Towards Algorithmic Typing for DOT (Short Paper)

Abel Nieto
University of Waterloo
Waterloo, ON, Canada
anietoro@uwaterloo.ca

Abstract

The Dependent Object Types (DOT) calculus formalizes key features of Scala. The $D_{<:}$ calculus is the core of DOT. To date, presentations of $D_{<:}$ have used declarative, as opposed to algorithmic, typing and subtyping rules. Unfortunately, algorithmic typing for full $D_{<:}$ is known to be an undecidable problem.

We explore the design space for a restricted version of $D_{<:}$ that has decidable typechecking. Even in this simplified $D_{<:}$, algorithmic typing and subtyping are tricky, due to the "bad bounds" problem. The Scala compiler bypasses bad bounds at the cost of a loss in expressiveness in its type system. Based on the approach taken in the Scala compiler, we present the *Step Typing* and *Step Subtyping* relations for $D_{<:}$. These relations are sound and decidable. They are not complete with respect to the original $D_{<:}$ typing rules.

CCS Concepts • **Software and its engineering → General programming languages**;

Keywords Scala, dependent object types, DOT calculus, algorithmic typing

ACM Reference Format:
Abel Nieto. 2017. Towards Algorithmic Typing for DOT (Short Paper). In *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3136000.3136003

1 Introduction

Would you rather have a typechecker that is run by the computer but is sometimes wrong, or one that is always right but needs to be run by hand?

"I want to have my cake and eat it too", you say. That is going to be difficult. On the one hand, the Scala compiler implements a typechecking algorithm that accepts or rejects Scala programs, but is occasionally wrong due to bugs. On the other hand, the DOT calculus is type-safe [Amin et al. 2016], but its typing rules can only be run manually via a proof assistant.

Why manually? The problem is that the typing rules are not syntax-directed, so an algorithm cannot be easily derived from them. For example, take the transitivity rule for subtyping, present in many calculi (DOT included):

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \quad (\textsc{Trans})$$

For a theorem prover this rule is no problem: get the human to provide a T for which the premises are satisfied, and then we can conclude $\Gamma \vdash S <: U$. For an algorithm, it is harder: how should it guess the right T? Iterating over the infinitely many possibilities is not an option.

The standard solution is to tweak the inference rules in such a way that the problematic rule is merged with others, becoming less general but more tractable. Here is how Kernel $F_{<:}$ [Cardelli et al. 1994] merges transitivity with type variable lookup:

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T} \quad (\textsc{Trans-TVar})$$

This is better: to determine whether X is a subtype of T, the typechecker can look up X in Γ, obtain the upper bound U, and recursively check whether $\Gamma \vdash U <: T$.

In this paper, we describe our work in progress towards algorithmic typing for $D_{<:}$ [Amin et al. 2016], a simple calculus that is the core of DOT. Our quest gets off to a bad start: $D_{<:}$ is a generalization of $F_{<:}$, the polymorphic lambda calculus with subtyping. Typing $F_{<:}$ is undecidable [Pierce 1994], which makes typing $D_{<:}$ also undecidable [Rompf and Amin 2015].

There is still hope, though. There are simpler versions of $F_{<:}$ with decidable typechecking. Could they be used as the basis for a simpler $D_{<:}$ with algorithmic typing rules?

This paper makes three contributions. First, we describe how even when the original source of undecidability is eliminated, the problem of bad bounds complicates algorithmic typing and subtyping of $D_{<:}$ (Section 2).

If bad bounds are so hard to deal with, how does the Scala compiler handle them? In fact, it does not. In Section 3, we show how the Scala compiler sidesteps the bad bounds problem by using a subtyping relation that is not transitive.

x, y, z	**Variable**	$S, T, U ::=$	**Type**
$v ::=$	**Value**	$\top$	top type
$\{A = T\}$	type tag	$\bot$	bottom type
$\lambda(x : T)t$	lambda	$\{A\colon S..T\}$	type declaration
$s, t, u ::=$	**Term**	$x.A$	path-dependent type
x	variable	$\forall(x : S)T$	dependent function
v	value		
$x\ y$	application		
let $x = t$ **in** u	let		

Figure 1. Abstract syntax of $D_{<:}$ [Amin et al. 2016]

Finally, in Section 4 we introduce the Step Typing and Subtyping relations. Step Typing and Subtyping are sound and decidable, but not complete, with respect to $D_{<:}$'s standard relations.

2 Bad Bounds

Pierce [2002] presents a design recipe for coming up with algorithmic typing rules for a calculus. We start with a set of declarative typing rules. We then modify the rules so that they are all syntax-directed, and prove them sound with respect to the declarative rules. Finally, we prove a *minimality* result: if a term can be typed, the algorithmic rules will type it with the most precise type. $\Gamma \vdash t\colon U \implies \Gamma \vdash_A t\colon T \wedge \Gamma \vdash T <: U$, where $\vdash_A$ is the algorithmic typing relation.

Below, we conjecture that there does not exist an algorithmic typing relation for $D_{<:}$ that satisfies the minimality condition.

$D_{<:}$ has a restricted form of types as values (Figure 1). A type tag $\{A = T\}$ defines A as an alias for T, and its type is the type declaration $\{A\colon T..T\}$. A type declaration $\{A\colon S..T\}$ can also have different lower and upper bounds, indicating that A is any type between S and T. If a variable x is mapped to a type declaration $\{A\colon S..T\}$ in the current type environment, then we can refer to the path-dependent type $x.A$: $\lambda(x : \{A\colon S..T\}) \ldots x.A \ldots$

A path-dependent type is related to the lower and upper bounds in its type declaration via subtyping:

$$\frac{\Gamma \vdash x\colon\ \{A\colon S..T\}}{\Gamma \vdash S <: x.A}\ (\textsc{<:-Sel}) \qquad \frac{\Gamma \vdash x\colon\ \{A\colon S..T\}}{\Gamma \vdash x.A <: T}\ (\textsc{Sel-<:})$$

Now notice what happens when <:-Sel and Sel-<: are combined with Trans, in a well-typed term $\lambda(e : \{E\colon \top..\bot\})\ \lambda(f : \top)\ \lambda(x : \top)\ f x$

How can $f x$ be well-typed, when f has type $\top$? The reason is that the application is typed in an environment where $\top <: e.E$ and $e.E <: \bot$, which means that $\top <: \bot$, because of transitivity. The entire type lattice collapses, so f can also be assigned type e.g. $\top \to \top$, making $f x$ type-correct.

In effect, a type declaration introduces not only a subtyping relation between a path-dependent type and its bounds, but also a subtyping relation between the bounds themselves. Amin et al. [2016] refers to these "strange" type declarations as having "bad bounds". Bad bounds affect minimality because a term can now be typed with two different types, neither of which is a subtype of the other. This leads us to the conjecture below.

Conjecture 2.1 (Impossibility of minimal typing). *Let* $\Gamma \vdash t\colon T$ *and* $\Gamma \vdash T <: U$ *be the typing and subtyping relations for* $D_{<:}$. *There does not exist a function* $\Gamma \vdash_A t\colon T$[1] *such that* $(\Gamma \vdash_A t\colon T \implies \Gamma \vdash t\colon T) \wedge (\Gamma \vdash t\colon U \implies \Gamma \vdash_A t\colon T \wedge \Gamma \vdash T <: U)$.

To see why the conjecture should be true, suppose such a function $\vdash_A$ exists. Now consider the term

$$\begin{aligned} A \equiv\ & \lambda(e : \{E\colon \forall(b : B)B..\forall(b : B)C\}) \\ & \quad \textbf{let } f = \lambda(b : B)b \textbf{ in} \\ & \quad\quad \textbf{let } b = \{V = \top\} \textbf{ in } f b. \end{aligned}$$

where B and C are syntactic abbreviations for types: $B \equiv \{V\colon \top..\top\}, C \equiv \{Z\colon \top..\top\}$.

Term A is a lambda abstraction. The argument e to the lambda is annotated with type $\{E\colon \forall(b : B)B..\forall(b : B)C\}$. This means that in the body of A we have $\forall(b : B)B <: \forall(b : B)C$. Since functions are covariant in their return type, we would usually need $B <: C$, but in this case B and C are type declarations with different labels (V and Z, respectively), so neither should be a subtype of the other. This means that e has bad bounds.

Within the body of A, f can be typed as $\forall(b : B)B$, but, because of the bad bounds (and subsumption), also as $\forall(b : B)C$. In turn, b has type $\{V\colon \top..\top\} = B$. In the application $f b$, we can give f the type $\forall(b : B)B$, in which case the type of the entire application will be B, or f can have type $\forall(b : B)C$, giving the application type C. The problem is that neither B nor C is a subtype of the other, making it unlikely that there is a minimal type.

More formally, while typechecking A, we will eventually descend into the environment

$$\Gamma_\star = e : \{E\colon \forall(b : B)B..\forall(b : B)C\}$$

The introduced bad bounds ensure $\Gamma_\star \vdash \forall(b : B)B <: \forall(b : B)C$. If w denotes the body of the lambda, $\Gamma_\star \vdash w\colon B$ and $\Gamma_\star \vdash w\colon C$.

Minimality implies $\Gamma_\star \vdash_A w\colon T$, with $\Gamma_\star \vdash T <: B$. By Lemma 2.1, $T = \bot$ or $T = \{V\colon V_1..V_2\}$. Similarly, $\Gamma_\star \vdash T <: C$, which means $T = \bot$ or $T = \{Z\colon Z_1..Z_2\}$. This means $T = \bot$. Because $\vdash_A$ is sound, we must have $\Gamma_\star \vdash w\colon \bot$, which does not seem like an obtainable judgment.

Lemma 2.1. $\Gamma_\star \vdash T <: \{X\colon X_1..X_2\} \implies T = \bot \vee T = \left\{X\colon X_1'..X_2'\right\}$.

It is not clear that the impossibility result, if true, carries over to DOT, because DOT has intersection types. A typing

[1]Notice $\Gamma \vdash_A t\colon T$ is a function, and not simply a relation. Therefore, the (Γ, t) pair is mapped to at most one type.

function for DOT might be able to produce the judgment $\Gamma_\star \vdash_A w\colon B \wedge C$, and $B \wedge C$ is plausibly a minimal typing for w.

2.1 Rejecting Bad Bounds

Now that we have seen how bad bounds lead to unintuitive behaviour and affect minimality, it is tempting to try to detect and reject types with bad bounds before they make it into the type environment.

One possible approach is to define a well-formedness relation on types $\Gamma \vdash T$ wf that identifies which types are free of bad bounds. Type declarations are well-formed if their bounds can be checked to be satisfiable:

$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash \{A\colon S..T\}\ \text{wf}} \quad \text{(WF-Decl)}$$

The rest of the types are all either trivially well-formed (e.g. $\top$) or are well-formed if their components are (function types).

There are at least two problems with this solution. First, it does not generalize well to recursive and intersection types, as explained in Amin et al. [2016]; Rapoport et al. [2017]. Additionally, even within $D_{<:}$, we can no longer type declarations where the bounds are abstract, leading to a loss in expressivity:

$\lambda(x : \{A\colon \bot..\top\})$
$\quad \lambda(y : \{B\colon \bot..\top\})$
$\quad\quad \lambda(z : \{C\colon x.A..y.B\})z$

The term above can no longer be typed if we require $\{C\colon x.A..y.B\}$ to be well-formed, because we will not have enough information to prove that $x.A <: y.B$ until x and y are instantiated.

3 Typing Scala

If bad bounds cause so much trouble, how does the Scala compiler[2] manage to typecheck them? In fact, Scala avoids dealing with bad bounds by restricting its subtyping relation to not be transitive.

Consider the example code below, which is a Scala version of our counterexample for minimality of $D_{<:}$. In $D_{<:}$, the code would typecheck, because the bounds on the abstract type declaration mean that Int => Int <: Int => String.

However, the snippet does not typecheck in Scala: there is no transitivity of subtyping!

```
trait BadBounds {
  type E >: Int => Int <: Int => String
  val f: Int => Int = (x) => x
  val f2: Int => String = f // Type Mismatch Error
  /* found: (Int => Int); required: Int => String */}
```

[2]By the "Scala compiler", we mean a June 2017 version of Dotty. In most cases, scalac behaves similarly to Dotty.

The information about the lower and upper bounds is not entirely lost. The compiler still uses it, but only when one of the two types in the subtype check is the abstract type. This patched-up version of the code typechecks:

```
trait BadBounds {
  type E >: Int => Int <: Int => String
  val f: Int => Int = (x) => x
  val e: E = f
  val f2: Int => String = e}
```

Here, the two subtype checks executed are $Int => Int <: E$ and $E <: Int => String$. Both of these involve E directly, so the type bounds are considered during the check. Indeed, inspection of the Dotty code shows it runs an algorithm similar to the one below (TB stands for "type bounds"):

```
def sub(t1: Type, t2: Type): Boolean = {
  val f = t2 match { case TB(l2, u2) => sub(t1, l2 )...}
  f || t1 match { case TB(l1, u1) => sub(u1, t2 )...}}
```

In addition to dropping transitivity, Scala's handling of bad bounds takes exponential time in the worst case. Let P_N denote the following program:

trait P_N {
  type $T_1 <: T_2$; ...; **type** $T_{N-1} <: T_N$; **type** T_N
  type $T_{2*N} >: T_{2*N-1}$; ...; **type** $T_{N+2} >: T_{N+1}$; **type** T_{N+1}
  val $v_1 : T_1$; **val** $v_{2*N} : T_{2*N} = v_1$}

Notice that $T_1 <: T_N$ via a chain of N upper bounds that are discoverable by the subtyping algorithm. The same holds for $T_{N+1} <: T_{2*N}$ via lower bounds. However, T_N is not a subtype of T_{N+1}, so a subtype check $\text{sub}(T_1, T_{2*N})$, triggered by the assignment val $v_{2*N} : T_{2*N} = v_1$, will fail only after at least N nested recursive calls. Since all the calls are eventually unsuccessful, this means there are at least 2^N recursive calls. We have experimentally verified that the compilation time increases exponentially for this family of programs.

4 Formalization

In this section, we present Step Typing and Step Subtyping, which form a sound, decidable typechecking algorithm for a subset of $D_{<:}$.

4.1 Exposure

The exposure relation $\Gamma \vdash T \Uparrow T'$ maps a type T to a supertype T' that is not path-dependent. If T is not a path-dependent type to start with, exposure behaves as the identity function. If T is the path-dependent type $x.A$, exposure traverses the type environment Γ, starting with the type of x, until it finds an upper bound that is not path-dependent, which it then returns:

$$\frac{\Gamma(x) = T \qquad \Gamma \vdash T \Uparrow \{A\colon S..U\} \qquad \Gamma \vdash U \Uparrow V}{\Gamma \vdash x.A \Uparrow V} \quad \text{(X-Path)}$$

A sample judgment would be: $x : \{A\colon \bot..\forall(z : \top)\top\}, y : \{B\colon \bot..x.A\} \vdash y.B \Uparrow \forall(y : \top)\top$. As is the case with some of the relations we will define later on, $\bot$ needs special treatment:

$$\frac{\Gamma(x) = T \qquad \Gamma \vdash T \Uparrow \bot}{\Gamma \vdash x.A \Uparrow \bot} \quad \text{(X-Bot)}$$

Exposure is used in places where the typechecker sees a path-dependent type, but needs a supertype of it that is a function or a type declaration. We base our exposure relation in both the exposure operation present in Kernel $F_{<:}$ [Pierce 2002] and the treatment of type bounds in Scala. Exposure preserves subtyping ($\Gamma \vdash T \Uparrow T' \implies \Gamma \vdash T <: T'$), and terminates.

4.2 Promotion and Demotion

The promotion relation $\Gamma \vdash T \Uparrow^x T'$ maps a type T to a supertype T' such that $x \notin fv(T')$. If $T = x.A$, then promotion looks up x in the environment, and then uses exposure to find a suitable upper bound U:

$$\frac{\Gamma(x) = T \qquad \Gamma \vdash T \Uparrow \{A\colon L..U\}}{\Gamma \vdash x.A \Uparrow^x U} \quad \text{(P-Up)}$$

Two points of note: we know that $x \notin fv(U)$ because $D_{<:}$ environments (unlike DOT's) are acyclic in a certain sense we do not fully define here. Second, in the premises, we do not simply expose $x.A$ ($\Gamma \vdash x.A \Uparrow T'$): that would make T' not a path-dependent type, which is too strong of a condition; we only need that T' does not contain x.

When promoting function types, we need to account for the argument being contravariant. This hints at the need of a relation that is the dual of promotion. The demotion relation $\Gamma \vdash T \Downarrow^x T'$ fulfills this role: it maps a type T to a subtype T' such that $x \notin fv(T')$. For every promotion inference rule, there is a corresponding one in the demotion relation. Promotion and demotion are combined to handle function types:

$$\frac{\Gamma \vdash S \Downarrow^x S' \qquad \Gamma, y : S' \vdash T \Uparrow^x T' \qquad y \neq x}{\Gamma \vdash \forall(y : S)T \Uparrow^x \forall(y : S')T'} \quad \text{(P-Lam)}$$

Notice how the argument type is demoted, while the result type is promoted. Type declarations are treated similarly. Finally, here is a sample promotion judgment: $x : \{A\colon \bot..\top\} \vdash \forall(y : x.A)x.A \Uparrow^x \forall(y : \bot)\top$.

Promotion and and demotion are adapted from the same-named relations present in Pierce and Turner [2000]. They remove all occurrences of the specified free variable ($\Gamma \vdash T \Uparrow^x T' \vee \Gamma \vdash T \Downarrow^x T' \implies x \notin fv(T)$), preserve subtyping ($\Gamma \vdash T \Uparrow^x T' \implies \Gamma \vdash T <: T' \wedge \Gamma \vdash T \Downarrow^x T' \implies \Gamma \vdash T' <: T$), and terminate.

4.3 Step Typing

We can now define Step Typing. Step Typing builds on the standard typing relation defined in Amin et al. [2016].

There are two rules in the standard typing relation that are not syntax-directed: Sub, which is needed when typing function applications, and Let, for typing let-expressions:

$$\frac{\Gamma \vdash t\colon T \qquad \Gamma \vdash T <: U}{\Gamma \vdash t\colon U} \quad \text{(Sub)} \qquad \frac{\Gamma \vdash t\colon T \qquad \Gamma, x : T \vdash u\colon U \qquad x \notin fv(U)}{\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ u : U\colon} \quad \text{(Let)}$$

Sub is not syntax-directed because the typechecker needs to "guess" the type U in the conclusion. Similarly, Let forces us to guess a type U where x is not free.

To fix these issues, Step Typing differs from the standard typing relation in two ways. First, it drops the subsumption rule: instead, when typing a function application it uses exposure to find a function type for the term in the function position:

$$\frac{\Gamma \vdash_S x\colon V \qquad \Gamma \vdash V \Uparrow \forall(z\colon S)T \qquad \Gamma \vdash_S y\colon U \qquad \Gamma \vdash_S U <: S}{\Gamma \vdash_S x\ y\colon [z := y]T} \quad \text{(T-All-E)}$$

Second, to make Let syntax-directed it uses promotion to remove all references to the bound variable in the returned type of a let-expression:

$$\frac{\Gamma \vdash_S t\colon T \qquad \Gamma, x\colon T \vdash_S u\colon U' \qquad \Gamma, x\colon T \vdash U' \Uparrow^x U}{\Gamma \vdash_S \textbf{let}\ x = t\ \textbf{in}\ u\colon U} \quad \text{(T-Let)}$$

4.4 Step Subtyping

The standard subtyping relation requires three changes: the first two to make the rules syntax-directed, and the last one to guarantee termination.

We drop the general reflexivity rule, replacing it with reflexivity of only path-dependent types. General reflexivity still holds, just not as an axiom:

$$\Gamma \vdash_S x.A <: x.A \quad \text{(S-Refl)}$$

Transitivity goes away: instead, whenever we are doing a subtype check involving a path-dependent type, we use exposure to find a lower or upper bound for it, and then recursively continue the subtype check on the bound. The upper bound case looks like

$$\frac{\Gamma(x) = T \qquad \Gamma \vdash T \Uparrow \{A\colon S_1..S_2\} \qquad \Gamma \vdash_S S_2 <: U}{\Gamma \vdash_S x.A <: U} \quad \text{(S-<:-Sel)}$$

Notice how, once again, we do not apply exposure directly to $x.A$: otherwise, we might miss some cases where S_2 is a path-dependent type by "overshooting". For example, if $\Gamma = x : \{A\colon \bot..\top\}, y : \{B\colon \bot..x.A\}$, we can derive $\Gamma \vdash_S y.B <:$

$x.A$ by a combination of S-<:-Sel and Refl, but $\Gamma \vdash y.B \Uparrow \top$, and we cannot derive $\Gamma \vdash_S \top <: x.A$.

Finally, so that the algorithm terminates, we only allow subtyping between function types with the same argument type (as opposed to the standard contravariant rule).

$$\frac{\Gamma, x\colon S \vdash_S T_1 <: T_2}{\Gamma \vdash_S \forall(x\colon S)T_1 <: \forall(x\colon S)T_2}\ \text{(S-All-<:-All)}$$

This is the same restriction used to make Kernel $F_{<:}$ decidable [Cardelli and Wegner 1985]. Even though Step Subtyping always terminates, it takes exponential time in the worst case to typecheck certain terms, like the P_N example of Section 3.

4.5 Metatheoretic Properties

Below, we summarize the metatheoretic properties of Step Typing and Subtyping. Proofs of these theorems can be found in the accompanying technical report [Nieto 2017].

Soundness: $\Gamma \vdash_S t\colon T \implies \Gamma \vdash t\colon T \land \Gamma \vdash_S S <: U \implies \Gamma \vdash S <: U$

Decidability: both relations are decidable. For Step Typing, we can use the size of the term being typed as a termination measure. For Step Subtyping, the termination measure is a weight function on types similar to the one in Pierce [2002] for Kernel $F_{<:}$.

Completeness: the relations are not complete. Any program that relies on a combination of bad bounds and transitivity to typecheck will fail to do so.

Subject Reduction: we do not currently know whether the subject-reduction property holds for Step Typing. This means we could have $\Gamma \vdash_S t\colon T$ and $t \longmapsto t'$, but t' can only be typed under the standard typing relation, and not Step Typing.

5 Related Work

$F_{<:}$: Pierce [1994] showed that algorithmic subtyping for $F_{<:}$ is undecidable. Kernel $F_{<:}$ [Cardelli and Wegner 1985] introduced the exposure operation. Pierce and Turner [2000] use the promotion and demotion operations to do local type inference on Kernel $F_{<:}$

$D_{<:}$: Rompf and Amin [2015] introduced $D_{<:}$ and proved it type-safe. The version of $D_{<:}$ we use comes from Amin et al. [2016], and uses ANF and small-step semantics.

DOT: on top of $D_{<:}$, DOT adds features like recursive and intersection types. There are many presentations of DOT [Amin et al. 2016, 2012, 2014; Rapoport et al. 2017; Rompf and Amin 2015], but they all use declarative typing rules.

Featherweight Scala: Cremet et al. [2006] introduced Featherweight Scala (FS_{alg}), which formalizes a subset of the Scala type system. They show that the calculus has decidable typing and subtyping. FS_{alg} has not been proven type-safe. Featherweight Scala is neither a subset nor a superset of $D_{<:}$, and differs from $D_{<:}$ in multiple ways: it is a class-based calculus with nominal typing and has call-by-name semantics. More relevant to our work, type members in FS_{alg} (which correspond to type declarations and type tags in $D_{<:}$) are either completely abstract (`type A`) or aliases (`type A = T`). It is not possible to assign lower or upper bounds to an abstract type member (`type A >: S <: T`), which is possible both in Scala and $D_{<:}$. Because bounds cannot be specified, it is not possible to create a custom subtyping lattice in FS_{alg}, so there is no bad bounds problem.

Scala: the Scala type system has been shown to be both unsound [Amin and Tate 2016] and undecidable [Bjarnason 2009, 2011]. Because Scala's type system is not formally specified, it is hard to say at any one point in time whether a specific proof of undecidability (or unsoundness) is still valid or not [Odersky 2016].

6 Conclusions

This paper described our work in progress towards a version of $D_{<:}$ with algorithmic typing. We showed how a combination of bad bounds and transitivity make it unlikely that a typing algorithm satisfying the minimality condition exists for $D_{<:}$, even after removing the known source of undecidability. We also showed how the Scala compiler deals with bad bounds by dropping transitivity of subtyping. Finally, we used prior work on decidable versions of $F_{<:}$, as well as the approach taken in the Scala compiler, to develop Step Typing and Subtyping. These relations are sound and decidable, but not complete, with respect to the standard relations.

Is the subset of $D_{<:}$ that Step Typing can type interesting? Maybe. We think a more conclusive answer will depend on whether the subject reduction property holds for Step Typing. Because Step Typing mimics the behaviour of the Scala compiler, we conjecture that the lack of transitivity does not, on its own, mean we cannot type "useful" programs (every single Scala program written to-date has been typed with a similar restriction in place). Additionally, we think that Step Typing might be capable of typing the encoding of $F_{<:}$ in $D_{<:}$ shown in Amin et al. [2016]. This seems plausible because even though the exposure operation does not bring back transitivity of subtyping in $D_{<:}$, the similar exposure operation in $F_{<:}$ does recover transitivity [Pierce 2002].

Future work will involve formalizing Step Typing and Subtyping in a mechanized proof, establishing subject reduction, investigating whether Step Typing is general enough to type the aforementioned encoding of $F_{<:}$, and extending Step Typing to DOT.

Acknowledgments

We would like to thank Marianna Rapoport for multiple helpful discussions about DOT. We would also like to thank Prabhakar Ragde, Ifaz Kabir and Paul He for proofreading this paper. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The Essence of Dependent Object Types*. Springer International Publishing, Cham.

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*.

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*. 233–249.

Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers. In *to appear in OOPSLA 2016*.

Runar Bjarnason. 2009. More Scala Typehackery. (2009). https://apocalisp.wordpress.com/2009/09/02/ Accessed: 2017-07-15.

Runar Bjarnason. 2011. Simple SKI Combinator Calculus in Scala's Type System. (2011). https://apocalisp.wordpress.com/2011/01/13/simple-ski-combinator-calculus-in-scalas-type-system/ Accessed: 2017-07-15.

Luca Cardelli, Simone Martini, John C Mitchell, and Andre Scedrov. 1994. An extension of system F with subtyping. *Information and Computation* 109, 1-2 (1994), 4–56.

Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 471–523.

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.*

Abel Nieto. 2017. Towards Algorithmic Typing for DOT. *CoRR* abs/1708.05437 (2017). http://arxiv.org/abs/1708.05437

Martin Odersky. 2016. Scaling DOT to Scala — Soundness. http://www.scala-lang.org/blog/2016/02/17/scaling-dot-soundness.html. (2016).

Benjamin C Pierce. 1994. Bounded quantification is undecidable. *Information and Computation* 112, 1 (1994), 131–165.

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Benjamin C Pierce and David N Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 1–44.

Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A Simple Soundness Proof for Dependent Object Types. In *Proceedings of the 2017 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2017*. To appear.

Tiark Rompf and Nada Amin. 2015. From F to DOT: Type Soundness Proofs with Definitional Interpreters. *CoRR* abs/1510.05216v1 (2015). http://arxiv.org/abs/1510.05216v1