Type Soundness for Dependent Object Types (DOT)

Tiark Rompf * Nada Amin†

*Purdue University, USA: {firstname}@purdue.edu
†EPFL, Switzerland: {first.last}@epfl.ch

Abstract

Scala's type system unifies aspects of ML modules, object-oriented, and functional programming. The Dependent Object Types (DOT) family of calculi has been proposed as a new theoretic foundation for Scala and similar expressive languages. Unfortunately, type soundness has only been established for restricted subsets of DOT. In fact, it has been shown that important Scala features such as type refinement or a subtyping relation with lattice structure break at least one key metatheoretic property such as environment narrowing or invertible subtyping transitivity, which are usually required for a type soundness proof.

The main contribution of this paper is to demonstrate how, perhaps surprisingly, even though these properties are lost in their full generality, a rich DOT calculus that includes recursive type refinement and a subtyping lattice with intersection types can still be proved sound. The key insight is that subtyping transitivity only needs to be invertible in code paths *executed at runtime*, with contexts consisting entirely of valid runtime objects, whereas inconsistent subtyping contexts can be permitted for code that is never executed.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords dependent object types, DOT, Scala, soundness

1. Introduction

Modern expressive programming languages such as Scala integrate and generalize concepts from functional programming, object oriented programming and ML-style module systems [32]. While most of these features are understood on a theoretical level in isolation, their combination is not, and the gap between formal type theoretic models and what is implemented in realistic languages is large.

In the case of Scala, developing a sound formal model that captures a relevant subset of its type system has been an elusive quest for more than a decade. After many false starts, the first mechanized soundness proof for a calculus of the DOT (Dependent Object Types) [4] family was finally presented in 2014 [5].

The calculus proved sound by Amin et al. [5] is μDOT, a core calculus that distills the essence of Scala's objects that may contain *type members* in addition to methods and fields, along with *path-dependent types*, which are used to access such type members. μDOT models just these two features–records with type members and type selections on variables–and nothing else. This simple system already captures some essential programming patterns, and it has a clean and intuitive theory. In particular, it satisfies the intuitive and mutually dependent properties of environment narrowing and subtyping transitivity, which are usually key requirements for a soundness proof.

Alas, Amin et al. also showed that adding other important Scala features such as type refinement, mixin composition, or just a bottom type breaks at least one of these properties, which makes a soundness proof much harder to come by.

The main contribution of this paper is to demonstrate how, perhaps surprisingly, even though these properties are lost in their full generality, a richer DOT calculus that includes both recursive type refinement and a subtyping lattice with full intersection and union types can still be proved sound. The key insight is that proper subtyping transitivity only needs to hold for types assigned to *runtime objects*, but not for arbitrary code that is never executed.

The paper is structured around the main contributions as follows:

- We present the first sound calculus of the DOT family that includes recursive type refinement and a subtyping lattice with intersection and union types: first informally with examples (Section 2), then formally (Section 3).
- We discuss the key properties that make a soundness proof difficult (Section 4).
- We present our soundness result. First, we give a small-step operational semantics (Section 5). Then we explain the proof strategy and key lemmas in details (Section 6).
- We offer some perspective on how foundational type-theoretic work influences practice (Section 7).

We discuss related work in Section 8 and offer concluding remarks in Section 9. Our mechanized Coq proofs are available from `http://oopsla16.namin.net`.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...
http://dx.doi.org/10.1145/2983990.2984008

2. Types in Scala and DOT

Scala is a large language that combines features from functional programming, object-oriented programming and module systems. Scala unifies many of these features (e.g. objects, functions, and modules) [32] but still contains a large set of distinct kinds of types. These can be broadly classified [28] into *modular* types:

```
named type: scala.collection.BitSet
compound type: Channel with Logged
refined type: Channel { def close(): Unit }
```

And *functional* types :

```
parameterized type: List[String]
existential type: List[T] forSome { type T }
higher-kinded type: List
```

While this variety of types enables programming styles appealing to programmers with different backgrounds (Java, ML, Haskell, ...), not all of them are essential. Further unification and an economy of concepts can be achieved by reducing functional to modular types as follows:

```
class List[Elem] {} ⇝ class List { type Elem }
List[String] ⇝ List { type Elem = String }
List[T] forSome { type T } ⇝ List
List ⇝ List
```

This unification is the main thrust of the calculi of the DOT family. A further thrust is to replace Scala's compound types `A with B` with proper intersection types `A & B`. It is worth noting that the correspondence between higher-kinded types and existential types suggested above is not exact, in particular with respect to well-kindedness constraints and partial application to type arguments. Before presenting our DOT variant in a formal setting in Section 3, we introduce the main ideas with some high-level programming examples.

Objects and First Class Modules In Scala and in DOT, every piece of data is conceptually an object and every operation a method call. This is in contrast to functional languages in the ML family that are stratified into a core language and a module system. Below is an implementation of a functional list data structure:

```
val listModule = new { m =>
  type List = { this =>
    type Elem
    def head(): this.Elem
    def tail(): m.List & { type Elem <: this.Elem }
  }
  def nil() = new { this =>
    type Elem = Bot
    def head() = error()
    def tail() = error()
  }
  def cons[T](hd: T)(tl: m.List & { type Elem <: T }) =
    new { this =>
    type Elem = T
    def head() = hd
    def tail() = tl
  }}
```

The actual `List` type is defined inside a container `listModule`, which we can think of as a first-class module. In an extended DOT calculus `error` may signify an 'acceptable' runtime error or exception that aborts the current execution and transfers control to an exception handler. In the case that we study, without such facilities, we can model the abortive behavior of `error` as a non-terminating function, for example `def error(): Bot = error()`.

Nominality through Ascription In most other settings (e.g. object-oriented subclassing, ML module sealing), nominality is enforced when objects are declared or constructed. Here we can just assign a more abstract type to our list module:

```
type ListAPI = { m =>
  type List <: { this =>
    type Elem
    def head(): this.Elem
    def tail(): m.List & { type Elem <: this.Elem }
  }
  def nil(): List & { type Elem = Bot }
  def cons[T]: T =>
    m.List & { type Elem <: T } =>
      m.List & { type Elem <: T }
}
```

Types `List` and `Elem` are abstract, and thus exhibit nominal as opposed to structural behavior. Since modules are just objects, it is perfectly possible to pick different implementations of an abstract type based on runtime parameters.

```
val mapImpl = if (size < 100) ListMap else HashMap
```

Polymorphic Methods In the code above, we have still used the functional notation `cons[T](...)` for parametric methods. We can desugar the type parameter T into a proper method parameter x with a modular type, and at the same time desugar the multi-argument function into nested anonymous functions:

```
def cons(x: { type A }) = ((hd: x.A) => ... )
```

References to T are replaced by a path-dependent type $x.A$. We can further desugar the anonymous functions into objects with a single `apply` method:

```
def cons(x: { type A }) = new {
  def apply(hd: x.A) = new {
    def apply(tl: m.List & { type Elem <: x.A }) =
      new { this =>
      type Elem = x.A
      def head() = hd
      def tail() = tl
    }}}
```

More generally, with these desugarings, we can encode the polymorphic λ-calculus System F [20, 35] and its extension with subtyping, System $F_{<:}$ [9], by defining a pointwise mapping over types and terms:

$$X \rightsquigarrow \texttt{x.A}$$
$$\forall(X <: S)T \rightsquigarrow \texttt{\{ def apply(x: \{ type A <: S \}): T \}}$$
$$\Lambda(X <: S)t \rightsquigarrow \texttt{new \{ def apply(x: \{ type A <: S \}) = t \}}$$
$$t[U] \rightsquigarrow \texttt{t.apply(\{ type A = U \})}$$

Apart from the translation of type variables into term variables with type members, another difference is that $F_{<:}$ supports only upper bounded type variables, while Scala and DOT allow both lower and upper bounds. This gives rise to very fine grained choices of translucency, i.e. how much implementation details are revealed for a partially abstract type, and also allows a precise modelling of covariance and contravariance. We will see an example of a lower bounded type member `{ type Config >: T}` below.

It is easy to see that while path-dependent types can encode System F, the reverse is not likely true. For example, the following function is expressible in Scala and DOT but does not have a System F equivalent: $\lambda(x : \{ \texttt{type A} \}).x$

Path-Dependent Types Let us consider another example to illustrate path-dependent types: a system of services, each with a specific type of configuration object. Here is the abstract interface:

```
type Service {
  type Config
  def default: Config
  def init(data: Config): Unit
}
```

We now create a system consisting of a database and an authentication service, each with their respective configuration types:

```
type DBConfig { def getDB: String }
type AuthConfig { def getAuth: String }
val dbs = new Service {
  type Config = DBConfig
  def default = new DBConfig { ... }
  def init(d:Config) = ... d.getDB ...
}
val auths = new Service {
  type Config = AuthConfig
  def default = new AuthConfig { ... }
  def init(d:Config) = ... d.getAuth ...
}
```

We can initialize `dbs` with a new `DBConfig`, and `auths` with a new `AuthConfig`, but not vice versa. This is because each object has its own specific `Config` type member and thus, `dbs.Config` and `auths.Config` are distinct *path-dependent* types. Likewise, if we have a service `lam: Service` without further refinement of its `Config` member, we can still call `lam.init(lam.default)` but we cannot create a `lam.Config` value directly, because `Config` is an abstract type in `Service`.

Intersection and Union Types At the end of the day, we want only one centralized configuration for our system, and we can create one by assigning an intersection type:

```
val globalConf: DBConfig & AuthConfig = new {
  def getDB = "myDB"
  def getAuth = "myAuth"
}
```

Since `globalConf` corresponds to both `DBConfig` and `AuthConfig`, we can use it to initialize both services:

```
dbs.init(globalConf)
auths.init(globalConf)
```

But we would like to abstract even more.

With the `List` definition presented earlier, we can build a list of services (using `::` as syntactic sugar for `cons`):

```
val services = auths::dbs::nil()
```

We define an initialization function for a whole list of services:

```
def initAll[T](xs:List[Service { type Config >: T }])(c: T) =
  xs.foreach(_ init c)
```

Which we can then use as:

```
initAll(services)(globalConf)
```

How do the types play out here? The definition of `List` and `cons` makes the type member `Elem` covariant. Thus, the type of `auths::dbs::nil()` corresponds to

```
List & {
  type Elem = Service & {
    type Config: (DBConfig & AuthConfig) ..
                 (DBConfig | AuthConfig)
  }
}
```

The notation `type T: S..U` denotes that type member `T` is lower bounded by `S` and upper bounded by `U`. Here, `Config` is lower bounded by the greatest lower bound (the intersection, `&`) and upper bounded by the least upper bound (the union, `|`) of the types `DBConfig` and `AuthConfig`. Since the `Config` member is lower bounded by `DBConfig & AuthConfig`, passing an object of that type to `init` is legal.

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

```
type PersistentService = Service { a =>
  def persist(config: a.Config)
}
```

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

```
type PersistentService = { a => Service & {
  def persist(config: a.Config)
}}
```

Our variant of DOT uses intersection types also to model the type of records with multiple type, value, or method members:

```
{ def foo(x: S1): U1   ;   def bar(y: S2): U2 } ⇝
{ def foo(x: S1): U1 } & { def bar(y: S2): U2 }
```

With this encoding of records, we benefit again from an economy of concepts.

Soundness? Our aim for DOT is a strong type soundness result of the form "well-typed programs don't go wrong", i.e. to show the total absence of runtime errors. Clearly we cannot expect the same strong result to hold for Scala, as Scala includes unsafe features like `null` values and unsafe casts. Especially `null` values make a strong soundness result impossible as they violate the key *canonical forms* property: a value that has a given type might not actually be a proper object of this type at runtime, but it might instead be `null`.

A more realistic notion of soundness needs to include the possibility of *benign* runtime failures. A desirable guarantee in this setting is that modules that do not introduce `null` values cannot be *blamed* for `NullPointerException`s, and modules that do not perform unsafe casts cannot be blamed for `ClassCastException`s.

In this light, it is important to note that Scala has some fundamental soundness bugs related to path-dependent types, even with respect to these weaker notions of soundness. Some manifestations of these bugs have been known for a long time (the Scala issue tracker lists unfixed soundness problems dating back at least to 2008 [40]), but they have been properly understood only recently, during the course of this work. We discuss the general issue next, with a particular example related to `null` values [6].

The key soundness challenge with path-dependent types is to avoid "bad bounds", i.e. type members like `type T: S..U` where `S` is *not* a subtype of `U`, e.g. `type T: String..Int`. If an object `x` were allowed to have such a type member, we could use it to safely treat a `String` first as type `x.T`, and then as `Int`, but of course this would be unsound: any attempt to treat a `String` as an `Int` will lead to a cast exception at runtime!

The formalization effort behind DOT has shown that preventing such "bad bounds" is harder than previously thought (see Section 4.3), and that the mechanisms implemented in the Scala compiler are not sufficient. More precisely, it is necessary to enforce constraints on type members and their bounds at object creation time (see Section 4.5). The canonical forms property ensures that an object that is supposed to have a type member `T: S..U` was indeed created with type member `T` set to a type *inbetween* `S` and `U` – thus providing constructive evidence that `S <: U`. However, `null` is incompatible with canonical forms: `null` is assignable to any other type, but it does not have any members on which constraints could be enforced. Thus it is clear that we can create a problem with type selections `x.T` when `x` is `null`. The potential danger of this has been realized early on in the history of Scala, and the Scala compiler has ad-hoc checks to prevent direct uses of `null`. However, with the new observation that "good bounds" are not preserved by narrowing (see Section 4.3), it becomes easy to thwart those checks via indirection as shown in the following example, which fails with a runtime cast exception:

```
trait A { type L >: Any}
def id1(a: A, x: Any): a.L = x
val p: A { type L <: Nothing } = null
def id2(x: Any): Nothing = id1(p, x)
id2("boom!")
```

The problem is that `id2` is a "safe" cast from `Any` to `Nothing`, which we cannot support in a sound language. The subtyping lattice collapses through a `null` path with bad bounds. The issue is described in more detail by Amin and Tate [6], who show that – remarkably – the same problem also exists in Java, even though Java does not have path-dependent types.

Since this example uses an explicit `null`, one might be tempted to dismiss the issue by saying "`null` is unsound anyways". But what is disturbing is that the result is not a `NullPointerException` but a `ClassCastException`, even though no "unsafe" casts are used. Also, no pointer is dereferenced, so while a `NullPointerException` could be more easily rationalized (e.g. as a "type-level" null deferencing), it would still be unexpected, even for weak notions of soundness.

The use of `null` is also not crucial, as we show in Section 7.1. The issue is deeper, and the take-away is that paths always need to refer to proper objects, which are guaranteed to have "good bounds" by construction, and thus serve as constructive evidence, i.e. witnesses for the subtyping relations they introduce via transitivity. As we discuss in Section 4, attempting to incorporate "good bounds" into well-formedness of types is not sufficient, as "good bounds" are not preserved under narrowing. Because they do not provide any actual witness ensuring that the subtyping relations added are valid, `null` values, unevaluated `lazy val`s, mutable variables, as well as arbitrary (non-terminating) term dependencies need to be excluded from paths. The core DOT calculus presented in this paper does not include these features, but we discuss potential avenues of extension in Section 7.2.

Our work on DOT helped identify and clarify these soundness issues, and identify potential fixes. In the case of `null`, there are no easy fixes: the DOT effort has underlined the necessity for future versions of Scala and similar languages to consider nullability explicitly in the type system.

3. Formal Model of DOT

Figure 1 shows the syntax and static semantics of the DOT calculus we study. For readability, we omit well-formedness requirements from the rules, and assume all types to be syntactically well-formed in the given environment. We write T^x when x is free in T. The type assignment syntax is overloaded: we use $\Gamma \vdash t : T$ for the usual term type assignment and $\Gamma \vdash t :! T$ for a path type assignment used in subtyping of type selections, specifically rules (SEL1) and (SEL2). We use $\Gamma \vdash t :_{(!)} T$ for rules that are defined for both cases, specifically rules (VAR), (VARUNPACK), and (SUB).

Syntax

x, y, z	Variable
L	Type label
m	Method label
$\Gamma ::= \emptyset \mid \Gamma, x : T$	Context

$S, T, U ::=$	Type
$\top$	top type
$\bot$	bottom type
$L : S..U$	type member
$m(x : S) : U^x$	method member
$x.L$	type selection
$\{z \Rightarrow T^z\}$	recursive self type
$T \wedge T$	intersection type
$T \vee T$	union type

$t, u ::=$	Term
x	variable reference
$\{z \Rightarrow \bar{d}\}$	new instance
$t.m(t)$	method invocation
$d ::=$	Initialization
$L = T$	type initialization
$m(x) = t$	method initialization

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

Lattice structure

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

$$\frac{\Gamma \vdash T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{(And11)} \qquad \frac{\Gamma \vdash T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2} \text{(Or21)}$$

$$\frac{\Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{(And12)} \qquad \frac{\Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2} \text{(Or22)}$$

$$\frac{\Gamma \vdash T <: T_1 \ , \ T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2} \text{(And2)} \qquad \frac{\Gamma \vdash T_1 <: T \ , \ T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T} \text{(Or1)}$$

Type and method members

$$\frac{\Gamma \vdash S_2 <: S_1 \ , \ U_1 <: U_2}{\Gamma \vdash L : S_1..U_1 <: L : S_2..U_2} \text{(Typ)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash U_1^x <: U_2^x}{\Gamma \vdash m(x : S_1) : U_1^x <: m(x : S_2) : U_2^x} \text{(Fun)}$$

Path selections

$$\Gamma \vdash x.L <: x.L \quad \text{(SelX)}$$

$$\frac{\Gamma_{[x]} \vdash x :_! (L : T..\top)}{\Gamma \vdash T <: x.L} \text{(Sel2)} \qquad \frac{\Gamma_{[x]} \vdash x :_! (L : \bot..T)}{\Gamma \vdash x.L <: T} \text{(Sel1)}$$

Recursive self types

$$\frac{\Gamma, z : T_1^z \vdash T_1^z <: T_2^z}{\Gamma \vdash \{z \Rightarrow T_1^z\} <: \{z \Rightarrow T_2^z\}} \text{(BindX)}$$

$$\frac{\Gamma, z : T_1^z \vdash T_1^z <: T_2}{\Gamma \vdash \{z \Rightarrow T_1^z\} <: T_2} \text{(Bind1)}$$

Transitivity

$$\frac{\Gamma \vdash T_1 <: T_2 \ , \ T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \text{(Trans)}$$

Type assignment $\boxed{\Gamma \vdash t :_{(!)} T}$

Variables, self packing/unpacking

$$\frac{\Gamma(x) = T}{\Gamma \vdash x :_{(!)} T} \text{(Var)}$$

$$\frac{\Gamma \vdash x : T^x}{\Gamma \vdash x : \{z \Rightarrow T^z\}} \text{(VarPack)} \qquad \frac{\Gamma \vdash x :_{(!)} \{z \Rightarrow T^z\}}{\Gamma \vdash x :_{(!)} T^x} \text{(VarUnpack)}$$

Subsumption

$$\frac{\Gamma \vdash t :_{(!)} T_1 \ , \ T_1 <: T_2}{\Gamma \vdash t :_{(!)} T_2} \text{(Sub)}$$

Method invocation

$$\frac{\Gamma \vdash t : (m(x : T_1) : T_2^x) \ , \ y : T_1}{\Gamma \vdash t.m(y) : T_2^y} \text{(TAppVar)}$$

$$\frac{\Gamma \vdash t : (m(x : T_1) : T_2) \ , \ t_2 : T_1}{\Gamma \vdash t.m(t_2) : T_2} \text{(TApp)}$$

Object creation

$$\frac{\Gamma, x : T_1^x \wedge \ldots \wedge T_n^x \vdash d_i : T_i^x \quad \text{(labels disjoint)} \quad \forall i, 1 \leq i \leq n}{\Gamma \vdash \{x \Rightarrow d_1 \ldots d_n\} : \{z \Rightarrow T_1^z \wedge \ldots \wedge T_n^z\}} \text{(TNew)}$$

Member initialization $\boxed{\Gamma \vdash d : T}$

$$\frac{\Gamma \vdash T <: T}{\Gamma \vdash (L = T) : (L : T..T)} \text{(DTyp)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (m(x) = t) : (m(x : T_1) : T_2)} \text{(DFun)}$$

Figure 1. DOT Syntax and Type System

Compared to the original DOT proposal [4], which used several auxiliary judgements such as membership and expansion, the presentation here is vastly simplified, and uses only subtyping to access function and type members. Compared to μDOT [5], the calculus is much more expressive, and includes key features like intersection and union types, which are absent in μDOT.

The Scala syntax used above maps to the formal notation in a straightforward way:

$$\begin{aligned} \texttt{\{ type T = Elem \}} &\rightsquigarrow T : \texttt{Elem..Elem} \\ \texttt{\{ type T >: S <: U \}} &\rightsquigarrow T : S..U \\ \texttt{\{ def m(x: T) = t \}} &\rightsquigarrow m(x) = t \\ \texttt{A \& B,\ \ A | B} &\rightsquigarrow A \wedge B,\ \ A \vee B \end{aligned}$$

In addition, top-level definitions of `val`s and `type`s desugar to local definitions, by the usual desugaring of let-bindings into applications. Intersection and union types, along with the $\bot$ and $\top$ types, provide a full subtyping lattice.

Note that each kind of object member, i.e. a type member $L = T$ or a method member $m(x) = t$, includes its specific label L or m, respectively. The same holds for the corresponding types. A sensible alternative would be to decouple object membership and the underlying primitive types for methods (having dependent function types instead) and types (having "type tag" types instead that can be dereferenced as types). Because objects are recursive in their self type, we find coupling the labels more intuitive in terms of reasoning.

When typing object creation, rules (TNEW) and (DTYP) ensure syntactically that object values have "good bounds", avoiding soundness issues due to lattice collapsing (which we explain in Section 4.3). First, labels are disjoint. Second, type member initialization requires exact aliases, not bounds. Finally, to avoid circular reasoning, the checking of member initialization is done without subsumption on the self type.

In subtyping, members of the same label and kind can be compared via rules (TYP) and (FUN). The type member upper bound, and method result type are covariant while the type member lower bound and the method parameter type are contravariant – as is standard. In rule (FUN), we allow some dependence between the parameter type and the return type of a method, when the argument is a variable. This is another difference to previous versions of DOT [4, 5], which did not have such dependent method types.

If a variable x can be assigned a type member L, then the type selection $x.L$ is valid, and can be compared with any upper bound when it is on the left, and with any lower bound when it is on the right, via rules (SEL1) and (SEL2). Furthermore, a type selection is reflexively a subtype of itself via rule (SELX). This rule is explicit, so that even abstract type members can be compared to themselves.

Finally, recursive self types can be compared to each other via (BINDX). They can also be dropped if the self identifier is not used, via rule (BIND1). During type assignment, the rules for variable packing and unpacking, (VARPACK) and (VARUNPACK), serve as introduction and elimination rules, enabling recursive types to be compared to other types as well. Since types can be recursive, and since subtype comparisons may introduce temporary bindings that may need to be unpacked, there are two *contractiveness restrictions* on type selections that ensure well-founded induction in the proofs (Section 6.2). First, the typing assignment judgement used in type selections, $\Gamma \vdash x :! T$, forbids (VARPACK). Note that this typing is used to match an unpacked type member type, anyways. Second, type selections restrict the environment to disregard bindings introduced after the self. In general, environments have the form $\Gamma = \overline{y : T}, \overline{z : T}$, with proper term bindings y followed by bindings z introduced by subtype comparisons. The notation $\Gamma_{[x]}$ in rules (SEL1) and (SEL2) signifies that all $z : T$ bindings to the right of x are dropped from Γ, where x can be either a proper term binding y or a z binding. While these restrictions are necessary for the proofs, they do not limit expressiveness of the type system in any significant way. Similar, and in many cases more impeding, restrictions are standard in type systems with recursive types [33]. Intuitively, note that the transitivity rule can always be used to relate type selections across the context. Note as well that there is no (BIND2) rule, symmetric to (BIND1), which is another kind of contractiveness restriction. We conjecture that these contractiveness restrictions could be lifted without breaking soundness, since we can always construct explicit conversion functions that use rules (VARPACK) and (VARUNPACK) on proper term bindings. However, removing these contractiveness restrictions would likely require different and harder to mechanize proof techniques such as a coinductive interpretation of subtyping.

In the version of DOT presented here, we make the transitivity rule (TRANS) explicit, although, as we will see in Section 4, we will sometimes need to ensure that we can eliminate uses of this rule from subtyping derivations so that the last rule is a structural one.

The aim of DOT is to be a simple, foundational calculus in the spirit of FJ [22], without committing to specific decisions for nonessential things. Hence, implementation inheritance, mixin strategy, and prototype vs class dispatch are not considered.

4. Static Properties of DOT

Having introduced the syntax and static semantics of DOT, we turn to its metatheoretic properties. Our main focus of interest will be type safety: establishing that well-typed DOT programs do not go wrong. Of course, type safety is only meaningful with respect to a dynamic semantics, which we will discuss in detail in Section 5. Here, we briefly touch some general static properties of DOT and then discuss specific properties of the subtyping relation, which (or their absence!) makes proving type safety a challenge.

Decidability Type assignment and subtyping are undecidable in DOT. This follows directly from the fact that DOT can encode $F_{<:}$, and that these properties are undecidable there.

Type Inference DOT has no principal types and no global Hindley-Milner style type inference procedure. But as in Scala, local type inference based on subtype constraint solving [34, 31] is possible, and in fact easier than in Scala due to the existence of universal greatest lower bounds and least upper bounds through intersection and union types. For example, in Scala, the least upper bound of the two types `C` and `D` is approximated by an infinite sequence:

```
trait A { type T <: A }
trait B { type T <: B }
trait C extends A with B { type T <: C }
trait D extends A with B { type T <: D }
lub(C, D) ~ A with B { type T <: A with B { type T <: ... } }
```

DOT's intersection and union types remedy this brittleness.

While the term syntax and type assignment given in Figure 1 is presented in Curry-style, without explicit type annotations except for type member initializations, a Church-style version with types on method arguments (as required for local type inference) is possible just as well. Our mechanized proof handles both Curry and Church style via optional parameter and return types. The Curry-style presentation glosses over explicit upcasts – a feature that can easily be added, for example through a typed `let` construct. Upcasts – whether explicit or implicit in the surface syntax – matter for nominality by ascription, as discussed in Section 2.

4.1 Properties of Subtyping

The relevant static properties we are interested in with regard to type safety are transitivity, narrowing, and inversion of subtyping and type assignment. They are defined as follows.

Inversion of subtyping (example for functions):

$$\frac{\Gamma \vdash m(x:S_1):U_1^x <: m(x:S_2):U_2^x}{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, x:S_2 \vdash U_1^x <: U_2^x} \text{ (InvFun)}$$

Transitivity of subtyping:

$$\frac{\Gamma \vdash T_1 <: T_2 \ , \ T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \qquad \text{(Trans)}$$

Narrowing:

$$\frac{\Gamma_1 \vdash T_1 <: T_2 \qquad \Gamma_2 \vdash T_3 <: T_4 \qquad \Gamma_1 = \Gamma_2(x \to T_1) \qquad \Gamma_2(x) = T_2}{\Gamma_1 \vdash T_3 <: T_4} \text{ (Narrow)}$$

where $\Gamma_1 = \Gamma_2(x \to T_1)$ denotes that Γ_1 is like Γ_2 except the binding for x maps to T_1.

On a high-level, the basic challenge for type safety is to establish that some value that e.g. has a function type actually is a function, with arguments and result corresponding to the given type. This is commonly known as the *canonical forms* property. Inversion of subtyping is required to extract the argument and result types from a given subtype relation between two function types, in particular to derive

$$T_2 <: T_1 \text{ and } U_1 <: U_2$$

from

$$m(x:T_1):U_1 <: m(x:T_2):U_2$$

when relating method types from a call site and the definition site.

Transitivity is required to collapse multiple subsumption steps that may have been used in type assignment. Narrowing can be seen as an instance of the Liskov substitution principle, preserving subtyping if a binding in the environment is replaced with a subtype. Narrowing is required for application, when the argument type is a subtype of the declared parameter type.

Unfortunately, as we will show next, these properties do *not* hold simultaneously in DOT, at least not in their full generality.

4.2 Inversion, Transitivity and Narrowing

First of all, let us take note that these properties are mutually dependent. In Figure 1, we have included (Trans) as an axiom. If we drop this axiom, then we obtain key rules like (InvFun) by direct inversion of the corresponding typing derivation, as only the structural rule (Fun) can match the pattern of comparing two function types. But then, we would need to prove (Trans) as a lemma.

Transitivity and narrowing are also mutually dependent. Transitivity requires narrowing in the following case, where we are given

$$\{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\} <: \{z \Rightarrow T_3\}$$

and want to derive:

$$\{z \Rightarrow T_1\} <: \{z \Rightarrow T_3\}$$

By inversion we obtain

$$z : T_1 \vdash T_1 <: T_2 \qquad z : T_2 \vdash T_2 <: T_3$$

and we narrow the right-hand derivation to $z : T_1 \vdash T_2 <: T_3$ before we apply transitivity inductively to obtain $z : T_1 \vdash T_1 <: T_3$ and thus $\{z \Rightarrow T_1\} <: \{z \Rightarrow T_3\}$.

Narrowing depends on transitivity in the case for type selections

$$x.L <: T \text{ or its counterpart } T <: x.L$$

Assume that we want to narrow x's binding from T_2 in Γ_2 to T_1 in Γ_1, with $\Gamma_1 \vdash T_1 <: T_2$. By inversion we obtain

$$x : (L : \bot..T)$$

and, disregarding rules (VarPack) and (VarUnpack) we can deconstruct this assignment as

$$\Gamma_2(x) = T_2 \qquad \Gamma_2 \vdash T_2 <: (L : \bot..T).$$

We first apply narrowing inductively and then use transitivity to derive the new binding

$$\Gamma_1(x) = T_1 \qquad \Gamma_1 \vdash T_1 <: T_2 <: (L : \bot..T).$$

On first glance, the situation appears to be similar to simpler calculi like $F_{<:}$, for which the transitivity rule can be shown to be admissible, i.e. implied by other subtyping rules and hence proved as a lemma and dropped from the definition of the subtyping relation. Unfortunately this is not the case in DOT.

4.3 Good Bounds, Bad Bounds

The transitivity axiom (or subsumption step in type assignment) is essential and cannot be dropped. Let us go through and see why we cannot prove transitivity directly.

First of all, observe that transitivity can only hold if all types in the environment have "good bounds", i.e. only members where the lower bound is a subtype of the upper bound. Through "bad" bounds in the context and subtyping transitivity, the subtyping lattice can collapse. For example, assume a binding with "bad" bounds like `Int..String`. Then the following subtype relation holds via transitivity

$$x : \{A = \texttt{Int..String}\} \vdash \texttt{Int} <: x.A <: \texttt{String}$$

but `Int` is not a subtype of `String`. Of course core DOT does not have `Int` and `String` types, but any other incompatible types can be taken as well.

But even if we take good bounds as a precondition, we cannot show

$$\{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\} <: \{z \Rightarrow T_3\}$$

because we would need to use $x : T_1$ in the extended environment for the inductive call, but we do not know that T_1 has indeed good bounds.

Of course we could modify the $\{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\}$ rule (BINDX) to require T_1 to have good bounds. But then this evidence would need to be narrowed, which unfortunately is not possible. Again, here is an example, considering a type under a context:

$$x : \{A : \bot..\top\} \vdash x.A \wedge \{B = \texttt{Int}\}$$

This type has good bounds, but when narrowing x in the context to the smaller type $\{A = \{B = \texttt{String}\}\}$ (which also has good bounds), its bounds become contradictory.

In summary, even if we assume good bounds in the environment, and strengthen our typing rules so that only types with good bounds are added to the environment, we may still end up with bad bounds due to narrowing and intersection types. This refutes one conjecture about possible proof avenues from earlier work on DOT [5].

Intuitively, all the questions related to bad bounds have a simple answer: We ensure good bounds at object creation time, so *why do we need to worry in such a fine-grained way?*

4.4 No (Simple) Substitution Lemma

As a final negative result, we illustrate how typing is not preserved by straightforward substitution in DOT, because of path-dependent types.

$$\Gamma, x : \{z \Rightarrow L : S^z..U^z \wedge m(\_) : T^z\} \vdash t^x : x.L$$

Now, consider substituting x in t with u, given

$$\Gamma \vdash u : \{z \Rightarrow L : S^z..U^z \wedge m(\_) : T^z\}$$

First, t^x might invoke method $x.m(\_)$ and therefore require assigning type $(m(\_) : T^x)$ to x. However, the self bind layer can only be removed if an object is accessed through a variable, otherwise we would not be able to substitute away the self identifier. Second, we cannot derive $\Gamma \vdash t^u : x.L$ with x removed from the environment, but we also cannot replace x in $x.L$ with a term u that is not an identifier. Hence, u cannot be an arbitrary term, because path-dependent types are not syntactically valid for arbitrary terms.

Intuitively, there is still hope for substitution, but only with forms that preserve syntactic validity of path-dependent types. Another option is to settle for a relaxed notion of path-dependent types that allows type selections on values [2].

4.5 There is Still Hope: Key Observations

Based on the discussion so far, we can make the following observations, which reveal a possible avenue for proving type soundness despite the difficulties:

Observation 1. *If objects can only be created with type aliases ($L = T$), not arbitrary bounds $T_1..T_2$, then runtime objects cannot have bad bounds.*

Observation 2. *If we assume a call-by-value evaluation strategy, then whenever we execute a method call at runtime, all variables are bound to existing objects.*

Observation 3. *The only place in a soundness proof where we rely on the* (INVFUN) *property is for the evaluation of such method calls.*

Taken together, these observations suggest that making a clear distinction between runtime values and other terms is a prerequisite for a successful soundness proof. Observation 3 further suggests that subtyping transitivity and narrowing can be treated as axioms in subtyping comparisons and when assigning types to static terms, as long as we can recover the (INVFUN) property in contexts that consist exclusively of runtime objects.

5. Operational Semantics

Type soundness is only meaningful with respect to a notion of evaluation: an operational semantics. In order to realize the soundness proof strategy outlined in Section 4.5, we need to pick a semantics that allows us to distinguish runtime values from normal terms. While such a distinction is particularly natural in big-step evaluators, the particular style of semantics does not matter so much, and different styles of semantics can be formally inter-derived using the techniques of Danvy et al. [12, 13, 1]. In our development, we have defined both big-step evaluators and small-step reduction semantics, and used them as the basis for mechanized proofs [36]. Here, we focus our presentation on a small-step reduction semantics, shown in Figure 2.

Syntax...

$x ::=$ Variable
- y Concrete Variable (i.e. Store Location)
- z Abstract Variable

$v ::= y$ Value (Store Location)

$\rho ::= \emptyset \mid \rho, y : \overline{d}$ Store

Reduction $\boxed{\rho_1\ t_1\ \rightarrow\ t_2\ \rho_2}$

$$\begin{array}{llllll} \rho & \{z \Rightarrow \overline{d^z}\} & \rightarrow & v & \rho, (v : \overline{d^v}) & \textbf{with } v \text{ fresh} \\ \rho & v_1.m(v_2) & \rightarrow & t^{v_2} & \rho & \textbf{if } \rho(v_1) \ni m(x) = t^x \\ \rho_1 & e[t_1] & \rightarrow & e[t_2] & \rho_2 & \textbf{if } \rho_1\ t_1\ \rightarrow\ t_2\ \rho_2 \end{array}$$
$$\textbf{where } e ::= [\,] \mid [\,].m(t) \mid v.m([\,])$$

Subtyping... $\boxed{\rho\ \Gamma\ \vdash\ S <: U}$

$$\frac{\rho(y) \ni (L = U) \qquad \rho\ \emptyset\ \vdash\ T <: U}{\rho\ \Gamma\ \vdash\ T <: y.L}\ (\text{SSEL2})$$

$$\frac{\rho(y) \ni (L = U) \qquad \rho\ \emptyset\ \vdash\ U <: T}{\rho\ \Gamma\ \vdash\ y.L <: T}\ (\text{SSEL1})$$

Type assignment... $\boxed{\rho\ \Gamma\ \vdash\ t :_{(!)}\ T}$

$$\frac{(y, \overline{d^y}) \in \rho \qquad \text{(labels disjoint)}\ \ \forall i, 1 \le i \le n \qquad \rho\ \emptyset, (x : T_1^x \wedge \ldots \wedge T_n^x)\ \vdash\ d_i^x : T_i^x}{\rho\ \Gamma\ \vdash\ y : T_1^y \wedge \ldots \wedge T_n^y}\ (\text{TLOC})$$

Figure 2. Small-Step Operational Semantics and Store Typing

We allocate all objects in a store-like structure, which grows monotonically. In this setting, store locations are the only values, i.e. passed around between functions and stored in environments. Figure 2 also shows necessary extensions to subtyping and type assignment, which we explain next.

A store-less alternative is also possible as detailed in Amin's thesis [2]. The key ideas are to remove the store indirection by treating values as concrete, and variables as abstract, and to relax the syntax of path-dependent types to also allow values as paths, in addition to variables.

5.1 Concrete Variables in Typing and Subtyping

To assign types to terms that occur during evaluation, we need to be able to handle path-dependent types that refer to variables bound in the store ρ, and type store locations. Accordingly, we extend the type system judgements with an additional store parameter ρ. The rules from Figure 1 are still valid, and merely thread the extra store parameter ρ unchanged. The additional cases are shown in Figure 2.

For concrete variables (i.e. store locations) in type selections, we use the precise type member given at object creation time by looking it up in the store, as defined by the new rules (SSEL1) and (SSEL2).

In this extended subtyping relation, we continue to use the same rules for type selections on variables bound in Γ, (SEL1) and (SEL2) from Figure 1. Hence, the auxiliary judgement $\rho\ \Gamma\ \vdash\ z :_!\ T$, which excludes (VARPACK), is only used for typing abstract variables.

Finally, we also need a new case for concrete variables in typing, (TLOC). It resembles (TNEW), except that it looks up the definitions in the store, it does not create a recursive type (because we can reuse the store location in the result type instead), and it does not consider the abstract context while checking the definitions (because all free variables in the store must be concrete).

6. Type Soundness

We discuss in detail our proofs for type soundness (Theorem 1), highlighting challenges and insights. Our statement of type safety is the following, folding together the usual notions of *progress* and *preservation* [41]:

Definition 1 (Type Safety)**.** *If a term t typechecks to some type T in a store and empty context $\rho\ \emptyset$ (i.e. $\rho\ \emptyset\ \vdash\ t : T$), then either the term t is a variable or the store-term configuration ρ, t steps to a configuration ρ', t' where the new store ρ' extends the old store ρ and the resulting configuration typechecks to the same type T (i.e. $\rho'\ \emptyset\ \vdash\ t' : T$).*

Note that Definition 1 assumes deterministic execution. Otherwise the statement would need to be modified to consider *all possible* following configurations.

We outline the main strategic choices next, with pointers to further subsections that describe the technical details. We reflect again on the higher-level strategy in Section 6.5.

Lenient Well-Formedness Any syntactically valid form (type or term) is considered well-formed, if all its free variables are bound in environments. Our strategy is to impose no semantics (e.g. good bounds for type members) on well-formedness, and ensure desired properties by construction only when they're needed (e.g. for runtime objects).

Lemma 1 (Regularity)**.** *If a judgement holds, all its forms are well-formed.*

Lemma 2 (Subtyping Reflexivity)**.** *Any well-formed type is a subtype of itself.*

This leniency pays off. As we saw in Section 4.3, trying to enforce semantic properties can break narrowing and other monotonicity properties. For example, the subtyping lattice is full just by definition, since the greatest lower bound and least upper bound of two types always exist, just by syntactic constructs (intersection and union).

"Pushback" For type soundness, we need to establish Canonical Forms (Section 6.3) properties: that if a concrete variable has a structural member type, then its definition in the store include a matching member definition.

This is challenging, as the evidence from the typing derivation might be indirect, because of subsumption and subtyping transitivity. Hence, we need to "pushback" such indirections.

We have two choices: pushback transitivity in subtyping (described in Section 6.1), or pushback value typing into directly invertible evidence (an independent alternative described in Section 6.6).

In both options, a key insight is that we only need to do the pushback in an empty abstract context, which circumvents the impossibility results of Section 4. This is where it pays off to distinguish between runtime values and only static types. Plus, for runtime values, we can rely on any properties enforced during their construction, in particular "good bounds".

Substitution By design, substitution is only needed to replace an abstract variable with a concrete one. Because the abstract variable might have a less precise type, there's also a narrowing step involved. Furthermore, type selections on concrete variables are defined more precisely than for abstract ones, which requires converting the evidence from subtyping with abstract type selections to concrete. We describe further in Section 6.2

6.1 Narrowing and Transitivity Pushback

In simpler type systems like $F_{<:}$, transitivity can be proved as a lemma, together with narrowing, in a mutual induction on the size of the middle type in a chain $T_1 <: T_2 <: T_3$ (see e.g. the POPLmark challenge documentation [7]).

Unfortunately, for subtyping in DOT, the same proof strategy fails, because subtyping may involve a type selection as the middle type: $T_1 <: z.L <: T_3$. Since proving transitivity becomes much harder, we adopt a strategy from previous DOT developments [5]: admit transitivity as an axiom (TRANS), but prove a 'pushback' lemma that allows to push uses of the axiom further up into a subtyping derivation, so that the top level becomes invertible. We denote this as *precise* subtyping $T_1 <:! T_2$.

Definition 2 (Precise Subtyping)**.** *If* $\rho\ \Gamma\ \vdash\ T_1 <: T_2$ *and the derivation does not end in rule* (TRANS) *then we say that* $\rho\ \Gamma\ \vdash\ T_1 <:! T_2$.

Such a strategy is reminiscent of cut elimination in natural deduction, and in fact, the possibility of cut elimination strategies is already mentioned in Cardelli's original work on $F_{<:}$ [9].

The narrowing lemma does not have any dependencies:

Lemma 3 (Narrowing)**.**

$$\frac{\rho\ \Gamma_1\ \vdash\ T_1 <: T_2 \qquad \rho\ \Gamma_2\ \vdash\ T_3 <: T_4 \qquad \Gamma_1 = \Gamma_2(z \to T_1) \qquad \Gamma_2(z) = T_2}{\rho\ \Gamma_1\ \vdash\ T_3 <: T_4}$$

Proof. By structural induction, and using the (TRANS) axiom. □

Lemma 4 (Transitivity Pushback)**.**

$$\frac{\rho\ \emptyset\ \vdash\ T_1 <: T_2}{\rho\ \emptyset\ \vdash\ T_1 <:! T_2}$$

Proof. We prove an auxiliary lemma by induction on the subtyping derivation $T_1 <: T_2$:

$$\frac{\rho\ \emptyset\ \vdash\ T_1 <: T_2\ ,\ T_2 <:! T_3}{\rho\ \emptyset\ \vdash\ T_1 <:! T_3}$$

using narrowing (Lemma 3) in cases (FUN), (BINDX), (BIND1). Transitivity pushback follows from the special case $T_2 = T_3$. □

Lemma 5 (Inversion of Subtyping)**.** *For example for function types:*

$$\frac{\rho\ \emptyset\ \vdash\ m(x : S_1) \to U_1^x <: m(x : S_2) \to U_2^x}{\rho\ \emptyset\ \vdash\ S_2 <: S_1 \qquad \rho\ x : S_2\ \vdash\ U_1^x <: U_2^x}$$

Proof. By transitivity pushback (Lemma 4) and inversion of the resulting derivation. □

Inversion of subtyping is only required in a concrete runtime context, without abstract component ($\Gamma = \emptyset$). Therefore, transitivity pushback is only required then. Transitivity pushback requires narrowing, but only for abstract bindings (those in Γ, never in ρ). Narrowing requires these bindings to be potentially imprecise, so that the transitivity axiom can be used to inject a step to a smaller type without recursing into the derivation. In summary, we need both (actual, non-axiom) transitivity and narrowing, but not at the same time.

The insight that transitivity pushback is only required in a concrete-only runtime context is crucial for supporting language features such as intersection types that may collapse the subtyping lattice in unrealizable contexts.

6.2 Bootstrapping Substitution and Canonical Forms

To mirror the effect of reduction $\rho\ x.m(y)\ \to\ t^y\ \rho$ given $\rho(x) \ni m(z) = t^z$, we need a substitution lemma for terms, but we also need to mirror the reduction on the type level, since types can refer to variables via type selections. More precisely, we need a substitution lemma that enables us to replace all type selections on an abstract variable $z.A$ with

concrete ones $y.A$. For this, we need to ensure that for a concrete variable with a structural type-member type, its definitions in the store include a corresponding type member – a Canonical Forms property for type members (see Section 6.3).

We cannot prove either of these properties directly and therefore need to bootstrap a mutual induction. The key induction measure will be an upper bound on the uses of (VARPACK) in a typing derivation.

Definition 3 (VARPACK Metric)**.** *Let* $\rho\ \Gamma\ \vdash_{\leq m}\ y : S$ *denote a derivation* $\rho\ \Gamma\ \vdash\ y : S$ *with no more than* m *uses of* (VARPACK).

Definition 4 (Substitution)**.** *Let Subst*(m) *denote:*

$$\frac{\rho\ \emptyset\ \vdash_{\leq m_1}\ y : S^y \qquad m_1 < m \qquad \rho\ z : S^z, \Gamma^z\ \vdash\ {T_1}^z <: {T_2}^z}{\rho\ \Gamma^y\ \vdash\ {T_1}^y <: {T_2}^y}$$

We go on to prove preliminary canonical forms lemmas, assuming the ability to perform substitution for a given m.

Lemma 6 (Pre-canonical Forms for Recursive Types)**.**

$$\frac{\rho\ \emptyset\ \vdash_{\leq m}\ \ y : \{z \Rightarrow T^z\} \qquad \text{Subst}(m)}{\rho\ \emptyset\ \vdash_{\leq m-1}\ \ y : T^y}$$

Proof. Any trailing uses of (SUB) can be accumulated into a single subtyping statement $\rho\ \emptyset\ \vdash\ \{z \Rightarrow T'^z\} <: \{z \Rightarrow T^z\}$, and the remaining derivation must end in (VARPACK). By inversion we obtain $\rho\ \emptyset\ \vdash_{\leq m-1}\ \ y : T'^y$. After transitivity pushback (Lemma 4), the subtyping derivation must end in (BINDX), from which we obtain $\rho\ z : T'^z\ \vdash\ T'^z <: T^z$. We can now apply our Subst(m) capability to obtain $\rho\ \emptyset\ \vdash\ T'^y\ <:\ T^y$. Applying subsumption (SUB) yields $\rho\ \emptyset\ \vdash_{\leq m-1}\ \ y : T^y$. □

Lemma 7 (Pre-canonical Forms for Type Members)**.**

$$\frac{\rho\ \emptyset\ \vdash_{\leq m}\ \ y : \{L : S..U\} \qquad \text{Subst}(m)}{\rho(y) \ni (L = T) \qquad \rho\ \emptyset\ \vdash\ S <: T <: U}$$

Proof. Again, we accumulate trailing uses of (SUB) into an invertible subtyping. If we hit (VARPACK), the resulting subtyping derivation must collapse into (BIND1). We finish the case by applying Lemma 6, Subst(m), and (SUB). Case (TLOC) is immediate from inversion of member case (DTYP). □

We are now ready to prove our substitution lemma, together with two helpers. The proof is by simultaneous induction, first over m, and then an inner induction over the size of the $<:$ or $:_!$ derivation.

Lemma 8 (Substitution for $<:$)**.** $\forall m.$*Subst*(m)*, i.e.:*

$$\frac{\rho\ \emptyset\ \vdash_{\leq m}\ y : S^y \qquad \rho\ z : S^z, \Gamma^z\ \vdash\ {T_1}^z <: {T_2}^z}{\rho\ \Gamma^y\ \vdash\ {T_1}^y <: {T_2}^y}$$

Proof. The key challenge is translating from the rules (SEL1) / (SEL2) to the rules (SSEL1) / (SSEL2). For (SEL2), if we are selecting $z.A$, we have $\rho\ z : S^z\ \vdash\ z :_!\ (L : T^z..\top)$, and with Lemma 10 we obtain $\rho\ \emptyset\ \vdash_{\leq m}\ y : (L : T^y..\top)$. We invoke canonical forms (Lemma 7), which yields $\rho(y) \ni (L = U)$ and $\rho\ \emptyset\ \vdash\ T^y <: U$. Note that we can apply Lemma 7 because of our outer induction on m. If we are selecting $z'.A$ with $z' \neq z$, we apply Lemma 9. Case (SEL1) is analogous. Note that the (SEL1)/(SEL2) rules are carefully set up so that $\Gamma_{[z]} = z : S$. □

Lemma 9 (Substitution for $:_!$)**.**

$$\frac{\rho\ \emptyset\ \vdash_{\leq m}\ y : S^y \qquad z' \neq z \qquad \rho\ z : S^z, \Gamma^z\ \vdash\ z' :_!\ T^z}{\rho\ \Gamma^y\ \vdash\ z' :_!\ T^y}$$

Proof. Straightforward. Case (SUB) uses Lemma 8. □

Lemma 10 (Substitution for $:$)**.**

$$\frac{\rho\ \emptyset\ \vdash_{\leq m}\ y : S^y \qquad \rho\ z : S^z, \Gamma^z\ \vdash\ z :_!\ T^z}{\rho\ \emptyset\ \vdash_{\leq m}\ y : T^y}$$

Proof. Case (VAR) is immediate. In case (VARUNPACK), we have $\rho\ z : S^z, \Gamma^z z :_!\ \{z_2 \Rightarrow T^{z_2}\}^z$ and by induction we obtain $\rho\ \emptyset\ \vdash_{\leq m}\ y : \{z_2 \Rightarrow T^{z_2}\}^y$, as required. In case (SUB), we have $\rho\ z : S^z, \Gamma^z z :_!\ T'^z$ and $\rho\ z : S^z, \Gamma^z z :_!\ T'^z <: T^z$. Applying the induction hypothesis and Lemma 8 and using (SUB) we get $\rho\ \emptyset\ \vdash_{\leq m}\ y : T'^y$. □

Finally, we can prove a substitution on terms.

Lemma 11 (Substitution in Term Typing)**.**

$$\frac{\rho\ (z : S), \Gamma^z\ \vdash\ t^z : T^z \qquad \rho\ \vdash\ y :_!\ S}{\rho\ \Gamma^y\ \vdash\ t^y : T^y}$$

Proof. Straightforward induction. Case (SUB) uses substitution of subtyping (Lemma 8). □

6.3 Inversion of Value Typing (Canonical Forms)

As a corollary of substitution (Lemma 8), we can extend the preliminary canonical forms lemmas to all m.

Lemma 12 (Canonical Forms for Type Members)**.**

$$\frac{\rho\ \emptyset\ \vdash\ \ y : \{L : S..U\}}{\rho(y) \ni (L = T) \qquad \rho\ \emptyset\ \vdash\ S <: T <: U}$$

Proof. Follows directly from Lemma 7 and Lemma 8. □

We also prove an additional canonical forms lemma for function members, which will be required by the main proof.

Lemma 13 (Canonical Forms for Method Members)**.**

$$\frac{\rho\ \emptyset\ \vdash\ \ y : m(x : S_2) \to U_2^x}{\begin{array}{c}\rho(y) \ni m(x) = t \\ \rho\ \emptyset, x : S_1\ \vdash\ t : U_1^x \\ \rho\ \emptyset\ \vdash\ m(x : S_1) \to U_1^x\ <:\ m(x : S_2) \to U_2^x\end{array}}$$

Proof. Analogous to Lemma 7. Case (TLOC) eliminates the self variable in the abstract context by applying substitution of term typing (Lemma 11). □

6.4 The Main Soundness Proof

Our main type safety theorem combines the usual *progress* and *preservation* lemmas into a single unified statement.

Theorem 1 (Type Safety)**.** *If* $\rho\ \emptyset \vdash t : T$*, then either* $t = y$ *and* $\rho(y) = \{\overline{d}\}$ *or* $\rho\ t \rightarrow t'\ \rho'$ *and* $\rho'\ \emptyset \vdash t' : T$

Proof. By structural induction. The most interesting case is the dependent application (TAPPVAR), if the receiver of the call is already evaluated to a store location y. We have $\rho\ \emptyset \vdash y.m(y_1) : U_2^{y_1}$ and, by inversion, $\rho\ \emptyset \vdash y : (m(x : S_2) : T_2^x)$ and $\rho\ \emptyset \vdash y_1 : S_1$. By canonical forms for method members (Lemma 13) we obtain $\rho(y) \ni m(x) = t^x$, $\rho\ x : S_1 \vdash t^x : U_1^x$, and $\rho \vdash m(x : S_1) : U_1^x <: m(x : S_2) : U_2^x$. We know that $\rho\ y.m(y_1) \rightarrow t^{y_1}\ \rho$, so we need to show that the result is well-typed with the same type: $\rho\ \emptyset \vdash t^{y_1} : U_2^{y_1}$. We apply inversion of subtyping (Lemma 5) and obtain $\rho\ \emptyset \vdash S_2 <: S_1$ and $\rho\ x : S_2 \vdash U_1^x <: U_2^x$. With (SUB), we have $\rho\ \emptyset \vdash y_1 : S_1$, and we can apply substitution of term typing (Lemma 11), to obtain $\rho\ \emptyset \vdash t^{y_1} : U_2^{y_1}$. With (SUB) and applying substitution of subtyping (Lemma 8) to derive $\rho\ \emptyset \vdash U_1^{y_1} <: U_2^{y_1}$, we arrive at the required $\rho\ \emptyset \vdash t^{y_1} : U_2^{y_1}$. □

6.5 Some Reflection

The soundness proof was set up carefully to avoid cycles between required lemmas. Where cycles did occur, as with transitivity and narrowing, we broke them using a pushback technique. Where this was cumbersome to do, as with substitution and canonical forms, we used mutual induction. A key property of the system is that, in general, we are very lenient about things outside of the concrete runtime store. The only place where we invert a dependent function type and go from abstract to concrete is in showing safety of the corresponding type assignment rules. This enables subtyping inversion and pushback to disregard abstract bindings for the most part.

When seeking to unpack recursive self types within lookups of type selections in subtype comparisons, the option of disregarding abstract bindings is no longer so easy. Every lookup of a variable, while considering a path-dependent type, may potentially need to unfold and invert self types. In particular, the substitution lemma itself that converts imprecise into precise bounds may unfold a self type. Then it will be faced with an abstract variable that first needs to be converted to a concrete variable. More generally, whenever we have a chain

$$\{z \Rightarrow T_1\} <: T <: \{z \Rightarrow T_2\},$$

we first need to apply transitivity pushback to perform inversion. But then, the result of inversion will yield another imprecise derivation

$$T_1 <: U <: T_2$$

which may be bigger than the original derivation due to transitivity pushback. So, we cannot process the result of the inversion further during an induction. This increase in size is a well-known property of cut elimination: removing uses of a cut rule (like our transitivity axiom) from a proof term may increase the proof size exponentially.

This is why type selections use the abstract variable type assignment, which disallow packing, relying on unpacking and subtyping instead.

Furthermore, for sub-derivations on a self type, the abstract context must be restricted to not include any binding added after the binding for the self type, so that the pushback lemma can be applied during substitution. In fact, this last restriction in the model is not just a technical device, it seems reasonable for soundness. Indeed, a self type might use a type that has bad bounds within a definition (for example, in a function parameter type, or a type member alias). When subtyping two recursive types, such nonsensical types might be added to the abstract context, but we do not want to unpack a self type by using such temporary bad evidence.

6.6 Alternative: Invertible Concrete Variable Typing

In the preceding sections, the presence of the subsumption rule in type assignment required us to prove various canonical forms lemmas. It is also possible to turn this around: design a type assignment relation for concrete variables that is directly invertible, and prove the subsumption property (upwards-closure with respect to subtyping) as a lemma.

Here we sketch this alternative, focusing on the set up of the auxiliary relation and lemmas rather than the proof details.

Invertible Concrete Variable Type Assignment We define the concrete variable type assignment, $\rho \vdash y :! T$, to be directly invertible by excluding subsumption, and instead relating a value in the store to each of its possible type. There is one case per syntactic form, except no case for $\bot$ and two cases for union types.

Definition 5 (Concrete Variable Type Assignment)**.** $\rho \vdash y :! T$ *is defined inductively by the following rules.*

$$\frac{y \in \rho}{\rho \vdash y :! \top} \quad \text{(V-TOP)}$$

$$\frac{\rho(y) \ni (L = T) \qquad \rho\ \emptyset \vdash S <: T\ ,\ T <: U}{\rho \vdash y :! (L : S..U)} \quad \text{(V-TYP)}$$

$$\frac{\begin{array}{c}(y, \overline{d^y}) \in \rho \\ \text{(labels disjoint)} \qquad \forall i, 1 \le i \le n \\ \rho\ \emptyset, (x : T_1^x \wedge \ldots \wedge T_n^x) \vdash d_i^x : T_i^x \\ \exists j,\ d_j^x = (m(z : S) = t^z) \quad T_j^x = m(z : S^x) : U^{x,z} \\ \rho\ \emptyset \vdash S' <: S^y \qquad \rho\ \emptyset, (z : S') \vdash U^{y,z} <: U'^z\end{array}}{\rho \vdash y :! (m(x : S') : U'^x)} \quad \text{(V-FUN)}$$

$$\frac{\rho(x) \ni (L = T) \qquad \rho \vdash y :_! T}{\rho \vdash y :_! (x.L)} \quad \text{(V-SEL)}$$

$$\frac{\rho \vdash y :_! T^y}{\rho \vdash y :_! \{z \Rightarrow T^z\}} \quad \text{(V-BIND)}$$

$$\frac{\rho \vdash y :_! T_1 \ , \ y :_! T_2}{\rho \vdash y :_! T_1 \wedge T_2} \quad \text{(V-AND)}$$

$$\frac{\rho \vdash y :_! T_1}{\rho \vdash y :_! T_1 \vee T_2} \quad \text{(V-OR1)}$$

$$\frac{\rho \vdash y :_! T_2}{\rho \vdash y :_! T_1 \vee T_2} \quad \text{(V-OR2)}$$

Bootstrapping Substitution and Widening We proceed in a way similar to Section 6.2 to bootstrap a mutual induction, but this time with swapped assumptions: canonical forms properties are now immediate, but we need to assume a widening (i.e. subsumption) capability.

Definition 6 (V-BIND Metric)**.** *Let $\rho \vdash_{\leq m} y :_! S$ denote a derivation $\rho \vdash y :_! S$ with no more than m uses of* (V-BIND).

Definition 7 (Widening)**.** *Let Widen(m) denote:*

$$\frac{\rho \vdash_m y :_! T \qquad \rho\, \emptyset \vdash T <: U}{\rho \vdash_{\leq m} y :_! U}$$

We prove three substitution lemmas, analogous to Lemmas 8,9, and 10, but predicated on Widen(m), and with respect to concrete variable typing ($\rho \vdash y :_! T$) instead of term typing ($\rho\, \emptyset \vdash y : T$).

Lemma 14 (Substitution for Subtyping)**.**

$$\frac{\rho \vdash_{\leq m} y :_! S^y \qquad \text{Widen}(m) \qquad \rho\, z : S^z, \Gamma^z \vdash {T_1}^z <: {T_2}^z}{\rho\, \Gamma^y \vdash {T_1}^y <: {T_2}^y}$$

Lemma 15 (Substitution for Abstract Variable Typing)**.**

$$\frac{\rho \vdash_{\leq m} y :_! S^y \qquad \text{Widen}(m) \qquad z' \neq z \qquad \rho\, z : S^z, \Gamma^z \vdash z' :_! T^z}{\rho\, \Gamma^y \vdash z' :_! T^y}$$

Lemma 16 (Substitution for Concrete Variable Typing)**.**

$$\frac{\text{Widen}(m) \qquad \rho \vdash_{\leq m} y :_! S^y \qquad \rho\, z : S^z, \Gamma^z \vdash z :_! T^z}{\rho \vdash_{\leq m} y :_! T^y}$$

Note that the abstract context in the premise of Lemma 16 disappears entirely from the conclusion, which is about concrete type assignment. This gives another intuition why the model restricts the abstract context when typing recursive types for type selections. Indeed, the extra abstract bindings that come from subtyping recursive types within recursive types might cause more derivations to hold, via lattice collapsing, in the abstract than in the concrete.

We can now prove that a general widening or subsumption rule is admissible.

Lemma 17 (Concrete Variable Widening)**.** $\forall m.$*Widen*(m)*, i.e.:*

$$\frac{\rho \vdash_m y :_! T \qquad \rho\, \emptyset \vdash T <: U}{\rho \vdash_{\leq m} y :_! U}$$

Finally, we can relate the normal typing relation and our concrete variable typing.

Lemma 18 (Concrete Variable Typing Inversion)**.** *Term typing of a concrete variable in an empty abstract context implies the same concrete variable type assignment.*

$$\frac{\rho\, \emptyset \vdash y : T}{\rho \vdash y :_! T}$$

7. Perspectives

7.1 DOT is Sound, but is Scala Sound?

It is not always clear how well results from a small formal model translate to a realistic language. In the case of DOT, the interleaved process of designing and proving sound a prescriptive core model of Scala's type system has provided valuable insights into the design space. Through debugging DOT models, we have uncovered several soundness issues in Scala. We have already discussed problems related to `null` values in Section 2. While it can be argued that `null` is a fundamentally unsound feature anyways, and therefore soundness issues involving `null` may be acceptable, we give two further examples here, which use only safe language features and thus constitute uncontroversial soundness violations.

In DOT, type members in new instances are restricted to aliases, so that "good bounds" are enforced syntactically rather than semantically. This restriction was added after it became clear that subtle situations could arise during object initialization with recursive types, where runtime contexts would be polluted by "bad" evidence that is temporary or justified only through circular reasoning. For a similar reason, there is no subsumption in member initialization, so type-checking at object creation (TNEW) can tie the recursive knot, without needing to check bounds.

Scala was designed and implemented before all these corner cases became apparent and allows more flexible bounds. Bounds are checked to be "good", but these checks are not sufficient. Here is an example where a concrete object with "bad bounds" slips through in Scala 2.11.8, causing a cast exception at runtime.

```
trait O { type A >: Any <: B; type B >: A <: Nothing }
val o = new O {}
def id(a: Any): Nothing = (a: o.B)
val n: Int = id("Boom!") // runtime cast exception
```

As another example, Scala allows `lazy vals` in paths of type selections, while trying to enforce realizability to prevent unsoundness due to "bad bounds" on non-terminating lazy paths that are never forced but appear in types. But from DOT, we know that realizability is not preserved by narrowing, and with a bit of work, we can exploit this insight and demonstrate the pitfalls of this approach.

```
trait A { type L <: Nothing }
trait B { type L >: Any}
trait U {
  type X <: B
  val p: X
  def id(x: Any): p.L = x // used in plausible context
}
trait V extends U {
  type X = B with A // unrealizable
  lazy val p: X = p // non-terminating
}
val v = new V {}
val n: Int = v.id("Boom!") // runtime cast exception
```

To conclude this section, we believe that formal work on core language models is important and, even though we do not and cannot consider a full language, this work still helps making full languages safer. In addition, formal work can also chart new territory and lead to more general and more regular full languages, by (cautiously) suggesting that a feature may be safer than previously thought and relaxing some constraints. For example, structural recursive types are more expressive in our DOT model than in Scala.

7.2 Scaling up: The Road Ahead

While we have seen in Section 2 that DOT can encode a large class of realistic Scala programming patterns, a number of non-quintessential but practically relevant features have been (sometimes deliberately) left out of DOT's formal model. Below is an attempt to classify these features according to their ease of integration with DOT and potential implications for type soundness.

Largely Orthogonal Features (1) Traits, Classes, Inheritance and Mixins: DOT does not model any "implementation reuse" mechanisms such as inheritance and mixins (which Scala has) and prototype dispatch (which does not currently exist in Scala, but would be interesting to consider). Most likely, such extensions will be through encodings, showing type preservation of a translation, and not through extending the DOT meta-theory itself. Some challenges: type member inheritance vs "good bounds" at creation time, open construction of nominal hierarchies. (2) Mutable State: this will require standard extensions to the operational semantics, and it is important that type selections remain *stable*, i.e. exclude mutable variables. As is standard, the interaction of mutation and polymorphism requires care [39], but previous work [36] has shown that mutation can be added to DOT-like type systems without interfering with soundness. (3) Implicits: Scala has both implicit parameters and implicit conversions, which are automatically inserted by the compiler based on types and scopes. Implicit parameters are useful for modeling type classes. Implicits do not introduce new types as typing rules and are unlikely to interfere with soundness.

Fitting Extensions (1) Full Paths: extend type selections to include chains of immutable field selections `x.a.b.C` in addition to variables `x.A`. Some challenges: path equality and reduction in type selections, field initialization, circular reasoning with self occurrences. (2) Singleton Types: `x.type` in addition to `x.A`, denoting the type that is only inhabited by the value of `x`. An interesting question is whether singleton types are already encodable in DOT.

Restricted Extensions (1) Laziness: DOT relies on the assumption of call-by-value evaluation. To model Scala's `lazy vals`, some restrictions are necessary. As a first approximation one can forbid type selections on `lazy vals`, but relaxations in combination with traits and class types may be possible, though challenging in terms of balancing flexibility and safety. Another avenue is deliberately forcing evaluation of `lazy vals` that occur in types. (2) By-Name Arguments: must not occur in type selections.

Debatable Extensions Type Projections, Type Lambdas and Higher-Kinded Types: These features are most likely not faithfully encodable in DOT. Type projections `T#A` are similar to type selections `x.A`, but crucially lose the guarantee of variable `x` pointing to an existing object. Thus, it is presently unclear how to approach soundness of type projections. For type lambdas, what is missing is a way to calculate the type resulting from a type application. This could be encoded through type projections or through dependent types beyond just paths. Simplified models are readily doable (as outlined in the introduction of Section 2), so it is not clear whether the unrestricted encoding is worth the extra complexity.

Incompatible Extensions As discussed in Section 2 and elsewhere [6], Scala's oblivious treatment of `null` pointers, which mainly exists for backwards compatibility with Java, seems to be fundamentally unsound. Future version of Scala can achieve soundness by (1) reflecting nullability in the type system, e.g. via union types, (2) preventing nullable expressions in type selections, similar to mutable variables or by-name arguments, and (3) tightening the rules for object member initialization so that `null` values cannot be observed during object creation.

8. Related Work

Scala Foundations Much work has been done on grounding Scala's type system in theory. Early efforts included νObj [30], Featherweight Scala [11] and Scalina [26], all of them more complex than what is studied here. None of them lead to a mechanized soundness result, and due to their inherent complexity, not much insight was gained why soundness was so hard to prove. DOT [4] was proposed as a simpler and more foundational core calculus, focusing on path-dependent types but disregarding classes, mixin linearization and similar questions. The original DOT formulation [4] had actual preservation issues because lookup was required to be precise. This prevented narrowing, as explained in Section 4. The originally proposed small step rewriting semantics with a store exposed the difficulty of relating paths at different stages of reductions.

The μDOT calculus [5] is the first calculus in the line with a mechanized soundness result, (in Twelf, based on total big-step semantics), but the calculus is much simpler than what is studied in this paper. Most importantly, μDOT lacks bottom, intersections and type refinement. Amin et al. [5] describe in great detail why adding these features causes trouble. Because of its simplicity, μDOT supports both narrowing and transitivity with precise lookup. The soundness proof for μDOT was also with respect to big-step semantics. However, the semantics had no concept of distinct runtime type assignment and would thus not be able to encode $F_{<:}$ and much less full DOT.

After the first version of this paper was circulated as a tech report [36], a soundness proof sketch for another DOT variant was proposed by Odersky [3]. While on the surface similar to the DOT version in this paper and in [36], there are some important differences that render the calculus in [3] much less expressive. First, the calculus is restricted to Administrative Normal Form (ANF) [18], requiring all intermediate subexpressions to be let-bound with explicit names. While seemingly a minor issue, reported difficulties in proving encodings of simpler calculi such as $F_{<:}$ that are not in ANF may suggest that this restriction is not entirely trivial. Second, and more importantly, the calculus does not support subtyping between recursive types (rules (BINDX) and (BIND1)), only their introduction and elimination as part of type assignment. This skirts most of the thorny issues in the proofs (see Section 6.2) and also limits the expressiveness of the calculus. For example, an identifier x bound to a refined type $\{z \Rightarrow T \wedge U^z\}$ can be treated as having type T, but if it instead has type $S \to \{z \Rightarrow T \wedge U^z\}$, it can not be assigned type $S \to T$. Instead, one has to eta-expand the term into a function (i.e. an object with a single method), `let`-bind the result of the call, and insert the required coercion to T. Similar considerations apply to types in other non-toplevel positions such as bounds of type members, but there it is not even clear if an analogue of eta-expansion is available. With this requirement for explicit conversions, the calculus in [3] does not, at least in our view, capture the essence of a type system based on subtyping. The results reported in the present paper predate those from [3], and they have no such restrictions: we provide a soundness proof, mechanized in Coq, for a calculus that is not restricted to ANF, and that supports subtyping between recursive types.

ML Module Systems 1ML [37] unifies the ML module and core languages through an elaboration to System F_ω based on earlier such work [38]. Compared to DOT, the formalism treats recursive modules in a less general way and it only models fully abstract vs fully concrete types, not bounded abstract types. Although an implementation is provided, there is no mechanized proof. In good ML tradition, 1ML supports Hindler-Milner style type inference, with only small restrictions. Path-dependent types in ML modules go back at least to SML [25], with foundational work on transparent bindings by Harper and Lillibridge [21] and Leroy [24]. MixML [14] drops the stratification requirement and enables modules as first-class values.

Other Related Languages Other languages and calculi that include features related to DOT's path dependent types include the family polymorphism of Ernst [15], Virtual Classes [17, 16, 27, 19], and ownership type systems like Tribe [10, 8]. Nominality by ascription is also achieved in Grace [23].

9. Conclusions

The key aim behind DOT is to build a solid foundation for Scala and similar languages from first principles. DOT has also been described as the essence of Scala: what remains after you "boil Scala on a slow flame and wait until all incidental features evaporate" [29].

We have presented the first soundness result for a variant of DOT that includes recursive type refinement and a subtyping lattice with full intersection types, demonstrating how the difficulties that prevented such a result previously can be overcome with a semantic model that exposes a distinction between static terms and runtime values.

We also think that it is important to convey not just that a calculus is sound in isolation, but also what assumptions the soundness proof relies on in order to evaluate the broader applicability of the work. In particular, our proof relies crucially on runtime values having only type members with good bounds, which the syntax enforces. Because of recursive types, such a property would be difficult to enforce semantically. It also relies on call-by-value semantics, in that it expects all variables that can partake in types to point to runtime values when a method body is evaluated.

Finally, in our own experience with DOT, the process of designing the calculus and proving it sound have been intertwined. As we understood the landscape better, we have been able to make the model more uniform yet powerful.

Acknowledgements

The initial design of DOT is due to Martin Odersky. Geoffrey Washburn, Adriaan Moors, Donna Malayeri, Samuel Grütter and Sandro Stucki have contributed to its development. For insightful discussions we thank Amal Ahmed, Jonathan Aldrich, Derek Dreyer, Sebastian Erdweg, Erik Ernst, Matthias Felleisen, Ronald Garcia, Paolo Giarrusso, Scott Kilpatrick, Grzegorz Kossakowski, Alexander Kuklev, Viktor Kuncak, Ondřej Lhoták, Alex Potanin, Jon Pretty, Didier Rémy, Lukas Rytz, Miles Sabin, Ilya Sergey, Jeremy Siek, Josh Suereth, Ross Tate, Eelco Visser, Philip Wadler and Jason Zaugg. Finally, we thank the anonymous reviewers for their thoughtful comments.

This research received funding from the European Research Council (ERC) under grant 587327 DOPPLER, from NSF under CAREER award 1553471, and from Purdue University through a faculty startup package.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, 2003.

[2] N. Amin. *Dependent Object Types*. PhD thesis, EPFL, 2016.

[3] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The essence of dependent object types. In *WadlerFest*, A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, 2016.

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

[5] N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014.

[6] N. Amin and R. Tate. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA*, 2016.

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

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

[9] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Inf. Comput.*, 109(1/2):4–56, 1994.

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

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

[12] O. Danvy and J. Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *J. Comput. Syst. Sci.*, 76(5):302–323, 2010.

[13] O. Danvy, K. Millikin, J. Munk, and I. Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theor. Comput. Sci.*, 435:21–42, 2012.

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

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

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

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

[18] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.

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

[20] J.-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. 1972.

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

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

[23] T. Jones, M. Homer, and J. Noble. Brand objects for nominal typing. In *ECOOP*, 2015.

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

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

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

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

[28] M. Odersky. The trouble with types. Presentation at Strange Loop, 2013.

[29] M. Odersky. The essence of Scala. `http://www.scala-lang.org/blog/2016/02/03/essence-of-scala.html`, February 2016.

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

[31] M. Odersky and K. Läufer. Putting type annotations to work. In *POPL*, 1996.

[32] M. Odersky and T. Rompf. Unifying functional and object-oriented programming with Scala. *Commun. ACM*, 57(4):76–86, 2014.

[33] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[34] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[35] J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[36] T. Rompf and N. Amin. From F to DOT: Type soundness proofs with definitional interpreters. Technical report, Purdue University, July 2015. `http://arxiv.org/abs/1510.05216`.

[37] A. Rossberg. 1ML - core and modules united (f-ing first-class modules). In *ICFP*, 2015.

[38] A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014.

[39] A. J. Summers. Modelling java requires state. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, page 10. ACM, 2009.

[40] G. A. Washburn. SI-1557: Another type soundness hole. `https://issues.scala-lang.org/browse/SI-1557`, 2008.

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

A. Mechanization in Coq

We outline the correspondence between the formalism on paper and its implementation in Coq (`oopsla16.namin.net`).

A.1 Model (`dot.v`)

A.1.1 Syntax (Figure 1)

`ty`	$S, T, U ::=$	Type	
`TTop`	$\top$	top type	
`TBot`	$\bot$	bottom type	
`TTyp` *L S U*	$L : S..U$	type member	
`TFun` *m S U*	$m(x : S) : U^x$	method member	
`TSel` *X L*	$x.L$	type selection	
`TBind` *T*	$\{z \Rightarrow T^z\}$	recursive self type	
`TAnd` *T T*	$T \wedge T$	intersection type	
`TOr` *T T*	$T \vee T$	union type	
`tm`	$t, u ::=$	Term	
`tvar` *b x*	x	variable reference	
`tobj` $\overline{d}$	$\{z \Rightarrow \overline{d}\}$	new instance	
`tapp` *t m t*	$t.m(t)$	method invocation	
`dm`	$d ::=$	Initialization	
`dty` *T*	$L = T$	type initialization	
`dfun` *[S] [T] t*	$m(x) = t$	method initialization	
`dms`	$\overline{d}$		

To demonstrate that both Church and Curry syle are possible, the parameter and return types `S` and `T` of a method initialization `dfun` *[S] [T] t* are optional.

For representing variable names in relation to an environment (context or store), we use a reverse de Bruijn convention, so that the name is invariant under environment extension. An environment is a list of right-hand sides. The older the binding, the more to the right, the smaller its number name. The name is uniquely determined by the position in the list as the length of the tail. For terms, a concrete variable y corresponds to `tvar true` y, and an abstract variable z corresponds to `tvar false` z. Similarly, in types, a concrete variable y corresponds to `TVar true` y, and an abstract variable z corresponds to `TVar false` z. Like in the paper, the y identifiers map to the store ρ and the z identifiers to the context Γ.

In addition, for types, we use a locally-nameless de Bruijn convention for variables under dependent types so that it's easy to substitute binders without variable capture. A variable x bound in T^x by a recursive type $\{x \Rightarrow T^x\}$ or a method member $m(x : S) : T^x$ is represented by `TVarB` *i* where *i* is the de Bruijn level, i.e. the number of other binders in scope in between a bound variable occurrence and its binder.

The labels L and m are not part of the Coq syntax for member initialization (`dty` and `dfun`), so that label disjointness is by design. Labels are auto-assigns from 0 at the right to $n-1$ at the left, for a sequence of length n. (Viewed as a list of right-hand sides, this is the same naming convention as for environments above.)

A.1.2 Small-Step Operational Semantics (Figure 2)

`step` $\rho\ t\ \rho'\ t'$	$\rho\ t\ \rightarrow\ t'\ \rho'$	Reduction

The `step` relation makes explicit the two congruence cases (`ST_App1`, `ST_App2`) of the reduction semantics.

A.1.3 Type System (Figures 1 & 2)

`stp` $\Gamma\ \rho\ S\ U\ n$	$\rho\ \Gamma\ \vdash\ S <: U$	Subtyping
`has_type` $\Gamma\ \rho\ t\ T\ n$	$\rho\ \Gamma\ \vdash\ t : T$	Typing
`dms_has_type` $\Gamma\ \rho\ \overline{d}\ T\ n$	$\rho\ \Gamma\ \vdash\ d : T$	Member Initialization
`htp` $\Gamma\ \rho\ x\ T\ n$	$\rho\ \Gamma\ \vdash\ x :_! T$	Abstract Variable Typing (for Subtyping)

The argument *n* at the end of each judgement denotes the *size* of derivation.

The relation `dms_has_type` is used in the premise of both (TNEW) mapping to `T_Obj` and (TLOC) mapping to `T_Vary`. It folds in member initialization, as it recursively maps over the sequence of member definitions. The resulting sequence of intersected types is associated to the right. For uniformity, an empty sequence adds an inner most $\top$ type.

The context restriction in subtyping type selection in rules (SEL1)/(SEL2) mapping to `stp_sel1`/`stp_sel2` is folded into the relation `htp`.

As we mention in Section 3, we omit routine well-formedness checks from the rules on paper for readability. In Coq, these correspond to `closed` conditions, which ensure that all the variables in a type are well-bound for the given environment and binding structure. The relation `closed` $|\Gamma|\ |\rho|\ k\ T$ ensures that `T` is well-bound in a context Γ, a store ρ and under at most $\leq k$ binders.

A.1.4 Type-Checked Examples (`dot_exs.v`)

As a sanity check, we ensure that our model can indeed type-check some intended examples, including a module for covariant lists as presented in Section 2.

A.2 Soundness Proofs (`dot.v, dot_soundness.v, dot_soundness_alt.v`)

The file `dot_soundness.v` presents the main development of the soundness proof, as presented in Section 6. The file `dot_soundness_alt.v` presents the alternative development of the soundness proof, briefly sketched in Section 6.6. In addition to the model, the file `dot.v` contains common infrastructure and lemmas.

A.2.1 Definitions

1. (Type Safety) – see also Theorem 1 – corresponds to `Theorem type_safety`.
2. (Precise Subtyping) corresponds to `Inductive stpp`.
3. (VARPACK Metric) corresponds to `Inductive htpy` as $\rho\ \Gamma\ \vdash_{\leq m}\ y : S$ maps to `htpy` $m\ \rho\ y\ S$ with $\Gamma = \emptyset$.
4. (Substitution) corresponds to `Definition Subst`.
5. (Concrete Variable Type Assignment) corresponds to `Inductive vtp`.
6. (V-BIND Metric) is built-into `Inductive vtp` as $\rho\ \vdash_{\leq m}\ y :_! S$ maps to `vtp` $m\ \rho\ y\ S\ n$.
7. (Widening) corresponds to `Lemma vtp_widen`.

A.2.2 Lemmas

1. (Regularity) corresponds to `Lemma stpd_reg1` and `Lemma stpd_reg2`.
2. (Subtyping Reflexivity) corresponds to `Lemma stpd_refl`.
3. (Narrowing) corresponds to `Lemma stp_narrow`.
4. (Transitivity Pushback) corresponds to `Lemma stp_trans_pushback`.
5. (Inversion of Subtyping) corresponds to Coq's `Inversion` after pushing back from `stp` to `stpp`.
6. (Pre-canonical Forms for Recursive Types) corresponds to `pre_canon_bind`.
7. (Pre-canonical Forms for Type Members) corresponds to `pre_canon_typ`.
8. (Substitution for $<:$) corresponds to the first projection in `Lemma subst_aux`. See also `Lemma stp_subst` and `Lemma all_Subst`.
9. (Substitution for $:_!$) corresponds to the second projection in `Lemma subst_aux`.
10. (Substitution for :) corresponds to the third projection of `Lemma subst_aux`.
11. (Substitution in Term Typing) corresponds to `Lemma hastp_subst`.
12. (Canonical Forms for Type Members) corresponds to `Lemma canon_typ`.
13. (Canonical Forms for Method Members) corresponds to `Lemma canon_fun`.
14. (Substitution for Subtyping) corresponds to `Lemma stp_subst_narrow0`.
15. (Substitution for Abstract Variable Typing) corresponds to the helper lemma `htp_subst_narrow02` within `htp_subst_narrow0`.
16. (Substitution for Concrete Variable Typing) corresponds to `Lemma stp_subst_narrowX`.
17. (Concrete Variable Widening) corresponds to `Lemma vtp_widen`.
18. (Concrete Variable Typing Inversion) corresponds to `Lemma hastp_inv`.