Dependent Object Types

Towards a foundation for Scala's type system

Nada Amin Adriaan Moors Martin Odersky
EPFL
first.last@epfl.ch

Abstract

We propose a new type-theoretic foundation of Scala and languages like it: the Dependent Object Types calculus (DOT). DOT models Scala's path-dependent types and abstract type members, as well as its mixture of nominal and structural typing through the use of refinement types. It makes no attempt to model inheritance or mixin composition. The calculus does not model what's currently in Scala: it is more normative than descriptive.

We show that DOT and its patched-up variants are not syntactically sound, by exhibiting counterexamples to preservation. Nevertheless, we sketch a proof of type-safety of the calculus via step-indexed logical relations.

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

General Terms Languages, Theory, Verification

Keywords calculus, objects, dependent types, step-indexed logical relations

1. Introduction

This paper presents a proposal for a new type-theoretic foundation of Scala and languages like it. The properties we are interested in modeling are Scala's path-dependent types and abstract type members, as well as its mixture of nominal and structural typing through the use of refinement types. Compared to previous approaches [5, 12], we make no attempt to model inheritance or mixin composition. Indeed we will argue that such concepts are better modeled in a different setting.

The calculus does not precisely describe what's currently in Scala. It is more normative than descriptive. The main point of deviation concerns the difference between Scala's compound type formation using **with** and classical type intersection, as it is modeled in the calculus. Scala, and the previous calculi attempting to model it, conflates the concepts of compound types (which inherit the members of several parent types) and mixin composition (which build classes from other classes and traits). At first glance, this offers an economy of concepts. However, it is problematic because mixin composition and intersection types have quite different properties. In the case of several inherited members with the same name, mixin composition has to pick one which overrides the others. It uses for that the concept of linearization of a trait hierarchy. Typically, given two independent traits T_1 and T_2 with a common method m, the mixin composition T_1 **with** T_2 would pick the m in T_2, whereas the member in T_1 would be available via a super-call. All this makes sense from an implementation standpoint. From a typing standpoint it is more awkward, because it breaks commutativity and with it several monotonicity properties.

In the present calculus, we replace Scala's compound types by classical intersection types, which are commutative. We also complement this by classical union types. Intersections and unions form a lattice wrt subtyping. This addresses another problematic feature of Scala: In Scala's current type system, least upper bounds and greatest lower bounds do not always exist. Here is an example: given two traits

```
trait A { type T <: A }
trait B { type T <: B }
```

The greatest lower bound of `A` and `B` is approximated by the infinite sequence

```
A with B { type T <: A with B { type T <: A with B {
  type T < ...
}}}
```

The limit of this sequence does not exist as a type in Scala.

This is problematic because glbs and lubs play a central role in Scala's type inference. The absence of universal glbs and lubs makes type inference more brittle and more unpredictable.

We propose DOT as a core calculus for path-dependent types. We present the calculus formally in section 2 and through examples in section 3. Though we show that the calculus does not satisfy the standard theorem of preservation in section 4, we contribute a proof sketch of type safety using logical relations in section 5. In section 6, we discuss choices and variants of the calculus, as well as related work, and conclude in section 7.

2. The DOT Calculus

The DOT calculus is a small system of dependent object-types. Figure 1 gives its syntax, reduction rules, and type assignment rules.

2.1 Notation

We use standard notational conventions for sets. The notation $\overline{X}$ denotes a set of elements X. Given such a set $\overline{X}$ in a typing rule, X_i denotes an arbitrary element of X. We use an abbreviation for preconditions in typing judgements. Given an environment Γ and

FOOL '12 October 22, 2012, Tucson, AZ, USA.

Syntax

x, y, z	Variable	$L ::=$	Type label
l	Value label	$\quad L_c$	class label
m	Method label	$\quad L_a$	abstract type label
$v ::=$	Value	$S, T, U, V, W ::=$	Type
$\quad x$	variable	$\quad p.L$	type selection
$t ::=$	Term	$\quad T\left\{z \Rightarrow \overline{D}\right\}$	refinement
$\quad v$	value	$\quad T \wedge T$	intersection type
$\quad \textbf{val}\ x = \textbf{new}\ c;\ t$	new instance	$\quad T \vee T$	union type
$\quad t.l$	field selection	$\quad \top$	top type
$\quad t.m(t)$	method invocation	$\quad \bot$	bottom type
$p ::=$	Path	$S_c, T_c ::=$	Concrete type
$\quad x$	variable	$\quad p.L_c \mid T_c\left\{z \Rightarrow \overline{D}\right\} \mid T_c \wedge T_c \mid \top$	
$\quad p.l$	selection	$D ::=$	Declaration
$c ::= T_c\left\{\overline{d}\right\}$	Constructor	$\quad L : S..U$	type declaration
$d ::=$	Initialization	$\quad l : T$	value declaration
$\quad l = v$	field initialization	$\quad m : S \to U$	method declaration
$\quad m(x) = t$	method initialization		
$s ::= \overline{x \mapsto c}$	Store	$\Gamma ::= \overline{x : T}$	Environment

Reduction $\boxed{t \mid s \ \to\ t' \mid s'}$

$$\frac{y \mapsto T_c\left\{\overline{l = v'}\ \overline{m(x) = t}\right\} \in s}{y.m_i(v) \mid s \ \to\ [v/x_i]t_i \mid s} \quad (\textsc{msel})$$

$$\frac{y \mapsto T_c\left\{\overline{l = v}\ \overline{m(x) = t}\right\} \in s}{y.l_i \mid s \ \to\ v_i \mid s} \quad (\textsc{sel})$$

$$\textbf{val}\ x = \textbf{new}\ c;\ t \mid s \ \to\ t \mid s, x \mapsto c \quad (\textsc{new})$$

$$\frac{t \mid s \ \to\ t' \mid s'}{e[t] \mid s \ \to\ e[t'] \mid s'} \quad (\textsc{context})$$

where evaluation context $\quad e ::= [\,] \mid e.m(t) \mid v.m(e) \mid e.l$

Type Assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\textsc{var})$$

$$\frac{\Gamma \vdash t \ni l : T'}{\Gamma \vdash t.l : T'} \quad (\textsc{sel})$$

$$\frac{\Gamma \vdash t \ni m : S \to T \quad \Gamma \vdash t' : T' \ ,\ T' <: S}{\Gamma \vdash t.m(t') : T} \quad (\textsc{msel})$$

$$\frac{\begin{array}{c} y \notin \mathit{fn}(T') \\ \Gamma \vdash T_c\ \textbf{wfe}\ ,\ T_c \prec_y \overline{L : S..U}, \overline{D} \\ \Gamma, y : T_c \vdash \overline{S <: U}\ ,\ \overline{d} : \overline{D}\ ,\ t' : T' \end{array}}{\Gamma \vdash \textbf{val}\ y = \textbf{new}\ T_c\left\{\overline{d}\right\};\ t' : T'} \quad (\textsc{new})$$

Declaration Assignment $\boxed{\Gamma \vdash d : D}$

$$\frac{\Gamma \vdash v : V' \ ,\ V' <: V}{\Gamma \vdash (l = v) : (l : V)} \quad (\textsc{vdecl})$$

$$\frac{\Gamma \vdash S\ \textbf{wfe} \quad \Gamma, x : S \vdash t : T' \ ,\ T' <: T}{\Gamma \vdash (m(x) = t) : (m : S \to T)} \quad (\textsc{mdecl})$$

Figure 1. The DOT Calculus : Syntax, Reduction, Type / Declaration Assignment

some predicates P and Q, the condition $\Gamma \vdash P\ ,\ Q$ is a shorthand for the two conditions $\Gamma \vdash P$ and $\Gamma \vdash Q$.

2.2 Syntax

There are four alphabets: Variable names x, y, z are freely alpha-renamable. They occur as parameters of methods, as binders for objects created by new-expressions, and as self references in refinements. Value labels l denote fields in objects, which are bound to values at run-time. Similarly, method labels m denote methods in objects. Type labels L denote type members of objects. Type labels are further separated into labels for abstract types L_a and labels for classes L_c. It is assumed that in each program every class label L_c is declared at most once.

We assume that the label alphabets l, m and L are finite. This is not a restriction in practice, because one can include in these alphabets every label occurring in a given program.

The terms t in DOT consist of variables x, y, z, field selections $t.l$, method invocations $t.m(t)$ and object creation expressions $\textbf{val}\ y = \textbf{new}\ c;\ t'$ where c is a constructor $T_c\left\{\overline{l = v}\ \overline{m(x) = t}\right\}$. The latter binds a variable y to a new instance of type T_c with fields $\overline{l}$ initialized to values $\overline{v}$ and methods $\overline{m}$ initialized to methods of one parameter $\overline{x}$ and body $\overline{t}$. The scope of y extends through the term t'.

Two sub-sorts of terms are values and paths. Values v consist of just variables. Paths p consist of just variables and field selections.

The types in DOT are denoted by letters S, T, U, V, or W. They consist of the following:

- Type selections $p.L$, which denote the type member L of path p.
- Refinement types $T\left\{z \Rightarrow \overline{D}\right\}$, which refine a type T by a set of declarations D. The variable z refers to the "self"-reference of the type. Declarations can refer to other declarations in the same type by selecting from z.
- Type intersections $T \wedge T'$, which carry the declarations of members present in either T or T'.
- Type unions $T \vee T'$, which carry only the declarations of members present in both T and T'.
- A top type $\top$, which corresponds to an empty object.
- A bottom type $\bot$, which represents a non-terminating computation.

A subset of types T_c are called *concrete types*. These are type selections $p.L_c$ of class labels, the top type $\top$, intersections of concrete types, and refinements $T_c\left\{z \Rightarrow \overline{D}\right\}$ of concrete types. Only concrete types are allowed in constructors c.

There are only three forms of declarations in DOT, which are all part of refinement types. A value declaration $l : T$ introduces a field with type T. A method declaration $m : S \to U$ introduces a method with parameter of type S and result of type U. A type declaration $L : S..U$ introduces a type member L with a lower bound type S and an upper bound type U. There are no type aliases, but a type alias can be simulated by a type declaration $L : T..T$ where the lower bound and the upper bound are the same type T.

Every field or type label can be declared only once in a set of declarations $\overline{D}$. A set of declarations can hence be seen as a map from labels to their declarations. Meets $\wedge$ and joins $\vee$ on sets of declarations are defined in Figure 2.

2.3 Reduction rules

Reduction rules $t \mid s \ \to\ t' \mid s'$ in DOT rewrite pairs of terms t and stores s, where stores map variables to constructors. There are three main reduction rules: Rule (MSEL) rewrites a method invocation $y.m_i(v)$ by retrieving the corresponding method definition from the store, and performing a substitution of the argument for the parameter in the body. Rule (SEL) rewrites a field selection $x.l$ by retrieving the corresponding value from the store. Rule (NEW) rewrites an object creation $\textbf{val}\ x = \textbf{new}\ c;\ t$ by placing the binding of variable x to constructor c in the store and continuing with term t. These reduction rules can be applied anywhere in a term where the hole $[\,]$ of an evaluation context e can be situated.

2.4 Type assignment rules

The last part of Figure 1 presents rules for type assignment.

Rules (SEL) and (MSEL) type field selections and method invocations by means of an auxiliary membership relation $\ni$, which determines whether a given term contains a given declaration as one of its members. The membership relation is defined in Figure 3 and is further explained in section 2.5.

The last rule, (NEW), assigns types to object creation expressions. It is the most complex of DOT's typing rules. To type-check an object creation $\textbf{val}\ y\ =\ \textbf{new}\ T_c\left\{\overline{l = v}\ \overline{m(x) = t}\right\};\ t'$, one verifies first that the type T_c is well-formed (see Figure 5 for a definition of well-formedness). One then determines the set of all declarations that this type carries, using the expansion relation $\prec$ defined in Figure 3. Every type declaration $L : S..U$ in this set must be realizable, i.e. its lower bound S must be a subtype of its upper bound U. Every field declaration $l : V$ in this set must have a corresponding initializing value of v of type V. These checks are made in an environment which is extended by the binding $y : T_c$. In particular this allows field values that recurse on "self" by referring to the bound variable x. Similarly, every method declaration $m : T \to W$ must have a corresponding initializing method definition $m(x) = t$. The parameter type T must be **wfe** (well-formed and expanding; see Figure 5), and the body t must type check to W in an environment extended by the bindings $y : T_c$ and $x : T$.

Instead of adding a separate subsumption rule, subtyping is expressed by preconditions in rules (MSEL) and (NEW).

2.5 Membership

Figure 3 presents typing rules for membership and expansion. The membership judgement $\Gamma \vdash t \ni D$ states that in environment Γ a term t has a declaration D as a member. The membership rules rely on expansion. There are two rules, one for paths (PATH-$\ni$) and one for general terms (TERM-$\ni$). For general terms, the "self"-reference of the type must not occur in the resulting declaration D, since, to guarantee syntactic validity, we can only substitute a path for the "self"-reference.

2.6 Expansion

The expansion relation $\prec$ is needed to typecheck the complete set of declarations carried by a concrete type that is used in a **new**-expression. Expansion is also used by the membership rules and in subtyping refinements on the right (see Figure 4).

Rule (RFN-$\prec$) states that a refinement type $T \prec_z \overline{D}$ expands to the conjunction of the expansion $\overline{D'}$ of T and the newly added declarations $\overline{D}$. Rule (TSEL-$\prec$) states that a type selection $p.L$ carries the same declarations as the upper bound U of L in T. Rules ($\wedge$-$\prec$) and ($\vee$-$\prec$) states that expansion distributes through meets and joins. Rule ($\top$-$\prec$) states that the top type $\top$ expands to the empty set. Rule ($\bot$-$\prec$) states that the bottom type $\bot$ expands to the bottom element $\overline{D_\bot}$ of the lattice of sets of declarations (recall Figure 2).

2.7 Subtyping

Figure 4 defines the subtyping judgement $\Gamma \vdash S <: T$ which states that in environment Γ type S is a subtype of type T. Sub-

$$
\begin{array}{lcll}
\mathit{dom}(\overline{D} \wedge \overline{D'}) & = & \mathit{dom}(\overline{D}) \cup \mathit{dom}(\overline{D'}) & \\
\mathit{dom}(\overline{D} \vee \overline{D'}) & = & \mathit{dom}(\overline{D}) \cap \mathit{dom}(\overline{D'}) & \\
(D \wedge D')(L) & = & L : (S \vee S')..(U \wedge U') & \text{if } (L : S..U) \in \overline{D} \text{ and } (L : S'..U') \in \overline{D'} \\
 & = & D(L) & \text{if } L \notin \mathit{dom}(\overline{D'}) \\
 & = & D'(L) & \text{if } L \notin \mathit{dom}(\overline{D}) \\
(D \wedge D')(m) & = & m : (S \vee S') \to (U \wedge U') & \text{if } (m : S \to U) \in \overline{D} \text{ and } (m : S' \to U') \in \overline{D'} \\
 & = & D(m) & \text{if } m \notin \mathit{dom}(\overline{D'}) \\
 & = & D'(m) & \text{if } m \notin \mathit{dom}(\overline{D}) \\
(D \wedge D')(l) & = & l : T \wedge T' & \text{if } (l : T) \in \overline{D} \text{ and } (l : T') \in \overline{D'} \\
 & = & D(l) & \text{if } l \notin \mathit{dom}(\overline{D'}) \\
 & = & D'(l) & \text{if } l \notin \mathit{dom}(\overline{D}) \\
(D \vee D')(L) & = & L : (S \wedge S')..(U \vee U') & \text{if } (L : S..U) \in \overline{D} \text{ and } (L : S'..U') \in \overline{D'} \\
(D \vee D')(m) & = & m : (S \wedge S') \to (U \vee U') & \text{if } (m : S \to U) \in \overline{D} \text{ and } (m : S' \to U') \in \overline{D'} \\
(D \vee D')(l) & = & l : T \vee T' & \text{if } (l : T) \in \overline{D} \text{ and } (l : T') \in \overline{D'}
\end{array}
$$

Sets of declarations form a lattice with the given meet $\wedge$ and join $\vee$, the empty set of declarations as the top element, and the bottom element $\overline{D_\bot}$, Here $\overline{D_\bot}$ is the set of declarations that contains for every term label l the declaration $l : \bot$, for every type label L the declaration $L : \top..\bot$ and for every method label m the declaration $m : \top \to \bot$.

Figure 2. The DOT Calculus : Declaration Lattice

Membership $\boxed{\Gamma \vdash t \ni D}$

$$
\frac{\Gamma \vdash p : T \ , \ T \prec_z \overline{D}}{\Gamma \vdash p \ni [p/z]D_i} \ (\text{PATH-}\ni) \qquad \frac{z \notin \mathit{fn}(D_i) \qquad \Gamma \vdash t : T \ , \ T \prec_z \overline{D}}{\Gamma \vdash t \ni D_i} \ (\text{TERM-}\ni)
$$

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

$$
\frac{\Gamma \vdash T \prec_z \overline{D'}}{\Gamma \vdash T\left\{z \Rightarrow \overline{D}\right\} \prec_z \overline{D'} \wedge \overline{D}} \ (\text{RFN-}\prec) \qquad \frac{\Gamma \vdash p \ni L : S..U \ , \ U \prec_z \overline{D}}{\Gamma \vdash p.L \prec_z \overline{D}} \ (\text{TSEL-}\prec)
$$

$$
\frac{\Gamma \vdash T_1 \prec_z \overline{D}_1 \ , \ T_2 \prec_z \overline{D}_2}{\Gamma \vdash T_1 \wedge T_2 \prec_z \overline{D}_1 \wedge \overline{D}_2} \ (\wedge\text{-}\prec) \qquad \frac{\Gamma \vdash T_1 \prec_z \overline{D}_1 \ , \ T_2 \prec_z \overline{D}_2}{\Gamma \vdash T_1 \vee T_2 \prec_z \overline{D}_1 \vee \overline{D}_2} \ (\vee\text{-}\prec)
$$

$$
\Gamma \vdash \top \prec_z \{\} \ (\top\text{-}\prec) \qquad \Gamma \vdash \bot \prec_z \overline{D_\bot} \ (\bot\text{-}\prec)
$$

Figure 3. The DOT Calculus : Membership and Expansion

typing is regular wrt **wfe**: if type S is a subtype of type T, then S and T are well-formed and expanding. Though this regularity limits our calculus to **wfe**-types, this limitation allows us to show that subtyping is transitive, as discussed in section 6.2.1.

2.8 Declaration Subsumption

The declaration subsumption judgement $\Gamma \vdash D <: D'$ in Figure 4 states that in environment Γ the declaration D subsumes the declaration D'. There are three rules, one for each kind (type, value, method) of declarations. Rule (TDECL-<:) states that a type declaration $L : S..U$ subsumes another type declaration $L : S'..U'$ if S' is a subtype of S and U is a subtype of U'. In other words, the set of types between S and U is contained in the set of types between S' and U'. Rule (VDECL-<:) states that a value declaration $l : T$ subsumes another value declaration $l : T'$ if T is a subtype of T'. Rule (MDECL-<:) is similar to (TDECL-<:), as the parameter type varies contravariantly and the return type covariantly.

Declaration subsumption is extended to a binary relation between sequences of declarations: $\overline{D} <: \overline{D'}$ iff $\forall D'_i, \exists D_j . D_j <: D'_i$.

2.9 Well-formedness

The well-formedness judgement $\Gamma \vdash T$ **wf** in Figure 5 states that in environment Γ the type T is well-formed.

A refinement type $T\left\{z \Rightarrow \overline{D}\right\}$ is well-formed if the parent type T is well-formed and every declaration in $\overline{D}$ is well-formed in an environment augmented by the binding of the self-reference z to the refinement type itself (RFN-WF).

A type selection $p.L$ is well-formed if L is a member of p, and the lower bound of L is also well-formed (TSEL-WF$_1$ and TSEL-WF$_2$). The latter condition has the effect that the lower bound of a type $p.L$ may not refer directly or indirectly to a type containing $p.L$ itself — if it would, the well-formedness judgement of $p.L$ would not have a finite proof. No such restriction exists for the upper bound of L if the lower bound is $\bot$ (TSEL-WF$_2$). The upper

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

$$\frac{\Gamma \vdash T \ \mathbf{wfe}}{\Gamma \vdash T <: T} \ (\text{REFL}) \qquad \frac{\Gamma \vdash T \ \mathbf{wfe}}{\Gamma \vdash \bot <: T} \ (\bot\text{-}<:)$$

$$\frac{\Gamma \vdash T\{z \Rightarrow \overline{D}\} \ \mathbf{wfe}\ ,\ S <: T\ ,\ S \prec_z \overline{D'} \quad \Gamma, z : S \vdash \overline{D'} <: \overline{D}}{\Gamma \vdash S <: T\{z \Rightarrow \overline{D}\}} \ (<:\text{-RFN}) \qquad \frac{\Gamma \vdash T\{z \Rightarrow \overline{D}\} \ \mathbf{wfe}\ ,\ T <: T'}{\Gamma \vdash T\{z \Rightarrow \overline{D}\} <: T'} \ (\text{RFN}\text{-}<:)$$

$$\frac{\Gamma \vdash p \ni L : S..U\ ,\ S <: U\ ,\ S' <: S}{\Gamma \vdash S' <: p.L} \ (<:\text{-TSEL}) \qquad \frac{\Gamma \vdash p \ni L : S..U\ ,\ S <: U\ ,\ U <: U'}{\Gamma \vdash p.L <: U'} \ (\text{TSEL}\text{-}<:)$$

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

$$\frac{\Gamma \vdash T_2 \ \mathbf{wfe}\ ,\ T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2} \ (<:\text{-}\vee_1) \qquad \frac{\Gamma \vdash T_2 \ \mathbf{wfe}\ ,\ T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \ (\wedge_1\text{-}<:)$$

$$\frac{\Gamma \vdash T_1 \ \mathbf{wfe}\ ,\ T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2} \ (<:\text{-}\vee_2) \qquad \frac{\Gamma \vdash T_1 \ \mathbf{wfe}\ ,\ T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \ (\wedge_2\text{-}<:)$$

$$\frac{\Gamma \vdash T \ \mathbf{wfe}}{\Gamma \vdash T <: \top} \ (<:\text{-}\top)$$

Declaration subsumption $\boxed{\Gamma \vdash D <: D'}$

$$\frac{\Gamma \vdash S' <: S\ ,\ T <: T'}{\Gamma \vdash (L : S..T) <: (L : S'..T')} \ (\text{TDECL}\text{-}<:) \qquad \frac{\Gamma \vdash S' <: S\ ,\ T <: T'}{\Gamma \vdash (m : S \to T) <: (m : S' \to T')} \ (\text{MDECL}\text{-}<:)$$

$$\frac{\Gamma \vdash T <: T'}{\Gamma \vdash (l : T) <: (l : T')} \ (\text{VDECL}\text{-}<:)$$

Figure 4. The DOT Calculus : Subtyping and Declaration Subsumption

bound may in fact refer back to the type. Hence, recursive class types and F-bounded abstract types are both expressible.

The other forms of types in DOT are all well-formed if their constituent types are well-formed.

Well-formedness extends straightforwardly to declarations with the judgement $\Gamma \vdash D$ **wf**. All declarations are well-formed if their constituent types are well-formed.

3. Examples

3.1 Sugar: Functions

Like in Scala, we can encode functions as objects with a special method. We will freely use the following sugar in the remaining of this paper. Note that the variable z must be fresh.

$$\begin{aligned} S \to_s T &\iff \top\{z \Rightarrow \mathit{apply} : S \to T\} \\ \mathbf{fun}\ (x : S)\ T\ t &\iff \mathbf{val}\ z = \mathbf{new}\ S \to_s T\ \{\mathit{apply}(x) = t\}\ ;\ z \\ (\mathbf{app}\ f\ x) &\iff f.\mathit{apply}(x) \\ (\mathbf{cast}\ T\ t) &\iff (\mathbf{app}\ (\mathbf{fun}\ (x : T)\ T\ x)\ t) \end{aligned}$$

In the remaining examples, for brevity, we will also use $\lambda x\!:\!S.t$ for $\mathbf{fun}\ (x : S)\ \_\ t$ where the return type can be somewhat inferred from the context.

3.2 Basics: Booleans, Error, ...

This program defines a *root* object with basic types (Unit, $\mathit{Boolean}$, Nat) and methods (true, false, error, zero). For brevity, we've omitted the code related to natural numbers. During the object creation, the method labels such as false are all initialized.

$$\begin{aligned} &\mathbf{val}\ \mathit{root} = \mathbf{new}\ \top\{r \Rightarrow \\ &\ \mathit{Unit} : \bot..\top \\ &\ \mathit{unit} : \top \to r.\mathit{Unit} \\ &\ \mathit{Boolean} : \bot..\top\{z \Rightarrow \\ &\qquad \mathit{ifNat} : (r.\mathit{Unit} \to_s r.\mathit{Nat}) \to_s (r.\mathit{Unit} \to_s r.\mathit{Nat}) \to_s r.\mathit{Nat} \\ &\qquad \} \\ &\ \mathit{false} : r.\mathit{Unit} \to r.\mathit{Boolean} \\ &\ \mathit{true} : r.\mathit{Unit} \to r.\mathit{Boolean} \\ &\ \mathit{error} : r.\mathit{Unit} \to \bot \\ &\ \ldots \\ &\}\{ \\ &\ \mathit{unit}(x) = \mathbf{val}\ u = \mathbf{new}\ \mathit{root}.\mathit{Unit};\ u \\ &\ \mathit{false}(u) = \\ &\qquad \mathbf{val}\ \mathit{ff} = \mathbf{new}\ \mathit{root}.\mathit{Boolean}\{ \end{aligned}$$

Well-formed types $\boxed{\Gamma \vdash T \textbf{ wf}}$

$$\frac{\Gamma \vdash T \textbf{ wf} \quad \Gamma, z : T\{z \Rightarrow \overline{D}\} \vdash \overline{D \textbf{ wf}}}{\Gamma \vdash T\{z \Rightarrow \overline{D}\} \textbf{ wf}} \text{ (RFN-WF)} \qquad \Gamma \vdash \top \textbf{ wf} \text{ (}\top\text{-WF)}$$

$$\Gamma \vdash \bot \textbf{ wf} \text{ (}\bot\text{-WF)}$$

$$\frac{\Gamma \vdash p \ni L : S..U \;,\; S \textbf{ wf} \;,\; U \textbf{ wf}}{\Gamma \vdash p.L \textbf{ wf}} \text{ (TSEL-WF}_1\text{)} \qquad \frac{\Gamma \vdash p \ni L : \bot..U}{\Gamma \vdash p.L \textbf{ wf}} \text{ (TSEL-WF}_2\text{)}$$

$$\frac{\Gamma \vdash T \textbf{ wf} \;,\; T' \textbf{ wf}}{\Gamma \vdash T \wedge T' \textbf{ wf}} \text{ (}\wedge\text{-WF)} \qquad \frac{\Gamma \vdash T \textbf{ wf} \;,\; T' \textbf{ wf}}{\Gamma \vdash T \vee T' \textbf{ wf}} \text{ (}\vee\text{-WF)}$$

Well-formed declarations $\boxed{\Gamma \vdash D \textbf{ wf}}$

$$\frac{\Gamma \vdash S \textbf{ wf} \;,\; U \textbf{ wf}}{\Gamma \vdash L : S..U \textbf{ wf}} \text{ (TDECL-WF)} \qquad \frac{\Gamma \vdash S \textbf{ wf} \;,\; U \textbf{ wf}}{\Gamma \vdash m : S \to U \textbf{ wf}} \text{ (MDECL-WF)}$$

$$\frac{\Gamma \vdash T \textbf{ wf}}{\Gamma \vdash l : T \textbf{ wf}} \text{ (VDECL-WF)}$$

Well-formed and expanding types $\boxed{\Gamma \vdash T \textbf{ wfe}}$

$$\frac{\Gamma \vdash T \textbf{ wf} \;,\; T \prec_z \overline{D}}{\Gamma \vdash T \textbf{ wfe}} \text{ (WFE)}$$

Figure 5. The DOT Calculus : Well-Formedness

```
        ifNat = λt: root.Unit →s root.Nat.
            λe: root.Unit →s root.Nat.
            (app e root.unit)
    };
    ff
 true(u) = ...
 error(u) = (app root.error u)
 ...
};
...
```

3.3 Lists

Polymorphic lists can be expressed using an abstract type member for the element type (*Elem*). We can instantiate a refinement of the list package to manipulate lists with a particular element type.

```
val genLists = new ⊤{g ⇒ ListPackage : ⊥..⊤{p ⇒
 Elem : ⊥..⊤
 List : ⊥..⊤{z ⇒
      isEmpty : root.Unit → root.Boolean
      head : root.Unit → p.Elem
      tail : root.Unit → p.List
     }
 nil : root.Unit → p.List
 cons : p.Elem → p.List →s p.List
}};
val natLists = new genLists.ListPackage{p ⇒
    Elem : root.Nat..root.Nat}{
 nil(u) = ...
 cons(hd) = ...
};
...
```

4. Counterexamples to Preservation

We sketch a proof that the DOT calculus is type-safe using logical relations in section 5. However, we first tried to prove the calculus type-safe using the standard theorems of preservation and progress [13, 14]. Unfortunately, for the calculus as presented, and any of its variants that we devised, preservation doesn't hold. In this section, we review some of the most salient counterexamples to preservation that we found. These counterexamples have been checked with PLT Redex [11].

4.1 TERM-∋ Restriction

There are two membership ($t \ni D$) rules: one for when the term t is a path, and one for an arbitrary term t. For paths, we can substitute the self-references in the declarations, but we cannot do so for arbitrary terms as the resulting types wouldn't be well-formed syntactically. Hence, the TERM-∋ has the restriction that self-occurrences are not allowed. Here is a counterexample related to this restriction.

Let X be a shorthand for the type:

$$\begin{array}{l}\top\{z \Rightarrow \\ \quad L_a : \top..\top \\ \quad l : z.L_a \\ \}\end{array}$$

Let Y be a shorthand for the type:

$$\begin{array}{l}\top\{z \Rightarrow \\ \quad l : \top \\ \}\end{array}$$

Now, consider the term

$$\begin{array}{l}\textbf{val}\ u = \textbf{new}\ X\ \{l = u\}\,; \\ (\textbf{app}\ (\lambda y{:}\top \to Y.(\textbf{app}\ y\ u))\ (\lambda d{:}\top.(\textbf{cast}\ X\ u))).l\end{array}$$

The term type-checks because the term $t = (\textbf{app}\ (\lambda y : \top \to Y.(\textbf{app}\ y\ u))\ (\lambda d : \top.(\textbf{cast}\ X\ u)))$ has type Y, so we can apply TERM-$\ni$ for l. However, the term t eventually steps to $(\textbf{cast}\ X\ u)$ which has type X, so we cannot apply TERM-$\ni$ for l because of the self-reference $(z.L_a)$.

4.2 Expansion Lost

Expansion is not preserved by narrowing. Here, we create two type selections that are mutually recursive in their upper bounds after narrowing: $z_0.C_2$ initially expands, but after narrowing, $z_0.C_2$ expands to what $z_0.A_2$ expands to, which expands to what $z_0.A_1$ expands to, which expands to what $z_0.A_2$ expands to, and thus we have an infinite expansion. Thus, the last new expression initially type-checks, but after narrowing, it doesn't because the precise expansion needed by NEW cannot be inferred.

$$\begin{array}{l}\textbf{val}\ x_0 = \textbf{new}\ \top\{z \Rightarrow A_1 : \bot..\top\{z \Rightarrow \\ \quad A_2 : \bot..\top \\ \quad A_3 : \bot..\top \\ \quad C_2 : \bot..z.A_2\}\}\,\{\}\,; \\ \textbf{val}\ x_1 = \textbf{new}\ \top\,\{z \Rightarrow C_1 : \bot..\top\,\{z \Rightarrow A_1 : \bot..x_0.A_1\}\}\,\{\}\,; \\ \textbf{val}\ x_2 = \textbf{new}\ x_1.C_1\,\{z \Rightarrow A_1 : \bot..x_0.A_1\,\{z \Rightarrow A_2 : \bot..z.A_3\}\}\,\{\}\,; \\ \textbf{val}\ x_3 = \textbf{new}\ x_1.C_1\,\{z \Rightarrow A_1 : \bot..x_0.A_1\,\{z \Rightarrow A_3 : \bot..z.A_2\}\}\,\{\}\,; \\ (\textbf{app}\ \lambda x{:}x_1.C_1.(\lambda z_0{:}x.A_1 \wedge x_3.A_1. \\ \qquad\qquad \textbf{val}\ z = \textbf{new}\ z_0.C_2;\ (\textbf{app}\ (\lambda x{:}\top.x)\ z)) \\ x_2)\end{array}$$

4.3 Well-Formedness Lost

Even well-formedness is not preserved by narrowing. The trick is that if the lower bound of a type selection is not $\bot$, then the bounds needs to be checked for well-formedness. Here, we create two type selections that are mutually recursive in their bounds after narrowing. $y.A$ is initially well-formed, but after narrowing, it isn't because we run into an infinite derivation trying to prove the well-formedness of its bounds.

$$\begin{array}{l}\textbf{val}\ v = \textbf{new}\ \top\,\{z \Rightarrow L : \bot..\top\,\{z \Rightarrow A : \bot..\top, B : z.A..z.A\}\}\,\{\}\,; \\ (\textbf{app}\ (\lambda x{:}\top\,\{z \Rightarrow L : \bot..\top\,\{z \Rightarrow A : \bot..\top, B : \bot..\top\}\}\,. \\ \quad \textbf{val}\ z = \textbf{new}\ \top\{z \Rightarrow \\ \qquad\qquad l : x.L \wedge \top\,\{z \Rightarrow A : z.B..z.B, B : \bot..\top\} \to \top\}\{ \\ \qquad\qquad l(y) = \textbf{fun}\ (a : y.A)\ \top\ a\}; \\ \quad (\textbf{cast}\ \top\ z)) \\ v)\end{array}$$

4.4 Path Equality

For preservation, we need to be able to relate path-dependent types after reduction. Here is a motivating example:

$$\begin{array}{l}\textbf{val}\ b = \textbf{new}\ \top\{z \Rightarrow \qquad X : \top..\top \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad l : z.X \qquad \}\,\{l = b\}\,; \\ \textbf{val}\ a = \textbf{new}\ \top\{z \Rightarrow i : \top\{z \Rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad X : \bot..\top \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad l : z.X\} \qquad \}\,\{i = b\}\,; \\ (\textbf{app}\ (\lambda x{:}\top.x)\ (\textbf{app}\ (\lambda x{:}a.i.X.x)\ a.i.l))\end{array}$$

$a.i.l$ reduces to $b.l$. $b.l$ has type $b.X$, so we need $b.X <: a.i.X$. This cannot be established with the current rules: it is not true in general, but true here because $a.i$ reduces to b. Hence, we need to acknowledge path equality for preservation to hold.

In section 6.2.3, we discuss our failure to patch the calculus for preservation to hold.

5. Type Safety via Logical Relations

We sketch a proof of type-safety of the DOT calculus via step-indexed logical relations [1, 2, 10].

5.1 Type Safety

Type-safety states that a well-typed program doesn't get stuck. More formally: If $\emptyset \vdash t : T$ and $t \,|\, \emptyset \ \to^*\ t' \,|\, s'$ then either t' is a value or $\exists t'', s''.t' \,|\, s' \to t'' \,|\, s''$.

Our strategy is to define a logical relation $\Gamma \vDash t : T$, such that $\Gamma \vdash t : T$ implies $\Gamma \vDash t : T$ implies type-safety.

5.2 Step-Indexed Logical Relations

In order to ensure that our logical relation is well-founded, we use a step index. For each step index k, we define the set of values and the set of terms that appear to belong to a given type, when taking at most k steps. $\Gamma \vDash t : T$ is then defined in terms of the step-indexed logical relation by requiring it to hold $\forall k$.

5.2.1 Set of Values

$\mathcal{V}_{k;\Gamma;s}[\![T]\!]$ defines the set of values that appear to have type T when taking at most k steps. Γ and s must agree: $dom(\Gamma) = dom(s)$ (ordered) and $\forall(x : T) \in \Gamma, x \in \mathcal{V}_{k;\Gamma;s}[\![T]\!]$. A variable y belongs to $\mathcal{V}_{0;\Gamma;s}[\![T]\!]$ simply by being in the store. In addition, it belongs to $\mathcal{V}_{k;\Gamma;s}[\![T]\!]$ for $k > 0$, if it defines all type, method and value labels

in the expansion of T appropriately for $j < k$ steps.

$$
\begin{array}{l}
\mathcal{V}_{k;\Gamma;s}[\![T]\!] = \{y \mid y \in \mathit{dom}(s) \ \wedge \ (\\
(\Gamma \vdash T \ \mathbf{wfe} \ \wedge \\
\quad \forall j < k, \\
\quad y \mapsto T_c \left\{ \overline{l = v} \ \overline{m(x) = t} \right\} \in s, \\
\quad \Gamma \vdash T \prec_y \overline{D}, \\
\quad (\forall L_i : S \to U \in \overline{D}, \\
\qquad \Gamma \vdash y \ni L_i : S'..U') \wedge \\
\quad (\forall m_i : S \to U \in \overline{D}, \\
\qquad t_i \in \mathcal{E}_{j;\Gamma,x_i:S;s}[\![U]\!]) \wedge \\
\quad (\forall l_i : V \in \overline{D}, \\
\qquad v_i \in \mathcal{V}_{j;\Gamma;s}[\![V]\!]))\vee \\
(T = T_1 \wedge T_2 \ \wedge \ y \in \mathcal{V}_{k;\Gamma;s}[\![T_1]\!] \ \wedge \ y \in \mathcal{V}_{k;\Gamma;s}[\![T_2]\!])\vee \\
(T = T_1 \vee T_2 \ \wedge \ (y \in \mathcal{V}_{k;\Gamma;s}[\![T_1]\!] \vee y \in \mathcal{V}_{k;\Gamma;s}[\![T_2]\!])) \\
)\}
\end{array}
$$

This relation captures the observation that the only ways for a term to get stuck is to have a field selection on an uninitialized field or a method invocation on an uninitialized method. However, a *potential pitfall* is that the value itself might occur in the types S, U, V, because we substitute it for the "self" occurrences in the expansion, so the relation makes sure that the required type labels exist.

5.2.2 Set of Terms

$\mathcal{E}_{k;\Gamma;s}[\![T]\!]$ defines the set of terms that appear to have type T when taking at most k steps. s must agree with a *prefix* of Γ, so Γ can additionally contain variables not in s. This is needed for checking methods in $\mathcal{V}$ above, and for relating open terms. If $k > 0$, $\mathcal{E}$ extends Γ and s so that they agree. It then states that if it can reduce t in the extended store to an irreducible term in $j < k$ steps, then this term must be in a corresponding $\mathcal{V}$ set with Γ now extended to agree with the store resulting from the reduction steps.

irred (t, s) is a shorthand for $\neg\exists t', s'.t \mid s \ \rightarrow \ t' \mid s'$. $\supseteq$ is used initially for the possibly shorter store to agree with the environment, and can extend both in many different ways. $\supseteq^!$ is used finally for the possibly shorter environment to agree with the store, and just extends the environment in one straightforward way: hence, it defines singleton sets.

$$
\begin{array}{l}
\mathcal{E}_{k;\Gamma;s}[\![T]\!] = \{t \mid \\
\quad k = 0 \vee (\forall j < k, \\
\qquad \forall (\Gamma'; s') \in \supseteq_k[\![\Gamma; s]\!], \\
\qquad t \mid s' \ \rightarrow^j \ t' \mid s'' \wedge \\
\qquad \text{irred } (t', s'') \rightarrow \\
\qquad \forall \Gamma'' \in \supseteq^!_{k;s''}[\![\Gamma']\!], \\
\qquad t' \in \mathcal{V}_{k-j-1;\Gamma'';s''}[\![T]\!]) \\
\}
\end{array}
$$

5.2.3 Extending the environment and the store

$\supseteq_k[\![\Gamma; s]\!]$ for $k > 0$ defines the set of completed environment and stores that agree on $k - 1$ steps, and that extend Γ and s. s must agree with a *prefix* of Γ. Both Γ and s are ordered maps. For s, s' extends s if s is a prefix of s'. For Γ, Γ' extends Γ if we get back Γ by keeping only the elements of Γ' that belong to Γ. Furthermore, a prefix of Γ' agrees with s.

$$
\begin{array}{l}
\supseteq_k[\![\Gamma; s]\!] = \{ \\
(\overline{x : T}^m, \overline{x_{ij} : T_{ij}}^{m \le i < n; 0 \le j < i_n}; s, \overline{x_{ij} \mapsto c_{ij}}^{m \le i < n; 0 \le j < i_n}) \mid \\
s = \overline{x \mapsto c}^m \ \wedge \ \Gamma = \overline{x : T}^n \ \wedge \\
m \le n \ \wedge \ \forall i, m \le i < n, \forall i_n, j, 0 \le j < i_n, \\
\forall T_{ij}, c_{ij}, T_{i(i_n-1)} = T_i, \forall n' \le n, i'_n \le i_n, \\
c_{ij} \in \mathcal{V}_{k-1;\overline{x:T}^m, \overline{x_{ij}:T_{ij}}^{m \le i' < n; 0 \le j < i'_n}; s, \overline{x_{ij} \mapsto c_{ij}}^{m \le i < n'; 0 \le j < i'_n}}[\![T_{ij}]\!] \\
\}
\end{array}
$$

5.2.4 Completing the environment to agree with the store

$\supseteq^!_{k;s}[\![\Gamma]\!]$ defines a singleton set of a completed environment that agrees with a store s by simply copying the constructor type from the store for each missing variable.

$$
\begin{array}{l}
\supseteq^!_{k;s}[\![\Gamma]\!] = \{\Gamma, \overline{x_i : T_{c_i}}^{m \le i < n} \mid \\
\qquad \Gamma = \overline{x : T}^m \ \wedge \ s = \overline{x \mapsto c}^n \\
\}
\end{array}
$$

5.2.5 Terms in the Logical Relation

$\Gamma \vDash t : T$ is simply defined as $t \in \mathcal{E}_{k;\Gamma;\emptyset}[\![T]\!], \forall k$.

5.3 Statements and Proofs

5.3.1 Fundamental Theorem

The fundamental theorem is the implication from $\Gamma \vdash t : T$ to $\Gamma \vDash t : T$. Type safety is a straightforward corollary of this theorem.

Proof: The proof is on induction on the derivation of $\Gamma \vdash t : T$. For each case, we need to show $t \in \mathcal{E}_{k;\Gamma;\emptyset}[\![T]\!], \forall k$. The non-trivial case is when $k > 0$ and for $(\Gamma'; s') \in \supseteq_k[\![\Gamma; s]\!]$ and some $j < k$, $t \mid s \ \rightarrow^j \ t' \mid s' \ \wedge$ irred (t', s'). Then, we need to show $t' \in \mathcal{V}_{k-j-1;\Gamma'';s'}[\![T]\!]$ for $\Gamma'' \in \supseteq^!_{\Gamma;k}[\![s']\!]\Gamma'$.

Case VAR: $\Gamma \vdash x : T$ knowing $(x : T) \in \Gamma$. $x \in \mathcal{V}_{k-1;\Gamma';s}[\![T]\!]$ follows from the definition of $\supseteq_k[\![\Gamma; \emptyset]\!]$.

Case SEL: $\Gamma \vdash t_1.l_i : T$ knowing $\Gamma \vdash t_1 : T_1, \Gamma \vdash T_1 \prec_z \overline{D}$, $l_i : V_i \in \overline{D}$ and knowing either that $t_1 = p_1 \ \wedge \ T = [p/z]V_i$ or that $z \notin \mathit{fn}(V_i) \ \wedge \ T = V_i$.

By operational semantics and induction hypothesis, $t_1 \mid s \ \rightarrow^{j-1} t'_1 \mid s'$ and irred (t'_1, s') and $t'_1 \in \mathcal{V}_{k-j+1-1;\Gamma';s'}[\![T_1]\!]$.

By operational semantics and the above, $t'_1.l_i \mid s' \ \rightarrow^1 \ t' \mid s'$, and we can conclude $t' \in \mathcal{V}_{k-j-1;\Gamma'';s'}[\![T]\!]$ from the clause for value labels of $t'_1 \in \mathcal{V}_{k-j;\Gamma'';s'}[\![T_1]\!]$.

Case MSEL: $\Gamma \vdash t_1.m_i(t_2) : T$ knowing $\Gamma \vdash t_1 : T_1$, $\Gamma \vdash t_2 : T_2, \Gamma \vdash T_1 \prec_z \overline{D}$, $m_i : S_i \to U_i \in \overline{D}$ and knowing either that $t_1 = p_1 \ \wedge \ S = [p/z]S_i \ \wedge \ T = [p/z]U_i$ or that $z \notin \mathit{fn}(S_i) \ \wedge \ z \notin \mathit{fn}(U_i) \ \wedge \ S = S_i \ \wedge \ T = U_i$, and knowing that $\Gamma \vdash T_2 <: S$.

By operational semantics and induction hypotheses, $t_1 \mid s \ \rightarrow^{j_1} t'_1 \mid s_1$ and irred (t'_1, s_1) and $t_2 \mid s \ \rightarrow^{j_2} \ t'_2 \mid s_2$ and irred (t'_2, s_2) and $t'_1 \in \mathcal{V}_{k-j_1-1;\Gamma_1;s_1}[\![T_1]\!]$ and $t'_2 \in \mathcal{V}_{k-j_2-1;\Gamma_2;s_2}[\![T_2]\!]$.

Because t_2 reduces to a value t'_2 starting in store s, it should also reduce to a value v_2 in the same number of steps starting in store s_1, since s_1 extends s. So let $t_2 \mid s_1 \ \rightarrow^{j_2} \ v_2 \mid s_{12}$ with $v_2 \in \mathcal{V}_{k-j_2-1;\Gamma_{12};s_{12}}[\![T_2]\!]$.

By the above and operational semantics, $t_1'.m_i(v_2) \mid s_{12} \ \rightarrow^1 [v_2/x_i]t_i \mid s_{12}$.

By the substitution lemma, $[v_2/x_i]t_i \in \mathcal{E}_{k-\max(j_1,j_2)-1;\Gamma_{12};s_{12}}[\![T]\!]$. Supposing, $[v_2/x_i]t_i \mid s_{12} \ \rightarrow^{j_3} \ t' \mid s'$, with $j_1 + j_2 + j_3 + 1 = j$, this completes the case, by monotonicity of $\mathcal{V}$.

Case NEW: $\Gamma \vdash$ **val** $y =$ **new** c; $t_b : T$ knowing ...

By operational semantics, **val** $y =$ **new** c; $t_b \mid s \ \rightarrow^1 \ t_b \mid s_b$ where $s_b = s, y \mapsto c$. So $t_b \mid s_b \ \rightarrow^{j-1} \ t' \mid s'$.

By induction hypotheses, $y \in \mathcal{V}_{k;\Gamma_b;s_b}[\![T_c]\!]$ and $t_b \in \mathcal{E}_{k;\Gamma_b;s_b}[\![T]\!]$.

Result follows by monotonicity of $\mathcal{V}$.

□

5.3.2 Substitution Lemma

The substitution lemma states that if (1) $v \in \mathcal{V}_{k_2;\Gamma_{12};s_{12}}[\![T_2]\!]$ and (2) $t \in \mathcal{E}_{k_1;\Gamma_1,x:S;s_1}[\![T]\!]$ and (3) $\Gamma \vdash T_2 <: S$ with (4) $x \notin fn(T)$ and Γ_1 extends Γ and Γ_{12} extends Γ_1 and s_{12} extends s_1 and Γ_1 agrees with s_1 and Γ_{12} agrees with s_{12} and a prefix of Γ_{12} agrees with s_1, then $[v/x]t \in \mathcal{E}_{\min(k_1,k_2);\Gamma_{12};s_{12}}[\![T]\!]$.

Proof Sketch: By (1) and (3), it should hold that (5) $v \in \mathcal{V}_{k_2;\Gamma_{12};s_{12}}[\![S]\!]$ by the subset semantics lemma. Since (2) holds, it should also hold that $t \in \mathcal{E}_{\min(k_1,k_2);\Gamma_{12},x:S;s_{12}}[\![T]\!]$ by the extended monotonicity lemma. Then, we can instantiate x in the complete store to map to what v maps to. This should be fine by (5) and monotonicity. Thus, $t \in \mathcal{E}_{\min(k_1,k_2);\Gamma_{12},x:S;s_{12},x\mapsto s_{12}(v)}[\![T]\!]$. Thanks to (4), we don't actually need x to be held abstract in the environment, because it won't occur in T or its expansion (a *potential pitfall* is whether its occurrences in t_i could still cause a check to fail through narrowing issues), so we can use the type of v in the environment instead of S for x: $t \in \mathcal{E}_{\min(k_1,k_2);\Gamma_{12},x:\Gamma_{12}(v);s_{12},x\mapsto s_{12}(v)}[\![T]\!]$. This implies what needs to be shown. □

5.3.3 Subset Semantics Lemma

The subset semantics lemma states that if $v \in \mathcal{V}_{k;\Gamma;s}[\![S]\!]$ and $\Gamma \vdash S <: U$, then $v \in \mathcal{V}_{k;\Gamma;s}[\![U]\!]$.

Proof Sketch: Because S is a subtype of U, it should hold that the expansion of S subsumes the expansion of U, when the "self" occurrences are of type S. Therefore, for $v \in \mathcal{V}_{k;\Gamma;s}[\![U]\!]$, we have fewer declarations to check than for $v \in \mathcal{V}_{k;\Gamma;s}[\![S]\!]$.

A *potential pitfall* is whether some types of the expansion of U can become non-expanding when the "self" occurrences are actually v instead of just abstractly of type S, causing a check to fail. Another worry is that such a non-expanding type results from narrowing of a parameter type. □

5.3.4 Extended Monotonicity Lemma

The extended monotonicity lemma states that if $t \in \mathcal{E}_{k;\Gamma,x:S;s}[\![T]\!]$ then $t \in \mathcal{E}_{j;\Gamma',x:S;s'}[\![T]\!]$ for $j \leq k$, Γ' extends Γ, s' extends s, and Γ agrees with s and a prefix of Γ' agrees with s.

Proof Sketch: For the monotonicity with regards to the step index, this follows directly from the definitions of $\mathcal{E}$ and $\mathcal{V}$. For the environment and the store, this follows by design from the definition of $\supseteq_k[\![\Gamma, x : S; s]\!]$. To extend the environment and the store for $x : S$, we can append as much as we want to Γ and s, to get Γ' and s', and then ignore the last element which is for $x : S$. □

6. Discussion

6.1 Why No Inheritance?

In the calculus we made the deliberate choice not to model any form of inheritance. This is, first and foremost, to keep the calculus simple. Secondly, there are many different approaches to inheritance and mixin composition, so that it looks advantageous not to tie the basic calculus to a specific one. Finally, it seems that the modelization of inheritance lends itself to a different approach than the basic calculus. For the latter, we need to prove type safety of the calculus. One might try to do this also for a calculus with inheritance, but our experience suggests that this complicates the proofs considerably. An alternative approach that might work better is to model inheritance as a form of code-reuse. Starting with an enriched type system with inheritance, and a translation to the basic calculus, one needs to show type safety wrt the translation. This might be easier than to prove type safety wrt reduction.

6.2 Variants of the DOT Calculus

6.2.1 Why limit the calculus to wfe-types?

Currently, the proof of type-safety via logical relations fundamentally relies on types having an expansion. However, this was not our original motivation for limiting the calculus to **wfe**-types.

Originally, subtyping was not regular wrt **wfe**. Roughly, all the **wfe** preconditions in subtyping were dropped. In this broader calculus, subtyping transitivity doesn't hold, because of the rule ($<:$-RFN) which requires expansion of the left type.

The problem is deep, as attested by this elaborate counterexample that is not so easily patched, and directly leads to a counterexample to preservation.

Consider an environment where u is bound to:

$$\begin{aligned}&\top\{u \Rightarrow\\ &\quad Bad : \bot..u.Bad\\ &\quad Good : \top\{z \Rightarrow L : \bot..\top\}..\top\{z \Rightarrow L : \bot..\top\}\\ &\quad Lower : u.Bad \wedge u.Good..u.Good\\ &\quad Upper : u.Good..u.Bad \vee u.Good\\ &\quad X : u.Lower..u.Upper\\ &\}\end{aligned}$$

Now, consider the types S, T, U defined in terms of u:

$$\begin{aligned}S &= u.Bad \wedge u.Good\\ T &= u.Lower\\ U &= u.X\{z \Rightarrow L : \bot..\top\}\end{aligned}$$

We have $S <: T$ and $T <: U$, but we cannot derive $S <: U$ because S doesn't expand.

Note that u is realizable, since each lower bound is a subtype of its upper bound. So it is straightforward to turn this counterexample to subtyping transitivity into a counterexample to preservation:

$$\begin{aligned}&\textbf{val } u = \textbf{new } \ldots;\\ &(\textbf{app } \lambda x : \top.x\\ &\ (\textbf{app } \lambda f : S \to U.f\\ &\ \ (\textbf{app } \lambda f : S \to T.f\\ &\ \ \ (\textbf{app } \lambda f : S \to S.f\\ &\ \ \ \ \lambda x : S.x))))\end{aligned}$$

The idea is to start with a function from $S \to S$ and cast it successively to $S \to T$ then $S \to U$. To typecheck the expression initially, we need to check $S <: T$ and $T <: U$. After some reduction steps, the first few casts vanish, and the reduced expression casts directly from $S \to S$ to $S \to U$, so we need to check $S <: U$.

6.2.2 Why not include the lambda-calculus instead of methods?

Originally, the DOT calculus included the lambda-calculus, and explicit methods were not needed since they could be represented

by a value label with a function type. However, the expansion of the function type was defined to be the empty set of declarations (like for $\top$), which caused a real breach of type-safety.

A concrete object could be a subtype of a function type without a function ever being defined. Consider:

$$\textbf{val } u = \textbf{new } \top\,\{z \Rightarrow C : \top \to \top..\top \to \top\}\,\{\}\,;$$
$$\textbf{val } f = \textbf{new } u.C\,\{\}\,;$$
$$\ldots$$

Now, f was a subtype of $\top \to \top$, but ($\textbf{app}\ f\ \ (\lambda x : \top.x)$) was stuck (and, rightfully, didn't typecheck). But we could use narrowing to create a counterexample to type safety: ($\textbf{app}\ (\lambda g : \top \to \top.(\textbf{app}\ g\ ()\lambda x\!:\!\top.x))\ f$).

Because of this complication, we decided to drop the lambda-calculus from DOT, and instead introduce methods with one parameter. Like in Scala, functions are then just sugar for objects with a special method.

An alternative design would have been to change the expansion of the function type to have a declaration for a special marker value label.

6.2.3 Why not patch the DOT calculus for preservation to hold?

We tried! However, the resulting calculi were not elegant, and furthermore, we still found issues with preservation. Below, we give a brief summary of one failed attempt to patch the calculus for preservation to hold.

Because many of the counterexamples to preservation are related to narrowing, we tried to make widening an explicit operation and change rules with implicit relaxations (MSEL and NEW) to be strict. From a typing perspective, the change was straightforward, but reduction became more complicated and dependent on typing because the type information in widenings needed to be propagated correctly.

We added path equality provisions in the subtyping rules, in the same spirit as the Tribe calculus [4].

Unfortunately, these two patches interacted badly, and we were left with a disturbing counterexample to type safety.

$$\textbf{val } a = \textbf{new } \top\,\{z \Rightarrow C : \bot..\top\,\{z \Rightarrow D : \bot..z.X, X : \bot..\top\}\}\,;$$
$$\textbf{val } b = \textbf{new } a.C\,\{z \Rightarrow X : \bot..\bot\}\,;$$
$$\textbf{val } c = \textbf{new } a.C;$$
$$\textbf{val } d = \textbf{new } (\textbf{cast } a.C\ b).D;$$
$$(\textbf{app}\ (\lambda x\!:\!\bot.x.foo)\ d)$$

Notice that d has type $\bot$ if the cast on b is ignored. This example didn't typecheck initially because the path-equality provisions only applied when objects are in the store, so the application was not well-typed. But if we started preservation in a store which had a, b, c and d then the application type-checked, because, through one of the path-equality provision, we could find that the type of d was a subtype of $\bot$. Now, of course, when we got to $d.foo$, reduction failed.

6.3 Related Work

In addition to Scala's previous models [5, 12], several calculi present some form of path-dependent types.

The *vc* calculus [7] models virtual classes with path-dependent types. *vc* restricts paths to start with "this", though it provides a way ("out") to refer to the enclosing object.

The Tribe calculus [4] builds an ownership types system [3] on top of a core calculus which models virtual classes. The soundness proof for the core calculus seems to be tied to the ownership types system.

Some ML-style module systems [8, 9] have a form of stratified path-dependent types. Because of the stratification, recursion is not allowed. In MixML [6] like in Scala, this restriction is lifted.

7. Conclusion

We have presented DOT, a calculus aimed as a new foundation of Scala and languages like it. DOT features path-dependent types, refinement types, and abstract type members.

Proving the DOT calculus type-safe has been an interesting adventure. We have shown that DOT does not satisfy preservation (also known as subject-reduction). However, the standard theorems of preservation and progress are just one way to prove type safety. We have sketched a plausible proof of type safety using the powerful method of logical relations.

Acknowledgments

We thank Amal Ahmed for many discussions and insights about applying logical relations to DOT. We thank Donna Malayeri and Geoffrey Washburn for preliminary work on DOT. We thank Tiark Rompf for helpful comments.

References

[1] A. J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

[2] A. J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, pages 69–83, 2006.

[3] N. R. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *OOPSLA*, pages 618–633, 2010.

[4] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: a simple virtual class calculus. In *AOSD*, pages 121–134, 2007.

[5] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In *MFCS*, pages 1–23, 2006.

[6] D. Dreyer and A. Rossberg. Mixin' up the ML module system. In *ICFP*, pages 307–320, 2008.

[7] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL*, pages 270–282, 2006.

[8] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, pages 123–137, 1994.

[9] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *ESOP*, pages 6–20, 2002.

[10] C. Hritcu and J. Schwinghammer. A step-indexed semantics of imperative objects. *Logical Methods in Computer Science*, 5(4), 2009.

[11] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *POPL*, pages 285–296, 2012.

[12] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP*, pages 201–224, 2003.

[13] B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

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