A Path to DOT: Formalizing Fully Path-Dependent Types

MARIANNA RAPOPORT, University of Waterloo, Canada
ONDŘEJ LHOTÁK, University of Waterloo, Canada

The Dependent Object Types (DOT) calculus aims to formalize the Scala programming language with a focus on *path-dependent types* — types such as $x.a_1 \ldots a_n.T$ that depend on the runtime value of a *path* $x.a_1 \ldots a_n$ to an object. Unfortunately, existing formulations of DOT can model only types of the form $x.A$ which depend on *variables* rather than general paths. This restriction makes it impossible to model nested module dependencies. Nesting small components inside larger ones is a necessary ingredient of a modular, scalable language. DOT's variable restriction thus undermines its ability to fully formalize a variety of programming-language features including Scala's module system, family polymorphism, and covariant specialization.

This paper presents the pDOT calculus, which generalizes DOT to support types that depend on paths of arbitrary length, as well as singleton types to track path equality. We show that naive approaches to add paths to DOT make it inherently unsound, and present necessary conditions for such a calculus to be sound. We discuss the key changes necessary to adapt the techniques of the DOT soundness proofs so that they can be applied to pDOT. Our paper comes with a Coq-mechanized type-safety proof of pDOT. With support for paths of arbitrary length, pDOT can realize DOT's full potential for formalizing Scala-like calculi.

CCS Concepts: • **Theory of computation** → *Program semantics*; • **Software and its engineering** → *Formal language definitions.*

Additional Key Words and Phrases: DOT, Scala, dependent types, paths

ACM Reference Format:
Marianna Rapoport and Ondřej Lhoták. 2019. A Path to DOT: Formalizing Fully Path-Dependent Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 145 (October 2019), 29 pages. https://doi.org/10.1145/3360571

1 INTRODUCTION

Path-dependent types embody two universal principles of modular programming: abstraction and composition.

$$\underbrace{\text{path-dependent}}_{\text{composition}}\ \underbrace{\text{type}}_{\text{abstraction}}$$

Abstraction allows us to leave values or types in a program unspecified to keep it generic and reusable. For example, in Scala, we can define trees where the node type remains abstract:

```
trait Tree {
    type Node
    val root: Node
    def add(node: Node): Tree
}
```

Authors' addresses: Marianna Rapoport, University of Waterloo, Canada, mrapoport@uwaterloo.ca; Ondřej Lhoták, olhotak@uwaterloo.ca, University of Waterloo, Canada.

2475-1421/2019/10-ART145
https://doi.org/10.1145/3360571

If an object `x` has type `Tree`, then the *path-dependent type* `x.Node` denotes the type of abstract nodes.

Composition is the ability to build our program out of smaller components. For example, if we are interested in a specific kind of tree, say a red-black tree, then we can refine the abstract `Node` type to contain a `Color` type:

```
trait RedBlackTree extends Tree {
    type Node <: { type Color }
}
```

This exemplifies composition in at least two ways: by having `RedBlackTree` extend `Tree` we have *inherited* its members; and by nesting the refined definition of `Node` within `RedBlackTree` we have used *aggregation*. If an object `r` is a `RedBlackTree`, then the path-dependent type `r.root.Color` allows us to traverse the composition and access the `Color` type member.

To fulfill their full potential with respect to composition, path-dependent types distinguish between paths that have different runtime values. For example, if we have apple and orange trees, we want to disallow mixing up their nodes:

```
val appleTree: Tree
val orangeTree: Tree
appleTree.add( orangeTree.root)  // expected: appleTree.Node, actual: orangeTree.Node
```

Here, the type system considers `appleTree.Node` and `orangeTree.Node` to be distinct and incompatible types because they depend on the runtime values of different objects.

Furthermore, path-dependent types allow Scala to unify modules and objects, so that the same language constructs can be used to specify the overall structure of a program as well as its implementation details. The unification of the module and term languages is witnessed by the following comparison with the ML module system: Scala objects correspond to ML modules, classes to functors, and interfaces to signatures [22].

The long struggle to formalize path-dependent types recently led to machine-verified soundness proofs for several variants of the Dependent Object Types (DOT) calculus [1, 3, 26]. In spite of its apparent simplicity DOT is an expressive calculus that can encode a variety of language features, and the discovery of its soundness proof was a breakthrough for the Scala community. Insights from the proof have influenced the design of Scala 3 and helped uncover soundness bugs in Scala and Java [5].

However, a crucial limitation is that the existing DOT calculi restrict path-dependent types to depend only on variables, not on general paths. That is, they allow the type `x.Node` (path of length 1) but not a longer type such as `r.root.Color` (length 2). We need to lift this restriction in order to faithfully model Scala which does allow general path-dependent types. More importantly, this restriction must be lifted to fulfill the goal of *scalable component abstraction* [22], in which modules of a program can be arbitrarily nested to form other, larger modules.

In this paper, we formalize and prove sound a generalization of the DOT calculus [1] with path-dependent types of arbitrary length. We call the new path-dependent calculus pDOT. Our Coq-verified proof is built on top of the proof of Rapoport et al. [24].

At this point, two questions naturally arise. Are fully path-dependent types really necessary? That is, do they provide additional expressiveness, or are they just syntactic sugar over variable-dependent types? And if fully path-dependent types are in fact useful, what are the barriers to adding them to DOT?

Why Fully Path-Dependent Types Are Necessary

The need for paths of arbitrary length is illustrated by the following simplified excerpt from the implementation of the Scala 3 ("Dotty") compiler:

Scala:

```
package dotty {
  package core {
    object types {
      class Type
      class TypeRef extends Type {
        val symb: core.symbols.Symbol}}
    object symbols {
      class Symbol {
        val tpe: core.types.Type}}}}
```

DOT pseudocode:

```
let dotty = new {
  val core = new {
    val types = new {
      type Type
      type TypeRef = Type & {
        val symb:  core.symbols.Symbol}}
    val symbols = new {
      type Symbol = {
        val tpe:  core.types.Type}}}}
```

Type references (`TypeRef`) are `Type`s that have an underlying class or trait definition (`Symbol`), while `Symbol`s in the language also have a `Type`. Additionally, `TypeRef`s and `Symbol`s are nested in different packages, `core.types` and `core.symbols`.

It is impossible to express the above type dependencies in DOT while maintaining the nested program structure, as shown on the right in DOT pseudocode (actual DOT syntax is introduced in Section 2.1). The DOT-like excerpt replicates nested Scala modules through objects and fields. Unfortunately, we run into problems when typing the `symb` field because its desired path-dependent type `core.symbols.Symbol` has a path of length two.

We are then tempted to find a workaround. One option is to try to reference `Symbol` as a path-dependent type of length one: `symbols.Symbol` instead of `core.symbols.Symbol`. However, this will not do because `symbols` is a field, and DOT requires that field accesses happen through the enclosing object (`core`). Another option is to move the definition of the `Symbol` type member to the place it is accessed from, to ensure that the path to the type member has length 1.

```
val types = new {
    type Type; type Symbol
    type TypeRef = Type & { val symb: this.Symbol }
}
```

However, such a transformation would require flattening the nested structure of the program whenever we need to use path-dependent types. This would limit encapsulation and our ability to organize a program according to its logical structure. Yet another approach is to assign the `symbols` object to a variable that is defined before the `dotty` object:

```
let symbols = new { type Symbol = { val tpe:  dotty.core.types.Type }} in
let dotty   = new ...
```

This attempt fails as well, as the `symbols` object can no longer reference the `dotty` package. For the above example this means that a `Symbol` cannot have a `Type` (see Section 2.2 for a minimal example of DOT's limited path expressivity).

This real-world pattern with multiple nested modules and intricate dependencies between them (sometimes even *recursive* dependencies, as in our example), leads to path-dependent types of length greater than one. Because path-dependent types are used in DOT to formalize features

like parametric and family polymorphism [12], covariant specialization [6], and wildcards, among others, a version of DOT with just variable-dependent types can only formalize these features in special cases. Thus, to unleash the full expressive power of DOT we need path-dependent types on paths of arbitrary length.

Why Fully Path-Dependent Types Are Hard

The restriction to types dependent on variables rather than paths is not merely cosmetic; it is fundamental. A key challenge in formalizing the DOT calculi is the *bad bounds* problem, discussed in Section 2.3.1: the occurrence of a type member in a program introduces new subtyping relationships, and these subtyping relationships could undermine type safety in the general case. To maintain type safety, the existing DOT calculi ensure that whenever a type $x.A$ is in scope, any code in the same scope will not execute until x has been assigned some concrete value; the value serves as evidence that type soundness has not been subverted. As we show in Section 2.3.2, if we allow a type to depend on a path, rather than a variable, we must extend this property to paths: we must show that whenever a scope admits a given path, that path will always evaluate to some stable value. The challenge of ensuring that the paths of type-selections always evaluate to a value is to rule out the possibility that paths cyclically alias each other, while at the same time keeping the calculus expressive enough to allow recursion. By contrast, the DOT calculus automatically avoids the problem of type selections on non-terminating paths (i.e. paths whose evaluation does not terminate) because in DOT all paths are variables, and variables are considered normal form.

A second challenge of extending DOT with support for general paths is to track *path equality*. Consider the following program:

```
val t1 = new ConcreteTree
val t2 = new ConcreteTree
val t3 = t2
```

A subclass of `Tree` such as `ConcreteTree` (not shown) refines `Node` with a concrete type that implements some representation of nodes. We want the types `t1.Node` and `t2.Node` to be considered distinct even though `t1` and `t2` are both initialized to the same expression. That way, we can distinguish the nodes of different tree instances. On the other hand, notice that the reference `t3` is initialized to be an alias to the same tree instance as `t2`. We therefore want `t2.Node` and `t3.Node` to be considered the same type.

How can the type system tell the difference between `t1.Node` and `t2.Node`, so that the former is considered distinct from `t3.Node`, but the latter is considered the same? Scala uses *singleton types* for this purpose. In Scala, `t3` can be typed with the singleton type `t2.type` which guarantees that it is an alias for the same object as `t2`. The type system treats paths that are provably aliased (as evidenced by singleton types) as interchangeable, so it considers `t2.Node` and `t3.Node` as the same type. We add singleton types to pDOT for two reasons: first, we found singleton types useful for formalizing path-dependent types, and second, enabling singleton types brings DOT closer to Scala.

This paper contributes the following:

1) The **pDOT** calculus, a generalization of DOT with **path-dependent types of arbitrary length** that lifts DOT's type-selection-on-variables restriction. Section 3 provides an intuition for pDOT's main ideas, and Section 4 presents the calculus in detail.
2) The first extension of DOT with **singleton types**, a Scala feature that, in addition to tracking path equality, enables the method chaining pattern and hierarchical organization of components [22].
3) A **Coq-mechanized type soundness proof** of pDOT that is based on the simple soundness proof by Rapoport et al. [24]. Our proof maintains the simple proof's modularity properties

which makes it easy to extend pDOT with new features. We describe the proof in Section 5 and include its Coq formalization in the accompanying artifact.

4) **Formalized examples**, presented in Section 6, that illustrate the expressive power of pDOT: the compiler example from this section that uses general path-dependent types, a method chaining example that uses singleton types, and a covariant list implementation. Our Coq proof can be found under https://git.io/dotpaths.

2 DOT: BACKGROUND AND LIMITATIONS

In this section, we survey the existing DOT calculus and discuss the challenges related to path-dependent types.

2.1 The DOT Calculus

We begin by reviewing the syntax of the DOT calculus of Amin et al. [1].A term t in DOT is a variable x, a value v, a function application $x\,y$, a field selection $x.a$, or a let binding **let** $x = t$ **in** u. The meanings of the terms are standard. The syntax is in Administrative Normal Form (ANF), which forces terms to be bound to variables by let bindings before they are used in a function application or as the base of a path. Values are either lambda abstractions, which are standard, or objects. An object $v(x\colon T)d$ defines a self variable x, which models the Scala `this` construct, specifies a self-type T for the object, and lists the field and type members d defined in the object, separated by the intersection operator $\wedge$. Both the type T and the member definitions d can recursively refer to the object itself through the self variable x.

A DOT type is one of the following:

- A dependent function type $\forall(x\colon S)\,T$ characterizes functions that take an argument of type S and return a result of type T. The result type T may refer to the parameter x.
- A recursive type $\mu\,(x\colon T)$ is the type of an object $v(x\colon T)d$. The type T describes the members of the object, and may refer to the recursive self variable x.
- A field declaration type $\{a\colon T\}$ classifies objects that have a field named a with type T.
- A type-member declaration type $\{A\colon S..U\}$ classifies objects that declare a type member A with the constraints that A is a supertype of S and a subtype of U.
- A type projection $x.A$ selects the member type A from the object referenced by the variable x.
- An intersection type $S \wedge T$ is the greatest subtype of both S and T. Unlike some other systems with intersection types, DOT does not define any distributive laws for intersections.
- The top ($\top$) and bottom ($\bot$) types are the supertype and subtype of all types, respectively.

In the following, we will write $v(x)d$ instead of $v(x\colon T)d$ if the self type of an object is not important. If a self variable is not used in the object we will denote it with an underscore: $v(\_)d$.

DOT's operational semantics is presented in Figure 4 on Page 17. The reduction relation operates on terms whose free variables are bound in a value environment γ that maps variables to values. In DOT, variables and values are considered normal form, i.e. they are irreducible. In particular, objects $v(x\colon T)d$ are values, and the fields of an object are not evaluated until those fields are selected. DOT fields are thus similar to Scala's `lazy vals` which declare immutable, lazily evaluated values (however, `lazy vals` differ from DOT fields because DOT does not memoize its fields).[1]

2.1.1 Recursion Elimination. The recursion-elimination rule is of particular interest in the context of paths:

[1] Evaluating fields strictly would require DOT to introduce a field initialization order which would complicate the calculus. DOT is designed to be a small calculus that focuses on formalizing path-dependent types and it deliberately leaves initialization as an open question. For a DOT with constructors and strict fields, see Kabir and Lhoták [17].

$$\frac{\Gamma \vdash x : \mu\,(z : T)}{\Gamma \vdash x : T\,[x/z]} \qquad (\text{Rec-E}_{\text{DOT}})$$

An object in DOT can recursively refer back to itself through the self variable. For example, the a member of the following object evaluates to the object itself (see the Scala version on the right):

let $x = \nu(z)\,\{a = z\}$ **in** ...

```
val x = new { val a = this }
```

Since types in DOT and Scala are dependent, the type of an object can also refer to the object itself:

let $y = \nu(z)\,\{A = T\} \wedge \{a = \lambda(x : z.A)\,x\}$ **in** ...

```
val y = new { type A = T
              val a = (x: this.A) => x }
```

The type of the field a depends on the type of the object containing it. In DOT, this is expressed using a recursive type. The type of our example object is $\mu\,(z : \{A : T..T\} \wedge \{a : \forall(x : z.A)\,z.A\})$.

Given the let binding above, what should be the type of $y.a$? In the recursive type of y, the type of the field a is $\forall(x : z.A)\,z.A$. However, because the self variable z is in scope only inside the object itself, the type $\forall(x : z.A)\,z.A$ does not make sense outside the object and cannot be used to type $y.a$. In the field selection, however, we have a name other than z for the object itself, the name y. Therefore, we can open the recursive type by replacing the self variable z with the external name of the object y, giving y the type $\{A : T\,[y/z]\,..T\,[y/z]\} \wedge \{a : \forall(x : y.A)\,y.A\}$. This is achieved by the recursion elimination typing rule. Now the path $y.a$ can have the type $\forall(x : y.A)\,y.A$. Notice that recursion elimination is possible only when we have a variable such as y as an external name for an object.

Just as we need to apply recursion elimination to the type of y before we can type a field selection $y.a$, we must also do the same before we can use a type-member selection $y.A$ (specifically, to conclude that $T\,[y/z] <: y.A <: T\,[y/z]$). The recursion elimination is necessary because the type T could also refer to the self variable z, and thus may not make any sense outside of the object. Recursion elimination replaces occurrences of z in the type T with the external name y, so that the resulting type is valid even outside the object. When we add path-dependent types to the calculus, an important consideration will be recursion elimination on paths rather than just variables.

2.2 Path Limitations: A Minimal Example

Consider the following example DOT object in which a type member B refers to a type member A that is nested inside the definition of a field c:

let $x = \nu(z)\,\{c = \nu(\_)\,\{A = z.B\}\} \wedge \{B = z.c.A\}$ **in** ...

```
val x = new { z ⇒
    val c: {type A} = new { type A = z.B }
    type B = z.c.A }
```

In the example, to reference the field c, we must first select the field's enclosing object x through its self variable z. As a result, the path to A leads through $z.c$ which is a path of length two. Since DOT does not allow paths of length two, this definition of B cannot be expressed in DOT without flattening the program structure so that all fields and type members become global members of one top-level object.

In the introduction, we illustrated how one might attempt to express the above in DOT by decomposing the path of length two into dereferences of simple variables, which would either lead to invalid programs or require flattening the program structure. We could try other ways of let-binding the inner objects to variables before defining the enclosing object, but all such attempts limit our ability to structure our program using nesting and recursion. A sequence of let bindings imposes a total ordering on the objects and restricts an object to refer only to objects that are

defined before it. In the presence of recursive references between the objects, as in this example, no valid ordering of the let bindings is possible while maintaining a nested object structure. To avoid this we could also try to transform the local variables into recursively defined fields of another object z', since the order in which fields are declared does not matter. However, then A would again need to refer to B through $z'.x.B$ (or $z'.y.B$) which has a path of length 2.

2.3 Challenges of Adding Paths to DOT

If restricting path-dependent types exclusively to variables limits the expressivity of DOT then why does the calculus impose such a constraint? Before we explain the soundness issue that makes it difficult to extend DOT with paths we must first review the key challenge that makes it difficult to ensure soundness of the DOT calculus.

2.3.1 Bad Bounds. Scala's abstract type members make it possible to define custom subtyping relationships between types. This is a powerful but tricky feature. For example, given any types S and U, consider the function $\lambda(x\colon \{A\colon S..U\})\, t$. In the body of the function, we can use $x.A$ as a placeholder for some type that is a supertype of S and a subtype of U. Some concrete type will be bound to $x.A$ when the function is eventually called with some specific argument. Due to transitivity of subtyping, the constraints on $x.A$ additionally introduce an assumption inside the function body that $S <: U$, because $S <: x.A <: U$ according to the type rules <:-Sel $_{\text{DOT}}$ and Sel-<: $_{\text{DOT}}$:

$$\frac{\Gamma \vdash x\colon \{A\colon S..T\}}{\Gamma \vdash S <: x.A} \quad (<:\text{-Sel}_{\,DOT}) \qquad \frac{\Gamma \vdash x\colon \{A\colon S..T\}}{\Gamma \vdash x.A <: T} \quad (\text{Sel-}<:_{\,DOT})$$

However, recall that S and U are arbitrary types, possibly with no existing subtyping relationship. The key to soundness is that although the function body is type-checked under the possibly unsound assumption $S <: U$, the body executes only when the function is called, and calling the function requires an argument that specifies a concrete type T to be bound to $x.A$. This argument type must satisfy the constraints $S <: T <: U$. Thus, the argument type embodies a form of *evidence* that the assumption $S <: U$ which is used to type-check the function body is actually valid.

More generally, given a term t of type $\{A\colon S..U\}$, we can rule out the possibility of bad bounds caused by the use of a dependent type $t.A$ if there exists some object with the same type $\{A\colon S..U\}$. This is because the object must bind the type member A to some concrete type T respecting the subtyping constraints $S <: T$ and $T <: U$, so the object is evidence that $S <: U$.

Existing DOT calculi ensure that whenever some variable x of type T is in scope in some term t, the term reduces only after x has already been assigned a value. The value assigned to x is evidence that T does not have bad bounds. To ensure that any code that uses the type $x.A$ executes only after x has been bound to a value of a compatible type, DOT employs a strict operational semantics. A variable x can be introduced by one of the three binding constructs: **let** $x = t$ **in** u, $\lambda(x\colon T)\, t$, or $\nu(x\colon T)d$. In the first case, x is in scope within u, and the reduction semantics requires that before u can execute, t must first reduce to a value with the same type as x. In the second case, x is in scope within t, which cannot execute until an argument value is provided for the parameter x. In the third case, the object itself is bound to the self variable x. In summary, the semantics ensures that by the time that evaluation reaches a context with x in scope, x is bound to a value, and therefore x's type does not introduce bad bounds. The issue of bad bounds has been discussed thoroughly in many of the previous papers about DOT [1, 2, 4, 24].

2.3.2 Naive Path Extension Leads to Bad Bounds. When we extend the type system with types $p.A$ that depend on paths rather than variables, we must take similar precautions to control bad bounds.

If a path p has type $\{A\colon S..U\}$ and some normal form n also has this type, then n must be an object that binds to type member A a type T such that $S <: T <: U$.

However, not all syntactic paths in DOT have this property. For example, in an object $v(x)\ \{a = t\}$, where t can be an arbitrary term, t could loop instead of reducing to a normal form of the same type. In that case, there is no guarantee that a value of the type exists, and it would be unsound to allow the path $x.a$ as the prefix of a path-dependent type $x.a.A$.

The following example, in which a function $x.b$ is typed as a record with field c, demonstrates this unsoundness (the Scala version cannot be typechecked):

$$
\begin{aligned}
&v(x:\\
&\quad \{a\colon \{C\colon (\forall(y\colon \top)\,\top)..\{c\colon \top\}\}\} \wedge \{b\colon \{c\colon \top\}\})\\
&\quad \{a = x.a\} \qquad\qquad\qquad\qquad \wedge \{b = \lambda(y\colon \top)\,y\}
\end{aligned}
$$

```
new {
  lazy val a: {type C >: Any ⇒ Any
                     <: {val c: Any}} = a
  lazy val b: {val c: Any} = (y: Any) ⇒ y }
```

Here, $x.b$ refers to a function $\lambda(y\colon \top)\,y$ of type $\forall(y\colon \top)\,\top$. If we allowed such a definition, the following would hold: $\forall(y\colon \top)\,\top <: x.a.C <: \{c\colon \top\}$. Then by subsumption, $x.b$, a function, has type $\{c\colon \top\}$ and therefore it must be an object. To avoid this unsoundness, we have to rule out the type selection $x.a.C$ on the non-terminating path $x.a$.

In general, if a path p has a field declaration type $\{a\colon T\}$, then the extended path $p.a$ has type T, but we do not know whether there exists a value of type T because $p.a$ has not yet reduced to a variable. Therefore, the type T could have bad bounds, and we should not allow the path $p.a$ to be used in a path-dependent type $p.a.A$.

The main difficulty we encountered in designing pDOT was to ensure that type selections occur only on terminating paths while ensuring that the calculus still permits non-terminating paths in general, since that is necessary to express recursive functions and maintain Turing completeness of the calculus.

3 MAIN IDEAS

In this Section, we outline the main ideas that have shaped our definition of pDOT. The pDOT calculus that implements these ideas in full detail is presented in Section 4.

3.1 Paths Instead of Variables

To support fully path-dependent types, our calculus needs to support paths in all places where DOT permitted variables. Consider the following example:

$$\textbf{let}\ x = v(y)\,\{a = v(z)\,\{B = U\}\}\ \textbf{in}\ x.a$$

```
val x = new { val a = new { type B = U }}; x.a
```

In order to make use of the fact that $U <: x.a.B <: U$, we need a type rule that reasons about path-dependent types. In DOT, this is done through the Sel-<: $_{DOT}$ and <:-Sel $_{DOT}$ rules mentioned in Section 2.3.1. Since we need to select B on a path $x.a$ and not just on a variable x, we need to extend the rules (merged into one here for brevity) to support paths:

$$\frac{\Gamma \vdash x\colon \{A\colon S..T\}}{\Gamma \vdash S <: x.A <: T}\ (<:\text{-Sel-}<:_{DOT}) \qquad \Rightarrow \qquad \frac{\Gamma \vdash p\colon \{A\colon S..T\}}{\Gamma \vdash S <: p.A <: T}\ (<:\text{-Sel-}<:)$$

However, before we can use this rule we need to also generalize the recursion elimination rule Rec-E$_{DOT}$, shown below. In the above example, how do we obtain the typing $\Gamma \vdash x.a\colon \{B\colon U..U\}$? The only identifier of the inner object is $x.a$, a path. The type of the path is $\mu(z\colon \{B\colon U..U\})$. In order to use the type member B, it is necessary to specialize this recursive type, replacing the recursive self variable z with the path $x.a$. This is necessary because the type U might refer to

the self variable z, which is not in scope outside the recursive type. Thus, in order to support path-dependent types, it is necessary to allow recursion elimination on objects identified by paths:

$$\frac{\Gamma \vdash x : \mu\,(y : T)}{\Gamma \vdash x : T\,[x/y]}\ (\textsc{Rec-E}_{DOT}) \qquad \Rightarrow \qquad \frac{\Gamma \vdash p : \mu\,(y : T)}{\Gamma \vdash p : T\,[p/y]}\ (\textsc{Rec-E})$$

By similar reasoning, we need to generalize all DOT variable-typing rules to path-typing rules. As we show later, we also have to generalize DOT's ANF syntax to use paths wherever DOT uses variables, so all the DOT reduction rules that operate on variables are generalized to paths in pDOT.

3.2 Paths as Identifiers

A key design decision of pDOT is to let *paths represent object identity*. In DOT, object identity is represented by variables, which works out because variables are irreducible. In pDOT, *paths* are *irreducible*, because reducing paths would strip objects of their identity and break preservation.

3.2.1 Variables are Identifiers in DOT. In the DOT calculus by Amin et al. [1], variables do not reduce to values for two reasons:

- *type safety*: making variables irreducible is necessary to maintain preservation, and
- *object identity*: to access the members of objects (which can recursively reference the object itself), objects need to have a name; reducing variables would strip objects of their identity.

If variables in DOT reduced to values, then in the previous example program, x would reduce to $v = \nu(y)\,\{a = \nu(z)\,\{B = U\}\}$. To maintain type preservation, for any type T such that $\Gamma \vdash x : T$, we also must be able to derive $\Gamma \vdash v : T$. Since $\Gamma \vdash x : \mu\,(y : \{a : \mu\,(z : \{B : U..U\})\})$, by recursion elimination $\textsc{Rec-E}_{DOT}$, $\Gamma \vdash x : \{a : \mu\,(z : \{B : U\,[x/y]..U\,[x/y]\})\}$. Does v also have that type? No!

$$\dfrac{\dfrac{\Gamma \vdash x : \mu\,(y : \{a : \mu\,(z : \{B : U..U\})\})}{\Gamma \vdash x : \{a : \mu\,(z : \{B : U\,[x/y]..U\,[x/y]\})\}}\ \textsc{Rec-E}_{DOT} \qquad \gamma : \Gamma \qquad \dfrac{\gamma(x) = v}{\gamma \mid x \longmapsto \gamma \mid v}\ \textsc{Hypothetical Var}_{DOT}}{\Gamma \vdash v : \{a : \mu\,(z : \{B : U\,[x/y]..U\,[x/y]\})\}}\ \textsc{preservation}_{DOT}$$

The value v has only the *recursive* type $\mu\,(y : \{a : \mu\,(z : \{B : U..U\})\})$. Since v is no longer connected to any specific name, no recursion elimination is possible on its type. In particular, it does not make sense to give this value the type $\{a : \mu\,(z : \{B : U\,[x/y]..U\,[x/y]\})\}$ because this type refers to x, but after the reduction, the value is no longer associated with this name.

The example illustrates that in DOT, variables represent the identity of objects. This is necessary in order to access an object's members: object members can reference the object itself, for which the object needs to have a name.

3.2.2 Paths are Identifiers in pDOT. In pDOT, *paths* represent the identity of objects and therefore they must be irreducible. Similarly to DOT, reducing paths would lead to unsoundness and strip nested objects of their identity. Making paths irreducible means that in pDOT, we cannot have an analog of DOT's field-selection reduction rule $\textsc{Proj}_{DOT}$.

Consider the field selection $x.a$ from the previous example. What is its type? By recursion elimination, $x.a$ has the type $\{B : U\,[x.a/z]..U\,[x.a/z]\}$. If pDOT had a path-reduction rule Proj analogous to DOT's $\textsc{Proj}_{DOT}$, then $x.a$ would reduce to $\nu(z)\{B = U\}$. However, that value does not have the type $\{B : U\,[x.a/z]..U\,[x.a/z]\}$; it only has the recursive type $\mu\,(z : \{B : U..U\})$.

$$\dfrac{\dfrac{\Gamma \vdash x.a : \mu\,(z : \{B : U..U\})}{\Gamma \vdash x.a : \{B : U\,[x.a/z]..U\,[x.a/z]\}}\ \textsc{Rec-E} \qquad \gamma : \Gamma \qquad \dfrac{\gamma(x) = \nu(y)\,\{a = \nu(z)\,\{B = U\}\}}{\gamma \mid x.a \longmapsto \gamma \mid \nu(z)\,\{B = U\}}\ \textsc{Hypothetical Proj}}{\Gamma \vdash \nu(z)\,\{B = U\} : \{B : U\,[x.a/z]..U\,[x.a/z]\}}\ \textsc{preservation}$$

The reduction step from $x.a$ to $v(z)\{B = U\}$ caused the object to lose its name. Since the non-recursive type of the term depends on the name, the loss of the name also caused the term to lose its non-recursive type. This reduction step violates type preservation and type soundness.

3.2.3 Well-Typed Paths Don't Go Wrong. If pDOT programs can return paths without reducing them to values, could these paths be nonsensical? The type system ensures that they cannot. In particular, we ensure that if a path p has a type then p either identifies some value, and looking up p in the runtime configuration terminates, or p is a path that cyclically aliases other paths. Additionally, as we will see in Section 5.2.3, the pDOT safety proof ensures that if a path has a function or object type, then it can be looked up to a value; if p can *only* be typed with a singleton type (or $\top$), then the lookup will loop.

3.3 Path Replacement

We introduce a *path replacement* operation for types that contain paths which reference the same object. If a path q is assigned to a path p then q *aliases* p. In the tree example from Section 1, `t3` aliases `t2`, but `t1` does not alias `t2`, even though they identify syntactically equal objects.

If q is an alias of p we want to ensure that we can use q in the same way as p. For example, any term that has type $T \to p.A$ should also have the type $T \to q.A$, and vice versa. In pDOT, we achieve this by introducing a subtyping relationship between *equivalent types*: if p and q are aliases, and a type T can be obtained from type U by *replacing* instances of p in U with q then T and U are equivalent. For example, $T \to q.A$ can be obtained from $T \to p.A$ by replacing p with q, and these types are therefore equivalent. We will precisely define the replacement operation in Section 4.2.

3.4 Singleton Types

To keep track of path aliases in the type system we use *singleton types*.

Suppose that a pDOT program assigns the path q to p, and that a type T can be obtained from U by replacing an instance of p with q. How does the type system know that T and U are equivalent? We could try passing information about the whole program throughout the type checker. However, that would make reasoning about types depend on reasoning about values, which would make typechecking more complicated and less modular [24].

Instead, we ensure that the type system keeps track of path aliasing using singleton types, an existing Scala feature. A singleton type of a path p, denoted p.type, is a type that is inhabited *only* with the value that is represented by p. In the tree example on Page 4, to tell the type system that `t3` aliases `t2`, we ensure that `t3` has the singleton type `t2.type`. This information is used to allow subtyping between aliased paths, and to allow such paths to be typed with the same types, as we will see in Section 4.2.

In pDOT, singleton types are an essential feature that is necessary to encode fully path-dependent types. However, this makes pDOT also the first DOT formalization of Scala's singleton types. In Section 6, we show a pDOT encoding of an example that motivates this Scala feature.

3.5 Distinguishing Fields and Methods

Scala distinguishes between fields (`val`s, immutable fields that are strictly evaluated at the time of object initialization) and methods (`def`s, which are re-evaluated at each invocation). By contrast, DOT unifies the two in the concept of a term member. Since the distinction affects which paths are legal in Scala, we must make some similar distinction in pDOT. Consider the following Scala program:

```
val x = new {
    val a: { type A } = t_a
```

```
    def b: { type B } = t_b }
val y: x.a.A
val z: x.b.B
```

Scala allows path-dependent types only on *stable* paths [9]. A `val` can be a part of a stable path but a `def` cannot. Therefore, the type selection `x.a.A` is allowed but `x.b.B` is not.

DOT unifies the two concepts in one:

$$\textbf{let}\ x = \nu(x)\,\{a = t_a\} \wedge \{b = t_b\} \qquad \textbf{in}\ \ldots$$

However, this translation differs from Scala in the order of evaluation. Scala's fields, unlike DOT's, are fully evaluated to values when the object is constructed. Therefore, a more accurate translation of this example would be as follows:

$$\begin{array}{ll} \textbf{let}\ a' = t_a & \textbf{in} \\ \textbf{let}\ x = \nu(x)\,\{a = a'\} \wedge \{b = \lambda(\_).\, t_b\} & \textbf{in}\ \ldots \end{array}$$

This translation highlights the fact that although Scala can initialize `x.a` to an arbitrary term, that term will be already reduced to a value before evaluation reaches a context that contains `x`. The reason is that the constructor for `x` will strictly evaluate all of `x`'s `val` fields when `x` is created.

To model the fact that Scala field initializers are fully evaluated when the object is constructed, we require field initializers in pDOT to be values or paths, rather than arbitrary terms. We use the name *stable term* for a value or path.

This raises the question of how to model a Scala method such as `b`. A method can still be represented by making the delayed evaluation of the body explicit: instead of initializing the field `b` with the method body itself, we delay the body inside a lambda abstraction. The lambda abstraction, a value, can be assigned to the field `b`. The body of the lambda abstraction can be an arbitrary term; it is not evaluated during object construction, but later when the method is called and the lambda is applied to some dummy argument.

3.6 Precise Self Types

DOT allows powerful type abstractions, but it demands *objects* as proof that the type abstractions make sense. An object assigns actual types to its type members and thus provides concrete evidence for the declared subtyping relationships between abstract type members. To make the connection between the object value and the type system, DOT requires the self type in an object literal to precisely describe the concrete types assigned to type members, and we need to define similar requirements for self types in pDOT.

In the object $\nu(x\colon \{A\colon T..T\})\,\{A = T\}$, DOT requires the self-type to be $\{A\colon T..T\}$ rather than some wider type $\{A\colon S..U\}$. This is not merely a convenience, but it is essential for soundness. Without the requirement, DOT could create and type the object $\nu(x\colon \{A\colon \top..\bot\})\,\{A = T\}$, which introduces the subtyping relationship $\top <: \bot$ and thus makes every type a subtype of every other type. Although we can require the actual assigned type T to respect the bounds (i.e. $\top <: T <: \bot$), such a condition is not sufficient to prohibit this object. The assigned type T and the bounds ($\top$ and $\bot$ in this example) can in general depend on the self variable, and thus the condition makes sense only in a typing context that contains the self variable with its declared self type. But in such a context, we already have the assumption that $\top <: x.A <: \bot$, so it holds that $\top <: T$ (since $\top <: x.A <: \bot <: T$) and similarly $T <: \bot$.

In pDOT, a path-dependent type $p.A$ can refer to type members not only at the top level, but also deep inside the object. Accordingly, we need to extend the precise self type requirement to apply recursively within the object, as follows:

(1) An object containing a type member definition $\{A = T\}$ must declare A with tight bounds, using $\{A\colon T..T\}$ in its self type.
(2) An object containing a definition $\{a = \nu(x\colon T)d\}$ must declare a with the recursive type $\mu\,(x\colon T)$, using $\{a\colon \mu\,(x\colon T)\}$ in its self type.
(3) An object containing a definition $\{a = \lambda(x\colon T)\,U\}$ must declare a with a function type, using $\{a\colon \forall(x\colon S)\,V\}$ in its self type.
(4) An object containing a definition $\{a = p\}$ must declare a with the singleton type $p.\mathsf{type}$, using $\{a\colon p.\mathsf{type}\}$ in its self type.

The first requirement is the same as in DOT. The second and third requirements are needed for soundness of paths that select type members from deep within an object. The fourth requirement is needed to prevent unsoundness in the case of cyclic references. For example, if we were to allow the object $\nu(x\colon \{a\colon \{A\colon \top..\bot\}\})\,\{a = x.a\}$ we would again have $\top <: \bot$. The fourth requirement forces this object to be declared with a precise self type: $\nu(x\colon \{a\colon x.a.\mathsf{type}\})\,\{a = x.a\}$. Now, $x.a$ no longer has the type $\{A\colon \top..\bot\}$, so it no longer collapses the subtyping relation. The precise typing thus ensures that cyclic paths can be only typed with singleton types but not function or object types, and therefore we cannot have type or term selection on cyclic paths.

Although both DOT and pDOT require precision in the self type of an object, the object itself can be typed with a wider type once it is assigned to a variable. For example, in DOT, if we have

$$\textbf{let}\ x = \nu(x\colon \{A\colon T..T\})\,\{A = T\}\ \textbf{in}\ \ldots$$

then x also has the wider type $\{A\colon \bot..\top\}$. Similarly, in pDOT, if we have

$$\textbf{let}\ x = \nu(x\colon \{a\colon \mu\,(y\colon \{b\colon \forall(z\colon T)\,U\}) \wedge \{c\colon x.a.b.\mathsf{type}\}\})d\ \textbf{in}\ \ldots$$

then x also has all of the following types: $\{a\colon \{b\colon \forall(z\colon T)\,U\}\}$, $\{a\colon \mu\,(y\colon \{b\colon \top\})\}$, $\{c\colon x.a.b.\mathsf{type}\}$, and $\{c\colon \forall(z\colon T)\,U\}$. In fact, the typings for this object in pDOT are more expressive than in DOT. Because DOT does not open types nested inside of field declarations, DOT cannot assign the first two types to x. In Section 4.2, we show one simple type rule that generalizes pDOT to open and abstract types of term members nested deeply inside an object. In Section 6, we encode several examples from previous DOT papers in pDOT and show that the real-world compiler example from Section 1 that uses types depending on long paths can be encoded in pDOT as well.

In summary, both DOT and pDOT require the self type in an object literal to precisely describe the values in the literal, but this does not limit the ability to ascribe a more abstract type to the paths that identify the object.

4 FROM DOT TO PDOT

The pDOT calculus generalizes DOT by allowing paths wherever DOT allows variables (except in places where variables are used as binders, such as x in $\lambda(x\colon T)\,t$).

4.1 Syntax

Figure 1 shows the abstract syntax of pDOT which is based on the DOT calculus of Amin et al. [1]. Differences from that calculus are indicated by shading.

The key construct in pDOT is a path, defined to be a variable followed by zero or more field selections (e.g. $x.a.b.c$). pDOT uses paths wherever DOT uses variables. In particular, field selections $x.a$ and function application $x\,y$ are done on paths: $p.a$ and $p\,q$. Most importantly, pDOT also generalizes DOT's types by allowing path-dependent types $p.A$ on paths rather than just on variables. Additionally, as described in Section 3.4, the pDOT calculus formalizes Scala's singleton types. A singleton type $p.\mathsf{type}$ is inhabited with only one value: the value that is assigned to the path p. A singleton type thus indicates that a term designates the same object as the path p. Just as

$x,\ y,\ z$		**Variable**
$a,\ b,\ c$		**Term member**
$A,\ B,\ C$		**Type member**
$p,\ q,\ r$	$:=$	**Path**
	x	variable
	$p.a$	field selection
$t,\ u$	$:=$	**Term**
	s	stable term
	$p\ q$	application
	let $x = t$ **in** u	let binding

s	$:=$	**Stable Term**
	p	path
	v	value
v	$:=$	**Value**
	$\nu(x: T)d$	object
	$\lambda(x: T)\ t$	lambda
d	$:=$	**Definition**
	$\{a = s\}$	field definition
	$\{A = T\}$	type definition
	$d \wedge d'$	aggregate definition

$S,\ T,\ U,\ V$	$:=$	**Type**
	$\top$	top type
	$\bot$	bottom type
	$\{a: T\}$	field declaration
	$\{A: S..T\}$	type declaration
	$S \wedge T$	intersection
	$\mu(x: T)$	recursive type
	$\forall(x: S)\ T$	depend. function
	$p.A$	type projection
	$p.\text{type}$	singleton type

Fig. 1. Abstract syntax of pDOT

a path-dependent type $p.A$ depends on the value of p, a singleton type $q.\text{type}$ depends on the value of q. Singleton types are therefore a second form of dependent types in the calculus.

4.2 pDOT Typing Rules

The typing and subtyping rules of pDOT are shown in Figures 2 and 3. The type system is based on the DOT of Amin et al. [1], and all changes are highlighted in gray.

4.2.1 From Variables to Paths. The first thing to notice in the pDOT typing and subtyping rules is that all variable-specific rules, except Var, are generalized to paths, as motivated in Section 3.1. The key rules that make DOT and pDOT interesting are the type-selection rules <:-Sel and Sel-<:. These rules enable us to make use of the type member in a path-dependent type. When a path p has type $\{A: S..U\}$, the rules introduce the path-dependent type $p.A$ into the subtyping relation by declaring the subtyping constraints $S <: p.A$ and $p.A <: U$. Thanks to these two rules, pDOT supports fully path-dependent types.

4.2.2 Object Typing. Similarly to the DOT calculus, the {}-I rule gives an object $\nu(x: T)d$ with declared type T which may depend on the self variable x the recursive type $\mu(x: T)$. The rule also checks that the definitions d of the object actually do have type T under the assumption that the self variable has this type. The object's definitions d are checked by the Definition typing rules. As discussed in Section 3.6, the rules assign a precise self type for objects, ensuring that paths are declared with singleton types, functions with function types, and objects with object types. For objects, the tight T condition ensures that all type members that can be reached by traversing T's fields have equal bounds, while still allowing arbitrary bounds in function types.

A difference with DOT is that pDOT's definition-typing judgment keeps track of the path that represents an object's identity. When we typecheck an outermost object that is not nested in any other object, we use the {}-I rule. The rule introduces x as the identity for the object and registers this fact in the definition-typing judgment. To typecheck an object that is assigned to a field a of another object p we use the Def-New rule. This rule typechecks the object's body assuming the object identity $p.a$ and replaces the self-variable of the object with that path. The definition-typing judgment keeps track of the path to the definition's enclosing object starting from the root of the

$$\Gamma ::= \varnothing \mid \Gamma,\ x\colon T \qquad \text{Type environment}$$

Term typing

$$\frac{\Gamma(x) = T}{\Gamma \vdash x\colon T}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash p\colon \{a\colon T\}}{\Gamma \vdash p.a\colon T}\ (\textsc{Fld-E}) \qquad \frac{\Gamma \vdash p\colon q.\mathsf{type} \qquad \Gamma \vdash q.a}{\Gamma \vdash p.a\colon q.a.\mathsf{type}}\ (\textsc{Sngl-E})$$

$$\frac{\Gamma,\ x\colon T \vdash t\colon U \qquad x \notin \mathrm{fv}(T)}{\Gamma \vdash \lambda(x\colon T)\,t\colon \forall(x\colon T)\,U}\ (\textsc{All-I}) \qquad \frac{\Gamma \vdash p.a\colon T}{\Gamma \vdash p\colon \{a\colon T\}}\ (\textsc{Fld-I}) \qquad \frac{\Gamma \vdash p\colon T\,[p/x]}{\Gamma \vdash p\colon \mu(x\colon T)}\ (\textsc{Rec-I})$$

$$\frac{\Gamma \vdash p\colon \forall(z\colon S)\,T \qquad \Gamma \vdash q\colon S}{\Gamma \vdash p\,q\colon T\,[q/z]}\ (\textsc{All-E}) \qquad \frac{\Gamma \vdash t\colon T \qquad \Gamma,\ x\colon T \vdash u\colon U \qquad x \notin \mathrm{fv}(U)}{\Gamma \vdash \mathbf{let}\ x = t\ \mathbf{in}\ u\colon U}\ (\textsc{Let}) \qquad \frac{\Gamma \vdash p\colon \mu(x\colon T)}{\Gamma \vdash p\colon T\,[p/x]}\ (\textsc{Rec-E})$$

$$\frac{\Gamma \vdash p\colon T \qquad \Gamma \vdash p\colon U}{\Gamma \vdash p\colon T \wedge U}\ (\&\text{-I}) \qquad \frac{x;\ \Gamma,\ x\colon T \vdash d\colon T}{\Gamma \vdash \nu(x\colon T)d\colon \mu(x\colon T)}\ (\{\}\text{-I}) \qquad \frac{\Gamma \vdash p\colon q.\mathsf{type} \qquad \Gamma \vdash q\colon T}{\Gamma \vdash p\colon T}\ (\textsc{Sngl-Trans})$$

$$\frac{\Gamma \vdash t\colon T \qquad \Gamma \vdash T <\colon U}{\Gamma \vdash t\colon U}\ (\textsc{Sub})$$

Definition typing

$$p;\ \Gamma \vdash \{A = T\}\colon \{A\colon T..T\}\ (\textsc{Def-Typ}) \qquad \frac{\Gamma \vdash q}{p;\ \Gamma \vdash \{a = q\}\colon \{a\colon q.\mathsf{type}\}}\ (\textsc{Def-Path})$$

$$\frac{\Gamma \vdash \lambda(x\colon T)\,t\colon \forall(x\colon U)\,V}{p;\ \Gamma \vdash \{a = \lambda(x\colon T)\,t\}\colon \{a\colon \forall(x\colon U)\,V\}}\ (\textsc{Def-All})$$

$$\frac{p.a;\ \Gamma \vdash d\,[p.a/y]\colon T\,[p.a/y] \qquad \mathsf{tight}\ T}{p;\ \Gamma \vdash \{a = \nu(y\colon T)d\}\colon \{a\colon \mu(y\colon T)\}}\ (\textsc{Def-New}) \qquad \frac{p;\ \Gamma \vdash d_1\colon T_1 \qquad p;\ \Gamma \vdash d_1\colon T_2 \qquad \mathrm{dom}(d_1),\ \mathrm{dom}(d_2)\ \text{disjoint}}{p;\ \Gamma \vdash d_1 \wedge d_2\colon T_1 \wedge T_2}\ (\textsc{AndDef-I})$$

Typeable paths

$$\frac{\Gamma \vdash p\colon T}{\Gamma \vdash p}\ (\textsc{Wf})$$

Tight bounds

$$\mathsf{tight}\ T = \begin{cases} U = V & \text{if } T = \{A\colon U..V\} \\ \mathsf{tight}\ U & \text{if } T = \mu(x\colon U) \text{ or } \{a\colon U\} \\ \mathsf{tight}\ U \text{ and } \mathsf{tight}\ V & \text{if } T = U \wedge V \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Fig. 2. pDOT typing rules

program. This way the type system knows what identities to assign to nested objects. For example, when typechecking the object assigned to $x.a$ in the expression

$$\mathbf{let}\ x = \nu(x)\,\{a = \nu(y)\,\{b = y.b\}\}\ \mathbf{in}\ \ldots$$

we need to replace y with the path $x.a$:

$$\dfrac{\dfrac{\Gamma,\ x\colon \{a\colon \mu(y\colon \{b\colon y.b.\mathsf{type}\})\} \vdash x.a}{x.a;\ \Gamma,\ x\colon \{a\colon \mu(y\colon \{b\colon y.b.\mathsf{type}\})\} \vdash \{b = x.a.b\}\colon \{b\colon x.a.b.\mathsf{type}\}}\ \textsc{Def-Path} \qquad \mathsf{tight}\ \{b\colon y.b.\mathsf{type}\}}{x;\ \Gamma,\ x\colon \{a\colon \mu(y\colon \{b\colon y.b.\mathsf{type}\})\} \vdash \{a = \nu(y)\,\{b = y.b\}\}\colon \{a\colon \mu(y\colon \{b\colon y.b.\mathsf{type}\})\}}\ \textsc{Def-New}$$
$$\Gamma \vdash \nu(x)\,\{a = \nu(y)\,\{b = y.b\}\} : \mu(x\colon \{a\colon \mu(y\colon \{b\colon y.b.\mathsf{type}\})\})\ \{\}\text{-I}$$

An alternative design of the Def-New rule can be to introduce a fresh variable y into the context (similarly to the {}-I rule). However, we would have to assign y the type $x.a.\mathsf{type}$ to register the

$$\Gamma \vdash T <: \top \quad (\textsc{Top}) \qquad \Gamma \vdash \bot <: T \quad (\textsc{Bot}) \qquad \Gamma \vdash T <: T \quad (\textsc{Refl})$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \ (\textsc{Trans}) \qquad \Gamma \vdash T \wedge U <: T \ (\textsc{And}_1\text{-}<:) \qquad \Gamma \vdash T \wedge U <: U \ (\textsc{And}_2\text{-}<:)$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \ (<:\text{-}\textsc{And}) \qquad \frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a\colon T\} <: \{a\colon U\}} \ (\textsc{Fld}\text{-}<:\text{-}\textsc{Fld})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A\colon S_1..T_1\} <: \{A\colon S_2..T_2\}} \ (\textsc{Typ}\text{-}<:\text{-}\textsc{Typ}) \qquad \frac{\Gamma \vdash p\colon \{A\colon S..T\}}{\Gamma \vdash S <: p.A} \ (<:\text{-}\textsc{Sel})$$

$$\frac{\Gamma \vdash p\colon q.\text{type} \qquad \Gamma \vdash q}{\Gamma \vdash T <: T\,[q/p]} \ (\textsc{Sngl}_{pq}\text{-}<:) \qquad \frac{\Gamma \vdash p\colon q.\text{type} \qquad \Gamma \vdash q}{\Gamma \vdash T <: T\,[p/q]} \ (\textsc{Sngl}_{qp}\text{-}<:)$$

$$\frac{\Gamma \vdash p\colon \{A\colon S..T\}}{\Gamma \vdash p.A <: T} \ (\textsc{Sel}\text{-}<:) \qquad \frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma,\, x\colon S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x\colon S_1)\, T_1 <: \forall(x\colon S_2)\, T_2} \ (\textsc{All}\text{-}<:\text{-}\textsc{All})$$

Fig. 3. pDOT subtyping rules

fact that these two paths identify the same object. We decided to simplify the rule by immediately replacing the nested object's self variable with the outer path to avoid the indirection of an additional singleton type.

4.2.3 Path Alias Typing. In pDOT singleton-type related typing and subtyping rules are responsible for the handling of aliased paths and equivalent types.

Singleton Type Creation. How does a path p obtain a singleton type? A singleton type indicates that in the initial program, a prefix of p (which could be all of p) is assigned a path q. For example, in the program

$$\textbf{let}\ x = v(x\colon\ \{a\colon x.\text{type}\} \wedge \{b\colon S\})\,\{a = x\} \wedge \{b = s\}\ \textbf{in}\ \ldots$$

the path $x.a$ should have the type $x.$type because $x.a$ is assigned the path x. The singleton type for $x.a$ can be obtained as follows. Suppose that in the typing context of the **let** body, x is mapped to the type of its object, $\mu\,(x\colon\ \{a\colon x.\text{type}\} \wedge \{b\colon S\})$. Through applying recursion elimination (Rec-E), field selection (Fld-E), and finally subsumption (Sub) with the intersection subtyping rule $\textsc{And}_1$-<:, we will obtain that $\Gamma \vdash x.a\colon x.$type.

In the above example, $x.a$ aliases x, so anything that we can do with x we should be able to do with $x.a$. Since x has a field b and we can create a path $x.b$, we want to be also able to create a path $x.a.b$. Moreover, we want to treat $x.a.b$ as an alias for $x.b$. This is done through the Sngl-E rule: it says that if p aliases q, and $q.a$ has a type (denoted with $\Gamma \vdash q.a$), then $p.a$ aliases $q.a$. This rule allows us to conclude that $\Gamma \vdash x.a.b\colon x.b.$type.

Singleton Type Propagation. In the above example we established that the path $x.a.b$ is an alias for $x.b$. Therefore, we want to be able to type $x.a.b$ with any type with which we can type $x.b$. The Sngl-Trans rule allows us to do just that: if p is an alias for q, then we can type p with any type with which we can type q. Using that rule, we can establish that $\Gamma \vdash x.a.b\colon S$ because $\Gamma \vdash x.b\colon S$.

Equivalent Types. As described in Section 3.3, we call two types equivalent if they are equal up to path aliases. We need to ensure that equivalent types are equivalent by subtyping, i.e. that they

are subtypes of each other. For example, suppose that $\Gamma \vdash p\colon q.\text{type}$, and the path r refers to an object $v(x)$ $\{a = p\} \wedge \{b = p\}$. Then we want to be able to type r with all of the following types:

$$\Gamma \vdash r\colon \{a\colon p.\text{type}\} \wedge \{b\colon p.\text{type}\} \qquad \Gamma \vdash r\colon \{a\colon p.\text{type}\} \wedge \{b\colon q.\text{type}\}$$
$$\Gamma \vdash r\colon \{a\colon q.\text{type}\} \wedge \{b\colon q.\text{type}\} \qquad \Gamma \vdash r\colon \{a\colon q.\text{type}\} \wedge \{b\colon p.\text{type}\}$$

The pDOT subtyping rules $\textsc{Sngl}_{pq}$-<: and $\textsc{Sngl}_{qp}$-<: allow us to assign these types to r by establishing subtyping between equivalent types. Specifically, if we know that $\Gamma \vdash p\colon q.\text{type}$ then the rules allow us to replace any occurrence of p in a type T with q, and vice versa, while maintaining subtyping relationships in both directions.

We express that two types are equivalent using the *replacement operation*. The operation is similar to the substitution operation, except that we replace paths with paths instead of variables with terms, and we replace only one path at a time rather than all of its occurrences. The statement $T\,[q/p] = U$ denotes that the type T contains one or more paths that start with p, e.g. $p.\overline{b_1}, \ldots, p.\overline{b_n}$, and that exactly one of these occurrences $p.\overline{b_i}$ is replaced with $q.\overline{b_i}$, yielding the type U. Note that it is not specified exactly in which occurrence of the above paths the prefix p is replaced with q. The precise definition of the replacement operation is presented in the accompanying technical report [25].

Given the path r from the above example, we can choose whether to replace the first or second occurrence of p with q; for example, we can derive

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash r\colon \{a\colon p.\text{type}\} \wedge \{b\colon p.\text{type}\}}\,\textsc{Rec-E} \qquad \dfrac{\Gamma \vdash p\colon q.\text{type} \quad \dfrac{\ldots}{\{a\colon p.\text{type}\} \wedge \{b\colon p.\text{type}\}\,[q/p] = \{a\colon p.\text{type}\} \wedge \{b\colon q.\text{type}\}}\,\textsc{Repl-And}_2}{\Gamma \vdash \{a\colon p.\text{type}\} \wedge \{b\colon p.\text{type}\} <: \{a\colon p.\text{type}\} \wedge \{b\colon q.\text{type}\}}\,\textsc{Sngl}_{pq}\text{-<:}}{\Gamma \vdash r\colon \{a\colon p.\text{type}\} \wedge \{b\colon q.\text{type}\}}\,\textsc{Sub}$$

To replace several occurrences of a path with another, we repeatedly apply $\textsc{Sngl}_{pq}$-<: or $\textsc{Sngl}_{qp}$-<:.

4.2.4 Abstracting Over Field Types. Finally, we describe one of the most interesting pDOT rules which adds significant expressivity to pDOT.

Consider a function $f = \lambda(x\colon \{a\colon T\})\ \ldots$ and a path p that refers to the object $v(x\colon \{a\colon q.\text{type}\})$ $\{a = q\}$, where $\Gamma \vdash q\colon T$. Since $\Gamma \vdash p\colon \mu(x\colon \{a\colon q.\text{type}\})$, by Rec-E, $\Gamma \vdash p\colon \{a\colon q.\text{type}\}$, assuming that q does not start with x. Therefore, since $\Gamma \vdash q\colon T$, we would like to be able to pass p into the function f which expects an argument of type $\{a\colon T\}$. Unfortunately, the typing rules so far do not allow us to do that because although q has type T, $q.\text{type}$ is *not* a subtype of T, and therefore $\{a\colon q.\text{type}\}$ is not a subtype of $\{a\colon T\}$.

The type rule Fld-I allows us to bypass that limitation. If a path p has a record type $\{a\colon T\}$ (and therefore $\Gamma \vdash p.a\colon T$), then the rule lets us type p with any type $\{a\colon U\}$ as long as $p.a$ can be typed with U.

For the above example, we can prove that $\Gamma \vdash p\colon \{a\colon T\}$ and pass it into f as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash p\colon \{a\colon q.\text{type}\}}{\Gamma \vdash p.a\colon q.\text{type}}\,\textsc{Fld-E} \qquad \Gamma \vdash q\colon T}{\Gamma \vdash p.a\colon T}\,\textsc{Sngl-Trans}}{\Gamma \vdash p\colon \{a\colon T\}}\,\textsc{Fld-I}$$

The Fld-I rule allows us to eliminate recursion on types that are nested inside fields, which is not possible in DOT. If a DOT function f expects a parameter of type $\{a\colon \mu(x\colon T)\}$, then in DOT, we cannot pass a variable y of type $\{a\colon \mu(x\colon T \wedge U)\}$ or a variable z of type $\{a\colon T\,[z.a/x]\}$ into f because there is no subtyping between recursive types, and there is no subtyping relationship between $\mu(x\colon T)$ and $T\,[z.a/x]$ (and the latter type might not exist in the first place due to the lack

$$\frac{\gamma(x) = v(x\colon T)\ldots\{a = t\}\ldots}{\gamma \mid x.a \longmapsto \gamma \mid t}\ (\textsc{Proj}_{\textsc{dot}})$$

$$\gamma ::= \varnothing \mid \gamma,\, x \mapsto v \qquad \textbf{Store}$$

$$\frac{\gamma(x) = \lambda(z\colon T)\, t}{\gamma \mid x\, y \longmapsto \gamma \mid t\,[y/z]}\ (\textsc{Apply}_{\textsc{dot}})$$

$$\frac{\gamma \vdash p \rightsquigarrow^* \lambda(z\colon T)\, t}{\gamma \mid p\, q \longmapsto \gamma \mid t\,[q/z]}\ (\textsc{Apply})$$

$$\gamma \mid \textbf{let}\ x = y\ \textbf{in}\ t \longmapsto \gamma \mid t\,[y/x]\ (\textsc{Let-Var}_{\textsc{dot}})$$

$$\gamma \mid \textbf{let}\ x = p\ \textbf{in}\ t \longmapsto \gamma \mid t\,[p/x]\ (\textsc{Let-Path})$$

$$\frac{x \notin \mathrm{dom}(\gamma)}{\gamma \mid \textbf{let}\ x = v\ \textbf{in}\ t \longmapsto \gamma,\, x \mapsto v \mid t}\ (\textsc{Let-Value}_{\textsc{dot}})$$

$$\frac{x \notin \mathrm{dom}(\gamma)}{\gamma \mid \textbf{let}\ x = v\ \textbf{in}\ t \longmapsto \gamma,\, x \mapsto v \mid t}\ (\textsc{Let-Value})$$

$$\frac{\gamma \mid t \longmapsto \gamma' \mid t'}{\gamma \mid \textbf{let}\ x = t\ \textbf{in}\ u \longmapsto \gamma' \mid \textbf{let}\ x = t'\ \textbf{in}\ u}\ (\textsc{Ctx}_{\textsc{dot}})$$

$$\frac{\gamma \mid t \longmapsto \gamma' \mid t'}{\gamma \mid \textbf{let}\ x = t\ \textbf{in}\ u \longmapsto \gamma' \mid \textbf{let}\ x = t'\ \textbf{in}\ u}\ (\textsc{Ctx})$$

Fig. 4. Operational semantics of DOT and pDOT

$$\frac{\gamma(x) = v}{\gamma \vdash x \rightsquigarrow v}\ (\textsc{Lookup-Step-Var})$$

$$\frac{\gamma \vdash p \rightsquigarrow v(x\colon T)\ldots\{a = s\}\ldots}{\gamma \vdash p.a \rightsquigarrow s\,[p/x]}\ (\textsc{Lookup-Step-Val})$$

$$\frac{\gamma \vdash p \rightsquigarrow q}{\gamma \vdash p.a \rightsquigarrow q.a}\ (\textsc{Lookup-Step-Path})$$

$$\gamma \vdash s \rightsquigarrow^* s\ (\textsc{Lookup-Refl}) \qquad \frac{\gamma \vdash s_1 \rightsquigarrow s_2 \qquad \gamma \vdash s_2 \rightsquigarrow^* s_3}{\gamma \vdash s_1 \rightsquigarrow^* s_3}\ (\textsc{Lookup-Trans})$$

Fig. 5. Value-environment path lookup

of fully path-dependent types). All of the above is possible in pDOT because both $y.a$ and $z.a$ can be typed with $\mu\,(x\colon T)$, which allows us to use the Fld-I rule and type y and z as $\{a\colon \mu\,(x\colon T)\}$.

4.3 Reduction Semantics

The operational semantics of pDOT is presented in Figure 4. pDOT's reduction rules mirror the DOT rules with three distinctions:

- *paths everywhere*: wherever DOT uses variables, pDOT uses paths;
- *no* $\textsc{Proj}_{\textsc{dot}}$: there is no reduction rule for field projection because in pDOT, paths are normal form (as motivated in Section 3.2.2);
- *path lookup*: pDOT uses the reflexive, transitive closure of the *path lookup* operation $\rightsquigarrow$ that generalizes variable lookup in value environments to paths.

The path lookup operation is presented in Figure 5. This operation allows us to look up a value that is nested deeply inside an object. If a path is a variable the lookup operation is a straightforward variable lookup (Lookup-Step-Var). If in a value environment γ, a path p is assigned

an object $v(x)\,\{a = s\}$ then $\gamma \vdash p.a \rightsquigarrow s\,[p/x]$ because the self variable x in s gets replaced with p (Lookup-Step-Val). If p is equal to another path q then $\gamma \vdash p.a \rightsquigarrow q.a$ (Lookup-Step-Path).

Finally, we want to be able to follow a sequence of paths in a value environment: for example, if $\gamma \vdash p \rightsquigarrow q$ and $\gamma \vdash q \rightsquigarrow v$, we want to conclude that looking up p yields v. This is done through the reflexive, transitive closure $\rightsquigarrow^*$ of the $\rightsquigarrow$ relation (Lookup-Refl and Lookup-Trans).

For example, looking up $x.a.c$ in the environment $\gamma = (y,\ v(y')\{b = v(y'')\{c = \lambda(z\colon \top)\,z\}\}),\ (x,\ v(x)\{a = y.b\})$ yields $\lambda(z\colon \top)\,z$:

$$\begin{array}{llll}
\gamma(x) = & v(x)\{a = & y.b\} & \gamma(y) = \quad v(y')\{b = v(y'')\{c = \lambda(z\colon \top)\,z\}\} \\
\gamma \vdash x \rightsquigarrow & v(x)\{a = & y.b\} & \gamma \vdash y \rightsquigarrow v(y')\{b = v(y'')\{c = \lambda(z\colon \top)\,z\}\} \\
\gamma \vdash x. & a \rightsquigarrow & y.b & \gamma \vdash y. \quad b \rightsquigarrow v(y'')\{c = \lambda(z\colon \top)\,z\} \\
\gamma \vdash x. & a.c \rightsquigarrow & y.b.c & \gamma \vdash y. \quad b. \quad c \rightsquigarrow \lambda(z\colon \top)\,z
\end{array}$$

$$\gamma \vdash x.a.c \rightsquigarrow^* \lambda(z\colon \top)\,z$$

The reduction rule that uses the lookup operation is the function application rule Apply: to apply p to q we must be able to look up p in the value environment and obtain a function. Since pDOT permits cycles in paths, does this mean that the lookup operation for this type rule might not terminate? Fortunately, pDOT's type safety ensures that this will not happen. As shown in Section 5.2.3, if $\Gamma \vdash p\colon \forall(T\colon U)$ then lookup of p eventually terminates and results in a function value. Therefore, a well-typed function application $p\,q$ always makes progress.

5 TYPE SAFETY

We implemented the type-safety proof of pDOT in Coq as an extension of the simple DOT soundness proof by Rapoport et al. [24]. Compared to the 2,051 LOC, 124 lemmas and theorems, and 65 inductive or function definitions in the simple DOT proof, the pDOT Coq formalization consists of 7,343 LOC, 429 lemmas and theorems, and 115 inductive or function definitions. Our paper comes with an artifact that presents the Coq formalization. A correspondence between the presentation of pDOT and the proof in the paper, and the Coq mechanization is presented in the accompanying technical report [25]. This section presents an overview of the key challenges and insights of proving pDOT sound. More details on the proof can be also found in the technical report.

Type safety ensures that any well-typed pDOT program does not get stuck, i.e. it either diverges or reduces to a normal form (a path or a value):

Theorem 5.1 (Type Soundness). *If $\vdash t\colon T$ then either t diverges, i.e. there exists an infinite reduction sequence $\varnothing \mid t \longmapsto \gamma_1 \mid t_1 \longmapsto \ldots \longmapsto \gamma_n \mid t_n \longmapsto \ldots$ starting with t, or t reduces to a normal form s, i.e. $\varnothing \mid t \longmapsto^* \gamma \mid s$, and $\Gamma \vdash s\colon T$ for some Γ such that $\gamma\colon \Gamma$.*

Since evaluating pDOT programs can result in paths (which are normal form), one might ask whether looking up those paths yields anything meaningful. As mentioned in Section 3.2.3, looking up any well-typed path in the runtime environment results either in a value or an infinite loop. To formulate the final soundness theorem that reasons about both term reduction and path lookup we define the following extended reduction relation $\twoheadrightarrow$:

$$\frac{\gamma \mid t \longmapsto \gamma' \mid t'}{\gamma \mid t \twoheadrightarrow \gamma' \mid t'} \qquad \frac{\gamma \vdash s \rightsquigarrow \gamma' s'}{\gamma \mid s \twoheadrightarrow \gamma' \mid s'}$$

We denote the reflexive, transitive closure of extended reduction as $\twoheadrightarrow^*$. Finally, we state the following extended soundness theorem:

Theorem 5.2 (Extended Type Soundness). *If* $\vdash t : T$ *then either* t *diverges, i.e. there exists an infinite reduction sequence* $\varnothing \mid t \twoheadrightarrow \gamma_1 \mid t_1 \twoheadrightarrow \ldots \twoheadrightarrow \gamma_n \mid t_n \twoheadrightarrow \ldots$ *starting with* t*, or* t *reduces to a value, i.e.* $\varnothing \mid t \twoheadrightarrow^* \gamma \mid v$.

Our proof follows the syntax-based approach to type soundness by Wright and Felleisen [32]. The two central lemmas of the proof are Progress and Preservation:

Lemma 5.3 (Progress). *Let* γ *be a value environment and* Γ *a typing environment. If i)* $\gamma : \Gamma$ *(i.e. if* $\gamma = \overline{(x_i, v_i)}$ *then* $\Gamma = \overline{(x_i, T_i)}$ *and* $\Gamma \vdash v_i : T_i$*), ii)* Γ *is* inert *(i.e. all types in* Γ *are the precise types of values, see Section 5.2.2), iii)* Γ *is* well-formed *(i.e. all paths in the types of* Γ *are typeable in* Γ*, see Section 5.2.1), and iv)* $\Gamma \vdash t : T$*, then* t *is in normal form or there exists a term* t' *and a value environment* γ' *such that* $\gamma \mid t \longmapsto \gamma' \mid t'$.

Lemma 5.4 (Preservation). *Let* γ *be a value environment and* Γ *a typing environment. If i)* $\gamma : \Gamma$*, ii)* Γ *is inert, iii)* Γ *is well-formed, iv)* $\Gamma \vdash t : T$*, and v)* $\gamma \mid t \longmapsto \gamma' \mid t'$*, then there exists an inert, well-formed typing environment* Γ' *such that* $\gamma' : \Gamma'$ *and* $\Gamma' \vdash t' : T$.

The pDOT proof follows the design principles laid out by Rapoport et al. [24] of separating the reasoning about types, variables (paths), and values from each other to ensure modularity and facilitate future extensions of pDOT.

5.1 Main Ideas of the DOT Safety Proof

The DOT type-safety proof addresses two main challenges:

1) *Rule out bad bounds:* Although bad bounds give rise to DOT typing contexts in which undesirable subtyping relationships hold, the proof needs to show that all reachable run-time states can be typed in well-behaved contexts.
2) *Induction on typing:* The DOT typing rules are very flexible, mirroring intuitive notions about which types a term ought to have. This flexibility requires rules that are opposites of each other and thus admit cycles in a derivation. The possibility of cycles impedes inductive reasoning about typing derivations.

The existing type safety proof defines a notion of *inert* types and typing contexts, a simple syntactic property of types that rules out bad bounds. Specifically, an inert type must be either a dependent function type or a recursive object type in which type members have equal bounds (i.e. $\{A : T..T\}$ rather than $\{A : S..U\}$). Crucially, the preservation lemma shows that reduction preserves inertness: that is, when $\gamma \mid t \longmapsto \gamma' \mid t'$ there is an *inert* typing environment Γ' that corresponds to γ' and in which t' has the required type.

The proof also employs the *proof recipe*, a stratification of the typing rules into multiple typing relations that rule out cycles in a typing derivation, but are provably as expressive as the general typing relation under the condition of an inert typing context. In particular, besides the general typing relation, the proof uses three intermediate relations: *tight* typing neutralizes the <:-Sel and Sel-<: rules that could introduce bad bounds, *invertible* typing contains introduction rules that create more complex types out of smaller ones, and *precise* typing contains elimination rules that decompose a type into its constituents.

5.2 Type Safety: From DOT to pDOT

The challenges of adapting the DOT soundness proof to pDOT can be classified into three main themes: adapting the notion of inert types to pDOT, adapting the stratification of typing rules to pDOT, and adapting the canonical forms lemma to changes in the operational semantics in pDOT.

$$\text{inert } \forall(x\colon T)\,U \qquad \frac{\text{record } T}{\text{inert } \mu\,(x\colon T)} \qquad \text{record } \{A\colon T..T\} \qquad \text{record } \{a\colon q.\text{type}\} \qquad \frac{\text{inert } T}{\text{record } \{a\colon T\}}$$

Fig. 6. Inert Types in pDOT

5.2.1 Inert Types in pDOT. The purpose of inertness is to prevent the introduction of a possibly undesirable subtyping relationship between arbitrary types $S <: U$ arising from the existence of a type member that has those types as bounds. If a variable x has type $\{A\colon S..U\}$, then $S <: x.A$ and $x.A <: U$, so by transitivity, $S <: U$.

A DOT type is *inert* if it is a function type or a recursive type $\mu\,(x\colon T)$ where T is a *record type*. A record type is either a type-member declaration with equal bounds $\{A\colon U..U\}$ or an arbitrary field declaration $\{a\colon S\}$. In DOT, this is sufficient to rule out the introduction of new subtyping relationships.

In pDOT, a new subtyping relationship $S <: U$ arises when a *path* p, rather than only a variable x, has a type member $\{A\colon S..U\}$. Therefore, the inertness condition needs to enforce equal bounds on type members not only at the top level of an object, but recursively in any objects nested deeply within the outermost object. Therefore, as shown in the inertness definition in Figure 6, a field declaration $\{a\colon T\}$ is inert only if the field type T is also inert. Moreover, since pDOT adds singleton types to DOT, the definition of a record type is also extended to allow a field to have a singleton type.

Both the DOT and pDOT preservation lemmas must ensure that reduction preserves inertness of typing contexts, but pDOT also requires preservation of a second property, well-formedness of typing contexts. A pDOT type can depend on paths rather than only variables, and the type makes sense only if the paths within it make sense; more precisely, well-formedness requires that any path that appears in a type should itself also be typeable in the same typing context. Without this property, it would be possible for the typing rules to derive types for ill-formed paths, and there could be paths that have types but do not resolve to any value during program execution.

5.2.2 Stratifying Typing Rules in pDOT. The language features that pDOT adds to DOT also create new ways to introduce cycles in a typing derivation. The stratification of the typing rules needs to be extended to eliminate these new kinds of cycles.

The notion of aliased paths is inherently symmetric: if p and q are aliases for the same object, then any term with type $p.A$ also has type $q.A$ and vice versa. This is complicated further because the paths p and q can occur deeply inside some complex type, and whether a term has such a type should be independent of whether the type is expressed in terms of p or q. A further complicating factor is that a prefix of a path is also a path, which may itself be aliased. For example, if p is an alias of q and $q.a$ is an alias of r, then by transitivity, $p.a$ should also be an alias of r.

The pDOT proof eliminates cycles due to aliased paths by breaking the symmetry. When p and q are aliased, either $\Gamma \vdash p\colon q.\text{type}$ or $\Gamma \vdash q\colon p.\text{type}$. The typing rules carefully distinguish these two cases, so that for every pair p, q of aliased paths introduced by a typing declaration, we know whether the aliasing was introduced by the declaration of p or of q.[2] A key lemma then proves that if we have any sequence of aliasing relationships $p_0 \sim p_1 \sim \cdots \sim p_n$, where for each i, either $\Gamma \vdash p_i\colon p_{i+1}.\text{type}$ or $\Gamma \vdash p_{i+1}\colon p_i.\text{type}$, we can reorder the replacements so that the ones of the first type all come first and the ones of the second type all come afterwards. More precisely, we

[2] An exceptional case is when p is declared to have type $q.\text{type}$ and q is declared to have type $p.\text{type}$. Fortunately, this case of a cycle turns out to be harmless because neither path is declared to have any other type other than its singleton type, and therefore neither path can be used in any interesting way.

can always find some "middle" path q such that $\Gamma \vdash p_0 : q.\text{type}$ and $\Gamma \vdash p_n : q.\text{type}$.[3] Therefore, we further stratify the proof recipe into two typing judgments the first of which accounts for the SNGL_{pq}-<: rule, and the second for the SNGL_{qp}-<: rule. This eliminates cycles in a typing derivation due to aliased paths, but the replacement reordering lemma ensures that it preserves expressiveness.

Another new kind of cycle is introduced by the field elimination rule FLD-E and the field introduction rule FLD-I that is newly added in pDOT. This cycle can be resolved in the same way as other cycles in DOT, by stratifying these rules in two different typing relations.

The final stratification of the pDOT typing rules requires 7 typing relations rather than the 4 required in the soundness proof for DOT. General and tight typing serve the same purpose as in the DOT proof, but pDOT requires three elimination and two introduction typing relations.

5.2.3 Canonical Forms in pDOT. Like many type soundness proofs, the DOT proof depends on canonical forms lemmas that state that if a variable has a function type, then it resolves to a corresponding function at execution time, and if it has a recursive object type, then it resolves to a corresponding object. The change from DOT to pDOT involves several changes to Canonical Forms.

Two changes are implied directly by the changes to the operational semantics. The DOT canonical forms lemmas apply to variables. Since the pDOT APPLY reduction rule applies to paths rather than variables, the canonical forms lemma is needed for paths. Since paths are normal forms in pDOT and there is no PROJ reduction rule for them, on the surface, pDOT needs a canonical forms lemma only for function types but not for object types. However, to reason about a path with a function type, we need to reason about the prefixes of the path, which have an object type. Therefore, the induction hypothesis in the canonical forms lemma for function types must still include canonical forms for object types. Moreover, since pDOT adds singleton types to the type system, the induction hypothesis needs to account for them as well.

A more subtle but important change is that lookup of a path in an execution environment is a recursive operation in pDOT, and therefore its termination cannot be taken for granted. An infinite loop in path lookup would be a hidden violation of progress for function application, since the APPLY reduction rule steps only once path lookup has finished finding a value for the path. Therefore, the canonical forms lemma proves that if a path has a function type, then lookup of that path does terminate, and the value with which it terminates is a function of the required type. The intuitive argument for termination requires connecting the execution environment with the typing environment: if direct lookup of path p yields another path q, then the context assigns p the singleton type $q.\text{type}$. But in order for p to have a function type, there cannot be a cycle of paths in the typing context (because a cycle would limit p to have only singleton types), and therefore there cannot be a cycle in the execution environment. The statement of the canonical forms lemma is:

LEMMA 5.5. *Let γ be a value environment and Γ be an inert, well-formed environment such that $\gamma : \Gamma$. If $\Gamma \vdash p : \forall(x : T)\, U$ then there exists a type T' and a term t such that i) $\gamma \vdash p \rightsquigarrow^* \lambda(x : T')\, t$, ii) $\Gamma \vdash T <: T'$, and iii) $\Gamma,\, x : T \vdash t : U$.*

This simple statement hides an intricate induction hypothesis and a long, tedious proof, since it needs to reason precisely about function, object, and singleton types and across all seven typing relations in the stratification of typing.

6 EXAMPLES

In this section, we present three pDOT program examples that illustrate different features of the calculus. All of the programs were formalized and typechecked in Coq.

[3] In degenerate cases, the middle path q might actually be p_0 or p_n.

To make the examples easier to read, we simplify the notation for objects $\nu(x\colon U)d$ by removing type annotations where they can be easily inferred, yielding a new notation $\nu(x \Rightarrow d')$:

- a type definition $\{A = T\}$ can be only typed with $\{A\colon T..T\}$, so we will skip type declarations;
- in a definition $\{a = p\}$, the field a is assigned a path and can be only typed with a singleton type; we will therefore skip the type $\{a\colon p.\mathsf{type}\}$;
- in a definition $\{a = \nu(x\colon T)d\}$, a is assigned an object that must be typed with $\mu\,(x\colon T)$; since we can infer T by looking at the object definitions, we will skip the typing $\{a\colon \mu\,(x\colon T)\}$;
- we inline the type of abstractions into the field definition (e.g. $\{a\colon \forall(x\colon T)\,U = \lambda(x\colon T)\,t\}$).

For readability we will also remove the curly braces around object definitions and replace the $\wedge$ delimiters with semicolons. As an example for our abbreviations, the object

$$\begin{aligned}\nu(x\colon\ &\{A\colon T..T\} \wedge \{a\colon p.\mathsf{type}\} \wedge \{b\colon \mu\,(y\colon U)\} \wedge \{c\colon \forall(z\colon S)\,V\})\\ &\{A = T\} \wedge \{a = p\} \wedge \{b = \nu(y\colon U)d\} \wedge \{c = \lambda(z\colon S')\,t\}\end{aligned}$$

will be encoded as

$$\nu(x \Rightarrow\ A = T;\ a = p;\ b = \nu(y \Rightarrow d');c\colon \forall(z\colon S)\,V = \lambda(z\colon S')\,t)$$

where $\nu(y \Rightarrow d')$ is the abbreviated version of $\nu(y\colon U)d$.

6.1 Class Encodings

Fully path-dependent types allow pDOT to define encodings for Scala's module system and classes, as we will see in the examples below.

In Scala, declaring a `class A(args)` automatically defines both a type `A` for the class and a constructor for `A` with parameters `args`. We will encode such a Scala class in pDOT as a type member A and a method `newA` that returns an object of type A:

$\nu(p \Rightarrow$
$\quad A = \mu\,(\mathbf{this}\colon \{\mathtt{foo}\colon \forall(\_)\,U\})\,;$
$\quad \mathtt{newA}\colon \forall(x\colon U)\,p.A$
$\quad\quad = \nu(\mathbf{this})\,\{\mathtt{foo} = \lambda(\_).\,x\})$

```
package p {
    class A(x: U) {
        def foo: U = x
    }}
```

To encode subtyping we use type intersection. For example, we can define a class `B` that extends `A` as follows:

$\nu(p \Rightarrow$
$\quad A = \mu\,(\mathbf{this}\colon \{\mathtt{foo}\colon \forall(\_)\,U\})\,;$
$\quad \mathtt{newA}\colon \forall(x\colon U)\,p.A$
$\quad\quad = \nu(\mathbf{this})\,\{\mathtt{foo} = \lambda(\_).\,x\})$

```
package p {
    class A(x: U) {
        def foo: U = x
    }}
```

6.2 Lists

As an example to illustrate that pDOT supports the type abstractions of DOT we formalize the covariant-list library by Amin et al. [1] in pDOT, presented in Figure 7 a). The encoding defines `List` as a data type with an element type A and methods `head` and `tail`. The library contains `nil` and `cons` fields for creating lists. To soundly formalize the list example, we encode `head` and `tail` as *methods* (`defs`) as opposed to `vals` by wrapping them in lambda abstractions, as discussed in Section 3.5. This encoding also corresponds to the Scala standard library where `head` and `tail` are `defs` and not `vals`, and hence one cannot perform a type selection on them.

By contrast, the list example by Amin et al. [1] encodes `head` and `tail` as fields without wrapping their results in functions. For a DOT that supports paths, such an encoding is unsound because

it violates the property that paths to objects with type members are acyclic. In particular, since no methods should be invoked on `nil`, its `head` and `tail` methods are defined as non-terminating loops, and `nil`'s element type is instantiated to $\bot$. If we allowed `nil.head` to have type $\bot$ then since $\bot <: \{A: \top..\bot\}$, we could derive $\top <:$ `nil.head.A` $<: \bot$.

6.3 Mutually Recursive Modules

The second example, presented in Figure 7 b), illustrates pDOT's ability to use path-dependent types of arbitrary length. It formalizes the compiler example from Section 1 in which the nested classes `Type` and `Symbol` recursively reference each other.

6.4 Chaining methods with singleton types

The last example focuses on pDOT's ability to use singleton types as they are motivated by Odersky and Zenger [22]. An example from that paper introduces a class C with an `incr` method that increments a mutable integer field x and returns the object itself (`this`). A class D extends C and defines an analogous `decr` method. The example shows how we can invoke a chain of `incr` and `decr` methods on an object of type D using singleton types: if C.`incr` returned an object of type C this would be impossible since C does not have a `decr` member, so the method's return type is `this.type`, a singleton type.

Our formalization of the example is displayed in Figure 7 c). Since pDOT does not support mutation, our example excludes the mutation side effect of the original example which is there to make the example more practical.

7 RELATED WORK

This section reviews work related to formalizing Scala with support for fully path-dependent types.

7.1 Early Class-based Scala Formalizations

Several predecessors of the DOT calculus support path-dependent types on paths of arbitrary length. The first Scala formalization, νObj [21], is a nominal, class-based calculus with a rich set of language features that formalizes object-dependent type members. Two subsequent calculi, Featherweight Scala (FS_{alg}) [8] and Scalina [20], build on νObj to establish Scala formalizations with algorithmic typing and with full support for higher-kinded types. All three calculi support paths of arbitrary length, singleton types, and abstract type members. Whereas FS_{alg} supports type-member selection directly on paths, νObj and Scalina allow selection $T\#A$ on types. A path-dependent type $p.A$ can thus be encoded as a selection on a singleton type: $p.\text{type}\#A$. νObj is the only of the above calculi that comes with a type-safety proof. The proof is non-mechanized.

Both pDOT and these calculi prevent type selections on non-terminating paths. νObj achieves this through a *contraction* requirement that prevents a term on the right-hand side of a definition from referring to the definition's self variable. At the same time, recursive calls can be encoded in νObj by referring to the self variable from a nested class definition. FS_{alg} ensures that paths are normalizing through a cycle detection mechanism that ensures that a field selection can appear only once as part of a path. Scalina avoids type selection $T\#A$ on a non-terminating type T by explicitly requiring T to be of a *concrete* kind, which means that T expands to a structural type R that contains a type member A. Although Scalina allows A to have upper and lower bounds, bad bounds are avoided because A also needs to be immediately initialized with a type U that conforms to A's bounds, which is more restrictive than DOT. In pDOT, it is possible to create cyclic paths but impossible to do a type selection on them because as explained in Section 3.6, cyclic paths that can appear in a concrete execution context cannot be typed with a type-member declaration.

a) A covariant list library in pDOT

```
ν(sci ⇒ List = μ(self: {A: ⊥..⊤} ∧ {head: ∀(_) self.A} ∧ {tail: ∀(_) (sci.List ∧ {A: ⊥..self.A})});
        nil: ∀(x: {A: ⊥..⊤}) sci.List ∧ {A: ⊥..⊥}
          = λ(x: {A: ⊥..⊤}) let result = ν(self ⇒ A = ⊥;
                                                  head: ∀(y: ⊤) self.A = λ(y: ⊤) self.head y;
                                                  tail: ∀(y: ⊤) (sci.List ∧ self.A) = λ(y: ⊤) self.tail y)
                                in result;
        cons: ∀(x: {A: ⊥..⊤}) ∀(hd: x.A) ∀(tl: sci.List ∧ {A: ⊥..x.A}) sci.List ∧ {A: ⊥..x.A}
          = λ(x: {A: ⊥..⊤}) λ(hd: x.A) λ(tl: sci.List ∧ {A: ⊥..x.A})
              let result = ν(self ⇒ A = x.A;
                                    head: ∀(_) self.A = λ_. hd
                                    tail: ∀(_) (sci.List ∧ self.A) = λ_. tl)
              in result)
```

b) Mutually recursive types in a compiler package: fully path-dependent types

```
ν(dc ⇒ types = ν(types ⇒ Type = μ(this: {symb: dc.symbols.Symbol});
                          newType: ∀(s: dc.symbols.Symbol) types.Type
                                = λ(s: dc.symbols.Symbol)
                                    let result' = ν(this ⇒ symb = s) in result');
        symbols = ν(symbols ⇒ Symbol = μ(this: {tpe: dc.types.Type});
                          newSymbol: ∀(t: dc.types.Type) symbols.Symbol
                                = λ(t: dc.types.Type)
                                    let result' = ν(this ⇒ tpe = t) in result'))
```

c) Chaining method calls using singleton types

```
let pkg = ν(p ⇒ C = μ(this: {incr: this.type});
                D = μ(this: p.C ∧ {decr: this.type});
                newD: ∀(_) p.D = λ_.
                    let result = ν(this ⇒ incr = this; decr = this)
                    in result)
in let d = pkg.newD _
in d.incr.decr
```

Fig. 7. Example pDOT encodings

A difference between pDOT and the above calculi is that to ensure type soundness, paths in pDOT are normal form. This is necessary to ensure that each object has a name, as explained in Section 3.2.2. νObj and FS_{alg} achieve type safety in spite of reducing paths by allowing field selection only on variables. This way, field selections always occur on named objects. Scalina does not require objects to be tied to names. In particular, its field selection rule E_Sel allows a field selection $\mathsf{new}\, T.a$ on an object if T contains a field definition $\{a = s\}$. The selection reduces to $s\,[^{\mathsf{new}\, T}/_{\mathbf{this}}]$, i.e. each occurrence of the self variable is replaced with a copy of $\mathsf{new}\, T$.

A second difference to pDOT is the handling of singleton types. In order to reason about a singleton type p.type, νObj, FS_{alg}, and Scalina use several recursively defined judgments (membership, expansion, and others) that rely on analyzing the shape and well-formedness of the type that p expands to. By contrast, pDOT contains one simple Sngl-Trans rule that allows a path to inherit the type of its alias. On the other hand, pDOT has the shortcoming that singleton typing is not reflexive. Unlike in the above systems and in Scala, pDOT lacks a type axiom $\Gamma \vdash p : p.\text{type}$. Such a rule would undermine the anti-symmetry of path aliasing which is essential to the safety proof.

None of the other calculi have to confront the problem of bad bounds. Unlike DOT, pDOT, and Scala, νObj and FS_{alg} do not support lower bounds of type members and have no unique upper and lower bounds on types. Scalina does have top and bottom types and supports bounds through interval kinds, but it avoids bad bounds by requiring types on which selection occurs to be concrete. In addition, it is unknown whether Scalina and FS_{alg} are sound.

Finally, the three type systems are nominal and class based, and include a large set of language features that are present in Scala. DOT is a simpler and smaller calculus that abstracts over many of the design decisions of the above calculi. Since DOT aims to be a base for experimentation with new language features, it serves well as a minimal core calculus for languages with type members, and the goal of pDOT is to generalize DOT to fully path-dependent types.

7.2 DOT-like Calculi

Amin et al. [2] present the first version of a DOT calculus. It includes type intersection, recursive types, unique top and bottom types, type members with upper and lower bounds, and path-dependent types on paths of arbitrary length. This version of DOT has explicit support for fields (`vals`) and methods (`defs`). Fields must be initialized to variables, which prevents the creation of non-terminating paths (since that would require initializing fields to paths), but it also limits expressivity. Specifically, just like in DOT by Amin et al. [1], path-dependent types cannot refer to nested modules because modules have to be created through methods, and method invocations cannot be part of a path-dependent type. The calculus is not type-safe, as illustrated by multiple counterexamples in the paper. In particular, this version of DOT does not track path equality which, as explained in the paper, breaks preservation.

To be type-safe, DOT must ensure that path-dependent types are invoked only on terminating paths. A possible strategy to ensure a sound DOT with support for paths is to investigate the conditions under which terms terminate, and to impose these conditions on the paths that participate in type selections. To address these questions, Wang and Rompf [31] present a Coq-mechanized semantic proof of strong normalization for the $\text{D}_{<:}$ calculus. $\text{D}_{<:}$ is a generalization of System $\text{F}_{<:}$ with lower- and upper-bounded type tags and variable-dependent types. The paper shows that recursive objects constitute the feature that enables recursion and hence Turing-completeness in DOT. Since $\text{D}_{<:}$ lacks recursive objects, it is strongly normalizing. Furthermore, the lack of objects and fields implies that this version of $\text{D}_{<:}$ can only express paths that are variables.

Hong et al. [15] present πDOT, a strongly normalizing version of a $\text{D}_{<:}$ without top and bottom types but with support for paths of arbitrary length. πDOT keeps track of path aliasing through path-equivalence sets, and the paper also mentions the possibility of using singleton types to

formalize path equality. Like the calculus by Wang and Rompf [31], this version of $D_{<:}$ is strongly normalizing due to the lack of recursive self variables. This guarantees that paths are acyclic. It also ensures that due to the lack of recursion elimination, reducing paths preserves soundness (unlike in pDOT, as explained in Section 3.2.2). πDOT comes with a non-mechanized soundness proof.

By contrast with these two papers, our work proposes a Turing-complete generalization with paths of arbitrary length of the full DOT calculus, which includes recursive objects and type intersections.

7.3 Other Related Languages and Calculi

Scala's module system shares many similarities with languages of the ML family. Earlier presentations of ML module systems [10, 14, 18] allow fine-grained control over type abstractions and code reuse but do not support mutually recursive modules, and separate the language of terms from the language of modules. *MixML* extends the essential features of these type systems with the ability to do both *hierarchical* and *mixin* composition of modules [28]. The language supports *recursive* modules which can be packaged as *first-class values*. The expressive power of *MixML*'s module system, plus support for decidable type-checking requires a set of constraints on the *linking* (module mixin) operation that restrict recursion between modules, including a total order on *label paths*, and yields a complex type system that closely models actual implementations of ML.

Rossberg et al. [29] and Rossberg [27] address the inherent complexity of ML module systems by presenting encodings of an ML-style module language into System F_ω. The latter paper presents *1ML*, a concise version of ML that fully unifies the language of modules with the language of terms. However, both formalizations exclude recursive modules.

A type system that distinguishes types based on the runtime values of their enclosing objects was first introduced by Ernst [12] in the context of *family polymorphism*. Notably, family polymorphism is supported by *virtual classes*, which can be inherited and overriden within different objects and whose concrete implementation is resolved at runtime. Virtual classes are supported in the *Beta* and *gbeta* programming languages [11, 19] (but not in Scala in which classes are statically resolved at compile time) and formalized by the *vc* and *Tribe* calculi [7, 13]. Paths in *vc* are relative to `this` and consist of a sequence of `out` keywords, which refer to enclosing objects, and field names. To track path equality, *vc* uses a normalization function that converts paths to a canonical representation, and to rule out cyclic paths it defines a partial order on declared names. *Tribe*'s paths can be both relative or absolute: they can start with a variable, and they can intermix class and object references. The calculus uses singleton types to track path equality and rules out cyclic paths by disallowing cyclic dependencies in its inheritance relation.

A difference between pDOT and all of *vc*, *Tribe*, and the ML formalizations is that pDOT does not impose any orderings on paths, and fully supports recursive references between objects and path-dependent types. In addition, pDOT's ability to define type members with both lower and upper bounds introduces a complex source of unsoundness in the form of *bad bounds* (alas, the cost for its expressiveness is that pDOT's type system is likely not decidable, as discussed below). Yet, by being mostly structurally typed, without having to model initialization and inheritance, pDOT remains general and small. Finally, by contrast to the above, pDOT comes with a mechanized type-safety proof.

7.4 Decidability

The baseline DOT calculus to which we add the path extensions is widely conjectured to have undecidable typechecking because it includes the features of $F_{<:}$, for which typechecking is undecidable [23]. Rompf and Amin [26] give a mapping from $F_{<:}$ to $D_{<:}$, a simpler calculus than DOT, and prove that if the $F_{<:}$ term is typeable then so is the $D_{<:}$ term, but the only-if direction and

therefore the decidability of $D_{<:}$ and DOT remain open problems, subject of active research [16]. The open question of decidability of DOT needs to be resolved before we can consider decidability of pDOT.

We believe that pDOT does not introduce additional sources of undecidability into DOT. One feature of pDOT that might call this into question is singleton types. In particular, Stone and Harper [30] study systems of singleton kinds that reason about types with non-trivial reduction rules, yet it remains decidable which types reduce to the same normal form. The singleton types of both Scala and pDOT are much simpler and less expressive in that only assignment of an object between variables and paths is allowed, but the objects are not arbitrary terms and do not reduce. Thus, the Scala and pDOT singleton types only need to track sequences of assignments. Thus, although decidability of pDOT is unknown because it is unknown for DOT, the singleton types that we add in pDOT are unlikely to affect decidability because they are significantly less expressive than the singleton types studied by Stone and Harper.

8 CONCLUSION

The DOT calculus was designed as a small core calculus to model Scala's type system with a focus on path-dependent types. However, DOT can only model types that depend on variables, which significantly under-approximates the behaviour of Scala programs. Scala and, more generally, languages with type members need to rely on fully path-dependent types to encode the possible type dependencies in their module systems without restrictions. Until now, it was unclear whether combining the fundamental features of languages with path-dependent types, namely bounded abstract type members, intersections, recursive objects, and paths of arbitrary length is type-safe.

This paper proposes pDOT, a calculus that generalizes DOT with support for paths of arbitrary length. The main insights of pDOT are to represent object identity through paths, to ensure that well-typed acyclic paths always represent values, to track path equality with singleton types, and to eliminate type selections on cyclic paths through precise object typing. pDOT allows us to use the full potential of path-dependent types. pDOT comes with a type-safety proof and motivating examples for fully path-dependent types and singleton types that are mechanized in Coq.

ACKNOWLEDGMENTS

We would like to thank Martin Odersky for suggesting the idea of extending DOT with paths of arbitrary length, and for the helpful discussions on early variants of pDOT. We thank Jaemin Hong, Abel Nieto, and the anonymous reviewers for their careful reading of our paper and the insightful suggestions that greatly helped improve it. We thank Lu Wang and Yaoyu Zhao for their contributions to the Coq proof. We thank Paolo Giarrusso and Ifaz Kabir for their thoughtful proofreading, and the helpful discussions on the DOT calculus which improved our understanding of the the expressiveness and limitations of DOT. We had other helpful discussions on DOT with Zhong Sheng Hu, Sukyoung Ryu, Derek Dreyer, Ilya Sergey, and Prabhakar Ragde. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] 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*. 249–272.

[2] Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent Object Types. In *International Workshop on Foundations of Object-Oriented Languages (FOOL 2012)*.

[3] Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 666–679.

[4] Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 233–249.

[5] Nada Amin and Ross Tate. 2016. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 838–848.

[6] Kim B. Bruce, Martin Odersky, and Philip Wadler. 1998. A Statically Safe Alternative to Virtual Types. In *ECOOP'98 - Object-Oriented Programming, 12th European Conference*. 523–549.

[7] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*. 121–134.

[8] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A Core Calculus for Scala Type Checking. In *Mathematical Foundations of Computer Science, 31st International Symposium, Slovakia*.

[9] Scala Documentation. 2018. Paths. Retrieved February 26, 2019 from https://www.scala-lang.org/files/archive/spec/2.11/03-types.html#paths

[10] Derek Dreyer, Karl Crary, and Robert Harper. 2003. A type system for higher-order modules. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*. 236–249.

[11] Erik Ernst. 1999. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Ph.D. Dissertation. Department of Computer Science, University of Aarhus, Århus, Denmark.

[12] Erik Ernst. 2001. Family Polymorphism. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. 303–326.

[13] Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. 270–282.

[14] Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. 123–137.

[15] Jaemin Hong, Jihyeok Park, and Sukyoung Ryu. 2018. Path Dependent Types with Path-equality. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala (Scala 2018)*. ACM, 35–39.

[16] Jason Hu and Ondrej Lhoták. 2019. Undecidability of $D_{<:}$ and Its Decidable Fragments. *CoRR* abs/1908.05294 (2019). http://arxiv.org/abs/1908.05294

[17] Ifaz Kabir and Ondřej Lhoták. 2018. κDOT: scaling DOT with mutation and constructors. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. ACM, 40–50.

[18] Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. 109–122.

[19] Ole Lehrmann Madsen and Birger Møller-Pedersen. 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89), New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings*. 397–406.

[20] Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Safe type-level abstraction in Scala. In *International Workshop on Foundations of Object-Oriented Languages (FOOL 2008)*.

[21] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. 2003. A Nominal Theory of Objects with Dependent Types. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*. 201–224.

[22] Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. 41–57.

[23] Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*. 305–315.

[24] Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A simple soundness proof for dependent object types. *PACMPL* 1, OOPSLA (2017), 46:1–46:27.

[25] Marianna Rapoport and Ondrej Lhoták. 2019. A Path To DOT: Formalizing Fully Path-Dependent Types. *CoRR* abs/1904.07298 (2019). http://arxiv.org/abs/1904.07298

[26] Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 624–641.
[27] Andreas Rossberg. 2018. 1ML — Core and modules united. *Journal of Functional Programming* 28 (2018), e22.
[28] Andreas Rossberg and Derek Dreyer. 2013. Mixin' Up the ML Module System. *ACM Trans. Program. Lang. Syst.* 35, 1 (2013), 2:1–2:84.
[29] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *J. Funct. Program.* 24, 5 (2014), 529–607.
[30] Christopher A. Stone and Robert Harper. 2006. Extensional equivalence and singleton types. *ACM Trans. Comput. Log.* 7, 4 (2006), 676–722.
[31] Fei Wang and Tiark Rompf. 2017. Towards Strong Normalization for Dependent Object Types (DOT). In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 27:1–27:25.
[32] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94.