A Simple Soundness Proof for Dependent Object Types

MARIANNA RAPOPORT, University of Waterloo, Canada
IFAZ KABIR, University of Waterloo, Canada
PAUL HE, University of Waterloo, Canada
ONDŘEJ LHOTÁK, University of Waterloo, Canada

Dependent Object Types (DOT) is intended to be a core calculus for modelling Scala. Its distinguishing feature is abstract type members, fields in objects that hold types rather than values. Proving soundness of DOT has been surprisingly challenging, and existing proofs are complicated, and reason about multiple concepts at the same time (e.g. types, values, evaluation). To serve as a core calculus for Scala, DOT should be easy to experiment with and extend, and therefore its soundness proof needs to be easy to modify.

This paper presents a simple and modular proof strategy for reasoning in DOT. The strategy separates reasoning about types from other concerns. It is centred around a theorem that connects the full DOT type system to a restricted variant in which the challenges and paradoxes caused by abstract type members are eliminated. Almost all reasoning in the proof is done in the intuitive world of this restricted type system. Once we have the necessary results about types, we observe that the other aspects of DOT are mostly standard and can be incorporated into a soundness proof using familiar techniques known from other calculi.

Our paper comes with a machine-verified version of the proof in Coq.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*;

Additional Key Words and Phrases: Scala, dependent object types, DOT calculus, type safety

ACM Reference Format:
Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A Simple Soundness Proof for Dependent Object Types. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 46 (October 2017), 27 pages. https://doi.org/10.1145/3133870

1 INTRODUCTION

2016 was an exciting year for those who desire a formalism to understand and reason about the unique features of Scala's type system. Mechanized soundness results were published for the Dependent Object Types (DOT) calculus and other related calculi [Amin et al. 2016; Amin and Rompf 2017; Rompf and Amin 2016]. These proofs were the culmination of an elusive search that spanned more than ten years. The chief subtleties and paradoxes inherent in DOT and the Scala type system, which made the proof so challenging, were documented along the way [Amin et al. 2012, 2014].

Since the DOT calculus exhibits such subtle and counterintuitive behaviour, and since the proofs are the result of such a long effort, it is to be expected that the proofs must be complicated. The calculus is dependently typed, so it is not surprising that the lemmas that make up the proofs reason about tricky relationships between types and values. In some contexts, the type system

Authors' addresses: Marianna Rapoport, University of Waterloo, Canada; Ifaz Kabir, University of Waterloo, Canada; Paul He, University of Waterloo, Canada; Ondřej Lhoták, University of Waterloo, Canada.

2475-1421/2017/10-ART46
https://doi.org/10.1145/3133870

admits typings that seem just plain wrong, and give no hope for soundness, so it seems necessary to have lemmas that reason simultaneously about the intricate properties of values, types, and the environments that they inhabit.

A core calculus needs to be easy to extend. Some extensions of DOT are necessary even just to model essential Scala features. As a prominent example, types in Scala may depend on paths $x.a_1.\cdots.a_n.A$ (where x is a variable, a_i are fields, and A is a type member), but types in the existing DOT calculi can depend only directly on variables ($x.A$). Path-dependent types are needed to model essential features such as classes and traits (as members nested in objects and packages) and the famous cake pattern [Odersky and Zenger 2005]. Another important Scala feature to be studied in DOT are implicit parameters. Moreover, language modifications and extensions are the raison d'être of a core calculus. DOT enables designers to experiment with exciting new features that can be added to Scala, to tweak them and reason about their properties before attempting to integrate them in the compiler with the complexity of the full Scala language.

The complexity of the proof is a hindrance to such extension and experimentation. Over the past ten years, DOT has been designed and re-designed to be just right, so that the brilliant lemmas that ensure its soundness hold and can be proven. When the DOT calculus is disrupted by a modification, it is difficult to predict which parts of the proof will be affected. Experimenting with modifications to DOT is difficult because each tweak requires many lemmas to be re-proven.

Our goal in this paper is a soundness proof that is simpler, more modular, and more intuitive. We aim to separate the concepts of types, values, and operational semantics, and to reason about one concept at a time. Then, if a language extension modifies only one concept, such as typing, the necessary changes are localized to the parts of the proof that deal with types. We also aim to isolate most of the reasoning in a simpler system that is immune to the paradox of bad bounds, the key challenge that plagued the long search for a soundness proof. In this system, our reasoning can rely on intuitive notions from familiar object calculi without dependent object types [Abadi and Cardelli 1996; Pierce 2002]. The results of this reasoning are lifted to the full DOT type system by a single, simple theorem.

The main focus of our proof is on types. Dependent object types are the one feature that distinguishes DOT, so we aim to decouple that one feature, which mainly affects the static type system, from other concerns. We focus on proving the properties that one expects of types, and deliberately keep the proof independent of other aspects, such as operational semantics and runtime values, which are similar in DOT as in other object calculi. Of course, a soundness proof must eventually speak about execution and values, but once we have the necessary theory to reason about types, these other concerns can be handled separately, at the end of the proof, using standard proof techniques. Our final soundness theorem is stated for the small-step operational semantics given by Amin et al. [2016], but that is only the final conclusion; the theory that we develop about dependent object types would be equally applicable in a proof for a big-step operational semantics.

In a sense, this paper moves in the opposite direction compared to other recent work related to DOT: this paper aims for a simpler proof of one specific calculus, while other work generalizes DOT with features from other calculi. The most significant addition in Rompf and Amin [2016] is subtyping between recursive types, which requires sophisticated proof techniques and induction schemes, but is not needed to model Scala. Amin and Rompf [2017] focuses on a family of calculi with some features similar to those in DOT, and on general proof techniques applicable to the whole family. While it is useful to generalize and compare DOT to other calculi, that is not the topic of this paper. This paper focuses inwards, on DOT itself, on only those features of DOT that are necessary for modelling Scala, with an aim to make the soundness proof of those specific features as simple and modular as possible.

The power of DOT is also its curse. DOT empowers a program to define a domain-specific type system with a custom subtyping lattice inside the existing Scala type system. This power has been used to encode in plain Scala expressive type systems that would otherwise require new languages to be designed. But this power also enables typing contexts that make no sense, in which types cannot be trusted and thus become meaningless. For example, a program could define typing contexts in which an object, which is not a function, nevertheless has a function type. Since such "crazy" contexts are possible, a soundness proof needs to consider them (but prove that they are harmless during execution).

Besides the general pursuit of modularity, the simplicity of our new proof depends on two main ingredients.

The first ingredient is *inert types* and *inert typing contexts*, which we will define in Section 3.2. The essential property of an inert type is that if all variables have inert types, then no unexpected subtyping relationships are possible, so types can be trusted, and none of the paradoxes are possible. We express this property more formally in Section 3.2. An important part of the soundness proof is to ensure that a term cannot evaluate until the types of all its free variables have been narrowed to inert types.

We define inertness as a concise, easily testable syntactic property of a type. The definition consists of only two non-recursive inference rules, so it can be easily inverted when it occurs in a proof. By contrast, existing DOT proofs achieve similar goals using properties that characterize types by the existence of values with specific relationships to those types. The benefit of our inertness property is that it involves only a type, not any values, and it is defined directly, not via existential quantification of some corresponding value.

The second ingredient is tight typing, a small restriction of the DOT typing rules with major consequences, which we will discuss in Section 3.3. We did not invent tight typing; it appears as a technical definition in the proof of Amin et al. [2016]. Our contribution is to identify and demonstrate just how useful and important tight typing is to a simple proof. Amin et al. [2016] use tight typing in a collection of technical lemmas mixed with reasoning about other concerns, such as general typing (the full typing rules of DOT) and correspondences between values and types. In our proof, however, tight typing takes centre stage; it is the main actor that enables intuition and simplicity.

Tight typing neutralizes the two DOT type rules that enable a program to define custom subtyping relationships. Tight typing immunizes the calculus: even if a typing context contains a type that is not inert, tight typing prevents it from doing any harm. The paradoxes that make it challenging to work with DOT disappear under tight typing. Without those two typing rules, the calculus behaves very differently, like object calculi without dependent object types, and our reasoning can rely on familiar properties that we are used to from these calculi.

Of course, DOT with tight typing is not at all the real DOT: it lacks the power to create customized type systems, and it is uninteresting; it is just another calculus with predictable behaviour. Theorem 3.3 in Section 3.3 bridges the gap by showing that in inert contexts, tight typing has all the power of general typing. Therefore, all the reasoning that we do in the intuitive environment of tight typing applies to the full power of DOT. Even our proof of Theorem 3.3 itself reasons entirely with tight typing, without having to deal with the paradoxes of general DOT typing, and without having to reason about relationships between types and values.

Combining these two ingredients, we contribute a unified general recipe that can be used whenever a proof about DOT needs to deduce information about a term from its type. Many of our lemmas follow this recipe. The first step of the recipe, which should be the first step of any reasoning about types in DOT, is to drop down from general typing to tight typing using Theorem 3.3. The purpose of the remaining steps is to make inductive reasoning as easy and systematic as possible.

Contributions. This paper presents a simplified and extensible soundness proof for the DOT calculus [Amin et al. 2016]. We contribute the following:

- A *modular* proof that reasons about types, values, and operational semantics separately.
- The concept of *inert* typing contexts, a syntactic characterization of contexts that rule out any non-sensical subtyping that could be introduced by abstract type members.
- A simple *proof recipe* for deducing properties of terms from their types in full DOT while reasoning only in a restricted, intuitive environment free from the paradoxes caused by abstract type members. Multiple lemmas follow the same recipe, and following the recipe can facilitate the development of new lemmas needed in future extensions for DOT.
- A *Coq formalization* of the DOT soundness proof presented in this paper.

Our Coq proof can be found at

https://git.io/simple-dot-proof

The rest of this paper is organized as follows. Section 2 describes the DOT type system and explains the problem of bad bounds, which is responsible for the complexity in DOT soundness proofs. Section 3 presents a detailed description of the simplified DOT soundness proof introduced in this paper. Section 4 summarizes the overall proof structure and explains how to extend the proof with new DOT features. Section 5 continues the discussion of the bad-bounds problem. Section 6 examines related work. We finish with concluding remarks in Section 7.

2 BACKGROUND

The proof in this paper proves type soundness of the variant of the DOT calculus defined by Amin et al. [2016].

2.1 DOT Syntax

		d ::=	**Definition**
x, y, z	**Variable**	$\{a = t\}$	field definition
a, b, c	**Term member**	$\{A = T\}$	type definition
A, B, C	**Type member**	$d \wedge d'$	aggregate definition
s, t, u ::=	**Term**	S, T, U ::=	**Type**
x	variable	$\forall(x\colon S)T$	dependent function
v	value	$\mu(x\colon T)$	recursive type
$x.a$	selection	$\{a\colon T\}$	field declaration
$x\,y$	application	$\{A\colon S..T\}$	type declaration
let $x = t$ in u	let binding	$x.A$	type projection
v ::=	**Value**	$S \wedge T$	intersection
$\lambda(x\colon T).t$	lambda	$\top$	top type
$\nu(x\colon T)d$	object	$\bot$	bottom type

Fig. 1. Abstract syntax of DOT [Amin et al. 2016]

We begin by describing the abstract syntax of the calculus, shown in Figure 1. The calculus defines two forms of *values*:

– A *lambda abstraction* $\lambda(x\colon T).t$ is a function with parameter x of type T and a body consisting of the term t.
– An *object* of type T with definitions d is denoted as $\nu(x\colon T)d$. The body of the object consists of the definitions d, which are a collection of field and type member definitions, connected through the intersection operator. The field definition $\{a = t\}$ assigns a term t to a field labeled a, and the type definition $\{A = U\}$ defines the type label A as an alias for the type U. The object also explicitly declares a recursive *self*, or "this", variable x. As a result, both T and d can refer to x.

A DOT *term* is a variable x, value v, field selection $x.a$, function application $x\,y$, or let binding $\mathsf{let}\ x = t\ \mathsf{in}\ u$. To keep the syntax simple, the DOT calculus uses administrative normal form (ANF); as a result, field selection and function application can involve only variables, not arbitrary terms.

A DOT *type* can be one of the following:

– A *dependent function* type $\forall(x\colon S)T$ is the type of a function with a parameter x of type S, and with the return type T, which can refer to the parameter x.
– A *recursive type* $\mu(x\colon T)$ declares an object type T which can refer to its self-variable x.
– A *field declaration* $\{a\colon T\}$ states that the field labeled a has type T.
– A *type declaration* $\{A\colon S..T\}$ specifies that an abstract type member A is a subtype of T and a supertype of S.
– A *type projection* $x.A$ is the type assigned to the type member labelled A of the object x (ANF allows type projection only on variables).
– An *intersection* type $S \wedge T$ is the most general subtype of both S and T.
– The *bottom* type $\bot$ and the *top* type $\top$ correspond to the bottom and top of the subtyping lattice, and are analogous to Scala's `Nothing` and `Any`.

Examples of DOT programs and their Scala equivalents can be found in Amin et al. [2016].

2.2 DOT Typing Rules

The DOT typing rules (which we will call the "general" typing relation throughout this paper) are presented in Figure 2.

The rules (All-I) and ({}-I) give types to values. An object $\nu(x\colon T)d$ has the recursive type $\mu(x\colon T)$, where the types T must match, and T must summarize the definitions d following the definition typing rules in Figure 2. Note that due to (Def-Typ), each of the type declarations in an object must have equal lower and upper bounds (i.e. an object $\nu(x\colon\ \{A\colon S..U\})\,\{A = T\}$ is only well-typed if $S = U = T$). The rules (Var), (All-E), ({}-E), (Let) give types to the other four forms of terms, and are unsurprising. The recursion introduction (Rec-I), recursion elimination (Rec-E), and intersection introduction (And-I) rules apply only to variables, but the subsumption rule (Sub) applies to all terms. The subtyping rules establish the top and bottom of the subtyping lattice (Top, Bot), define reflexivity and transitivity (Refl, Trans), and basic subtyping rules for intersection types (And$_1$-<:, And$_2$-<:, <:-And). As is commonplace, dependent functions are covariant in the return type and contravariant in the parameter type (All-<:-All). Field typing is covariant by the rule (Fld-<:-Fld), whereas type member declarations are contravariant in the lower bound and covariant in the upper bound via (Typ-<:-Typ). The most interesting rules that distinguish DOT are (<:-Sel) and (Sel-<:), which introduce an object-dependent type $x.A$ and define subtyping between it and its bounds. As we will see, these rules are responsible for much of the complexity of the safety proof.

2.3 Bad Bounds

The type selection subtyping rules (<:-Sel) and (Sel-<:) enable users to define a type system with a custom subtyping lattice. If a program defines a function $\lambda(x\colon\ \{A\colon S..U\}).t$, then t is typed in

Term typing

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \colon T} \quad (\textsc{Var})$$

$$\frac{\Gamma \vdash t \colon T \qquad (\Gamma,\, x \colon T) \vdash u \colon U \qquad x \notin \mathsf{fv}(U)}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u \colon U} \quad (\textsc{Let})$$

$$\frac{(\Gamma,\, x \colon T) \vdash t \colon U \qquad x \notin \mathsf{fv}(T)}{\Gamma \vdash \lambda(x \colon T).t \colon \forall(x \colon T)U} \quad (\textsc{All-I})$$

$$\frac{\Gamma \vdash x \colon T}{\Gamma \vdash x \colon \mu(x \colon T)} \quad (\textsc{Rec-I})$$

$$\frac{\Gamma \vdash x \colon \forall(z \colon S)T \qquad \Gamma \vdash y \colon S}{\Gamma \vdash x\, y \colon\ [^{y}/_{z}]\,T} \quad (\textsc{All-E})$$

$$\frac{\Gamma \vdash x \colon \mu(z \colon T)}{\Gamma \vdash x \colon\ [^{x}/_{z}]\,T} \quad (\textsc{Rec-E})$$

$$\frac{(\Gamma,\, x \colon T) \vdash d \colon T}{\Gamma \vdash \nu(x \colon T)d \colon \mu(x \colon T)} \quad (\{\}\text{-I})$$

$$\frac{\Gamma \vdash x \colon T \qquad \Gamma \vdash x \colon U}{\Gamma \vdash x \colon T \wedge U} \quad (\textsc{And-I})$$

$$\frac{\Gamma \vdash x \colon\ \{a \colon T\}}{\Gamma \vdash x.a \colon T} \quad (\{\}\text{-E})$$

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

Definition typing rules

$$\frac{\Gamma \vdash t \colon U}{\Gamma \vdash \{a = t\} \colon\ \{a \colon U\}} \quad (\textsc{Def-Trm})$$

$$\Gamma \vdash \{A = T\} \colon\ \{A \colon T..T\} \quad (\textsc{Def-Typ})$$

$$\frac{\Gamma \vdash d_1 \colon T_1 \qquad \Gamma \vdash d_2 \colon T_2 \qquad \mathsf{dom}(d_1),\ \mathsf{dom}(d_2)\ \text{disjoint}}{\Gamma \vdash d_1 \wedge d_2 \colon T_1 \wedge T_2} \quad (\textsc{AndDef-I})$$

Subtyping rules

$$\Gamma \vdash T <: \top \quad (\textsc{Top})$$

$$\Gamma \vdash \bot <: T \quad (\textsc{Bot})$$

$$\Gamma \vdash T <: T \quad (\textsc{Refl})$$

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a \colon T\} <: \{a \colon U\}} \quad (\textsc{Fld-<:-Fld})$$

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

$$\Gamma \vdash T \wedge U <: T \quad (\textsc{And}_1\textsc{-<:})$$

$$\Gamma \vdash T \wedge U <: U \quad (\textsc{And}_2\textsc{-<:})$$

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

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

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

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A \colon S_1..T_1\} <: \{A \colon S_2..T_2\}} \quad (\textsc{Typ-<:-Typ})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad (\Gamma,\, x \colon S_2) \vdash T_1 <: T_2}{\Gamma \vdash \forall(x \colon S_1)T_1 <: \forall(x \colon S_2)T_2} \quad (\textsc{All-<:-All})$$

Fig. 2. DOT Type Rules [Amin et al. 2016]

a context in which S is considered a subtype of U, because $S <: x.A <: U$. The soundness proof must ensure that such a user-defined subtyping lattice does not cause any harm, i.e., cannot cause a violation of type soundness of the overall calculus.

Let S be the object type $\{a\colon \top\}$ and U be the function type $\forall(z\colon \top)\top$. Then the following is a valid and well-typed DOT term:

$$\lambda(x\colon \{A\colon S..U\}).\mathsf{let}\ y = \nu(y\colon S)\ \{a = y.a\}\ \ \mathsf{in}\ y\ y$$

How is this possible? The inner term $y\ y$ is a function application applying y to itself, but y is bound by the let to an object, not a function. How can y appear in a function application when it is not a function? This is possible because y has the object type S, and in the body of the lambda, we have the subtyping chain $S <: x.A <: U$. The declaration of the lambda asserts that $x.A$ is a supertype of S and a subtype of U, and therefore introduces the new custom subtyping relationship $S <: U$. Inside the body of the lambda, the object type S is a subtype of the function type U, so since the object y has type S, it also has the function type U. The function application of object y to itself is therefore well-typed in this context.

This is crazy, the reader may be thinking. Indeed, in an environment in which subtyping can be arbitrarily redefined, types cannot be trusted. In particular, we cannot conclude from the fact that y has the function type S that it is indeed a function; actually, it is an object. The seemingly obvious fix is to require S to be a subtype of U when the parameter x of the lambda is declared to have type $\{A\colon S..U\}$. But as we will discuss in Section 5, this seemingly obvious fix does not work, and the struggle to try to make it work has caused much of the difficulty in the ten-year struggle for a DOT soundness proof.

How can DOT be sound then, when it is so crazy? After all, the function application $y\ y$ is well-typed but its evaluation gets stuck, because y is not a function, so how can DOT be sound? The key is that the DOT semantics is call-by-value. In order to invoke the body of the lambda, one must provide an argument value to pass for the parameter x. This value must contain a type assigned to A that is both a supertype of U and a subtype of S. If no such type exists, then no such argument value can exist, so the lambda cannot be called, so its body containing the crazy application $y\ y$ cannot ever be executed. Therefore, this term is not a counterexample to the soundness of the DOT type system.

Why should DOT have such a strange feature? The ability to define a custom subtyping lattice turns out to be very useful. For example, we can define the term:

$$\lambda(x\colon \{A\colon \bot..\top\} \wedge \{B\colon x.A..x.C\} \wedge \{C\colon \bot..\top\}).t$$

In the body t of this lambda, we can make use of unspecified opaque types A, B, and C, making use of only the condition that $A <: B <: C$. We can use this feature to define arbitrary type systems within the language. For example, Scalas and Yoshida [2016] have implemented session types, a feature that usually requires a custom-designed language, inside plain Scala. As another example, Osvald et al. [2016] used this ability to define a lattice of lifetimes within the Scala type system for categorizing values that cannot outlive different stack frames. Even the well-known Scala cake pattern [Odersky and Zenger 2005] is built using this feature.

To reconcile a custom subtyping lattice with a sound language, we only need to force the programmer to provide evidence that the custom lattice does not violate any familiar assumptions (e.g., it does not make object types subtypes of function types). This evidence takes the form of an argument value that must be passed to the lambda before the body that uses the custom type lattice can be allowed to execute. This value must be an object that provides existing types that satisfy the specified custom subtyping constraints. In our example, this is easy: it suffices to pass the same type, such as $\top$, for all three type parameters, since $\top <: \top <: \top$. However, the types are opaque:

when checking the body of the lambda, the type checker cannot use the fact that $A = B = C = \top$; the body must type-check even under only the assumptions that $A <: B <: C$.

Since DOT programs can exhibit unexpected subtyping lattices in some contexts, and since this is unavoidable, an essential feature of a soundness proof is to clearly distinguish contexts in which types can be trusted, because any custom subtyping relationships have been justified by actual type arguments, from contexts in which types cannot be trusted, because they could have been derived from arbitrary unjustified custom subtyping relationships. In Section 3.2, we will formally define this property that types can be trusted, and define a simple syntactic characterization of *inert* typing contexts that guarantee this property. In earlier DOT soundness proofs, the trusted types property was not precisely defined, and typing contexts in which there are no bad bounds were defined more indirectly, not in terms of the types themselves, but in terms of the existence of values having those types.

3 PROOF

3.1 Overview

We will first outline the general recipe that we use to reason throughout the proof about the meaning of a type. The details of each step will be discussed in the following subsections. We present the overview on an example proof of Lemma 3.9, which will be introduced in Section 3.5, but the specific example is unimportant; most of the reasoning throughout the proof follows the same steps, through the same typing relations, in the same order, using the same reasoning techniques.

Usually, we know that some term has some type (e.g. $\Gamma \vdash x\colon \{a\colon T\}$), and we seek to interpret what the type tells us about the term, and to determine how the type of the term was derived. In this example, we seek more detailed information about x, for example that the typing context Γ assigns it an object type $\Gamma(x) = \mu(x\colon \cdots \wedge \{a\colon T'\} \wedge \cdots)$, or the shape of the value that it will hold at run time (e.g. an object $\nu(x\colon \cdots \wedge \{a\colon T'\} \wedge \cdots)(\cdots \wedge \{a = t'\} \wedge \cdots)$).

Each such derivation follows the same sequence of steps (although sometimes only a subsequence of the steps is necessary):

$$
\cfrac{\cfrac{\cfrac{\cfrac{\text{inert } \Gamma \qquad \Gamma \vdash x\colon \{a\colon T\}}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} x\colon \{a\colon T\}}\ \textsc{Theorem 3.3 } (\vdash \text{ to } \vdash_{\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#\#} x\colon \{a\colon T\}}\ \textsc{Theorem 3.6 } (\vdash_{\#} \text{ to } \vdash_{\#\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} x\colon \{a\colon T'\} \qquad \Gamma \vdash T' <: T}\ \textsc{Induction on } \vdash_{\#\#}}{\text{inert } \Gamma \qquad \Gamma(x) = \mu(x\colon \cdots \wedge \{a\colon T'\} \wedge \cdots) \qquad \Gamma \vdash T' <: T}\ \textsc{Inversion of } (\{\}\text{-I-!})
$$

Although there are four steps, each individual step is quite simple. More importantly, each step is modular, independent of the other steps, and the proof techniques at each step are either directly reusable (theorems) or easily adaptable (induction) to proofs of properties other than this specific lemma.

The derivation starts with general typing ($\Gamma \vdash x\colon \{a\colon T\}$), the typing relation of the DOT calculus. The key property that makes reasoning possible is that the typing context Γ is inert. Inert contexts will be defined in Section 3.2. Inertness ensures that customized subtyping in the program does not introduce unexpected subtyping relationships. If the context were not inert, any type could have been customized to have arbitrary subtypes and be inhabited by arbitrary terms, so it would be impossible to draw any conclusions about a term from its type.

Knowing that the typing context is inert, we apply Theorem 3.3 ($\vdash$ to $\vdash_{\#}$) to get a tight typing ($\Gamma \vdash_{\#} x\colon \{a\colon T\}$), which will be discussed in Section 3.3. A tight typing is immune to any unexpected subtyping relationships that the program may have defined, so our reasoning can now rely on familiar intuitions about what types ought to mean about their terms.

However, the tight typing rules are not amenable to inductive proofs. Theorem 3.6 ($\vdash_{\#}$ to $\vdash_{\#\#}$) gives invertible typing ($\Gamma \vdash_{\#\#} x\colon \{a\colon T\}$), which is specifically designed to make inductive reasoning as easy as possible. Invertible typing will be discussed in Section 3.4.

By induction on invertible typing, we obtain a property of all of the precise types $\Gamma \vdash_{!} x\colon \{a\colon T'\}$ that could have caused x to have the general type $\{a\colon T\}$. Informally, the precise typing means that the type $\Gamma(x)$ given to x by the typing context is an object type containing a field a of type T'. We will present precise typing in Section 3.3. Precise typing is also amenable to straightforward induction proofs, so we can use one to obtain $\Gamma(x)$.

3.2 Inert Typing Contexts

Recall the function $\lambda(x\colon \{A\colon S..U\}).t$ that we discussed in Section 2.3. If the function appears in a context Γ, its body is type checked in an extended context $(\Gamma,\, x\colon \{A\colon S..U\})$. The extended context adds a new subtyping relationship $(\Gamma,\, x\colon \{A\colon S..U\}) \vdash S <: U$ that might not have held in the original context Γ. In particular, the extended context could introduce a subtyping relationship that does not make sense, such as $\forall(x\colon S)T <: \mu(x\colon U)$, or $\top <: \bot$. To control such unpredictable contexts, we define the notion of inert typing contexts and inert types.

Definition 3.1. A *typing context* Γ is *inert* if the type $\Gamma(x)$ that it assigns to each variable x is inert.

Definition 3.2. A *type* U is *inert* if

- U is a dependent function type $\forall(x\colon S)T$, or
- U is a recursive type $\mu(x\colon T)$, where T is an intersection of field declarations $\{a\colon S\}$ and tight type declarations $\{A\colon S..S\}$, and the type labels A of the tight type declarations are distinct. A type declaration $\{A\colon S..U\}$ is tight if its bounds S and U are the same.

An inert typing context has the following useful property.

Property 1 (Inert Context Guarantee). *Let Γ be any inert typing context, t be a closed term and U be a closed type. If $\Gamma \vdash t\colon U$, then $\vdash t\colon U$.*

The significance of this property is that in an inert typing context, a term t does not have any "unexpected" types that it would not have in an empty typing context. For example, we can be sure that in an inert typing context, a function value will not have an object (recursive) type, and an object will not have a function type. Though we do not directly apply the property in the proof, it is useful for intuitive reasoning about typing and subtyping in inert typing contexts.

Every value has an inert type (as long as the value is well formed, i.e., as long as it has any type at all). This is because the two base typing rules for values, (All-I) and ({}-I), and the definition typing rules that they depend on, always assign an inert type to the value. The converse is not true: not every inert type is inhabited by a value. For example, we cannot construct a value of type $\lambda(x\colon \top).\bot$.

Returning to the example, suppose now that the function is invoked with some value v bound to a variable y: let $y = v$ in $(\lambda(x\colon \{A\colon S..U\}).t)\ y$. Recall that the body t is typed with the assumption that $S <: U$. Type checking the overall term ensures that the argument y provides *evidence* for that assumption. Specifically, the value v has an inert type, so y has an inert type. The typing rule for function application requires subtyping between the argument and parameter types, so the type of y must have a member $\{A\colon T..T\}$ with $S <: T$ and $T <: U$. (The bounds T of the type member must be tight because the type is inert.) The type T that y provides is evidence that justifies the assumption $S <: U$ under which the body t of the function was type checked. During execution, when the function is called, all occurrences of x in the body t will be replaced by y before evaluation of the body begins. In general, the semantics ensures that before it begins evaluating a term (such as t), the term has a type in a context in which all non-inert types (such as the type of x) have been narrowed to inert types (such as the type of y).

3.3 Tight Typing

Tight term typing

$$\frac{\Gamma(x) = T}{\Gamma \vdash_{\#} x\colon T}\ (\textsc{Var-\#}) \qquad \frac{\Gamma \vdash_{\#} t\colon T \quad (\Gamma,\, x\colon T) \vdash u\colon U \quad x \notin \mathsf{fv}(U)}{\Gamma \vdash_{\#} \mathsf{let}\ x = t\ \mathsf{in}\ u\colon U}\ (\textsc{Let-\#})$$

$$\frac{(\Gamma,\, x\colon T) \vdash t\colon U \quad x \notin \mathsf{fv}(T)}{\Gamma \vdash_{\#} \lambda(x\colon T).t\colon \forall(x\colon T)U}\ (\textsc{All-I-\#}) \qquad \frac{\Gamma \vdash_{\#} x\colon T}{\Gamma \vdash_{\#} x\colon \mu(x\colon T)}\ (\textsc{Rec-I-\#})$$

$$\frac{\Gamma \vdash_{\#} x\colon \forall(z\colon S)T \quad \Gamma \vdash_{\#} y\colon S}{\Gamma \vdash_{\#} x\,y\colon\ [y/z]\,T}\ (\textsc{All-E-\#}) \qquad \frac{\Gamma \vdash_{\#} x\colon \mu(z\colon T)}{\Gamma \vdash_{\#} x\colon\ [x/z]\,T}\ (\textsc{Rec-E-\#})$$

$$\frac{(\Gamma,\, x\colon T) \vdash d\colon T}{\Gamma \vdash_{\#} \nu(x\colon T)d\colon \mu(x\colon T)}\ (\{\}\text{-I-\#}) \qquad \frac{\Gamma \vdash_{\#} x\colon T \quad \Gamma \vdash_{\#} x\colon U}{\Gamma \vdash_{\#} x\colon T \wedge U}\ (\textsc{And-I-\#})$$

$$\frac{\Gamma \vdash_{\#} x\colon\ \{a\colon T\}}{\Gamma \vdash_{\#} x.a\colon T}\ (\{\}\text{-E-\#}) \qquad \frac{\Gamma \vdash_{\#} t\colon T \quad \Gamma \vdash_{\#} T <: U}{\Gamma \vdash_{\#} t\colon U}\ (\textsc{Sub-\#})$$

Tight subtyping

$$\Gamma \vdash_{\#} T <: \top\ (\textsc{Top-\#}) \qquad \Gamma \vdash_{\#} T \wedge U <: T\ (\textsc{And}_1\text{-<:-\#}) \qquad \frac{\Gamma \vdash_{\#} T <: U}{\Gamma \vdash_{\#} \{a\colon T\} <: \{a\colon U\}}\ (\textsc{Fld-<:-Fld-\#})$$

$$\Gamma \vdash_{\#} \bot <: T\ (\textsc{Bot-\#}) \qquad \Gamma \vdash_{\#} T \wedge U <: U\ (\textsc{And}_2\text{-<:-\#})$$

$$\Gamma \vdash_{\#} T <: T\ (\textsc{Refl-\#}) \qquad \frac{\Gamma \vdash_{!} x\colon\ \{A\colon T..T\}}{\Gamma \vdash_{\#} T <: x.A}\ (\text{<:-}\textsc{Sel-\#})$$

$$\frac{\Gamma \vdash_{\#} S <: T \quad \Gamma \vdash_{\#} T <: U}{\Gamma \vdash_{\#} S <: U}\ (\textsc{Trans-\#}) \qquad \frac{\Gamma \vdash_{\#} S <: T \quad \Gamma \vdash_{\#} S <: U}{\Gamma \vdash_{\#} S <: T \wedge U}\ (\text{<:-}\textsc{And-\#}) \qquad \frac{\Gamma \vdash_{!} x\colon\ \{A\colon T..T\}}{\Gamma \vdash_{\#} x.A <: T}\ (\textsc{Sel-<:-\#})$$

$$\frac{\Gamma \vdash_{\#} S_2 <: S_1 \quad \Gamma \vdash_{\#} T_1 <: T_2}{\Gamma \vdash_{\#} \{A\colon S_1..T_1\} <: \{A\colon S_2..T_2\}}\ (\textsc{Typ-<:-Typ-\#}) \qquad \frac{\Gamma \vdash_{\#} S_2 <: S_1 \quad (\Gamma,\, x\colon S_2) \vdash T_1 <: T_2}{\Gamma \vdash_{\#} \forall(x\colon S_1)T_1 <: \forall(x\colon S_2)T_2}\ (\textsc{All-<:-All-\#})$$

Fig. 3. Tight Typing Rules [Amin et al. 2016]

Although inert contexts provide the assurance of Property 1 (Inert Context Guarantee), in our proofs, we often need to reason even in contexts that are not inert. Moreover, even when we know that a context is inert, it would be difficult to express the important consequences of the inert context in every proof that deals with the general DOT typing and subtyping rules.

Precise variable typing

$$\frac{\Gamma(x) = T}{\Gamma \vdash_! x\colon T}\ (\textsc{Var-!}) \qquad \frac{\Gamma \vdash_! x\colon \mu(z\colon T)}{\Gamma \vdash_! x\colon\ [^x/_z]\,T}\ (\textsc{Rec-E-!}) \qquad \frac{\Gamma \vdash_! x\colon T \wedge U}{\Gamma \vdash_! x\colon T}\ (\textsc{And}_1\text{-E-!}) \qquad \frac{\Gamma \vdash_! x\colon T \wedge U}{\Gamma \vdash_! x\colon U}\ (\textsc{And}_2\text{-E-!})$$

Precise value typing

$$\frac{(\Gamma,\, x\colon T) \vdash t\colon U \qquad x \notin \mathsf{fv}(T)}{\Gamma \vdash_! \lambda(x\colon T).t\colon \forall(x\colon T)U}\ (\textsc{All-I-!}) \qquad \frac{(\Gamma,\, x\colon T) \vdash d\colon T}{\Gamma \vdash_! \nu(x\colon T)d\colon \mu(x\colon T)}\ (\{\}\text{-I-!})$$

Fig. 4. Precise Typing Rules [Amin et al. 2016]

Tight typing [Amin et al. 2016] is a slight restriction of general typing that can bridge the gap between the unpredictability of the general DOT typing rules in arbitrary typing contexts and the predictable assurances of Property 1 in inert typing contexts. The tight typing rules are presented in Figure 3. They are almost the same as the general DOT typing rules, except that the (<:-Sel-#) and (Sel-<:-#) rules have the restricted premise $\Gamma \vdash_! x\colon\ \{A\colon T..T\}$, so they can be applied only when the bounds T of the type member A are tight. Precise typing, denoted $\vdash_!$, is defined in Figure 4. The precise type of a variable x is the type $\Gamma(x)$ given to it by the typing context Γ, possibly decomposed using the elimination rules, so that if $\Gamma(x)$ is an object type such as $\mu(x\colon \cdots \wedge \{A\colon T..T\} \wedge \cdots)$, then x also has just the type member $\{A\colon T..T\}$ as a precise type. For values, precise typing applies only the base case rules (All-I) and ({}-I) from general typing. In premises of rules that extend the typing context (All-I-#, Let-#, {}-I-#), tight typing reverts to general typing in the extended context.

We observe two useful properties of tight typing that together combine to make it especially convenient for reasoning about DOT typing. The first property is that tight typing extends the benefits of Property 1 (Inert Context Guarantee) to *all* typing contexts, not only inert ones:

Property 2 (Tight Typing Guarantee). *Let Γ be any typing context, t be a closed term and U be a closed type. If $\Gamma \vdash_\# t\colon U$, then $\vdash_\# t\colon U$ and $\vdash t\colon U$.*

The general typing rules that enable DOT programs to define new user-defined subtyping relationships, (<:-Sel) and (Sel-<:), are restricted in tight typing to (<:-Sel-#) and (Sel-<:-#), which allow only to give an alias to an existing type, but not to introduce new subtyping between existing types.

Property 2 makes reasoning in tight typing easy: we never have to worry about unexpected custom subtyping relationships being introduced by the program, and we do not need to reason about whether we are in an inert typing context, because tight typing gives the guarantee in all contexts.

Although tight typing satisfies the desirable intuitive Property 2, it is not DOT. In particular, tight typing does not, in general, enable a program to use a custom-defined subtyping lattice that is the key feature of dependent object types. We would like the best of both worlds: to allow DOT programs to enjoy the full power of general typing, yet to reason about our proofs with the intuitive tight typing. For this, we need the second property of tight typing.

The second important property of tight typing is that in an inert typing context, tight typing is equivalent to general DOT typing:

THEOREM 3.3 ($\vdash$ TO $\vdash_\#$). *If Γ is an inert context, then $\Gamma \vdash t\colon T$ implies $\Gamma \vdash_\# t\colon T$, and $\Gamma \vdash S <: U$ implies $\Gamma \vdash_\# S <: U$.*

We delay giving the proof of the theorem until after some discussion.

These two properties motivate and justify our recommendation that tight typing should be at the core of all reasoning about the meaning of types in DOT. Tight typing is predictable, like the type systems of familiar calculi without dependent object types, yet in an inert typing context, it has the same power as general DOT typing. Therefore, every proof with a premise involving general typing and an inert typing context should immediately apply Theorem 3.3 ($\vdash$ to $\vdash_\#$) to drop down into the intuitive environment of tight typing for the rest of the reasoning.

What if we do not have an inert context as a premise, and therefore cannot apply Theorem 3.3? In that case, we should not reason about the meanings of types at all. As we saw in Section 2.3, in such a context, a term could be given an arbitrary type by custom subtyping rules. Therefore, we cannot deduce anything about a term from its type, and it would be futile to try.

In summary, inert contexts, tight typing, and Theorem 3.3 that justifies reasoning in tight typing should be the cornerstones of any reasoning about the meaning of types in the DOT calculus.

How shall we prove Theorem 3.3, then? It is tempting to prove the theorem by trying to compare various properties of the tight and general typing *relations*, the closures of the tight and general typing *rules*. This approach was taken in the proof of Amin et al. [2016] for a related theorem (with the same conclusion but different premises). The typing relations are very different from each other (general typing is much more powerful), but the rules that give rise to them are quite similar. It is much easier, therefore, to instead show that the *rules* are equivalent in an inert context. The only rules in general typing missing from tight typing are the (<:-SEL) and (SEL-<:) rules. Our goal is therefore to replace these rules with a lemma:

LEMMA 3.4 (SEL-<: REPLACEMENT). *If Γ is an inert context, then if $\Gamma \vdash_\# x\colon \{A\colon S..U\}$, then $\Gamma \vdash_\# S <: x.A$ and $\Gamma \vdash_\# x.A <: U$.*

One nice property of this lemma is that it is stated entirely in terms of tight typing. Thus, to prove it, we can ignore the unpredictable world of general typing, and work exclusively in the intuitive world of tight typing.

But how can we prove it? We would like to apply the (<:-SEL-#) and (SEL-<:-#) rules. Their premises are $\Gamma \vdash_! x\colon \{A\colon T..T\}$. Therefore, we need to *invert* tight typing, to show the following:

LEMMA 3.5 (SEL-<:-# PREMISE). *If Γ is an inert context, then if $\Gamma \vdash_\# x\colon \{A\colon S..U\}$, then there exists a type T such that $\Gamma \vdash_! x\colon \{A\colon T..T\}$, $\Gamma \vdash_\# S <: T$, and $\Gamma \vdash_\# T <: U$.*

We will discuss how to invert tight typing to prove this lemma in Section 3.4.

Using Lemma 3.5 (Sel-<:-# Premise), proving Lemma 3.4 (Sel-<: Replacement) is easy:

PROOF OF LEMMA 3.4 (SEL-<: REPLACEMENT). Apply Lemma 3.5 (Sel-<:-# Premise), then (<:-SEL-#) and (SEL-<:-#), to get $\Gamma \vdash_\# S <: T <: x.A <: T <: U$. The result follows by (TRANS-#). □

Using Lemma 3.4 (Sel-<: Replacement), proving Theorem 3.3 ($\vdash$ to $\vdash_\#$) is now also quite easy.

PROOF OF THEOREM 3.3 ($\vdash$ TO $\vdash_\#$). The proof is by mutual induction on the tight typing and subtyping derivations of $\Gamma \vdash t\colon T$ and $\Gamma \vdash S <: U$. In general, for each rule of general typing, we invoke the corresponding rule of tight typing. The premises of the tight typing rules differ from those of the general typing rules in that they require tight typing in rules that do not extend the context. Since the unextended context is inert, the general premise implies the tight premise by the induction hypothesis. Premises that do extend the context use general typing, so nothing needs to be proven for them. The exception is the (<:-SEL) and (SEL-<:) rules. Lemma 3.4 (Sel-<: Replacement) is an exact replacement for these rules, so we just apply it. Despite the long explanation, the proof in Coq is only two lines long. □

3.4 Inversion of Tight Typing

Although reasoning with tight typing is intuitive because it obeys Property 2 (Tight Typing Guarantee), we often need to invert the tight typing rules to prove properties such as Lemma 3.5 (Sel-<:-# Premise), which we used in the proof of Lemma 3.4 (Sel-<: Replacement). More generally, we need to prove that if $\Gamma \vdash_{\#} x\colon T$, where T is of a certain form, then $\Gamma(x) = U$, and there is a certain relationship between T and U.

The obvious approach to proving such inversion properties is by induction on the derivation of the tight typing. This usually fails, however, because of cycles in the tight typing rules. Each language construct typically has both an introduction and an elimination rule, and the two form a cycle. For example, if $\Gamma \vdash_{\#} x\colon T$, then $\Gamma \vdash_{\#} x\colon \mu(x\colon T)$ by (Rec-I-#), so again $\Gamma \vdash_{\#} x\colon T$ by (Rec-E-#). Such cycles block inductive proofs because a proposition $\Gamma \vdash_{\#} x\colon T$ is justified by $\Gamma \vdash_{\#} x\colon \mu(x\colon T)$, which in turn is justified by the original proposition $\Gamma \vdash_{\#} x\colon T$. The solution is to define a set of acyclic, invertible rules on which induction is easy, and to prove that the invertible rules induce the same typing relation as the cyclic tight typing rules.

The construction of the invertible typing rules is simplified by two restrictions:

(1) We only ever need to invert typing rules in inert typing contexts.
(2) We only ever need to invert typings of variables and values, not of arbitrary terms.

In the invertible rules, we can thus exclude rules that cannot apply to variables or values, and rules that cannot apply to inert types or to types derived from inert types.

It remains to decide, when facing a cycle of two rules that introduce and eliminate a given language construct, which one of the two rules to remove and which one to keep in the acyclic, invertible rule set. In general, because a construct can be introduced an unbounded number of times in tight typing, we must keep the introduction rule. For example, if x has type T, then x also has type $\mu(y\colon \mu(y'\colon \mu(y''\colon T)))$, and the invertible rules must generate this type. On the other hand, the base case of the typing rules for variables, the rule (Var-#), gives each variable x the type $\Gamma(x)$, which in an inert context is an inert type, and can therefore be a recursive type containing an intersection type. Since the tight typing rules eliminate the recursion and the intersection, the invertible rules must also eliminate them. It seems that we have reached a contradiction: the invertible rules must have both introduction and elimination rules for recursive and intersection types.

The solution is to split the invertible rules into two phases. The first phase of rules contains all the elimination rules. After all necessary eliminations have been performed, a second phase containing only introduction rules can then perform all necessary introductions. By splitting the rules into two phases, we ensure that no derivation can cycle between introductions and eliminations, so the rules are invertible. It turns out that we already have rules for the first phase: the precise typing rules introduced in Section 3.3 already contain all of the elimination rules that apply to variables and values, and eliminate from the type of a variable all constructs that can appear in an inert type. (Note that even the general DOT typing rules remove recursive and intersection types only from the types of variables, not values.) To construct the invertible introduction rules, we propose the following recipe:

(1) Start with the tight typing rules.
(2) Inline the subsumption rule (inline the subtyping rules into the typing rules). This simplifies the construction, so we define only one relation instead of two separate typing and subtyping relations.

(3) Specialize the terms in all rules to variables and values, and remove all rules that cannot apply to variables or values.
(4) Remove all elimination rules.
(5) Remove all rules that cannot apply in an inert context. Specifically, this means the (Bot-#) rule, because it has $\Gamma \vdash_{\#} x\colon \bot$ as a premise, but this typing cannot be derived by any of the other remaining rules starting from an inert type given to a variable by the (Var-#) rule or to a value by the (All-I-#) and ({}-I-#) rules.

Invertible Typing for Variables

$$\frac{\Gamma \vdash_{!} x\colon T}{\Gamma \vdash_{\#\#} x\colon T} \quad (\textsc{Var-\#\#})$$

$$\frac{\Gamma \vdash_{\#\#} x\colon \forall(z\colon S)T \qquad \Gamma \vdash_{\#} S' <: S \qquad (\Gamma, y\colon S') \vdash T <: T'}{\Gamma \vdash_{\#\#} x\colon \forall(z\colon S')T'} \quad (\textsc{All-<:-All-\#\#})$$

$$\frac{\Gamma \vdash_{\#\#} x\colon \{a\colon T\} \qquad \Gamma \vdash_{\#} T <: U}{\Gamma \vdash_{\#\#} x\colon \{a\colon U\}} \quad (\textsc{Fld-<:-Fld-\#\#})$$

$$\frac{\Gamma \vdash_{\#\#} x\colon T \qquad \Gamma \vdash_{\#\#} x\colon U}{\Gamma \vdash_{\#\#} x\colon T \wedge U} \quad (\textsc{And-I-\#\#})$$

$$\frac{\Gamma \vdash_{\#\#} x\colon \{A\colon T..U\} \qquad \Gamma \vdash_{\#} T' <: T \qquad \Gamma \vdash_{\#} U <: U'}{\Gamma \vdash_{\#\#} x\colon \{A\colon T'..U'\}} \quad (\textsc{Typ-<:-Typ-\#\#})$$

$$\frac{\Gamma \vdash_{\#\#} x\colon S \qquad \Gamma \vdash_{!} y\colon \{A\colon S..S\}}{\Gamma \vdash_{\#\#} x\colon y.A} \quad (\textsc{Sel-\#\#})$$

$$\frac{\Gamma \vdash_{\#\#} x\colon T}{\Gamma \vdash_{\#\#} x\colon \mu(x\colon T)} \quad (\textsc{Rec-I-\#\#})$$

$$\frac{\Gamma \vdash_{\#\#} x\colon T}{\Gamma \vdash_{\#\#} x\colon \top} \quad (\textsc{Top-\#\#})$$

Invertible Typing for Values

$$\frac{\Gamma \vdash_{!} v\colon T}{\Gamma \vdash_{\#\#} v\colon T} \quad (\textsc{Val-\#\#}_v)$$

$$\frac{\Gamma \vdash_{\#\#} v\colon T \qquad \Gamma \vdash_{\#\#} v\colon U}{\Gamma \vdash_{\#\#} v\colon T \wedge U} \quad (\textsc{And-I-\#\#}_v)$$

$$\frac{\Gamma \vdash_{\#\#} v\colon S \qquad \Gamma \vdash_{!} y\colon \{A\colon S..S\}}{\Gamma \vdash_{\#\#} v\colon y.A} \quad (\textsc{Sel-\#\#}_v)$$

$$\frac{\Gamma \vdash_{\#\#} v\colon \forall(z\colon S)T \qquad \Gamma \vdash_{\#} S' <: S \qquad (\Gamma, y\colon S') \vdash T <: T'}{\Gamma \vdash_{\#\#} v\colon \forall(z\colon S')T'} \quad (\textsc{All-<:-All-\#\#}_v)$$

$$\frac{\Gamma \vdash_{\#\#} v\colon T}{\Gamma \vdash_{\#\#} v\colon \top} \quad (\textsc{Top-\#\#}_v)$$

Fig. 5. Invertible Typing Rules

By applying this recipe to the tight typing rules, we arrive at the invertible typing rules shown in Figure 5. We must now prove that the typing relation induced by the invertible typing rules is equal to the typing relation induced by the tight typing rules (restricted to inert contexts and to variables and values):

Theorem 3.6 ($\vdash_{\#}$ to $\vdash_{\#\#}$). *If Γ is an inert context, t is a variable or a value, and $\Gamma \vdash_{\#} t\colon T$, then $\Gamma \vdash_{\#\#} t\colon T$.*

Proof. We first prove as a helper lemma that if $\Gamma \vdash_{\#\#} t\colon T$ and $\Gamma \vdash_{\#} T <: U$ then $\Gamma \vdash_{\#\#} t\colon U$ by induction on the derivation of $\Gamma \vdash_{\#} T <: U$. The main proof is by induction on the derivation of

$\Gamma \vdash_{\#} t \colon T$. Although we said that induction on tight typing usually fails because the rules have cycles, in this specific case, the induction is quite straightforward because invertible typing is part of the induction hypothesis. The inductive cases for elimination rules, which would usually lead to cycles in the induction, are all discharged using the invertible typing in the induction hypothesis. □

With this theorem, inversion proofs such as the proof of Lemma 3.5 (Sel-<:-# Premise) become easy inductions on the invertible typing rules:

Proof of Lemma 3.5 (Sel-<:-# Premise).

$$\dfrac{\dfrac{\text{inert}\,\Gamma \qquad \Gamma \vdash_{\#} x\colon \{A\colon S..U\}}{\text{inert}\,\Gamma \qquad \Gamma \vdash_{\#\#} x\colon \{A\colon S..U\}}\ \textsc{Theorem 3.6}\ (\vdash_{\#}\ \textsc{to}\ \vdash_{\#\#})}{\text{inert}\,\Gamma \qquad \Gamma \vdash_{!} x\colon \{A\colon T..T\} \qquad \Gamma \vdash_{\#} S <: x.T \qquad \Gamma \vdash_{\#} x.T <: U}\ \textsc{Induction on}\ \vdash_{\#\#}$$

□

We will see more lemmas that follow the same proof strategy in the next section.

3.5 Extending to Values

In general, soundness proofs require canonical-forms lemmas that show that if a value has a given type, then it is a particular form of value. Following our theme of a modular proof that deals with one concept at a time, we do most of our work at the level of types, following the same general recipe.

Because the DOT syntax enforces ANF, before a value can be used for anything interesting, it must first be assigned to a variable through a let expression. Suppose a variable x is bound to a value v by let $x = v$ in t and the variable x is used somewhere inside t. From the type U of the use of x, we would like to deduce the form of the value v.

We proceed in two steps. First, from a type U such that $\Gamma' \vdash x\colon U$, where Γ' is the typing context used to type the use of x occurring inside t, we follow the proof recipe to deduce the type $\Gamma'(x)$ given to x by the typing context. The typing context Γ' is constructed by the premises of the (Let) typing rule, which extends an existing typing context Γ to the typing context Γ' by adding a binding $(x : T)$. Here, T is some type such that $\Gamma \vdash v\colon T$. Therefore, $\Gamma'(x)$ is this T, and we have, in general, that $\Gamma \vdash v\colon \Gamma'(x)$ and thus also $\Gamma' \vdash v\colon \Gamma'(x)$.

For the second step, we know $\Gamma' \vdash v\colon T$, where the type T has been identified by the first step, and we wish to deduce the precise type of v, and thence invert the precise value typing rules to obtain the form of v.

The following lemmas instantiate these two steps, first for dependent function types, and then for field member types.

Lemma 3.7 ($\forall$ to $\Gamma(x)$).

$$\dfrac{\textit{inert}\,\Gamma \qquad \Gamma \vdash z\colon \forall(x\colon T)U}{\Gamma(z) = \forall(x\colon T')U' \qquad \Gamma \vdash T <: T' \qquad (\Gamma,\, x\colon T) \vdash U' <: U}$$

Lemma 3.8 ($\forall$ to λ).

$$\dfrac{\textit{inert}\,\Gamma \qquad \Gamma \vdash v\colon \forall(x\colon T)U}{v = \lambda(x\colon T').t \qquad \Gamma \vdash T <: T' \qquad (\Gamma,\, x\colon T) \vdash t\colon U}$$

Lemma 3.9 (μ to $\Gamma(x)$).

$$\frac{\textit{inert}\ \Gamma \qquad \Gamma \vdash x \colon \{a \colon T\}}{\Gamma(x) = \mu(x \colon \cdots \wedge \{a \colon T'\} \wedge \cdots) \qquad \Gamma \vdash T' <: T}$$

Lemma 3.10 (μ to ν).

$$\frac{\textit{inert}\ \Gamma \qquad \Gamma \vdash v \colon \mu(x \colon S) \qquad S = \cdots \wedge \{a \colon T\} \wedge \cdots}{v = \nu(x \colon S)(\cdots \wedge \{a = t\} \wedge \cdots) \qquad \Gamma \vdash t \colon T}$$

The proofs of all of the lemmas follow the same general proof recipe that we introduced for Lemma 3.9 in Section 3.1. We show the proof of Lemma 3.8 here, and proofs of the other three lemmas in the Appendix.

Proof of Lemma 3.8 ($\forall$ to λ).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{inert}\ \Gamma \qquad \Gamma \vdash v \colon \forall(x \colon T)U}{\text{inert}\ \Gamma \qquad \Gamma \vdash_{\#} v \colon \forall(x \colon T)U}\ \textsc{Theorem 3.3 } (\vdash \textsc{ to } \vdash_{\#})}{\text{inert}\ \Gamma \qquad \Gamma \vdash_{\#\#} v \colon \forall(x \colon T)U}\ \textsc{Theorem 3.6 } (\vdash_{\#} \textsc{ to } \vdash_{\#\#})}{\text{inert}\ \Gamma \qquad \Gamma \vdash_{!} v \colon \forall(x \colon T')U' \qquad \Gamma \vdash T <: T' \qquad (\Gamma, x \colon T') \vdash U' <: U}\ \textsc{Induction on } \vdash_{\#\#}}{v = \lambda(x \colon T').t \qquad (\Gamma, x \colon T') \vdash t \colon U' \qquad \Gamma \vdash T <: T' \qquad (\Gamma, x \colon T') \vdash U' <: U}\ \begin{matrix}\textsc{Inversion}\\ \textsc{of (All-I-!)}\end{matrix}}{v = \lambda(x \colon T').t \qquad (\Gamma, x \colon T) \vdash t \colon U' \qquad \Gamma \vdash T <: T' \qquad (\Gamma, x \colon T) \vdash U' <: U}\ \textsc{Narrowing}}{v = \lambda(x \colon T').t \qquad (\Gamma, x \colon T) \vdash t \colon U \qquad \Gamma \vdash T <: T'}\ (\textsc{Sub})$$

□

Since the return type of a dependent function type depends on the parameter type, this proof and the proof of Lemma 3.7 rely on a standard narrowing property, which states that making a typing context more precise by substituting one of the types by its subtype preserves the typing and subtyping relations.

Lemma 3.11 (Narrowing). *Suppose* $\Gamma(x) = T$ *and* $\Gamma[x \colon T'] \vdash T' <: T$. *Then* $\Gamma \vdash t \colon U$ *implies* $\Gamma[x \colon T'] \vdash t \colon U$, *and* $\Gamma \vdash S <: U$ *implies* $\Gamma[x \colon T'] \vdash S <: U$.

Narrowing is proved for DOT by Amin et al. [2016]. The proof is standard, with no issues specific to DOT, by induction on the typing and subtyping rules.

3.6 Operational Semantics

In general, a type soundness proof for a given operational semantics shows that if a term t has a type T, then it steps to another term of the same type (in a small-step semantics), or it reduces to a value of the same type (in a big-step semantics). In both cases, the first step of the proof is to deduce the form of the value from its type. The second step is then to apply the evaluation relation to obtain a new term (or value), apply the typing rules to obtain its type, and (hopefully) conclude that it has the original type T. The techniques presented in the preceding sections solve the first step for DOT. In this section, we consider the second step, highlighting aspects that differ from standard techniques for other calculi.

A key concept in any type soundness proof is that if an overall term has a type, then each subterm that the semantics evaluates must also have a type in some appropriate typing context Γ. This applies both in big-step semantics to each subterm that the evaluation function recursively evaluates, and in small-step semantics to each subterm that can appear in the hole of an evaluation context. In order to use the theory developed in the preceding sections to prove soundness of any

DOT semantics, whether big-step or small-step, an essential extension to this concept is that the typing context Γ in which the subterm to be evaluated has a type must be *inert*. If the typing context were not inert, we would not be able to conclude anything about the subterm being evaluated from its type. The various existing semantics that have been defined for DOT have the property that each subterm that the semantics evaluates is indeed typable in some inert context. Note that it is possible for a well-typed DOT program to have subterms not typable in any inert context, but only in positions where they do not reduce, such as in a function whose parameter type is uninhabited.

$$e ::= [\,] \mid \mathsf{let}\ x = [\,]\ \mathsf{in}\ t \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \qquad \textbf{evaluation context}$$

$$\frac{t \longmapsto t'}{e[t] \longmapsto e[t']} \qquad (\textsc{Term})$$

$$\frac{v = \lambda(z\colon T).t}{\mathsf{let}\ x = v\ \mathsf{in}\ e[x\,y] \longmapsto \mathsf{let}\ x = v\ \mathsf{in}\ e[[{}^{y}\!/\!{}_{z}]\,t]} \qquad (\textsc{Apply})$$

$$\frac{v = \nu(x\colon T)\ldots\{a = t\}\ldots}{\mathsf{let}\ x = v\ \mathsf{in}\ e[x.a] \longmapsto \mathsf{let}\ x = v\ \mathsf{in}\ e[t]} \qquad (\textsc{Project})$$

$$\mathsf{let}\ x = y\ \mathsf{in}\ t \longmapsto [{}^{y}\!/\!{}_{x}]\,t \qquad (\textsc{Let-Var})$$

$$\mathsf{let}\ x = \mathsf{let}\ y = s\ \mathsf{in}\ t\ \mathsf{in}\ u \longmapsto \mathsf{let}\ y = s\ \mathsf{in}\ \mathsf{let}\ x = t\ \mathsf{in}\ u \qquad (\textsc{Let-Let})$$

Fig. 6. DOT Operational Semantics [Amin et al. 2016]

$$\frac{e \text{ contains the binding } \mathsf{let}\ x = \lambda(z\colon T).t}{e[x\,y] \longmapsto e[[{}^{y}\!/\!{}_{z}]\,t]} \qquad (\textsc{Apply})$$

$$\frac{e \text{ contains the binding } \mathsf{let}\ x = \nu(x\colon T)\ldots\{a = t\}\ldots}{e[x.a] \longmapsto e[t]} \qquad (\textsc{Project})$$

$$e[\mathsf{let}\ x = [y]\ \mathsf{in}\ t] \longmapsto e[[{}^{y}\!/\!{}_{x}]\,t] \qquad (\textsc{Let-Var})$$

$$e[\mathsf{let}\ x = [\mathsf{let}\ y = s\ \mathsf{in}\ t]\ \mathsf{in}\ u] \longmapsto e[\mathsf{let}\ y = s\ \mathsf{in}\ \mathsf{let}\ x = t\ \mathsf{in}\ u] \qquad (\textsc{Let-Let})$$

Fig. 7. DOT Operational Semantics with Inlined (Term) Rule

The variant of DOT that we have studied in this paper is from Amin et al. [2016], which defines the small-step semantics with evaluation contexts shown in Figure 6. Accordingly, we prove soundness using the progress and preservation approach of Wright and Felleisen [1994]. To make the reduction rules more convenient to work with, we have inlined the (Term) rule into each of the other rules in Figure 7. As a result, the conclusions of the (Apply) and (Project) rules operate on the complicated contexts of the form $e[\mathsf{let}\ x = v\ \mathsf{in}\ e[\,]]$. We rewrite these contexts into just $e[\,]$ and add a premise that e contains the binding $\mathsf{let}\ x = v$.

Since the semantics is small step, we must show that if an overall term $u = e[t]$ has a type, then t has a type in an appropriate, inert typing context Γ. The next definitions make precise which typing contexts are appropriate for a given evaluation context e, and the next lemma gives us the typing context that we need.

Definition 3.12. An evaluation context e is *well-typed* with respect to a typing context Γ, written $e : \Gamma$, if whenever e contains the binding let $x = v$, then $\Gamma \vdash v \colon \Gamma(x)$.

Definition 3.13. The *domain* of an evaluation context e is the set of variables that are bound to values by let bindings in e.

Lemma 3.14 (Context Type). *If $\vdash e[t] \colon U$, then there exists an inert typing context Γ and a type T such that $e : \Gamma$, the domains of e and Γ are equal, and $\Gamma \vdash t \colon T$.*

Proof. We prove a stronger induction hypothesis: for any Γ', if $\Gamma' \vdash e[t] \colon U$, then there exists an inert typing context Γ and a type T such that $e : \Gamma', \Gamma$, the domains of e and Γ are equal, and $\Gamma', \Gamma \vdash t \colon T$. The lemma will then follow by setting $\Gamma' = \emptyset$. The proof is by induction on the derivation of $\Gamma' \vdash e[t] \colon U$. The interesting case is the (Let) typing rule. For each let binding $x_i = v_i$ in e, the premises of the rule yield a type T_i of v_i to which x_i can be bound in the typing context Γ being constructed. In order to construct an *inert* Γ, the type that we add must be inert, which may not be the case for T_i. The following lemma ensures that every value that has a type T_i also has a precise type T_i' that is a subtype of T_i. Every precise type of a value is inert, so we add T_i' to Γ instead of T_i to keep Γ inert. □

Lemma 3.15 (Value Typing). *If $\Gamma \vdash v \colon T$, then there exists a type T' such that $\Gamma \vdash_! v \colon T'$ and $\Gamma \vdash T' <: T$.*

Proof. [Amin et al. 2016] The proof is by induction on the derivation of $\Gamma \vdash v \colon T$, and is short because only three typing rules apply to values: (All-I), ({}-I), and (Sub). In the first two cases, the precise type of v coincides with the general type. The subsumption case is handled by using the induction hypothesis and transitivity of subtyping. □

With these results, we can prove the standard progress theorem that every typable term is a normal form or reduces to some other term.

Definition 3.16. [Amin et al. 2016] A *normal form* is a term generated by the grammar:

$$n ::= v \mid x \mid \text{let } x = v \text{ in } n$$

Theorem 3.17 (Progress). *If $\vdash u \colon U$, then u is a normal form or there is some u' such that $u \longmapsto u'$.*

Proof. Let e be the longest evaluation context such that $e[t] = u$, constructed by greedily applying the production rules of the evaluation context grammar, preferring $e ::= \text{let } x = v \text{ in } e$ over $e ::= \text{let } x = [\,] \text{ in } t$, and preferring the latter over $e ::= [\,]$, for as long as there exists a t such that $e[t] = u$. For every u, some evaluation context e always exists because $[u] = u$.

Then, by Lemma 3.14, there exists an inert typing context Γ and type T such that $e : \Gamma$, the domains of e and Γ are equal, and $\Gamma \vdash t \colon T$. We proceed by induction on the derivation of $\Gamma \vdash t \colon T$, in each case finding a reduction rule that applies. The interesting cases are (All-E) and ({}-E).

In the premises of (All-E), variable x has type $\forall(z \colon S)T$. Lemma 3.7 ($\forall$ to $\Gamma(x)$) tells us that Γ binds x to a compatible function type. The equality of the domains of e and Γ ensures that e binds x to some value v, and $e : \Gamma$ ensures that v has a compatible function type. Finally, Lemma 3.8 ($\forall$ to λ) tells us the v is a lambda, so the (Apply) reduction rule can be applied.

The ({}-E) case is similar, but using Lemma 3.9 (μ to $\Gamma(x)$) and Lemma 3.10 (μ to ν) instead of Lemmas 3.7 and 3.8, respectively, and the (Project) reduction rule instead of (Apply). □

In the above proof, our Coq formulation omits proving that the procedure of constructing the longest evaluation context e for which there exists a t such that $e[t] = u$ terminates, i.e., that we cannot keep adding an unbounded number of value let bindings of the form let $x = v$ to e and always finding a subterm t such that $e[t] = u$.

We can now prove the standard preservation theorem.

Theorem 3.18 (Preservation). *If $\vdash u\colon U$ and $u \longmapsto u'$, then $\vdash u'\colon U$.*

Proof. Since $u \longmapsto u'$, there must be some evaluation contexts and subterms such that $u = e[t]$ and $u' = e'[t']$. We use the observation that in the reduction relation, e and e' are very similar; precisely, for each case, we can find a subterm t_0 such that $e'[t_0] = e[t] = u$. Using Lemma 3.14, we obtain an inert Γ and T such that $e' : \Gamma$ and $\Gamma \vdash t_0\colon T$. The proof proceeds by induction on the derivation of this typing. The interesting cases are again (All-E) and ({}-E), and we apply Lemmas 3.7 to 3.10 to obtain values with the necessary types like in the proof of Theorem 3.17 (Progress). In each case, we obtain that $\Gamma \vdash t'\colon T$, with the (All-E) case requiring the substitution lemma defined below.

Now we have $\vdash e'[t_0]\colon U$, $\Gamma \vdash t_0\colon T$, and $\Gamma \vdash t'\colon T$, and we need to prove $\vdash e'[t']\colon U$. The overall approach is to apply induction to the structure of e', and in the inductive step, to apply another induction on the derivation of $\Gamma \vdash t_0\colon T$. Specifically, if the evaluation context e' is empty, then $e'[t_0] = t_0$, $T = U$, $\Gamma = \emptyset$, and $e'[t'] = t'$, so $\Gamma \vdash e'[t']\colon U$. If e' is not empty, then it ends in either let $x = v$ in [] or let $x =$ [] in s. The inductive step moves this final let out of the evaluation context into the term by replacing the term t_0 with let $x = v$ in t_0 or let $x = t_0$ in s, respectively. The same replacement is also applied to t'. Each such move of a value let binding out of e' and into t_0 and t' preserves the fact that if $\Gamma \vdash t_0\colon T$, then $\Gamma \vdash t'\colon T$, which we prove by induction on the derivation of $\Gamma \vdash t_0\colon T$. □

Lemma 3.19 (Substitution). *If $(\Gamma, x\colon S) \vdash t\colon T$ and $\Gamma \vdash y\colon [y/x]\,S$ then $\Gamma \vdash [y/x]\,t\colon [y/x]\,T$.*

The lemma is proven by Amin et al. [2016]. The proof is standard, with no issues specific to DOT, by induction on the typing and subtyping rules. Thanks to the use of A-normal form in the DOT syntax, function application and therefore substitution is needed only for substituting variables for other variables.

Our paper comes with a Coq-formalized version of the presented proof. It is based on the original Coq proof by Amin et al. [2016].

4 PROOF STRUCTURE AND EXTENSIONS

This section summarizes the structure of the proof and discusses how the proof is affected by changes and extensions of the DOT calculus.

4.1 Proof Structure

The dependencies between the main lemmas in the proof are summarized in the diagram in Figure 8. The gray nodes and solid lines denote the lemmas in the proof for the DOT calculus that we have presented in this paper. The white boxes and dotted lines correspond to changes needed to prove soundness of an example extension of the calculus that will be described in Section 4.3.

The final progress and preservation theorems depend on the four applications of the proof recipe to prove canonical forms for values and variables of object and function type (written in the figure as $\forall$ to λ, μ to ν, $\forall$ to $\Gamma(x)$, and μ to $\Gamma(x)$). Each application of the proof recipe uses Theorem 3.3 ($\vdash$

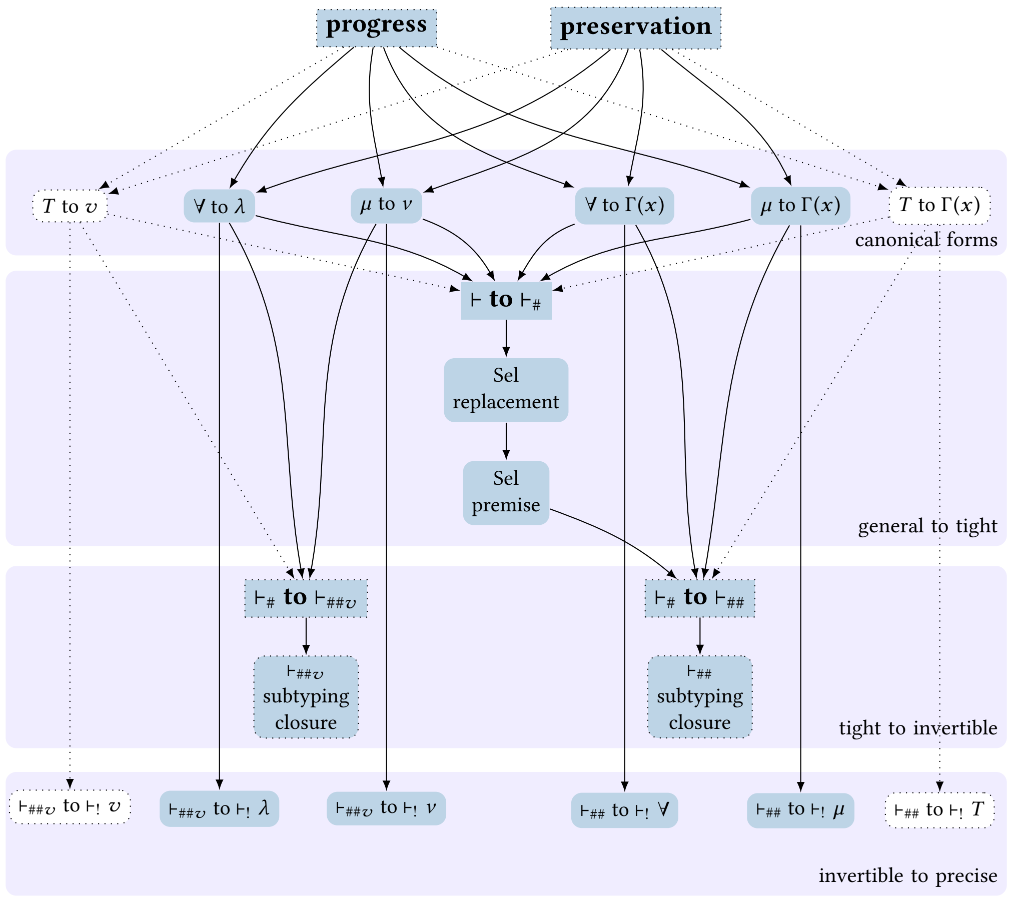
progress
preservation
T to v
$\forall$ to λ
μ to ν
$\forall$ to $\Gamma(x)$
μ to $\Gamma(x)$
T to $\Gamma(x)$
canonical forms
$\vdash$ to $\vdash_{\#}$
Sel replacement
Sel premise
general to tight
$\vdash_{\#}$ to $\vdash_{\#\#v}$
$\vdash_{\#\#v}$ subtyping closure
$\vdash_{\#}$ to $\vdash_{\#\#}$
$\vdash_{\#\#}$ subtyping closure
tight to invertible
$\vdash_{\#\#v}$ to $\vdash_{!}$ v
$\vdash_{\#\#v}$ to $\vdash_{!}$ λ
$\vdash_{\#\#v}$ to $\vdash_{!}$ ν
$\vdash_{\#\#}$ to $\vdash_{!}$ $\forall$
$\vdash_{\#\#}$ to $\vdash_{!}$ μ
$\vdash_{\#\#}$ to $\vdash_{!}$ T
invertible to precise

Fig. 8. Dependencies between main lemmas in the proof. Gray nodes denote existing lemmas. White nodes denote lemmas that would need to be added if DOT were extended with a new type T and a new value v. Dotted lines correspond to lemmas that will need to be added or modified for the extension.

to $\vdash_{\#}$) and Theorem 3.6 ($\vdash_{\#}$ to $\vdash_{\#\#}$) to convert general typing to tight typing and then to invertible typing. Theorem 3.3 depends on Lemma 3.4 (Sel-<: Replacement) and Lemma 3.5 (Sel-<:-# Premise). Theorem 3.6 depends on a subtyping closure helper lemma. After using the theorems to obtain invertible typing, we invert the invertible typing in each of the four cases to obtain either the type $\Gamma(x)$ assigned to a variable x by the typing context Γ or the form of the value v with the given type. The four light large boxes in the figure indicate the canonical-forms lemmas, and the three phases of the proof recipe (conversion of general to tight typing, tight to invertible typing, and inversion of invertible typing).

4.2 Modifications of the Calculus

The most common expected extensions of a calculus are the addition of new forms of values and terms, of new forms of types and typing rules, and changes to the evaluation rules. Most extensions will change multiple aspects (e.g., add a new form of value and an associated type), but we discuss each change individually. In the next section, we will present a specific example of an extension that makes all of these kinds of changes.

The only part of our proof that deals with values are the two pairs of canonical forms lemmas in Section 3.5 and the final progress and preservation theorems. A new form of value will require an additional pair of canonical-forms lemmas. The lemma can follow the general recipe: it will apply Theorem 3.3 ($\vdash$ to $\vdash_{\#}$) and Theorem 3.6 ($\vdash_{\#}$ to $\vdash_{\#\#}$). It does not need to reason with the general DOT typing rules, but only to invert the invertible typing obtained from Theorem 3.6. This last inversion step should be easy, because invertible typing is designed to be easily invertible. The addition of a new form of value is illustrated in the dependence graph by the two white nodes on the left side of the graph.

The only part of the proof that deals with terms in general are the final progress and preservation theorems. The only non-trivial change required when adding a new term is that if new reduction rules are added for the new term to the operational semantics, cases for the new reduction rules need to be added to the progress and preservation theorems. This is illustrated in the dependence graph by the dotted outlines of the nodes representing those two theorems.

Adding a new form of type is a more significant change. Given general typing rules for the new type, we must incorporate the changes into the tight, invertible, and precise typing rules. Tight typing differs from general typing only in its handling of abstract type members and type projections, so changes unrelated to those features can be incorporated directly into tight typing. A change involving abstract type members or type projections requires corresponding modifications to tight typing. Property 2 (Tight Typing Guarantee) gives a modular specification to guide the design of such modifications. Specifically, we know that as long as the modified tight typing rules satisfy the property and we can prove Theorem 3.3 ($\vdash$ to $\vdash_{\#}$), then the proof recipe and the rest of the whole soundness proof will continue to hold without requiring non-trivial changes. To incorporate the modifications into invertible and precise typing, it suffices to follow the general recipe outlined in Section 3.4. Specifically, we must classify the new tight typing rules as either introducing or eliminating a syntactic construct, and then add them to either invertible or precise typing, respectively. Adding new typing rules requires adding the corresponding cases to the proofs of Theorem 3.3 ($\vdash$ to $\vdash_{\#}$) and Theorem 3.6 ($\vdash_{\#}$ to $\vdash_{\#\#}$). To illustrate this, we add a dotted outline around the node representing Theorem 3.6 and the subtyping closure lemma used in its proof. In the proof of Theorem 3.3 ($\vdash$ to $\vdash_{\#}$), all cases except (<:-Sel) and (Sel-<:) are so simple that Coq discharges them automatically, so we do not add a dotted outline to the node to indicate new cases in the proof.

A change to the evaluation rules of the calculus does not affect any of the reasoning in Sections 3.1 to 3.5, since those sections are independent of any particular evaluation semantics. Only the final progress and preservation theorems are affected.

4.3 Case Study: Adding Mutation to DOT

We have applied our proof technique to prove soundness of the mutable extension of DOT of Rapoport and Lhoták [2017], which adds mutable reference cells and associated reference types. This extension of the calculus involved adding the following to the soundness proof:

- a new case to the definition of inert types: any reference type T is inert;
- an additional case to the definitions of invertible typing for variables and invertible typing for values v;
- two new canonical forms lemmas for values and variables of reference type following the proof recipe;
- the corresponding cases to Theorem 3.6 ($\vdash_{\#}$ to $\vdash_{\#\#}$) and the subtyping closure lemma that it depends on, to Theorem 3.17 (Progress), and to Theorem 3.18 (Preservation).

These changes are shown using dotted lines and white nodes in the dependence graph in Figure 8. The overall structure of the dependencies between the lemmas did not change. The new canonical forms lemmas followed the proof recipe that we have described in this paper. In the proofs of some lemmas, we had additional new cases to prove, but the structure of the proof of each lemma did not change. In general, we found that the new lemmas and new cases in the existing lemmas were easy to prove.

5 THE STRUGGLE FOR "GOOD" BOUNDS

A recurring theme in previous work on DOT has been the struggle to enforce "good" bounds. A type member declaration $\{A\colon S..U\}$ is considered to have "good" bounds if $S <: U$. If all type members could be forced to maintain "good" bounds, it would prevent an object of type $\mu(x\colon \{A\colon S..U\})$ from introducing a new, possibly non-sensical subtyping relationship $S <: U$ from $S <: x.A <: U$ and transitivity. Many of the challenges along the way to defining a sound DOT calculus arose from the negative interaction between "good" bounds and other properties, such as narrowing and transitivity. For example, although both $\{A\colon \bot..\bot\}$ and $\{A\colon \top..\top\}$ have "good" bounds, the narrowed type $\{A\colon \bot..\bot\} \wedge \{A\colon \top..\top\}$ causes trouble: in the function $\lambda(x\colon \{A\colon \bot..\bot\} \wedge \{A\colon \top..\top\}).t$, the body t is type-checked in a typing context in which $\top <: x.A <: \bot$.

Not only do "good" bounds interact poorly with other desirable properties, but even defining precisely what "good" bounds are is surprisingly elusive. Informally, bounds are "good" if $S <: U$. But in what typing context should this subtyping relationship hold? In deciding whether the type $\mu(x\colon \{A\colon S..U\})$ should be allowable, it seems appropriate to respect the recursion implied by μ and use a context that includes x; that is, to require that $(\Gamma,\ x\colon \{A\colon S..U\}) \vdash S <: U$. But this statement is always true regardless of the types S and U because it is self-justifying: $(\Gamma,\ x\colon \{A\colon S..U\}) \vdash S <: x.A <: U$. If we decide instead to exclude the self-reference x from the context used to decide whether $S <: U$, we exclude many desirable types from the definition of "good" bounds. For example, we consider "bad" the type $\mu(x\colon \{A\colon \bot..\top\} \wedge \{B\colon x.A..x.C\} \wedge \{C\colon \bot..\top\})$ that innocently defines three type members with $A <: B <: C$, because $x.A$ cannot be a subtype of $x.C$ without x in the context. We also consider "bad" the following type that defines two type members $A <: B$ constrained to be function types: $\mu(x\colon \{A\colon \bot..\forall(y\colon \bot)\top\} \wedge \{B\colon x.A..\forall(y\colon \bot)\top\})$. Again, $x.A$ cannot be a subtype of $\forall(y\colon \bot)\top$ without x in the context. Finally, such a definition of "good" bounds restricts the applicability of type aliases: the following type defines A and B as aliases for $\top$ and $\bot$, respectively, but cannot use these aliases in the bounds of C because $x.B \not<: x.A$ in a context without x: $\mu(x\colon \{A\colon \top..\top\} \wedge \{B\colon \bot..\bot\} \wedge \{C\colon x.B..x.A\})$. Although it would be possible to come up with some definition of "good" bounds that handles these specific examples, the definition of what was intended to be an obvious and intuitive concept would become very complicated, and other more sophisticated counterexamples would probably continue to exist. Thus, it appears that trying to enforce "good" bounds, and even trying to define what "good" bounds are, is a dead end.

By contrast, inert types obey a purely syntactic property that is easily defined and checked, without requiring a subtyping judgment in some typing context that would have to be specified. The property provided by an inert typing context can be stated precisely and formally (Property 1 (Inert Context Guarantee)).

6 RELATED WORK

6.1 DOT Soundness Proofs

The work most closely related to ours is Amin et al. [2016], which defines and proves sound the variant of the DOT calculus for which we have developed our alternative soundness proof. That work also defines tight typing, though it does not use it as pervasively as our proof does.

A central notion of that proof is store correspondence, a relationship between typing contexts and stores of runtime values. A typing context Γ corresponds to a store s if for every variable x, $\Gamma \vdash_! s(x) : \Gamma(x)$. Typing and subtyping in a context Γ that corresponds to some store s have similar predictable behaviour as they do in an inert context. Part of the proof consists of lemmas that relate internal details of values in stores with internal details of types in corresponding contexts. By contrast, the property of inert contexts is independent of values, so our proof does not depend on such lemmas.

Another central notion is "possible types": if a typing context Γ corresponds to some store s, and s assigns to variable x the value v, then the possible types of the triple (Γ, x, v) include all types T such that $\Gamma \vdash x : T$. Possible types serve a similar purpose as our invertible typing rules, to facilitate induction proofs. Unlike invertible typing, possible types depend on the runtime value v of x. The possible types lemma relates general typing in a context with a corresponding store to possible types. It serves a similar purpose as our Theorem 3.6 ($\vdash_\#$ to $\vdash_{\#\#}$) (which relates tight to invertible typing), but its proof is more complicated, because it depends on sublemmas that relate types to values in the context corresponding to the store, and on general typing.

Amin et al. [2016] also prove a similar result as Theorem 3.3 ($\vdash$ to $\vdash_\#$): the general to tight lemma states that in a context Γ for which there exists some corresponding runtime store s, general typing implies tight typing. We prove Theorem 3.6 ($\vdash_\#$ to $\vdash_{\#\#}$) first, which makes proving Theorem 3.3 ($\vdash$ to $\vdash_\#$) easy. The proof of Amin et al. [2016] does the analogous steps in the opposite order: it proves the general to tight lemma first, and the possible types lemma afterwards, using the general to tight lemma in its proof. The proof of the general to tight lemma is thus complicated because it cannot make use of possible types. Another complication is that the proof of the general to tight lemma, like the proof of the possible types lemma, depends on sublemmas that relate types to values in the context corresponding to the store.

Rompf and Amin [2016] define a variant of the DOT calculus with additional features, most significantly subtyping between recursive types. This adds significant complexity to the proof: Lemmas 6 to 11 are needed only because of this feature. However, subtyping between Scala's types can be already modelled by subtyping between type members in DOT. Scala has nominal subtyping between classes and traits that are explicitly declared to be subtypes using an `extends` clause. A class or trait declaration in Scala corresponds in DOT to a type member definition in some package x that gives a label A to a recursive type. The recursive type is used to define the members of the class, and the recursion is necessary so that members of the class can refer to the object of the class `this`. A subclass B of A can be declared as the type member definition $B = x.A \wedge \mu(z : x.A \wedge T)$, where the type T describes the additional members that B adds to A. Then $x.B$ is a subtype of $x.A$, and given a variable of type $x.B$, it is possible to access both members that were declared in A and members that were added in B. This DOT encoding models the Scala subtyping between classes A and B without requiring subtyping between recursive types, and it can be expressed in the DOT of Amin et al. [2016].

Unlike Amin et al. [2016] and our proof, the proof of Rompf and Amin [2016] does not use tight typing, the typing relation that neutralizes the two type rules that enable a DOT program to introduce non-sensical subtyping relationships in a custom type system. Instead, the proof uses "precise subtyping", a restriction of general subtyping to relationships whose derivation does not end in the transitivity rule.

6.2 History of Scala Calculi

Odersky et al. [2003] introduce νObj, a calculus to formalize Scala's path-dependent types. νObj includes abstract type members, classes, compound (non-commutative) mixin composition, and singleton types, among other features. However, the calculus lacks several essential Scala features, such as the ability to define custom lower bounds for type members, and has no top and bottom types. Additionally, νObj, unlike Scala, has classes as first-class values. νObj comes with a type soundness proof. The paper also shows that type checking for νObj is undecidable. Cremet et al. [2006] propose Featherweight Scala, which is similar to νObj, but without classes as first-class values. The paper shows that type inference in Featherweight Scala is decidable, but does not prove type safety. Scalina, introduced by Moors et al. [2008], presents a formalization for higher-kinded types in Scala, but also without a soundness proof.

Amin et al. [2012] present the first DOT. DOT has fewer syntax-level features than νObj: there are no classes, mixins, or inheritance. However, some of the previously missing crucial Scala features are now present. The calculus allows refinement of abstract type members through commutative intersections, combining nominal with structural typing. Type members can have custom lower and upper bounds, and the type system contains a bottom and top type. The paper comes without a type safety proof, but it explains the challenges and provides counterexamples to preservation. The paper shows how the environment narrowing property makes proving soundness complicated: replacing a type in the context with a more precise version can impose a new subtyping relationship, which could disagree with the existing ones.

Amin et al. [2014] have the first mechanized soundness proof for μDOT, a simplified calculus that excludes refinements, intersections, and the bottom and top types, and uses big-step semantics. The paper proposes the idea to circumvent bad bounds by reasoning about types that correspond to runtime values.

Amin et al. [2016] and Rompf and Amin [2016] build on this store correspondence idea, to establish the first mechanized soundness proofs for DOT calculi with support for type intersection and refinement, and top and bottom types. The two calculi and soundness proofs were discussed in the previous section.

6.3 Other Related Calculi

Path-dependent types were first introduced in the context of *family polymorphism* by Ernst [2001]. In family polymorphism, groups of types can form *families* that correspond to a specific object. Two types from the same class are considered incompatible if the types are associated with different runtime objects.

Family polymorphism is the foundation of *virtual classes*, which were introduced in the Beta programming language [Madsen and Møller-Pedersen 1989] and further developed in gbeta [Ernst 1999]. Virtual classes are nested classes that can be extended or redefined (overridden), and are dynamically resolved through late binding. Family polymorphism allows for a fine-grained distinction between classes that have the same static path, yet belong to different runtime objects and can thus have different implementations.

Virtual classes were first formalized and proved type safe in the *vc* calculus [Ernst et al. 2006]. vc is a class-based, nominally-typed calculus with a big-step semantics. To create path-based types, the keyword out is used to refer to an enclosing object. With its support for classes, inheritance, and mutation of variables, vc is more complex than DOT, whose purpose is to serve as a simple core calculus for Scala. Additionally, Scala has no support for virtual classes: the language does not allow class overriding, and its classes are resolved statically at compile time.

Tribe by Clarke et al. [2007] is a simpler, more general calculus inspired by vc. One of the main distinctions to vc is that variables, and not just enclosing objects (out), can be used as paths for path-dependent types. This makes the calculus more general, as it can express subtyping relationships between classes with arbitrary absolute paths. Tribe comes with a type-safety proof, which is based on a small-step semantics. Expanding paths to allow variables brings Tribe closer to DOT. However, the complexity of the type system, resulting from modelling classes and inheritance, and the modelling of virtual classes, which are not present in Scala, leaves DOT more suitable as a core calculus for Scala.

Amin and Rompf [2017] offer a survey of mechanized soundness proofs for big-step, DOT-like calculi using definitional interpreters. The paper explores a family of calculi ranging from System F to System $D_{<:>}$ and general proof techniques that can be applied to this entire family. The paper discusses similarities and differences between System $D_{<:>}$ and DOT.

7 CONCLUSION

DOT [Amin et al. 2016] is the result of a long effort to develop a core calculus for Scala. Now that there is a sound version of the calculus, we would like to extend it with other Scala features, such as classes, mixin composition, side effects, implicit parameters, etc. DOT can be also used as a platform for developing new language features and for fixing Scala's soundness issues [Amin and Tate 2016]. But these applications are hindered by the complexity of the existing soundness proofs, which interleave reasoning about variables, types, and runtime values, and their complex interactions.

We have presented a simplified soundness proof for the DOT calculus, formalized in Coq. The proof separates the reasoning about types, typing contexts, and values from each other. The proof depends on the insight of inert typing contexts, a syntactic characterization of contexts that rule out any non-sensical subtyping that could be introduced by abstract type members. The central lemmas of the proof follow a general proof recipe for deducing properties of terms from their types in full DOT while reasoning only in a restricted, intuitive environment free from the paradoxes caused by abstract type members. The same recipe can be followed to prove similar lemmas when the calculus is modified or extended. The result is a simple, modular proof that is well suited for developing extensions.

A APPENDIX

Proof of Lemma 3.7 ($\forall$ to $\Gamma(x)$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{inert } \Gamma \qquad \Gamma \vdash x\colon \forall(y\colon T)U}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} x\colon \forall(y\colon T)U}\ \text{Theorem 3.3 } (\vdash \text{ to } \vdash_{\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#\#} x\colon \forall(y\colon T)U}\ \text{Theorem 3.6 } (\vdash_{\#} \text{ to } \vdash_{\#\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} x\colon \forall(y\colon T')U' \qquad \Gamma \vdash T <: T' \qquad (\Gamma, y\colon T') \vdash U' <: U}\ \text{Induction on } \vdash_{\#\#}}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} x\colon \forall(y\colon T')U' \qquad \Gamma \vdash T <: T' \qquad (\Gamma, y\colon T) \vdash U' <: U}\ \text{Narrowing}}{\text{inert } \Gamma \qquad \Gamma(x) = \forall(y\colon T')U' \qquad \Gamma \vdash T <: T' \qquad (\Gamma, y\colon T) \vdash U' <: U}\ \text{Induction on } \vdash_{!}$$

□

Proof of Lemma 3.9 (μ to $\Gamma(x)$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{inert } \Gamma \qquad \Gamma \vdash x\colon \{a\colon T\}}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} x\colon \{a\colon T\}}\ \text{Theorem 3.3 } (\vdash \text{ to } \vdash_{\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#\#} x\colon \{a\colon T\}}\ \text{Theorem 3.6 } (\vdash_{\#} \text{ to } \vdash_{\#\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} x\colon \{a\colon T'\} \qquad \Gamma \vdash T' <: T}\ \text{Induction on } \vdash_{\#\#}}{\text{inert } \Gamma \qquad \Gamma(x) = \mu(x\colon \cdots \wedge \{a\colon T'\} \wedge \cdots) \qquad \Gamma \vdash T' <: T}\ \text{Induction on } \vdash_{!}$$

□

Proof of Lemma 3.10 (μ to ν).

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{inert } \Gamma \qquad \Gamma \vdash v\colon \mu(x\colon S) \qquad S = \cdots \wedge \{a\colon T\} \wedge \cdots}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} v\colon \mu(x\colon S) \qquad S = \cdots \wedge \{a\colon T\} \wedge \cdots}\;\textsc{Theorem 3.3 } (\vdash \text{ to } \vdash_{\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#\#} v\colon \mu(x\colon S) \qquad S = \cdots \wedge \{a\colon T\} \wedge \cdots}\;\textsc{Theorem 3.6 } (\vdash_{\#} \text{ to } \vdash_{\#\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} v\colon \mu(x\colon S) \qquad S = \cdots \wedge \{a\colon T\} \wedge \cdots}\;\textsc{Induction on } \vdash_{\#\#}}{\text{inert } \Gamma \qquad v = \nu(x\colon S)(\cdots \wedge \{a = t\} \wedge \cdots) \qquad \Gamma \vdash t\colon T}\;\textsc{Inversion of } (\{\}\text{-I-!})$$

□

ACKNOWLEDGMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

Martín Abadi and Luca Cardelli. 1996. *A Theory of Objects.* Springer.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 249–272. DOI: http://dx.doi.org/10.1007/978-3-319-30936-1_14

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

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. DOI: http://dx.doi.org/10.1145/3009837

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 233–249. DOI: http://dx.doi.org/10.1145/2660193.2660216

Nada Amin and Ross Tate. 2016. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 838–848. DOI: http://dx.doi.org/10.1145/2983990.2984004

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007 (ACM International Conference Proceeding Series)*, Brian M. Barry and Oege de Moor (Eds.), Vol. 208. ACM, 121–134. DOI: http://dx.doi.org/10.1145/1218563.1218578

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.*

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

Erik Ernst. 2001. Family Polymorphism. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science)*, Jørgen Lindskov Knudsen (Ed.), Vol. 2072. Springer, 303–326. DOI: http://dx.doi.org/10.1007/3-540-45337-7_17

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 270–282. DOI: http://dx.doi.org/10.1145/1111037.1111062

Ole Lehrmann Madsen and Birger Møller-Pedersen. 1989. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89), New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings.*, George Bosworth (Ed.). ACM, 397–406. DOI: http://dx.doi.org/10.1145/74877.74919

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

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. 2003. A Nominal Theory of Objects with Dependent Types. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings (Lecture Notes in Computer Science)*, Luca Cardelli (Ed.), Vol. 2743. Springer, 201–224. DOI: http://dx.doi.org/10.1007/978-3-540-45070-2_10

Martin Odersky and Matthias Zenger. 2005. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 41–57. DOI: http://dx.doi.org/10.1145/1094811.1094815

Leo Osvald, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 234–251. DOI: http://dx.doi.org/10.1145/2983990.2984009

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

Marianna Rapoport and Ondřej Lhoták. 2017. Mutable WadlerFest DOT. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs (FTfJP'17).* ACM, New York, NY, USA, Article 7, 6 pages. DOI: http://dx.doi.org/10.1145/3103111.3104036

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 624–641. DOI: http://dx.doi.org/10.1145/2983990.2984008

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 21:1–21:28. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.21

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94.