# Foundations of Path-Dependent Types

Nada Amin[*] Tiark Rompf [†*] Martin Odersky[*]

[*]EPFL: `{first.last}@epfl.ch`
[†]Oracle Labs: `{first.last}@oracle.com`

## Abstract

A scalable programming language is one in which the same concepts can describe small as well as large parts. Towards this goal, Scala unifies concepts from object and module systems. An essential ingredient of this unification is the concept of objects with type members, which can be referenced through path-dependent types. Unfortunately, path-dependent types are not well-understood, and have been a roadblock in grounding the Scala type system on firm theory.

We study several calculi for path-dependent types. We present $\mu$DOT which captures the essence – DOT stands for Dependent Object Types. We explore the design space bottom-up, teasing apart inherent from accidental complexities, while fully mechanizing our models at each step. Even in this simple setting, many interesting patterns arise from the interaction of structural and nominal features.

Whereas our simple calculus enjoys many desirable and intuitive properties, we demonstrate that the theory gets much more complicated once we add another Scala feature, type refinement, or extend the subtyping relation to a lattice. We discuss possible remedies and trade-offs in modeling type systems for Scala-like languages.

***Categories and Subject Descriptors*** D.3.3 [*Language Constructs and Features*]: Abstract data types, Classes and objects, polymorphism; D.3.1 [*Formal Definitions and Theory*]: Syntax, Semantics; F.3.3 [*Studies of Program Constructs*]: Object-oriented constructs, type structure; F.3.2 [*Semantics or Programming Languages*]: Operational semantics

***General Terms*** Languages, Theory

***Keywords*** calculus, objects, dependent types

## 1. Introduction

A scalable programming language is one in which the same concepts can describe small as well as large parts. Towards this goal, Scala unifies concepts from object and module systems. An essential ingredient of this unification is to support objects that contain *type members* in addition to fields and methods.

To make any use of type members, programmers need a way to refer to them. This means that types must be able to refer to objects, i.e. contain terms that serve as static approximation of a set of dynamic objects. In other words, some level of dependent types is required; the usual notion is that of *path-dependent* types.

Despite many attempts, Scala's type system has never been fully formalized. Newer languages like Kotlin and Ceylon have shied away from adding a corresponding feature. Previous type systems that studied similar constructs have mainly focused on aspects such as implementation inheritance, family polymorphism and virtual classes.

Our motivation is twofold. First, we believe objects with type members are not fully understood. It is not clear what causes the complexity, which pieces of complexity are essential to the concept or accidental to a language implementation or calculus that tries to achieve something else. Second, we believe objects with type members are really useful. They can encode a variety of other, usually separate type system features. Most importantly, they unify concepts from object and module systems, by adding a notion of nominality to otherwise structural systems.

The main contribution of this paper is to liberate path-dependent types from questions of inheritance, refinements, etc. and show that a core calculus of objects with type members has a clean and intuitive theory, with many desirable properties including type safety. We demonstrate that much of the perceived complexity of type members in a full language is the result of interaction with other features. In particular we show that adding another Scala feature, type refinement, or giving the subtyping relation lattice structure breaks the intuitive behavior in at least one aspect and makes a proof of type safety much harder to establish. We discuss the resulting trade-offs and possible directions for the type system designer.

### 1.1 Motivating Example in Scala

The key use case for path-dependent types is to model nominality through abstract type members, as we show through an example in Scala inspired by [15].

***Nominality through Abstract Type Members*** An animal eats food of a certain type that depends on the animal. We model an animal with a `trait Animal` that has an abstract type member `type Food`. The variable `a` denotes the self (i.e. `this`) object, in the scope defining the trait `Animal`. So we can refer to the abstract type member `type Food` as a type through a type selection: `a.Food` – `a.Food` is a *path-dependent type*, in general, a chain, starting with an immutable variable, of immutable field selections, ending with a type selection. Notice that the type selection `a.Food` can appear both covariantly, as in the method `def gets`, where it is the return type, and contravariantly, as in the method `def eats`, where it is a parameter type.

```
trait Animal { a =>
  type Food
  def eats(food: a.Food): Unit = {}
  def gets: a.Food
}
```

A cow is an animal that eats grass. A lion is an animal that eats meat. We can model these two animals by refining the `trait Animal`.

```
trait Grass
trait Meat
trait Cow extends Animal with Meat {
  type Food = Grass
  def gets = new Grass {}
}
trait Lion extends Animal {
  type Food = Meat
  def gets = new Meat {}
}
```

Now, let's have `leo` the Lion eat `milka` the Cow. This is possible, because `Cow` is a subtype of `Meat`:

```
val leo = new Lion {}
val milka = new Cow {}
leo.eats(milka)
```

The lion `leo` can also eat whatever it gets:

```
leo.eats(leo.gets)
```

On the other hand, if we have `lambda` the (unknown) Animal, then we cannot feed it `milka` the Cow. After all, `lambda` the Animal might well be a Cow – or even, `milka` itself:

```
val lambda: Animal = milka
lambda.eats(milka) // type mismatch
                   // found   : Cow
                   // required: lambda.Food
```

Still, `lambda` can eat whatever it gets:

```
lambda.eats(lambda.gets)
```

The path-dependent type arising from the type member `Food` drives this example. In the trait `Animal`, the type `Food` is a fully abstract type member. Conceptually, its lower bound is bottom (the uninhabited type) and its upper bound is top (the type of all values). In the traits `Cow` and `Lion`, we refine the type member `Food` – this is allowed, as long as the bounds in the subtype are not wider than the bounds in the supertype. On the one hand, the Scala type system admits that `leo` the Lion eats `milka` the Cow, because `leo.Food`, appearing in a contravariant position, is lower-bounded by the type `Meat` and the type `Cow` is a subtype of the type `Meat`. On the other hand, the Scala type system refuses that `lambda` the unknown Animal eats `milka` the Cow – because `lambda.Food`, being fully abstract, is lower-bounded by bottom – and of course, the type `Cow` is not a subtype of bottom. Still, `lambda` the (unknown) Animal can eat whatever it gets, because that has type `lambda.Food` – through subtyping reflexivity, we get nominality as an emergent property of type members and path-dependent types.

***Composition and Refinement through Subtyping Lattice*** Note that even though we have used traits and inheritance in this example, these mechanisms are not essential here for the subtyping relations. In Section 2, we show a variant that uses only path-dependent types and achieves type abstraction through ascription (up-cast) instead of inheritance.

To keep the core calculus simple, we deliberately choose not to model inheritance and mixin composition. Still, we want the core calculus to be rich enough so that we could express inheritance and mixing composition by translating them to the core. For this requirement, we find it important to also explore a calculus with a complete subtyping lattice, such that meets and joins are defined for all types. With such a core calculus, programming patterns involving greatest lower bounds and least upper bounds would be naturally handled. For example, two animals might want to share a bite:

```
def share(a1: Animal)(a2: Animal)(bite: a1.Food with a2.Food) {
  a1.eats(bite)
  a2.eats(bite)
}
```

Two lions, `leo` and `simba`, can indeed share a bite:

```
val simba = new Lion {}
share(leo)(simba)(leo.gets) // ok
```

However, `lambda` the (unknown) Animal cannot share a bite with `leo` the Lion:

```
share(leo)(lambda)(leo.gets) // error: type mismatch
                             // found   : Meat
                             // required: leo.Food with lambda.Food
```

Even without a full subtyping lattice, we will see that just adding unrestricted refinements breaks some intuitive properties that we expect of subtyping, such as transitivity and environment narrowing. The Scala compiler has ad-hoc restrictions to prevent these issues on an implementation level, but nobody has a proof that these tweaks are sufficient.

## 1.2 Contributions

In this paper, we explore the foundations of path-dependent types. We make the following contributions:

- We distill the essence of path-dependent types with a simple system of types – $\mu\mathrm{DOT}_T$ – comprised of just records with type members and type selections on variables (Section 3). We show that this simple system captures the essential programming patterns (Section 3.1) and satisfies the intuitive and mutually dependent properties of environment narrowing and subtyping transitivity (Section 3.2). Finally, we show that adding type refinement or extending subtyping to a lattice breaks at least one of these intuitive properties in their full generality (Section 3.3).
- We augment our simple system of types into a full-fledged but still simple calculus, $\mu$DOT, with a term syntax, term typing, big-step operational semantics, and value typing. In big-step evaluation, path-dependent types close over their defining environments, which we model through a "two-headed" subtyping judgement with an environment for each type (Section 4).
- We provide a mechanized proof of the type-safety of $\mu$DOT, highlighting the reusable insights and techniques. The proof is set up with extensibility in mind: being based on big-step semantics, it does not depend on a general environment narrowing property for arbitrary terms, but only on a weaker statement for paths starting with a self name, which is required to uphold subtyping transitivity (Section 5).
- We demonstrate a fundamental trade-off between nominality, refinements, and subtyping transitivity. We discuss how we can still build sound type systems that combine these features (but may incur other restrictions) by carefully orchestrating multiple subtyping notions. For example, we can use a nominal system for type assignment and "erase" to a transitive system for the proofs. Another option is to combine a "strong" subtyping notion (with environment narrowing) with the regular one, if we can limit regular transitivity to only rely on "strong" but not regular environment narrowing (Section 7).

We survey related work in Section 8 and offer a concluding discussion in Section 9.

## 2. The Essence of Scala

Before formally defining $\mu$DOT, we quickly recap how objects with type members can encode other language features. This will set the stage for the formal development in Sections 3 and 4, and clarify the subset of Scala modeled by the calculi.

```
type Meat = {
  type IsMeat = Any
}
type Grass = {
  type IsGrass = Any
}
type Animal = { a =>
  type Food
  def eats(food: a.Food): Unit
  def gets: a.Food
}
type Cow = {
  type IsMeat = Any
  type Food <: Grass
  def eats(food: Grass): Unit
  def gets: Grass
}
type Lion = {
  type Food = Meat
  def eats(food: Meat): Unit
  def gets: Meat
}
```

```
def newMeat  = new {
  type IsMeat = Any
}
def newGrass = new {
  type IsGrass = Any
}
def newCow   = new {
  type IsMeat = Any
  type Food = Grass
  def eats(food: Grass) = ()
  def gets = newGrass
}
def newLion = new {
  type Food = Meat
  def eats(food: Meat) = ()
  def gets = newMeat
}

val milka = newCow
val leo = newLion
leo.eats(milka)
```

**Figure 1.** Cow and Lion example expressed in Scala subset that maps to $\mu$DOT

We first revisit the introductory example from Section 1.1. In this paper, we are not concerned with traits, classes, or implementation inheritance at all, so we present a version that does not use these features in Figure 1. This restricted subset of Scala maps directly to $\mu$DOT. Each trait maps to the type it represents and each concrete trait in addition to a constructor that builds objects of that type. We express purely nominal subtyping relations through the presence of certain type members (e.g. `IsMeat`). Thus, we use type members to represent *all* nominal types, not only some.

In Scala, function literals of type `A=>B` are represented as objects of a trait `Function1[A,B]` that implement a single method `def apply(x:A):B`. This desugaring carries over directly to $\mu$DOT.

### 2.1 Type Refinement

Subtyping allows us to treat a type as a less precise one. Scala provides a dual mechanism that enables us to create a more precise type by *refining* an existing one.

We could use refinement in the definition of `Cow`:

```
type Cow = Animal { a =>
  type Food <: Grass
  def gets: Grass
}
```

Here, we are expressing `Cow` as a more precise version of `Animal`.

This example can easily be expressed without refinement, because the definition of `Animal` is concrete. If the type `Animal` were more abstract, for example only upper-bounded by instead of aliasing its defining record, then the definition of `Cow` expressed without refinement would be lacking, as it would not capture the intention that `Cow` is a subtype of `Animal`.

When refining an abstract type, we speak of "open refinement". As an aside, if the type `Cow` was an open refinement of an abstract type `Animal`, then it would also be difficult to create a new `Cow` from scratch, since one would not know how to conform to its `Animal` nature.

We can think of refinement as a special case of intersection, between a type and a record type which defines additional members or refines already defined members. One difference is that a refinement may refer to members of the type being refined, as if the "self" variable closes over both types of the intersection, while one usually expects each type of an intersection to be well-formed independently.

For example, we can express the type `Hoarder` as a refinement of the type `Animal`, with an additional member that refers to the abstract type member `Food` of `Animal`:

```
type Hoarder = Animal { a =>
  def stash(food: a.Food)
}
```

To express the type `Hoarder` by desugaring the refinement into an intersection type, we need a "self" variable (here `a`) to close over the intersection type, in order to refer to the abstract type member `Food` of `Animal`:

```
type Hoarder = { a =>
  Animal & {
    def stash(food: a.Food)
  }
}
```

The core $\mu$DOT calculus does not support intersections or refinements. We describe the problems with naive extensions in Section 3.3 and possible remedies in Section 7.

### 2.2 Mixins != Intersection Types

In Scala, mixins aren't quite intersection types. Scala, and the previous calculi attempting to model it, conflate the concepts of compound types (which inherit the members of several parent types) and mixin composition (which build classes from other classes and traits). At first glance, this offers an economy of concepts. However, it is problematic because mixin composition and intersection types have quite different properties. In the case of several inherited members with the same name, mixin composition has to pick one which overrides the others. It uses for that the concept of linearization of a trait hierarchy. Typically, given two independent traits $T_1$ and $T_2$ with a common method $m$, the mixin composition $T_1$ `with` $T_2$ would pick the $m$ in $T_2$, whereas the member in $T_1$ would be available via a super-call. All this makes sense from an implementation standpoint. From a typing standpoint it is more awkward, because it breaks commutativity and with it several monotonicity properties.

### 2.3 Subtyping Lattice

Scala currently lacks a full subtyping lattice, because greatest lower bounds and least upper bounds do not always exist. Here is an example:

```
trait A { type T <: A }
trait B { type T <: B }
trait C extends A with B { type T <: C }
trait D extends A with B { type T <: D }
```

The least upper bound of types `C` and `D` does not have a finite representation because the type member `T` can be refined arbitrarily:

```
  val cond: Boolean
  val o: A with B{type T <: A with B{type T <: A with B\*...*\} =
    if (cond) (new C{}) else (new D{})
```

On the other hand, the type inference engine must arbitrarily settle eagerly on a finite type to represent the least upper bound. This approximation can be a source of inefficiency, brittleness and impredictability:

```
val i = if (cond) (new C{}) else (new D{})  // type inferred
val i1 = (i : A with B)                     // ok
val i2 = (i : A with B{type T <: A with B}) // ok
val i3 = (i : A with B{type T <: A with B{type T <: A with B}})
 // error: type mismatch;
 // found   : A with B{type T <: A with B}
 // required: A with B{type T <: A with B{type T <: A with B}}
```

If the core calculus had classical intersection and union types, then the type inference engine could simply lazily construct the least upper bound of types `C` and `D` as its union $\texttt{C} \vee \texttt{D}$.

## 3. Objects with Type Members: $\mu$DOT$_T$

To illustrate the semantics of path-dependent types, we start with a simple system of types, comprising only record types with type members, and path-dependent types selecting a member type. Figures 2 and 3 specify its syntax and its semantics.

| | |
|---|---|
| $x, y, z$ | Variable |
| $S, U, T ::=$ | Type |
| $\quad \{z \Rightarrow \overline{D}\}$ | Record Type |
| $\quad x.L$ | Type Selection |
| $D ::=$ | Member Declaration |
| $\quad$ **type** $L : S..U$ | Type Member |
| $\quad$ **type** $L <: U$ | Type Member (Upper-Bounded) |

**Figure 2.** $\mu$DOT$_T$: Syntax of Types

The empty record type $\{z \Rightarrow\}$ is a natural top type for this system. We will see later that a bottom type is semantically complex, and so we exclude it from this simple system. However, we still want to reason about fully abstract types (conceptually, with lower bound bottom and upper bound top) – hence, we include the special case of a type member that is only upper-bounded: the syntax "**type** $L$ <: $U$" defines a type member which is only upper-bounded by a type $U$, while the syntax "**type** $L : S..U$", a type member which is both lower-bounded by a type $S$ and upper-bounded by a type $U$.

The semantics of $\mu\text{DOT}_T$ is specified through four judgements: subtyping, well-formedness, expansion and membership.

For subtyping, we admit reflexivity on all well-formed types. Additionally, type selections can be structurally "opened", on the left through the upper bound, and on the right through the lower bound (only if it exists). Record types are directly compared only with other record types, in order to keep the self-binding variable in sync.

For well-formedness, in this simple system, we check "proper" bounds both when they are declared and when they are used. Though this double-checking restricts the use of recursion in the types, it also ensures that a type always expands, which is necessary for environment narrowing.

Subtyping is regular, in that it implies well-formedness. In REC-<:-REC, we explicitly ensure that the record type on the right-hand side is well-formed, because the induction hypothesis of regularity only ensures that the right-hand declarations are well-formed in a context where the self type is bound to the left-hand record type. This is not enough: we want to rule out right-hand record types referring to members that only exist on the (presumably more precise) left-hand side.

A type expands to all the declarations in the record type obtained by following the upper bounds of type selections (if any) until reaching a record type. In the expansion judgement, $\Gamma \vdash T \prec_z \overline{D}$, the subscript $z$ denotes the self variable in the declarations $\overline{D}$. Membership looks up a particular declaration in the expansion.

Though we later discuss why augmenting this simple system of types with a subtyping lattice is problematic, we would like to point out here that it also can offer an economy of concepts in at least two aspects. First, introducing a bottom type nullifies the need for the special case of type members that are only bounded from above. Second, intersection types enable declarations to be specified and checked separately, so only membership, not expansion, is required. Surprisingly, membership need not be related to subtyping – the only requirement on membership is uniqueness.

## 3.1 Nominal Abstraction through Ascription

A record type `S` is a subtype of a record type `U` if it defines at least all the type members of `U`, and for each one of those, the bounds it specifies are no wider. This requirement matches our zoo example of section 1.1, where the supertype `Animal` specified the `Food` type member with fully abstract bounds, and the subtype `Cow` narrowed the `Food` type member to alias the type `Grass`. Let's define such record types in $\mu\text{DOT}_T$ – note that we use abbreviations, not part of the object language, for convenience, to avoid repeating the same record types.

```
abbrev Grass  =  { g => type IsGrass: {id => } .. {id => } }
abbrev Animal =  { a => type Food <: { f => } }
abbrev Cow    =  { a => type Food: Grass .. Grass }
```

Our semantics enables us to establish the subtyping proposition `Cow <: Animal` (via REC-<:-REC). Assuming the binding `a: Cow`, we can establish the subtyping propositions `Grass <: a.Food <: Grass` (via <:-TSEL and TSEL-<:), but not assuming the binding `a: Animal`. So if we have a `Cow a`, we can upcast it to an `Animal`, and hide the fact that it can eat `Grass`, but still be able to feed it `a.Food` nominally (via REFL) – a form of data abstraction.

## 3.2 Environment Narrowing and Subtyping Transitivity

The $\mu\text{DOT}_T$ system satisfies a number of intuitive and general properties. We discuss two important ones next.

***Environment Narrowing*** If we have an Animal, and later find out it is a Cow, this should not invalidate any findings we made about the Animal – that is anything we prove assuming `a: Animal` should still hold assuming `a: Cow`. More generally, anything proved assuming the binding `x: U` should still hold assuming the binding `x: S` if the type `S` is a subtype of the type `U`. This intuitive monotonic property is known as environment narrowing.

Environment narrowing plays a key role in type preservation proofs via small-step operational semantics. With path-dependent types, environment narrowing is also needed to establish subtyping transitivity. Furthermore, the proof of environment narrowing uses subtyping transitivity, so the two proofs need to be set up through a mutual induction.

$$\frac{\Gamma^a, (x:U), \Gamma^b \vdash T <: T' \qquad \Gamma^a \vdash S <: U}{\Gamma^a, (x:S), \Gamma^b \vdash T <: T'} \quad (\text{<:-NARROW})$$

***Subtyping Transitivity*** Subtyping transitivity is another intuitive property.

$$\frac{\Gamma \vdash S <: T\ ,\ T <: U}{\Gamma \vdash S <: U} \qquad (\text{<:-TRANS})$$

Since types may be path-dependent types, we rely on environment narrowing to establish subtyping transitivity. In a chain, $\Gamma \vdash S <: T$ and $\Gamma \vdash T <: U$ both derived by REC-<:-REC, we prove $\Gamma \vdash S <: U$ through narrowing the right-hand assumption of the premises to match the left-hand assumption.

Even in this simple type system, the proof of environment narrowing and subtyping transitivity is not trivial. We explain the problems and our proof methodology in Section 6.

## 3.3 No Naive Subtyping Lattice or Refinements

The $\mu\text{DOT}_T$ system is set up carefully, so that well-formed type members always have "good" bounds, i.e. the lower bound is always a subtype of the upper bound. This restriction is essential for subtyping transitivity, in order to conclude that $S <: U$ whenever $S <: x.L$ (via TSEL-<:) and $x.L <: U$ (via <:-TSEL).

In general, subtyping allows us to treat a type as a less precise one. We would like to add a dual mechanism that would enable us to create a more precise type by *refining* an existing one. Where subtyping allows us to widen the bounds of a type member, refinement would enable us to narrow its bounds.

The situation becomes quite complicated if we allow open refinement, i.e. combining refinements from different places. If type bounds are narrowed to two disjoint ranges, the composition would no longer be well-formed. Unfortunately this situation is not easy to determine. The particular problematic case occurs when refining a path-dependent type $x.L$. Assuming a certain type $T$ for $x$, a refinement $U$ of $T$ might look valid, but environment narrowing tells us that we can replace the binding of $x$ with type $S <: T$, which might be a refinement of $T$ that is incompatible with $U$.

To make the discussion more concrete, let us consider extending the subtyping relation of $\mu\text{DOT}_T$ to a lattice by adding bottom and top types, and classical intersection and union constructors ($\wedge$ and $\vee$). Intersection types are a general encoding of open refinements.

As an example, consider the type `abbrev T = a.Food` $\wedge$ `Grass`. Assuming `a` is an `Animal`, this resulting type is at most inhabited by all things `Grass`. Assuming `b` of type `T`, `b.IsGrass` would have perfectly good bounds. But suppose, we now know that the object `a` furthermore is a `Lion`, and `a.Food` is then `Meat`, defined as follows:

```
abbrev No   = { n => type IsNot: {id => } .. {id => } }
abbrev Meat = { g => type IsGrass: No .. No }
abbrev Lion = { a => type Food: Meat .. Meat }
```

We now know that the type `T` is unrealizable, and assuming `b: T`, the type selection `b.IsGrass` would have "bad" bounds: the lower bound would be `{ id => }` while the upper bound would have a type member `IsNot` – so clearly, it would not be a supertype.

What went wrong? Once we introduce a subtyping lattice, we can discover that we are in an impossible situation only after narrowing, when more precise information about a variable is available. The canonical impossible situation is a variable of type bottom, and it is completely unclear what bounds a type selection on a bottom variable should assume.

What bounds should a type selection on a bottom variable be? Since we can always narrow the type of a variable to bottom, the bounds should be whatever they were before the narrowing – so perhaps, arbitrary as long as "good"? The problem now is that we have no leverage for subtyping transitivity when the middle man is `x.L` assuming `x` has type bottom. Indeed, the arbitrary bounds used to derive `S <: x.L` would bear no relation to the arbitrary bounds used to derive `x.L <: U`, so how would we confirm that `S <: U`?

Perhaps, we could admit that the lattice collapses in an unrealizable context, but then we would need to be able to detect unrealizable contexts and explicitly permit even more collapsing. Otherwise, it is possible to get into a situation where narrowing an unrealizable context breaks an established proposition.

We conclude the discussion at this point by noting that there is no easy way to add refinements to our type system, and that any extension in this direction will likely sacrifice general environment narrowing. Note however that this does not rule out the possibility of weaker statements, for example a form of environment narrowing that is just enough to carry the transitive case of REC-<:-REC. We will come back to these considerations in section 7.

## 4. The $\mu$DOT Calculus

We now complete the system of types $\mu\text{DOT}_T$ to a full-fledged calculus $\mu$DOT by adding a term syntax, term typing, a big-step operational semantics and value typing. Figures 4, 5 and 6 specify the additions to syntax and (static and dynamic) semantics.

| | |
|---|---|
| $D ::= \ldots$ | Member Declaration |
| **def** $m : S \to U$ | Method Member |
| $s, t, u ::=$ | Term |
| $x$ | Variable |
| **new** $(z \Rightarrow \overline{I})$ | Object Creation |
| $t.m(t')$ | Method Invocation |
| $I ::=$ | Member Initialization |
| **type** $L : S..U$ | Type Member |
| **type** $L <: U$ | Type Member (Upper-Bounded) |
| **def** $m(x : S) : U = t$ | Method Member |
| $v ::=$ | Value |
| $<z \Rightarrow \overline{E}$ **in** $H>$ | Closure |
| $E ::=$ | Run-Time Member Definitions |
| **def** $m(x) = t$ | Method Member |
| $H ::= \overline{x : v}$ | Run-Time Environment |

**Figure 4.** $\mu$DOT: Syntax

Regarding syntax, we augment record types with declarations for method members. Our term language comprises variables, object creation expressions, and method invocations. Our values are closures, which package method definitions and the lexically scoped run-time environment of the object definition site.

Regarding semantics, the typing of terms is syntax-directed (one case per shape of terms), plus an additional non-algorithmic subsumption case which applies to any term. New object creation expressions ensure that the type resulting from the initialization is well-formed and that the method definitions type-check, while the self-binding variable and the parameter are in scope. For method invocations, note that we avoid membership, but rely on subsumption to find a record with exactly the method declaration needed – since membership is only defined on variables, not arbitrary terms.

Now, that we have a full-fledged calculus, let us illustrate some of the possibilities offered by path-dependent types. In the examples, we use syntactic sugar for let-binding a variable: **val** $x : T = t^x; (t : U)$ desugars into (**new** (_ $\Rightarrow$ **def** app$(x : T) : U = t)$).app$(t^x)$.

By widening the bounds of a type member, we can abstract from implementation details. At one extreme, we can widen the bounds of a type member so that it is only upper-bounded, rendering its type selection nominal and controlling the creation process of objects of this nominal type. Here is an example:

```
val o: {o =>
  type A <: {a => }
  def m(x: {a => }): o.A
} = new (o =>
  type A:  {a => } .. {a => }
  def m(x: {a => }): o.A = x
);
o.m(new(a =>)): o.A
```

From the inside of the object creation of o, the type member A is an alias for the empty record type, the top type. From the outside, the bounds are widened, so that it can no longer be opened from below. Thus, from the outside, the only way to get an instance of the type o.A is to call the method o.m. What's surprising is that this method o.m is only the identity function, and yet, from the outside, its argument type is not a subtype of its return type. In a small-step setting, invoking such an identity method would violate type preservation!

This example suggests that preservation of types and preservation of type abstractions are incompatible, at least without some restrictions on where path-dependent types can appear. This problem has been studied in the context of ML modules [6, 20]. Here is an example similar to the one discussed above in Standard ML [13, 23], using an opaque ascription:

```
signature O = sig
  type A
  val m : int -> A
end

structure MyO :> O = struct
  type A = int
  fun m(x) = x
end

MyO.m(10): MyO.A;  (* ok *)
10: int;           (* ok *)
10: MyO.A;         (* error *)
MyO.m(10): int;    (* error *)
```

In our formalism, we use a big-step operational semantics, which simplifies relating path-dependent types. Still, we will see in section 5 that we still need to relate path-dependent types across different lexical environments.

## 5. Type Safety of $\mu$DOT

We prove the type-safety of the $\mu$DOT calculus. Our approach is based on Siek's three easy lemmas [28], which were inspired by the *vc* paper [11]. The formal development is available online at `oopsla14.namin.net`.

We first highlight our proof of type preservation, showing that if a term type-checks and evaluates to a value, then that value type-checks in agreement. To prove full type-safety, type preservation is not enough. We extend our evaluation relation into a total relation by explicitly specifying the time-out and stuck cases. Figure 7 presents such an extended total evaluation relation. Given this extended total evaluation relation, we show that if a term type-checks, then, given any evaluation (and we can always get one for a given $n$, since the evaluation is total), the result of the evaluation is either a timeout or a value which type-checks in agreement – but crucially, the result is not stuck.

$$\frac{\Gamma \vdash t : T \quad H : \Gamma \quad H \vdash t \Downarrow v}{\Gamma \vdash v : T} \quad (\text{PRESERVATION})$$

$$\frac{\Gamma \vdash t : T \quad H : \Gamma \quad H \vdash t \Downarrow_n r}{(r = \textbf{timeout}) | (r = v \;\&\; \Gamma \vdash v : T)} \quad (\text{SAFETY})$$

Being based on big-step semantics, the proof does not depend on a general environment narrowing property for typing of arbitrary terms, which would be unlikely to hold in extensions of the calculus. Instead, it is centered around a collection of inversion lemmas that rely only on subtyping transitivity to factor out subsumption introduced by SUB and VSUB. Thus, the proof model carries over to possible extensions of $\mu$DOT that support only a limited environment narrowing statement to carry the REC-<:-REC case in subtyping transitivity but not environment narrowing in its general form.

Once all the inversion lemmas described in section 5.2 are in place, the proof of type preservation is rather straightforward, but one additional mechanism is needed when invoking a method from a closure, which we highlight next.

### 5.1 "Two-Headed" Subtyping across Environments

When we invoke a closure, we evaluate its body in the run-time environment from the closure's definition site, not its site of invocation. Therefore, since variables are lexically scoped, we may have different run-time – and hence, static – environments at each site. With path-dependent types, not only terms depend on the environment but also types. Thus, when we invoke a closure, we need to relate the parameter type of the definition site with the argument type of the invocation site. Likewise, we need to relate the type of the value returned from the method body with the expected return type at the invocation site. Effectively, a type closes over its defining environment, and subtyping needs to relate two types with respect to their two corresponding environments.

This leads us to a two-headed subtyping relation with an environment on each side:

$$\Gamma \vdash T <: T' \dashv \Gamma'$$

If $\Gamma = \Gamma'$, we continue to write $\Gamma \vdash T <: T'$.

Most subtyping rules extend straightforwardly to two environments – the only tweaking is to identify and correctly relate self-variables across environments: when extending each of the two environments with a self-variable, we record that they are related, and then, we exploit this information when comparing type selections through reflexivity.

For the semantics of typing values, we allow subsumption across environments (rule VSUB in Figure 5), which precisely models the method call case. The inversion lemmas crucially depend on transitivity of the two-headed subtyping relation to resolve the subsumption case.

### 5.2 Inversion Lemmas

Except for the two-environment trick to resolve path-dependent types of different lexical scopes, our proof of type-safety looks otherwise unsurprising for a system that combines big-step semantics and subtyping. In particular, we use inversion lemmas as expected to factor out subsumption. In summary, the proof of type safety is by induction on the evaluation, and each case in the evaluation is solved through an inversion lemma. Each inversion lemma captures the essential constraints of the typing derivation for a particular shape of terms – since evaluation is syntax-directed, there is an inversion lemma for each case in the syntax of terms. Each proof of an inversion lemma proceeds by induction on the typing derivation – there are always two cases to consider: the particular base case for this shape of term, and the subsumption case. Since evaluating an application relies on the function evaluating to a closure, we also need an inversion lemma on value typing for this case. The inversion lemma are described in Figures 8 and 9.

## 6. Proving Transitivity and Narrowing

We have stated in Section 3.2 that $\mu$DOT enjoys the properties of environment narrowing and subtyping transitivity:

$$\frac{\Gamma^a, (x : U), \Gamma^b \vdash T <: T' \qquad \Gamma^a \vdash S <: U}{\Gamma^a, (x : S), \Gamma^b \vdash T <: T'} \quad (<:\text{-NARROW})$$

$$\frac{\Gamma \vdash S <: T \ , \ T <: U}{\Gamma \vdash S <: U} \quad (<:\text{-TRANS})$$

While simple to state, proving these lemmas is actually quite involved. We outline several failed proof attempts below before describing our final successful proof strategy.

***Attempt 0: Independent Proofs*** This clearly doesn't work: narrowing and transitivity are mutually dependent.

Since types may be record types, subtyping transitivity relies on environment narrowing. In a chain,

$$\{z \Rightarrow \overline{D_1}\} <: \{z \Rightarrow \overline{D_2}\} <: \{z \Rightarrow \overline{D_3}\}$$

both derived by REC-<:-REC, we need to prove $\{z \Rightarrow \overline{D_1}\} <: \{z \Rightarrow \overline{D_3}\}$ through narrowing the right-hand assumption of the premises

$$z : \{z \Rightarrow \overline{D_2}\} \vdash \{z \Rightarrow \overline{D_2}\} <: \{z \Rightarrow \overline{D_3}\}$$

to match the left-hand assumption

$$z : \{z \Rightarrow \overline{D_1}\} \vdash \{z \Rightarrow \overline{D_2}\} <: \{z \Rightarrow \overline{D_3}\}$$

and then apply transitivity again to derive

$$z : \{z \Rightarrow \overline{D_1}\} \vdash \{z \Rightarrow \overline{D_1}\} <: \{z \Rightarrow \overline{D_3}\}$$

which completes the case.

Narrowing also uses transitivity: consider `T <: p.L` with bounds `S..U` that are narrowed to `S'..U'`. We have `T <: S` and `S <: S'` and derive `T <: S <: S'`.

***Attempt 1: Induction on Subtyping Derivations*** We can try a proof by induction on the subtyping derivations in the premise. For the termination measure, we use the size of the derivations – at least one of them decreases when invoking an induction hypothesis. To deal with derivations in contravariant position, we keep a notion of polarity so that we can switch the two derivations, and the polarity to maintain a decreasing measure.

Problem: both the covariant and contravariant version must use narrowing on the right hand derivation, but that one only gets smaller in one of them.

***Attempt 2: Induction on the Middle Type*** For $\Gamma \vdash T_1 <: T_2 <: T_3$, we can try induction on $T_2$. This is the strategy commonly employed for proofs of system $F_{<:}$, which also requires a mutually inductive proof for transitivity and narrowing (see e.g. the PoplMark Challenge [3]).

Problem: the key difference to $F_{<:}$ is that there, only subtyping rules of the form $X <: T$ exist but not $T <: X$, for type variables $X$.

In $\mu$DOT however we have both variants, and we need to handle the case $T_1 <: p.L <: T_2$, the analogue of which cannot occur in $F_{<:}$. Given bounds $S..U$, we obtain from the premises derivations $T_1 <: S <: U$ and $S <: U <: T_2$, but we cannot use induction on them because neither of $S$ nor $U$ is a syntactic subterm of $p.L$.

***Attempt 3: Induction on Well-Formedness Witness*** Instead of induction on $T_2$, we could do induction on $\Gamma \vdash T_2$ **wf**. Well-formedness of a type selection can include well-formedness of the bounds, so the previous case goes through.

Problem: we run in cycles.

Since only finite expansions are ever needed in practice, a possible solution (which we have explored but not finished) may be to parameterize subtyping over the maximum derivation depth needed.

***Attempt 4: Delaying Transitivity*** Define $\Gamma \vdash T <:^\star T'$ that admits transitivity as an additional axiom. The narrowing proof becomes independent of transitivity.

Problem: inversion lemmas become much harder – for example, relating method members in VINV-CLO-M, because one cannot reason structurally on subtyping derivations.

***Success: Push-Back of Transitivity Axioms*** From $T <:^\star T'$ compute $T <: T'$ by pushing top-level uses of the transitivity axiom one level down into the derivation.

The issue is again that $T <:^\star T'$ might resolve to $T <:^\star p.L <:^\star T'$ through the transitivity axiom, which, assuming bounds $S..U$ leaves us with a chain $T <:^\star S <:^\star U <:^\star T'$ whose elements may again use transitivity at the top level.

Our solution involves two separate steps. First we define a one-step inversion that takes $T_1 <: T_2$ and $T_2 <: T_3$ where $T_2 \neq p.L$ and returns $T_1 <: T_3$.

As a second step, we tackle chains resulting from $T_1 <:^\star T_n$. We start with $T_n$ and accumulate all the primitive steps in a list, i.e. successively build up a representation of $T_1 <: \ldots, <: (T_{n-2} <: (T_{n-1} <: T_n))$ from the right. Whenever we add a new derivation on the left, we check if we are in a $T_1 <: p.L <: T_2$ situation and replace $p.L$ with its bounds.

This gives us a list of steps without $p.L$s in the middle, so we can just go through and reduce with the one-step inversion to obtain the final $T_1 <: T_n$ result.

***Outlook*** This layered proof structure seems well-suited to extensions. The main requirement is that we can't narrow to types that don't expand.

The hope is that any *realizable* type, i.e. type that can be assigned as a result of an object creation, must expand. Furthermore, for soundness, the static expansion relied on to type-check an object creation must be exact – see the aside in the discussion of open refinements in Section 2.1.

For extensions to $\mu$DOT, in big-step style, we might thus get away with a restricted form of narrowing, which is only valid when narrowing to a realizable context, guaranteed to be realizable because it is derived from typing values with precisely known expansions.

In a small-step style proof, the trade-offs might be different: one might admit unrestricted narrowing and confine transitivity pushback to realizable contexts. Perhaps, one way to achieve this option would be to admit subsumption anywhere – even on paths of type selections – except where precise expansion is required during type-checking of object creations. Then, narrowing would be trivial but transitivity would need to be carefully controlled: in the body of a method with an unrealizable parameter type, it might be possible to collapse the lattice by deriving $\top <: \bot$ or other absurd relations through the subsumption of bad bounds to incompatible good bounds via transitivity. For example, take a parameter $z$ of unrealizable type $\{z \Rightarrow \textbf{type}\, X : \top..\top\} \wedge \{z \Rightarrow \textbf{type}\, X : \bot..\bot\}$. Now, within the body, we could derive $\top <: \bot$ via transitivity on $z.X$: the judgement $\top <: z.X$ subsuming $z$ to $\{z \Rightarrow \textbf{type}\, X : \top..\top\}$ and the judgement $z.X <: \bot$ subsuming $z$ to $\{z \Rightarrow \textbf{type}\, X : \bot..\bot\}$. Clearly, the proof of transitivity pushback cannot (and should not) be extended to such inconsistent middle men.

## 7. Reconciling Nominality and Refinements

The $\mu$DOT calculus supports nominality but not general refinement. We can also build sound type systems with general refinements but without nominality. Building a sound calculus which combines both features is harder. Indeed, as detailed in Section 6, we cannot expect subtyping to enjoy the intuitive properties of narrowing and transitivity in all contexts, once extensions can express unrealizable types or incompatible subtypes.

We explore the design space and trade-offs around supporting both nominality and general refinements in the same calculus. We first define these two features, and then discuss the tension between them.

**Nominality** As in $\mu$DOT, a type selection can be treated nominally, based on its name, even in the context of subtyping types where the selected type member has different bounds. In particular, this subtyping proposition should hold as it does in $\mu$DOT:

```
{ z => type A: X .. X;       def id(z.A):z.A } <:
{ z => type A <: { a => };   def id(z.A):z.A }
```

where `X` is a valid but irrelevant type. Notice the resemblance with establishing that `Cow` is a subtype of `Animal`.

**General Refinements** If we admit types $\{z \Rightarrow T \wedge T'\}$ instead of just $\{z \Rightarrow \overline{D}\}$ we obtain general open refinements, powerful enough to express types arising through OOP constructs like inheritance and mixins.

Historically, we developed our understanding of the design space by iteratively adding features in a bottom-up fashion. In our exploration of general refinements, we started with a calculus that had only one subtyping relation, which was two-headed to support the typing of both terms and values as explained in Section 5.1.

Given that setting, essentially, we have two choices for defining the REC-<:-REC rule in subtyping, assuming we have general refinements. We describe the choices starting with the empty context for exposition.

$$\frac{z : T \vdash T <: T' \dashv z : T'}{\{z \Rightarrow T\} <: \{z \Rightarrow T'\}} \qquad \text{(A)}$$

$$\frac{z : T \vdash T <: T' \dashv z : T}{\{z \Rightarrow T\} <: \{z \Rightarrow T'\}} \qquad \text{(B)}$$

The choice boils down to whether we use the right-hand type (A) or left-hand type (B) for the self in the right-hand context. Changing nothing else, choice (A) achieves transitivity but gives up nominality because reflexivity no longer applies to `z.A` once the environments on each side diverge; choice (B) achieves nominality but gives up transitivity because environment narrowing no longer applies once we admit intersection types. Notice that $\mu$DOT makes choice (B) without giving up transitivity, because narrowing applies.

Now, let us go ahead with choice (A), but "patch" the calculus to admit $z : T \vdash z.A <: z.A \dashv z : T'$ to restore reflexive nominality. This naive patch breaks transitivity. Suppose `T` defines `type A: S .. U` and `T'` defines `type A: S' .. U'` with strictly wider bounds. Then, `S` $<:$ `z.A(S .. U)` $<:$ `z.A(S' .. U')`, but we cannot show `S` $<:$ `z.A(S' .. U')` since `S'` $<:$ `S`, not the other way around! We can "patch" once more by ignoring lower bounds, and always comparing upper bounds. Then, we give up nominality as type ascription.

***Two Kinds of Subtyping: Static and Dynamic*** Because our calculus combines a big-step operational semantics with a type system based on subtyping, we discovered a way out of this dilemma: we can use different subtyping semantics when typing expressions (statically) and values (dynamically). For expressions, we want to be strict in order to achieve nominality, even at the expense of transitivity. For values, we want to be lenient, in order to ensure transitivity. In such a setting, we mechanically proved that the calculus is type-safe as long as the static strict semantics imply the dynamic lenient semantics.

For example, we can apply the final "patch" that ignores lower bounds only to the lenient dynamic calculus, so that the strict static calculus still has nominality as type ascription. However, this system can only be proven sound, if the first "patch" that restores reflexive nominality is admitted only if the bounds on the left-hand side are no wider than the bounds on the right-hand side. Hence, reflexive nominality is supported only in covariant position, which is rather limiting. With additional machinery to flip the bounds in contravariant positions, it might possible to restore full reflexive nominality using the same underlying technique. In short, the key take-away is that the semantics of subtyping need not be the same for terms and values; they can be more relaxed when typing values than when typing terms without affecting soundness.

Another avenue to try is the following: value typing can be more precise than term typing, because the exact run-time environments are known. Instead of using a static approximation of a run-time environment, we can actually construct a precise and unique static environment from a run-time environment, given the precise and unique static expansion of each value. We can then use this precise static environment, known to be realizable, to ensure that all static approximations of it are compatible. This way, we can take subtyping derivations coming from inversion lemmas of term typing – defined on an approximate static environment – and narrow these subtyping derivations to the more precise static environment constructed from the value environment.

***Two Kinds of Subtyping: Strong and Weak*** Let us now take a step back and consider an alternative point in the design space, which again uses two subtyping relations but in a different way.

First, we observe that our type safety proof does not rely on a general environment narrowing statement, but only on a single use of environment narrowing in one particular case of the subtyping transitivity proof: the middle type is a general refinement and both subtyping subderivations are derived by REC-<:-REC.

Now, the idea is to strengthen the premise of REC-<:-REC to require just as much additional fuel to enable environment narrowing in this particular case. This is where the second, alternative subtyping relation comes in: instead of using the regular subtyping relation $<:$ we can re-define REC-<:-REC to use a stronger relation $<<:$ in the premise:

$$\frac{z : T \vdash T <<: T'}{\{z \Rightarrow T\} <: \{z \Rightarrow T'\}} \qquad \text{(C)}$$

If we are able to define $<<:$ such that it is transitive, supports environment narrowing, and implies regular subtyping $<:$, we achieve transitivity of $<:$ by design.

A necessary requirement for environment narrowing of $<<:$ is implied by the original definition of REC-<:-REC: if $T <<: T'$, then for all (non-method) declarations $D'$ in (the expansion of) $T'$, there must exist a declaration $D$ in (the expansion of) $T$, such that the latter subsumes the former, $z : T \vdash D <: D'$.

We conjecture that, for $\mu$DOT extended with a subtyping lattice, this requirement may also be sufficient and could be taken as a definition of $<<:$, provided that expansion for intersection types is defined appropriately. This conjecture seems in line with the realizability requirements for narrowing defined at the end of Section 6.

***Resolution*** In some sense, the two-headed subtyping judgement as we envisioned it at the outset was trying to achieve too much. On the one hand, it was intended to opaquely relate types across distinct environments to reconcile each definition site of a closure with some call site. On the other hand, it was also trying to maintain the notion of nominality through reflexivity of path-dependent types of Section 3.1, in the face of new extensions. However, we cannot expect "environment hopping" to be sensibly defined, when a supertype might have two incompatible subtypes – which is already the case in $\mu$DOT, for example: let the supertype be $U = \{z \Rightarrow \textbf{type}\, X <: \top\}$ and the two subtypes be $S_1 = \{z \Rightarrow \textbf{type}\, X : \top..\top\}$ and $S_2 = \{z \Rightarrow \textbf{type}\, X : \{x \Rightarrow \textbf{def}\, m : \top \to \top\}..\{x \Rightarrow \textbf{def}\, m : \top \to \top\}\}$. Now, it wouldn't be valid to hop from $G_2 = z : S_2$ to $G_1 = z : S_2$ via $G_U = z : U$, in the subtyping chain $\emptyset \vdash \{x \Rightarrow \textbf{def}\, m : \top \to \top\} <: z.X \dashv G_2$, then $G_2 \dashv z.X <: z.X \vdash G_U$, and finally $G_U \dashv z.X <: z.X \vdash G_1$ as the transitive conclusion $\emptyset \vdash \{x \Rightarrow \textbf{def}\, m : \top \to \top\} <: z.X \dashv G_1$ would not be sound.

The layered proof structure of Section 6 resolves this dual use of the two-headed subtyping judgement by limiting its use to the first intention – relating distinct environments. Therefore, reflexivity through nominality only holds when the two environments are exactly the same. Then, hopping between distinct approximations of the same run-time environment is allowed only through narrowing to the most precise environment (which is known to be realizable even in extensions as it is constructed from typing values), thus ensuring compatibility of environments across several hops.

## 8. Related Work

***ML Module Systems*** Dependent types have been known in programming languages at least since the ML module system. The original approach to modules in SML [21] could expose the implementation of a type member in a transparent binding, but equations between type members of an abstract signature were only introduced by Harper and Lillibridge [14] and Leroy [19]. Unlike the calculi studied here, their systems are stratified, i.e. definitions can only depend on earlier definitions in the same type, which allows to ensure well-formedness of definitions by construction. Hence, recursion is not allowed. Also, type bounds are not considered. Similarly to Scala's approach, the MixML language [8] drops the stratification requirement and also allows modules as first class values.

***Path-dependent Types in OO Languages*** In object-oriented programming languages, path-dependent types were first proposed for family polymorphism by Ernst [9]. Formal treatments of them in object-oriented languages have been studied by Odersky et al.[7, 26]. Their models are considerably more complicated than the setting studied here because they also need to express inheritance including mixin linearization and some notion of classes.

Other research generalizes this even further by introducing virtual classes [10, 12, 25]. Virtual classes abstract not only over the type, but also over classes and inheritance: in a class extension, the actual superclass used at run-time can be a subclass of the class that appears statically in the extends clause. This feature is powerful but also hard to control, because the unknown superclass might introduce bindings which conflict with those in the subclass. Consequently, type systems for virtual classes either need additional restrictions or more type machinery to control these interactions.

The Tribe calculus [5] builds an ownership type system [4] on top of a core calculus which models virtual classes. The soundness proof for the core calculus seems to be tied to the ownership types system.

The *vc* calculus [11] models virtual classes with path-dependent types. *vc* restricts paths to start with "this", though it provides a way ("out") to refer to the enclosing object. In $\mu$DOT, any variable including self-binding variables can take part in a type selection. Like $\mu$DOT, *vc* is defined using big-step evaluation rules.

***Foundations for Scala*** Since the early days of Scala, grounding its type system on firm theory has been an area of active research, and several calculi were proposed to capture various aspects of the language. Alas, over the course of 10 years no verifiable type safety proof for any of these systems has been established.

Notable previous efforts include $\nu$Obj [26], Featherweight Scala [7] and Scalina [24]. The $\nu$Obj calculus relies on features beyond the full Scala language, for example classes as first-class values, and contains a comparatively large type language, including distinct notions of singleton types, type selections, record types, class types and compound types, which make $\nu$Obj rather unwieldy in practice and not very suitable to extensions. In particular, subtyping does not provide unique upper or lower bounds, and mixin composition is not commutative. Type checking in $\nu$Obj was shown to be undecidable through an encoding of the $F_{<:}$ system. The $\nu$Obj paper [26] claims a type soundness result, but there is no machine-verified proof. Featherweight Scala was an attempt at a calculus with decidable type checking, but soundness was explicitly left as future work [7]. Scalina [24] was proposed as a formal underpinning for introducing higher-kinded types in Scala. Among other constructs, it contains concepts such as un-members, un-types and un-kinds to model contravariant refinements. Soundness of Scalina has not been established.

All these previous systems have in common that their models are considerably more complicated than the setting studied here, in particular due to the choice of including particular notions of classes, implementation inheritance and mixin linearization. Due to this complexity, none of the previous formal models proved suitable as a core calculus that could serve as a basis for mechanized proofs of various extensions, and more importantly, none of the previous calculi provided much insight into *why* type soundness was hard to prove.

***The Essence of Path Dependent Types*** Compared to previous work that aimed to describe an existing language more or less precisely, $\mu$DOT focuses on the essence of path-dependent types and nothing else. It reduces complexity as far as eschewing fields in the term language, and having only variables as paths. Despite and due to such restrictions,

$\mu$DOT is more general than previous models as it does not assume a class-based language or any other particular choice of implementation reuse.

Previous work [2] on a core calculus for path-dependent types based on a small-step operational semantics with a store to identify objects explained the difficulty of relating paths at different stages of reductions. In a big-step approach, the challenge is to bootstrap self-references without compromising expressivity: since values are closures – if a closure now has fields initialized by terms during object creation, those terms need to be evaluated before the object.

This present work has provided valuable insights why previous efforts to formalize the Scala type system have failed, and it shows that a core calculus, without the baggage of a full language, has a clean and intuitive theory.

To the best of our knowledge, $\mu$DOT is the first calculus with path-dependent types that has a fully mechanized type soundness proof.

## 9. Conclusion and Future Work

We conclude our exploration of path-dependent types with a few lessons from the past, and avenues for the future.

***Process*** We have tried various approaches for designing a core calculus for path-dependent types, and proving it sound. In total, we went through about 30 variations of mechanized models. At the risk of oversimplifying, we classify the approaches in two dimensions: top-down vs. bottom-up and proof-driven vs. query-driven. Top-down means starting with the full calculus, and bottom-up means adding the wanted features one-by-one. Proof-driven means trying to show some general result about the calculus, and query-driven means trying out some particular examples. Both when working bottom-up or top-down, we found it essential to alternate proof-driven and query-driven phases; however, these phases play different roles in each approach. In a top-down approach, we go proof-driven until we get stuck, and then we go query-driven to pinpoint a counterexample. In a bottom-up approach, we go query-driven to understand the expressivity and limitations of what we proved. Switching from a top-down to a bottom-up approach resulted in more reusable insights in terms of which feature combinations are problematic.

***Tools*** During our journey, we have used many tools, including Coq [22], Dafny [18], Twelf [27], Redex [17]. Dafny worked well for exploring the proof structure top-down in an efficient high-level way. Redex worked well for query-driven phases, and pinpointing counterexamples. Twelf worked well in a bottom-up process alternating proving and querying – and surprisingly, in our current workflow, we use Twelf almost exclusively (almost, because for queries, we use Twelf as a backend only by tacking a frontend written in Scala to translate examples to the AST used in the Twelf models). Though Twelf is rather low-level, being a proof checker more than a proof assistant, in a bottom-up process, it is insightful to see exactly which steps of the proof break when adding a new feature. Though Twelf's query engine is rather simple, not to mention incomplete, we were surprised by the examples we were able to express after careful re-ordering of the clauses despite using blatantly non-algorithmic rules like subsumption. It has also been insightful to sometimes get a confirmed exhaustively checked negative result for a query, showing that it is inexpressible in the model. More importantly, we found it crucial to use the same system for proving and querying, not only for convenience but especially for confidence.

***Big-Step Operational Semantics*** We believe that big-step operational semantics deserves to be more widely used. It is often dismissed because of the tedious requirement of explicit "stuck" cases. In practice, we find it convenient to prove just preservation first, and then extend the preservation lemma to a full type safety theorem in a separate iteration, by adding the necessary step indexes. We also find that when working bottom-up on the *type* system, the evaluation stays the same, and so the core proof remains the same – what changes are the proofs of the subtyping properties such as environment weakening and transtivity.

In small-step operational semantics based on rewriting [31], the substitution lemma is crucial and often tricky. The substitution model is nice conceptually, but may not actually correspond to evaluation of programs. The main difficulty is assigning types to partially evaluated terms – an almost artificial requirement if the end goal is to show that "well-typed terms don't get stuck".

***Foundations*** We hope that $\mu$DOT will serve as a "featherweight" calculus in the spirit of [16], on top of which more features can be studied or into which other features can be translated. For example, expressing Ross Tate's Mixed-Site Variance [30] on top of a DOT-like calculus would be an interesting case study. Of course a long-term goal is to provide a firm basis for future versions of Scala and similar languages, in form of type systems with fewer but more powerful building blocks: traits, mixin composition / refinements, path-dependent types – but nothing more. Since $\mu$DOT does not model any notion of classes or inheritance, we anticipate that it may be a useful foundation for class-less languages as well. In particular, we believe that the ability to model nominal abstraction as ascription in an otherwise structural type system will be useful in the context of gradual typing [29].

***Mind the Gap!*** A key contribution of this paper is to document important insights why previous attempts to formalize the Scala type system have failed, and to show that a core calculus, without the baggage of a full language, has a clean and intuitive theory – thus, exposing the sausage factory of designing calculi, and the mine fields in the landscape [1].

**Subtyping of Types** $\boxed{\Gamma \vdash S <: U}$

$$\frac{\Gamma \vdash T\ \textbf{wf}}{\Gamma \vdash T <: T}\ (\textsc{Refl})$$

$$\frac{\Gamma \vdash x \ni \textbf{type}\ L <: U\ ,\ U <: U'}{\Gamma \vdash x.L <: U'}\ (\textsc{TSel}_u\text{-}<:)$$

$$\frac{\Gamma \vdash x \ni \textbf{type}\ L : S..U\ ,\ S' <: S\ ,\ S <: U}{\Gamma \vdash S' <: x.L}\ (<:\text{-}\textsc{TSel})$$

$$\frac{\Gamma \vdash \{z \Rightarrow \overline{D'}\}\ \textbf{wf}\ ;\ \overbrace{\Gamma, z : \{z \Rightarrow \overline{D}\} \vdash D_i <: D'_i}^{\forall i, D'_i}}{\Gamma \vdash \{z \Rightarrow \overline{D}\} <: \{z \Rightarrow \overline{D'}\}}\ (\textsc{Rec-}<:\textsc{-Rec})$$

$$\frac{\Gamma \vdash x \ni \textbf{type}\ L : S..U\ ,\ U <: U'\ ,\ S <: U}{\Gamma \vdash x.L <: U'}\ (\textsc{TSel-}<:)$$

**Subtyping of Declarations** $\boxed{\Gamma \vdash D <: D'}$

$$\frac{\begin{array}{c}\Gamma \vdash S' <: S\ ,\ U <: U' \\ \Gamma \vdash S <: U\ ,\ S' <: U'\end{array}}{\Gamma \vdash (\textbf{type}\ L : S..U) <: (\textbf{type}\ L : S'..U')}\ (\textsc{TDecl-}<:\textsc{-TDecl})$$

$$\frac{\Gamma \vdash U <: U'\ ,\ S <: U}{\Gamma \vdash (\textbf{type}\ L : S..U) <: (\textbf{type}\ L <: U')}\ (\textsc{TDecl-}<:\textsc{-TDecl}_u)$$

$$\frac{\Gamma \vdash U <: U'}{\Gamma \vdash (\textbf{type}\ L <: U) <: (\textbf{type}\ L <: U')}\ (\textsc{TDecl}_u\text{-}<:\textsc{-TDecl}_u)$$

---

**Well-Formed Types** $\boxed{\Gamma \vdash T\ \textbf{wf}}$

$$\frac{\Gamma \vdash x \ni \textbf{type}\ L : S..U\ ,\ S <: U}{\Gamma \vdash x.L\ \textbf{wf}}\ (\textsc{TSel-WF})$$

$$\frac{\overbrace{\Gamma, z : \{z \Rightarrow \overline{D}\} \vdash D_i\ \textbf{wf}}^{\forall i, D_i}}{\Gamma \vdash \{z \Rightarrow \overline{D}\}\ \textbf{wf}}\ (\textsc{Rec-WF})$$

$$\frac{\Gamma \vdash x \ni \textbf{type}\ L <: U\ ,\ U\ \textbf{wf}}{\Gamma \vdash x.L\ \textbf{wf}}\ (\textsc{TSel}_u\textsc{-WF})$$

**Well-Formed Declarations** $\boxed{\Gamma \vdash D\ \textbf{wf}}$

$$\frac{\Gamma \vdash S <: U}{\Gamma \vdash (\textbf{type}\ L : S..U)\ \textbf{wf}}\ (\textsc{TDecl-WF})$$

$$\frac{\Gamma \vdash U\ \textbf{wf}}{\Gamma \vdash (\textbf{type}\ L <: U)\ \textbf{wf}}\ (\textsc{TDecl}_u\textsc{-WF})$$

---

**Expansion** $\boxed{\Gamma \vdash T \prec_z \overline{D}}$

$$\frac{\Gamma \vdash x \ni \textbf{type}\ L : S..U\ ,\ U \prec_x \overline{D}}{\Gamma \vdash x.L \prec_x \overline{D}}\ (\textsc{TSel-}\prec)$$

$$\frac{\Gamma \vdash x \ni \textbf{type}\ L <: U\ ,\ U \prec_x \overline{D}}{\Gamma \vdash x.L \prec_x \overline{D}}\ (\textsc{TSel}_u\text{-}\prec)$$

$$\Gamma \vdash \{z \Rightarrow \overline{D}\} \prec_z \overline{D}\quad (\textsc{Rec-}\prec)$$

**Membership** $\boxed{\Gamma \vdash x \ni D}$

$$\frac{\begin{array}{c}(x : T) \in \Gamma \\ \Gamma \vdash T \prec_x \overline{D}\end{array}}{\Gamma \vdash x \ni D_i}\ (\textsc{Var-}\ni)$$

---

**Figure 3.** $\mu$DOT$_T$: Semantics of Types

**Typing of Terms** $\boxed{\Gamma \vdash t : T}$

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \quad (\text{VAR})$$

$$\frac{\begin{array}{c}\overline{I_i : D_i} \\ \Gamma \vdash \{z \Rightarrow \overline{D}\}\ \mathbf{wf} \\ \overbrace{\Gamma, x : S, z : \{z \Rightarrow \overline{D}\} \vdash t : U}^{\forall i, I_i = (\mathbf{def}\ m(x:S):U = t)}\end{array}}{\Gamma \vdash \mathbf{new}\ (z \Rightarrow \overline{I}) : \{z \Rightarrow \overline{D}\}} \quad (\text{NEW})$$

$$\frac{\Gamma \vdash t : \{\_ \Rightarrow \mathbf{def}\ m : S \to U\}\ ,\ t' : S}{\Gamma \vdash t.m(t') : U} \quad (\text{APP})$$

$$\frac{\Gamma \vdash t : T\ ,\ T <: T'}{\Gamma \vdash t : T'} \quad (\text{SUB})$$

**Conversion of Initializations to Declarations** $\boxed{I : D}$

$$(\mathbf{type}\ L : S..U) : (\mathbf{type}\ L : S..U) \quad (\text{T-I2D})$$

$$(\mathbf{def}\ m(x : S) : U = t) : (\mathbf{def}\ m : S \to U) \quad (\text{M-I2D})$$

$$(\mathbf{type}\ L <: U) : (\mathbf{type}\ L <: U) \quad (\text{T}_u\text{-I2D})$$

---

**Typing of Values** $\boxed{\Gamma \vdash v : T}$

$$\frac{\begin{array}{c} H : \Gamma^H \\ \overbrace{D_i^T = (\mathbf{type}\ L_i : S_i^T..U_i^T)\ \mathbf{or}\ (\mathbf{type}\ L_i <: U_i^T)}^{\forall i \le N} \\ \overbrace{D_i^M = (\mathbf{def}\ m_i : S_i^M \to U_i^M)}^{\forall i, E_i = (\mathbf{def}\ m_i(x_i) = t_i)} \\ \overbrace{\Gamma^H, z : \{z \Rightarrow \overline{D^T D^M}\}, x : S_i^M \vdash t_i : U_i^M}^{\forall i, E_i} \\ \Gamma^H, z : \{z \Rightarrow \overline{D^T D^M}\} \vdash \{z \Rightarrow \overline{D^T D^M}\} <: T \dashv \Gamma \end{array}}{\Gamma \vdash \texttt{<}z \Rightarrow \overline{E}\ \mathbf{in}\ H\texttt{>} : T} \quad (\text{VCLO})$$

$$\frac{\begin{array}{c}\Gamma \vdash v : T \\ \Gamma \vdash T <: T' \dashv \Gamma'\end{array}}{\Gamma' \vdash v : T'} \quad (\text{VSUB})$$

**Typing of Run-Time Environments** $\boxed{\mathrm{H} : \Gamma}$

$$\emptyset : \emptyset \quad (\text{WFE-NIL})$$

$$\frac{\begin{array}{c}(\overline{x : v}) : (\overline{x : T}) \\ (\overline{x : T}, y : T_y) \vdash v_y : T_y\end{array}}{(\overline{x : v}, y : v_y) : (\overline{x : T}, y : T_y)} \quad (\text{WFE-CONS})$$

---

**Subtyping of Declarations ...** $\boxed{\Gamma \vdash D <: D'}$

$$\frac{\Gamma \vdash S' <: S\ ,\ U <: U'}{\Gamma \vdash (\mathbf{def}\ m : S \to U) <: (\mathbf{def}\ m : S' \to U')} \quad (\text{MDECL-<:-MDECL})$$

**Well-Formed Declarations ...** $\boxed{\Gamma \vdash D\ \mathbf{wf}}$

$$\frac{\Gamma \vdash S\ \mathbf{wf}\ ,\ U\ \mathbf{wf}}{\Gamma \vdash (\mathbf{def}\ m : S \to U)\ \mathbf{wf}} \quad (\text{MDECL-WF})$$

**Figure 5.** $\mu$DOT: Extra Static Semantics

**Evaluation** $\boxed{H \vdash t \Downarrow v}$

$$\frac{(x : v) \in H}{H \vdash x \Downarrow v} \quad (\text{EVAR})$$

$$\frac{H \vdash t \Downarrow \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>} \quad H \vdash t' \Downarrow v' \quad H^t, z : \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>}, x_i : v' \vdash t_i \Downarrow v_i}{H \vdash t.m_i(t') \Downarrow v_i} \quad (\text{EAPP})$$

$$\frac{E = \overbrace{\textbf{def}\, m_i(x_i) = t_i}^{\forall i, I_i = (\textbf{def}\, m_i(x_i : S_i) : U_i = t_i)}}{H \vdash \textbf{new}\,(z \Rightarrow \overline{I}) \Downarrow \texttt{<}z \Rightarrow \overline{E} \textbf{ in } H\texttt{>}} \quad (\text{ENEW})$$

**Figure 6.** $\mu$DOT: Dynamic Semantics

**Total Evaluation** $\boxed{H \vdash t \Downarrow_n (v | \textbf{timeout} | \textbf{stuck})}$

$$\frac{(x : v) \in H}{H \vdash x \Downarrow_{n+1} v} \quad (\text{TE-VAR})$$

$$\frac{E = \overbrace{\textbf{def}\, m_i(x_i) = t_i}^{\forall i, I_i = (\textbf{def}\, m_i(x_i : S_i) : U_i = t_i)}}{H \vdash \textbf{new}\,(z \Rightarrow \overline{I}) \Downarrow_{n+1} \texttt{<}z \Rightarrow \overline{E} \textbf{ in } H\texttt{>}} \quad (\text{TE-NEW})$$

$$\frac{H \vdash t \Downarrow_{n+1} \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>} \quad H \vdash t' \Downarrow_{n+1} v' \quad H^t, z : \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>}, x_i : v' \vdash t_i \Downarrow_n v_i}{H \vdash t.m_i(t') \Downarrow_{n+1} v_i} \quad (\text{TE-APP})$$

$$H \vdash t \Downarrow_0 \textbf{timeout} \quad (\text{TE-TIMEOUT})$$

$$\frac{H \vdash t \Downarrow_{n+1} \textbf{timeout}}{H \vdash t.m_i(t') \Downarrow_{n+1} \textbf{timeout}} \quad (\text{TE-APP-TIMEOUT-1})$$

$$\frac{H \vdash t \Downarrow_{n+1} \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>} \quad H \vdash t' \Downarrow_{n+1} \textbf{timeout}}{H \vdash t.m_i(t') \Downarrow_{n+1} \textbf{timeout}} \quad (\text{TE-APP-TIMEOUT-2})$$

$$\frac{H \vdash t \Downarrow_{n+1} \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>} \quad H \vdash t' \Downarrow_{n+1} v' \quad H^t, z : \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>}, x_i : v' \vdash t_i \Downarrow_n \textbf{timeout}}{H \vdash t.m_i(t') \Downarrow_{n+1} \textbf{timeout}} \quad (\text{TE-APP-TIMEOUT-3})$$

$$\frac{H \vdash t \Downarrow_{n+1} \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>} \quad m \notin \overline{m_i}}{H \vdash t.m(t') \Downarrow_{n+1} \textbf{stuck}} \quad (\text{TE-APP-STUCK-1})$$

$$\frac{H \vdash t \Downarrow_{n+1} \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>} \quad H \vdash t' \Downarrow_{n+1} \textbf{stuck}}{H \vdash t.m_i(t') \Downarrow_{n+1} \textbf{stuck}} \quad (\text{TE-APP-STUCK-2})$$

$$\frac{H \vdash t \Downarrow_{n+1} \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>} \quad H \vdash t' \Downarrow_{n+1} v' \quad H^t, z : \texttt{<}z \Rightarrow \overline{m_i(x_i) = t_i} \textbf{ in } H^t\texttt{>}, x_i : v' \vdash t_i \Downarrow_n \textbf{stuck}}{H \vdash t.m_i(t') \Downarrow_{n+1} \textbf{stuck}} \quad (\text{TE-APP-STUCK-3})$$

**Figure 7.** $\mu$DOT: Total Dynamic Semantics

If a variable type-checks to a type, inversion asserts that the variable is bound to a subtype in the typing environment:

$$\frac{\Gamma \vdash x : T}{\begin{array}{c}(x : T') \in \Gamma \\ \Gamma \vdash T' <: T\end{array}} \quad \text{(INV-VAR)}$$

If a method invocation type-checks, inversion asserts that the term invoked has an arrow type, whose parameter type matches the type of the argument, and whose return type is a subtype of the type of the invocation:

$$\frac{\Gamma \vdash t.m(t') : T}{\begin{array}{c}\Gamma \vdash t : \{\_ \Rightarrow \textbf{def}\, m : S \to U\} \\ \Gamma \vdash t' : S \\ \Gamma \vdash U <: T\end{array}} \quad \text{(INV-APP)}$$

If a new object creation type-checks, inversion asserts that the result type is a subtype of a record type matching the initialization of the object:

$$\frac{\Gamma \vdash \textbf{new}\ (z \Rightarrow \overline{I}) : T}{\begin{array}{c}\overline{I_i : D_i} \\ \overbrace{\Gamma, x : S, z : \{z \Rightarrow \overline{D}\} \vdash t : U}^{\forall i, I_i = (\textbf{def}\, m(x:S):U=t)} \\ \Gamma \vdash \{z \Rightarrow \overline{D}\} <: T\end{array}} \quad \text{(INV-NEW)}$$

**Figure 8.** Inversion of Typing of Terms

If a closure type-checks to an arrow type, then inversion asserts that the closure holds an appropriate definition for that method:

$$\frac{\Gamma \vdash \texttt{<}z \Rightarrow \overline{E}\ \textbf{in}\ H\texttt{>} : \{\_ \Rightarrow \textbf{def}\, m : S \to U\}}{\begin{array}{c}H : \Gamma^H \\ (\textbf{def}\, m(x) = t) \in \overline{E} \\ \Gamma^H, z : T, x : S \vdash t : U \\ \Gamma^H, z : T \vdash \{\_ \Rightarrow \textbf{def}\, m : S \to U\} <: \{\_ \Rightarrow \textbf{def}\, m : S \to U\} \dashv \Gamma \\ \Gamma^H, z : T \vdash \texttt{<}z \Rightarrow \overline{E}\ \textbf{in}\ H\texttt{>} : T\end{array}} \quad \text{(VINV-CLO-M)}$$

**Figure 9.** Inversion of Typing of Values

## Acknowledgments

We thank Adriaan Moors, Donna Malayeri and Geoffrey Washburn for previous work on DOT. We thank Amal Ahmed, Derek Dreyer, Samuel Grütter, Scott Kilpatrick, Viktor Kuncak, Lukas Rytz and Jeremy Siek for insightful discussions.

This research was supported by the European Research Council (ERC) under grant 587327 DOPPLER.

## References

[1] N. Amin and T. Rompf. Mind the gap: Artifacts vs insights in pl theory. In *OBT*, 2014. URL `http://popl-obt-2014.cs.brown.edu/papers/gap.pdf`.

[2] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *FOOL*, 2012.

[3] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark Challenge. In *TPHOLs*, 2005.

[4] N. R. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *OOPSLA*, 2010.

[5] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: a simple virtual class calculus. In *AOSD*, 2007.

[6] J. Courant. An applicative module calculus. In *TAPSOFT: Theory and Practice of Software Development*. 1997.

[7] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In *MFCS*, 2006.

[8] D. Dreyer and A. Rossberg. Mixin' up the ML module system. In *ICFP*, 2008.

[9] E. Ernst. Family polymorphism. In *ECOOP*, 2001.

[10] E. Ernst. Higher-order hierarchies. In *ECOOP*, 2003.

[11] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL*, 2006.

[12] V. Gasiunas, M. Mezini, and K. Ostermann. Dependent classes. In *OOPSLA*, 2007.

[13] R. Harper. Programming in standard ml, 2013.

[14] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.

[15] A. Igarashi and B. C. Pierce. Foundations for virtual types. *Inf. Comput.*, 175(1):34–49, 2002.

[16] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3), 2001.

[17] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *POPL*, 2012.

[18] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, 2010.

[19] X. Leroy. Manifest types, modules and separate compilation. In *POPL*, 1994.

[20] X. Leroy. A modular module system. *Journal of Functional Programming*, 10:269–303, May 2000.

[21] D. Macqueen. Using dependent types to express modular structure. In *POPL*, 1986.

[22] The Coq development team. *The Coq proof assistant reference manual*, 2012. URL `http://coq.inria.fr`. Version 8.4.

[23] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[24] A. Moors, F. Piessens, and M. Odersky. Safe type-level abstraction in Scala. In *FOOL*, 2008.

[25] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA*, 2004.

[26] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP*, 2003.

[27] F. Pfenning and C. Schürmann. In *CADE*, 1999.

[28] J. Siek. Type safety in three easy lemmas. `http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html`, 2013.

[29] J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, 2007.

[30] R. Tate. Mixed-site variance. In *FOOL*, 2013.

[31] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.