta(X) = \mathtt{getters}_\Delta(\Delta(X))$$
$$\mathtt{getters}_\Delta(\text{Object}) = \bullet$$

$$\frac{\text{class } C[\overline{X <: N}](\overline{f : U}) \ldots}{\mathtt{getters}_\Delta(C[\overline{T}]) = [\overline{T/X}]\overline{f : U}}$$

$$\frac{\text{trait } C[\ldots] \ldots}{\mathtt{getters}_\Delta(C[\overline{T}]) = \bullet}$$

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 GT-CLASS also ensures that we cannot inherit from multiple unrelated classes and that getters in sub-classes and super-classes have matching types:

$$\texttt{getters}_\Delta(T_1 \ \& \ T_2) =$$

$$\begin{cases} \texttt{getters}_\Delta(T_1) \cup \texttt{getters}_\Delta(T_2) & \text{if both sides are defined} \\ \texttt{getters}_\Delta(T_1) & \text{if it is defined} \\ \texttt{getters}_\Delta(T_2) & \text{otherwise} \end{cases}$$

4.2.2 $\texttt{mtype}_\Delta(\texttt{m}, T)$: The Type of the Method *m* in *T*.

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

```
trait L { def foo(): A }
trait R { def foo(): B }
```

What is the type of `x.foo()`? In Java (and FJ&λ) 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:

$$\frac{\begin{array}{c}\texttt{mtype}_\Delta(m, T_1) = [\overline{Y <: P}] \to \overline{S} \to U_1 \\ \texttt{mtype}_\Delta(m, T_2) = [\overline{Y <: P}] \to \overline{S} \to U_2 \\ \Delta_m = \Delta, \overline{Y <: P} \\ U_{12} = \begin{cases} U_1 & \text{if } \Delta_m \vdash U_1 <: U_2 \\ U_2 & \text{if } \Delta_m \vdash U_2 <: U_1 \\ U_1 \ \& \ U_2 & \text{otherwise} \end{cases}\end{array}}{\texttt{mtype}_\Delta(m, T_1 \ \& \ T_2) = [\overline{Y <: P}] \to \overline{S} \to U_{12}}$$

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

$$\texttt{mtype}_\Delta(m, X) = \texttt{mtype}_\Delta(m, \Delta(X))$$

$$\frac{\text{def } m[\overline{Y <: P}](\overline{x : T}) : T \underset{\sim\sim\sim}{=} \ldots \in \texttt{mdecls}(N)}{\texttt{mtype}_\Delta(m, N) = [\overline{Y <: P}] \to \overline{T} \to T}$$

$$\frac{\texttt{parents}(N) = N_1, \ldots, N_n \quad \text{def } m \ldots \notin \texttt{mdecls}(N)}{\texttt{mtype}_\Delta(m, N) = \texttt{mtype}_\Delta(m, N_1 \ \& \ \ldots \ \& \ N_n)}$$

$$\frac{\texttt{mtype}_\Delta(m, T_1) \text{ defined} \qquad \texttt{mtype}_\Delta(m, T_2) \text{ undefined}}{\texttt{mtype}_\Delta(m, T_1 \ \& \ T_2) = \texttt{mtype}_\Delta(m, T_1)}$$

$$\frac{\texttt{mtype}_\Delta(m, T_1) \text{ undefined} \qquad \texttt{mtype}_\Delta(m, T_2) \text{ defined}}{\texttt{mtype}_\Delta(m, T_1 \ \& \ T_2) = \texttt{mtype}_\Delta(m, T_2)}$$

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-abstracting" 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 modelized in FJ&λ which does not allow an abstract method to override a concrete one [5]. To represent this we define the mutually recursive $\texttt{mnames}_{abs}(N)$ and $\texttt{mnames}_{con}(N)$ to be the sets of names of respectively abstract and concrete members of N:

$$\frac{\begin{array}{c}\texttt{mdecls}(N) = \overline{\text{def } m_{abs} \ldots}; \overline{\text{def } m_{con} \ldots = \ldots} \\ \overline{P} = \texttt{parents}(N)\end{array}}{\begin{array}{l}\texttt{mnames}_{con}(N) = \overline{m_{con}} \cup (\overline{\texttt{mnames}_{con}(P)} \setminus \overline{m_{abs}}) \\ \texttt{mnames}_{abs}(N) = \overline{m_{abs}} \cup (\overline{\texttt{mnames}_{abs}(P)} \setminus \texttt{mnames}_{con}(P))\end{array}}$$

GT-CLASS then takes care of checking that $\texttt{mnames}_{abs}$ is empty for proper classes. For convenience we also define the set of all method names in N as:

$$\texttt{mnames}(N) = \texttt{mnames}_{abs}(N) \cup \texttt{mnames}_{con}(N)$$

5 Erasure

Our target calculus is FJ&λ without lambdas or intersections, we name the resulting fragment Featherweight Java with Default methods (FJD) [6].

5.1 Type Erasure

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

$$|X|_\Delta = |\Delta(X)|_\Delta$$
$$|C[\ldots]|_\Delta = C$$

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 `mh` in [Bettini et al. 2018, p. 15]

[6] FJI was already taken by Featherweight Java with Inner classes [Igarashi and Pierce 2002].

$$\frac{x : T \in \Gamma}{\Delta, \Gamma \vdash x : T} \text{ [GT-VAR]}$$

$$\frac{\Delta, \Gamma \vdash e_0 : T_0 \qquad \texttt{getters}_\Delta(T_0) = \overline{f : T}}{\Delta, \Gamma \vdash e_0.f_j : T_j} \text{ [GT-FIELD]}$$

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash e_0 : T_0 \qquad \texttt{mtype}_\Delta(m, T_0) = [\overline{Y <: P}] \to \overline{U} \to U_0 \\ \Delta \vdash \overline{V}\ \mathsf{ok} \qquad \Delta \vdash \overline{V} <: [\overline{V/Y}]\overline{P} \qquad \Delta, \Gamma \vdash \overline{e : S} \qquad \Delta \vdash \overline{S} <: [\overline{V/Y}]\overline{U}\end{array}}{\Delta, \Gamma \vdash e_0.m[\overline{V}](\overline{e}) : [\overline{V/Y}]U_0} \text{ [GT-INVK]}$$

$$\frac{\Delta \vdash N\ \mathsf{ok} \qquad \texttt{getters}_\Delta(N) = \overline{f : T} \qquad \Delta, \Gamma \vdash \overline{e : S} \qquad \Delta \vdash \overline{S <: T}}{\Delta, \Gamma \vdash \mathsf{new}\, N(\overline{e}) : N} \text{ [GT-NEW]}$$

Figure 4. Syntax Directed Typing Rules

$$\frac{\Delta = \overline{X <: N}, \overline{Y <: P} \qquad \Delta \vdash \overline{T}, T_0, \overline{P}\ \mathsf{ok} \qquad \Delta; \overline{x : T}, \mathsf{this} : C[\overline{X}] \vdash e_0 : S \qquad \Delta \vdash S <: T_0}{\mathsf{def}\ m[\overline{Y <: P}](\overline{x : T}) : T_0 = e_0\ \texttt{OK IN}\ C[\overline{X <: N}]} \text{ [GT-METHOD]}$$

$$\frac{\begin{array}{c}\overline{X <: N} \vdash \overline{N}, \overline{U}, \overline{T}, P, \overline{Q}\ \texttt{OK} \qquad \texttt{getters}_\varnothing(P) = \overline{g : U} \qquad \overline{M}\ \texttt{OK IN}\ C[\overline{X <: N}] \qquad \texttt{isClass}(P) \\ \overline{\texttt{isTrait}(Q)} \qquad \text{If } \texttt{mimpl}(m, C[\overline{X}]) \text{ is defined then } \texttt{isValid}(m, C) \text{ must also be defined} \qquad \texttt{mnames}_{abs}(C) = \bullet\end{array}}{\mathsf{class}\, C[\overline{X <: N}](\overline{g : U}, \overline{f : T}) \lhd P(\overline{g})\,,\ \overline{Q}\,\{\overline{M}\}\ \texttt{OK}} \text{ [GT-CLASS]}$$

$$\frac{\overline{X <: N} \vdash \overline{N}, \overline{Q}\ \texttt{OK} \qquad \overline{H}, \overline{M}\ \texttt{OK IN}\ C[\overline{X <: N}] \qquad \overline{\texttt{isTrait}(Q)}}{\mathsf{trait}\, C[\overline{X <: N}] \lhd \overline{Q}\,\{\overline{H}; \overline{M}\}\ \texttt{OK}} \text{ [GT-TRAIT]}$$

Figure 5. Declaration Typing Rules

C, D, E	**type**
$L ::=$	**class declaration**
$\mathsf{class}\, C \lhd C\,,\ \overline{C}\,\{\overline{C\,f}; K; \overline{M}\}$	proper class
$\mathsf{interface}\, C \lhd \overline{C}\,\{\overline{H}; \overline{M}\}$	interface
$H ::= C\, m(\overline{C\,x})$	**abstract method**
$M ::= H = e$	**concrete method**
f, g	**class field**
x, y, z	**variable**
$e ::=$	**expression**
x	variable
$e.f$	field access
$e.m(\overline{e})$	method call
$\mathsf{new}\, C(\overline{e})$	object
$(C)e$	cast

Figure 6. Syntax of FJD

$\Delta \vdash S <:_{FGJ} T$ then $|S|_\Delta <:_{FJ} |T|_\Delta$ (Lemma A.3.5$_{FGJ}$) which reduces the amount of casts that need to be inserted when erasing expressions to a minimum (Theorem 4.5.3$_{FGJ}$).

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 $|T_1|_\Delta$ and $|T_2|_\Delta$ [7]. Nevertheless, since we're trying to preserve as much structure as possible, it seems logical to define:

$$|T_1 \ \&\ T_2|_\Delta = \texttt{erasedGlb}(|T_1|_\Delta, |T_2|_\Delta)$$

where `erasedGlb` always returns one of its arguments. In fact this is what both Java and Scala do, but they differ on the implementation of `erasedGlb`:

- Java simply defines $\texttt{erasedGlb}(T_1, T_2) = T_1$ [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.

[7] Technically, subtyping would be preserved if we erased all types to Object, but this would defeat the point since it would require many more casts in expression erasure and impede interoperability between Scala and Java.

$$|x|_{\Delta,\Gamma} = x \text{ [E-VAR]}$$

$$\frac{\Delta;\Gamma \vdash e_0 : T_0 \qquad |T_0|_\Delta = C}{|e_0.f|_{\Delta,\Gamma} = |e_0|^C_{\Delta,\Gamma}.f} \text{ [E-FIELD]}$$

$$\frac{\Delta;\Gamma \vdash e_0 : T_0 \qquad \mathtt{erasedReceiver}_\Delta(m, T_0) = C \qquad \mathtt{mtype}_{\mathtt{FJD}}(m_C, C) = \overline{U} \to U_0 \qquad e_i' = |e_i|^{U_i}_{\Delta,\Gamma}}{|e_0.m[\overline{V}](\overline{e})|_{\Delta,\Gamma} = |e_0|^C_{\Delta,\Gamma}.m_C(\overline{e_i'})} \text{ [E-INVK]}$$

$$\frac{|N|_\Delta = C \qquad \mathtt{fields}_{\mathtt{FJD}}(C) = \overline{f : T} \qquad e_i' = |e_i|^{T_i}_{\Delta,\Gamma}}{|\mathsf{new}\, N(\overline{e})|_{\Delta,\Gamma} = \mathsf{new}\, C(\overline{e'})} \text{ [E-NEW]}$$

Figure 7. Expression Erasure

$$\frac{\Gamma = \overline{x : T}, \mathsf{this} : C[\overline{X}] \qquad \Delta = \overline{X <: N}, \overline{Y <: P}}{|\mathsf{def}\ m[\overline{Y <: P}](\overline{x : T}) : T_0 = e_0|_{\overline{X<:N},C} = |T_0|_\Delta\, m_C(\overline{|T|_\Delta\, x})\ \{\mathsf{return}\ \ |e_0|^{|T_0|}_{\Delta,\Gamma};\}} \text{ [E-METHOD]}$$

$$\frac{\begin{array}{c}\Delta = \overline{X <: N} \\ K = C(\overline{|U|_\Delta\, g}, \overline{|T|_\Delta\, f})\{\mathsf{super}(\overline{g}); \mathsf{this}.\overline{f} = \overline{f};\} \qquad \overline{M'} = \overline{|M|_{\Delta,C}} \cup \{\, \mathtt{bridges}(\mathbf{m}, C) \mid \forall \mathbf{m} \in \mathtt{mnames}(C) \,\}\end{array}}{|\mathsf{class}\, C[\overline{X <: N}](\overline{g : U}, \overline{f : T}) \lhd P(\overline{g})\,,\ \overline{Q}\ \{\overline{M}\}| = \mathsf{class}\, C \lhd |P|_\Delta\,,\ \overline{|Q|_\Delta}\ \{\overline{|T|_\Delta\, f}; K; \overline{M'}\}} \text{ [E-CLASS]}$$

$$\frac{\Delta = \overline{X <: N} \qquad \overline{M'} = \overline{|M|_{\Delta,C}}}{|\mathsf{trait}\, C[\overline{X <: N}] \lhd \overline{Q}\ \{\overline{M}\}| = \mathsf{interface}\, C \lhd \overline{|Q|_\Delta}\ \{\overline{M'}\}} \text{ [E-TRAIT]}$$

Figure 8. Class Table Erasure

- On the other hand, Scala 2 defines `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; Shipilëv 2020]). Unfortunately, completely specifying the behavior of Scala 2 here is extremely hard because it inadvertently depends on implementation details of the compiler [8]
- Scala 3 preserves the two properties from Scala 2 mentioned above and additionally ensures that erasure preserves commutativity of intersection ($|T_1 \ \&\ T_2|_\Delta = |T_2 \ \&\ T_1|_\Delta$) by applying a tie-break based on the lexographical order of the names of the compared types. The following pseudo-code accurately specifies its behavior [9]:

```
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:

```
trait X; trait Y; trait Z extends X
```

Then `|(X & Y) & Z| = Z` but `|X & (Y & Z)| = X`. The problem is that while the lexicographic ordering by itself is total, it is applied inconsistently because *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 [10] 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:

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

[8] See https://github.com/lampepfl/dotty/blob/master/compiler/src/dotty/tools/dotc/core/unpickleScala2/Scala2Erasure.scala for the unsavory details.

[9] The complete implementation also special-cases value types and array types which we do not model in our calculus, see `erasedGlb` in https://github.com/lampepfl/dotty/blob/master/compiler/src/dotty/tools/dotc/core/TypeErasure.scala

[10] 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:

$$\frac{e' = |e|_{\Delta,\Gamma} \qquad \Gamma \vdash_{FJD} e' : S}{|e|^T_{\Delta,\Gamma} = \begin{cases} e' & \text{if } S <:_{FJD} T \\ (T)e' & \text{otherwise} \end{cases}}$$

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 $\texttt{getters}_\Delta(T_0) = \overline{f : T}$ then $\texttt{fields}_{\texttt{FJD}}(|T_0|_\Delta) = \overline{f : |T|_\Delta}$ and E-FIELD is straight-forward, but finding the right cast for the receiver of a method call is more involved.

5.2.1 $\texttt{erasedReceiver}_\Delta(m, N)$: 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 $|\texttt{x}|_{\Delta,\Gamma}$ 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:

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

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

$$\texttt{erasedReceiver}_\Delta(m, T_1 \,\&\, T_2) = \begin{cases} \texttt{erasedReceiver}_\Delta(m, T_1) & \text{if } \texttt{mtype}_\Delta(m, T_1) \text{ is defined} \\ \texttt{erasedReceiver}_\Delta(m, T_2) & \text{otherwise} \end{cases}$$

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 new Y(); }
  Object 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 foo_L() }
interface R { X foo_R() }
class A ◁ Object, L, R {
  Y foo_A { return new Y(); }
```

```
  Object foo_L { return this.foo_A(); }
  X foo_R { return this.foo_A(); }
}
```

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:

$$\frac{\begin{array}{c}\mathtt{mtype}_{\mathsf{FJD}}(m_E, E) = \overline{T} \to T_0 \\ \mathtt{mtype}_{\mathsf{FJD}}(m_D, D) = \overline{U} \to U_0 \\ x_0 = \mathsf{this}.m_D(\overline{e}) \qquad e_i = \begin{cases} x_i & \text{if } T_i = U_i \\ (U_i)x_i & \text{otherwise} \end{cases}\end{array}}{\mathtt{bridge}(m_E, m_D) = T_0\, m_E(\overline{T\, x})\, \{\mathsf{return}\ \ e_0;\}}$$

$$\frac{\begin{array}{c}\mathtt{mimpl}(m, N) = D[\overline{T}] \\ \overline{E[...]} = \{\, \mathbf{n} \in \mathcal{L}(N) \setminus D[\overline{T}] \mid \mathsf{def}\ m\ ... \in \mathtt{mdecls}(\mathbf{n})\}\end{array}}{\mathtt{bridges}(m, N) = \overline{\mathtt{bridge}(m_E, m_D)}}$$

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 [Wasserrab 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&λ [Bettini et al. 2018]. The semantics of intersection types were then generalized beyond what Java supports in FJP&λ [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 FJP&λ into FJ&λ. Pathless Scala can be seen as a generalization of FJP&λ, but we found it easier to extend FGJ with traits and intersections rather than to extend FJP&λ with polymorphism and generalize its interfaces to traits. However, FJ&λ 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&λ 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&λ 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 https://github.com/scala/bug/issues/4214.

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 straight-forward 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

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, Cham, Switzerland, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*. Association for Computing Machinery, New York, NY, USA.

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A monotonic superclass linearization for Dylan. In *ACM SIGPLAN Notices*. Vol. 31. Association for Computing Machinery, New York, NY, USA, 69–82. https://doi.org/10.1145/236337.236343

Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 2018. Java & Lambda: a Featherweight Story. *Logical Methods in Computer Science* Volume 14, Issue 3 (2018). https://doi.org/10.23638/LMCS-14(3:17)2018 arXiv:1801.05052

Gilad Bracha, Norman Cohen, Christian Kemper, Martin Odersky, David Stoutamire, Kresten Thorup, and Philip Wadler. 2003. *Adding Generics to the Java Programming Language: Public Draft Specification Version 2.0.* http://www.javainthebox.net/laboratory/J2SE1.5/LangSpec/Generics/materials/adding_generics-2_2-ea/spec10.pdf

Cliff Click and John Rose. 2002. Fast subtype checking in the HotSpot JVM. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/583810.583821

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A core calculus for Scala type checking. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1–23. https://doi.org/10.1007/11821069_1

Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. 2019. Intersection Types in Java: Back to the Future. In *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*. Springer, Cham, Switzerland, 68–86. https://doi.org/10.1007/978-3-030-22348-9_6

Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. 2020. Deconfined Intersection Types in Java. In *Recent Developments in the Design and Implementation of Programming Languages (OpenAccess Series in Informatics (OASIcs), Vol. 86)*, Frank S. de Boer and Jacopo Mauro (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:25. https://doi.org/10.4230/OASIcs.Gabbrielli.3

Sébastien Jean R. Doeraene. 2018. *Cross-Platform Language Design*. Ph.D. Dissertation. EPFL. https://doi.org/10.5075/epfl-thesis-8733

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *Proc. ACM Program. Lang.* 4, ICFP (Aug 2020), 1–29. https://doi.org/10.1145/3408996

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2015. *The Java Language Specification, Java SE 8 Edition*. Oracle. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Atsushi Igarashi and Benjamin C. Pierce. 2002. On Inner Classes. *Inform. And Comput.* 177, 1 (Aug 2002), 56–89. https://doi.org/10.1006/inco.2002.3092

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

Ifaz Kabir and Ondřej Lhoták. 2018. κDOT: scaling DOT with mutation and constructors. In *Scala 2018: Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. Association for Computing Machinery, New York, NY, USA, 40–50. https://doi.org/10.1145/3241653.3241659

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. *The Java Virtual Machine Specification, Java SE 8 Edition*. Oracle. https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

Martin Odersky et al. 2021. *The Scala Language Specification, Scala 2.13 Edition*. EPFL. https://www.scala-lang.org/files/archive/spec/2.13/

Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. *SIGPLAN Not.* 40, 10 (Oct 2005), 41–57. https://doi.org/10.1145/1103845.1094815

Marianna Rapoport. 2019. *A Path to DOT: Formalizing Scala with Dependent Object Types*. Ph.D. Dissertation. University of Waterloo. http://hdl.handle.net/10012/15322

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). *SIGPLAN Not.* 51, 10 (Oct 2016), 624–641. https://doi.org/10.1145/3022671.2984008

Aleksey Shipilëv. 2020. *The Black Magic of (Java) Method Dispatch*. https://shipilev.net/blog/2015/black-magic-method-dispatch

Sandro Stucki and Paolo G. Giarrusso. 2021. A theory of higher-order subtyping with type intervals. *Proc. ACM Program. Lang.* 5, ICFP (Aug 2021), 1–30. https://doi.org/10.1145/3473574

Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. 2006. An operational semantics and type safety proof for multiple inheritance in C++. *SIGPLAN Not.* 41, 10 (Oct 2006), 345–362. https://doi.org/10.1145/1167515.1167503