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 (DOT) calculus. DOT models Scala's path-dependent types, abstract type members and its mixture of nominal and structural typing through the use of refinement types. The core formalism makes no attempt to model inheritance and mixin composition. DOT normalizes Scala's type system by unifying the constructs for type members and by providing classical intersection and union types which simplify greatest lower bound and least upper bound computations.

In this paper, we present the DOT calculus, both formally and informally. We also discuss our work-in-progress to prove type-safety of the calculus.

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

1. Introduction

A scalable programming language is one in which the same concepts can describe small as well as large parts. Towards this goal, Scala unifies concepts from object and module systems. An essential ingredient of this unification is objects with type members. Given a stable path to an object, its type members can be accessed as types, called path-dependent types.

This paper presents Dependent Object Types (DOT), a small object calculus with path-dependent types. In addition to path-dependent types, types in DOT are built from refinements, intersections and unions. A refinement extends a type by (re-)declaring members, which can be types, values or methods.

We propose DOT as 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, 14], 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 DOT 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 A and B, where each declares an abstract upper-bounded type member T,

```
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 greatest lower bounds and least upper bounds play a central role in Scala's type inference. For example, in order to infer the type of an **if** expression such as

```
if (cond) ((a: A) ⇒ c: C) else ((b: B) ⇒ d: D)
```

type inference tries to compute the greatest lower bound of A and B and the least upper bound of C and D. The absence of universal greatest lower bounds and least upper bounds makes type inference more brittle and more unpredictable.

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

Compared to Scala, DOT also simplifies type members. In DOT, a type member is declared by its lower and upper bounds. Here is an example inspired by [11]:

```
trait Food
trait Animal {
  type Meal <: Food
  def eat(meal: Meal) {}
}
```

`Meal` is a type member with a lower bound of $\bot$ and an upper bound of `Food`. It is possible to instantiate an `Animal` without further specifying `Meal`, though it would not be possible to feed it. Now, we define `Cow` by refining `Animal`.

```
trait Grass extends Food
trait Cow extends Animal {
  type Meal = Grass
}
```

In Scala, the type alias **type** `Meal = Grass` defines a concrete binding for `Meal`. In DOT, such a type alias is declared by giving it identical lower and upper bounds. Now, we can instantiate `Cows` and feed them `Grass`.

DOT has a notion of concrete vs abstract type members, which is used to distinguish which types members are instantiable. In the example above, all types introduced by **trait** would be concrete type members in DOT with lower bounds of $\bot$ and upper bounds describing the **extends** clause: $\top$ for `Food`, a refinement of $\top$ with type member `Meal` and method member `eat` for `Animal`, `Food` for `Grass` and a refinement of `Animal` with type member `Meal` for `Cow`. Concrete type members typically have a lower bound of $\bot$ so that they are purely nominal: e.g. one needs to directly or indirectly instantiate a `Cow` to have an object of type `Cow`.

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. In section 4, we show that the calculus does not satisfy the standard theorem of preservation, also known as subject reduction. However, we still believe that the calculus is type-safe, as explained 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 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 **val** $y =$ **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 value, method 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 **val** $x =$ **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.

Syntax

x, y, z	Variable
l	Value label
m	Method label
$v ::=$	Value
x	variable
$t ::=$	Term
v	value
$\textbf{val}\ x = \textbf{new}\ c;\ t$	new instance
$t.l$	field selection
$t.m(t)$	method invocation
$p ::=$	Path
x	variable
$p.l$	selection
$c ::= T_c\left\{\overline{d}\right\}$	Constructor
$d ::=$	Initialization
$l = v$	field initialization
$m(x) = t$	method initialization
$s ::= \overline{x \mapsto c}$	Store

$L ::=$	Type label
L_c	class label
L_a	abstract type label
$S, T, U, V, W ::=$	Type
$p.L$	type selection
$T\left\{z \Rightarrow \overline{D}\right\}$	refinement
$T \wedge T$	intersection type
$T \vee T$	union type
$\top$	top type
$\bot$	bottom type
$S_c, T_c ::=$	Concrete type
$p.L_c \mid T_c\left\{z \Rightarrow \overline{D}\right\} \mid T_c \wedge T_c \mid \top$	
$D ::=$	Declaration
$L : S..U$	type declaration
$l : T$	value declaration
$m : S \to U$	method declaration
$\Gamma ::= \overline{x : T}$	Environment

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

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

$$\textbf{val}\ x = \textbf{new}\ c;\ t \mid s \ \to\ t \mid s, x \mapsto c \qquad \text{(NEW)}$$

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

$$\frac{t \mid s \ \to\ t' \mid s'}{e[t] \mid s \ \to\ e[t'] \mid s'} \qquad \text{(CONTEXT)}$$

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

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(VAR)}$$

$$\frac{\Gamma \vdash t \ni l : T'}{\Gamma \vdash t.l : T'} \qquad \text{(SEL)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \ni m : S \to T \\ \Gamma \vdash t' : T' \,,\ T' <: S\end{array}}{\Gamma \vdash t.m(t') : T} \qquad \text{(MSEL)}$$

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

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

$$\frac{\Gamma \vdash v : V' \,,\ V' <: V}{\Gamma \vdash (l = v) : (l : V)} \qquad \text{(VDECL)}$$

$$\frac{\begin{array}{c}\Gamma \vdash S\ \textbf{wfe} \\ \Gamma, x : S \vdash t : T' \,,\ T' <: T\end{array}}{\Gamma \vdash (m(x) = t) : (m : S \to T)} \qquad \text{(MDECL)}$$

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

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

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 $\overline{D_\perp}$, Here $\overline{D_\perp}$ is the set of declarations that contains for every term label l the declaration $l : \perp$, for every type label L the declaration $L : \top..\perp$ and for every method label m the declaration $m : \top \to \perp$.

Figure 2. The DOT Calculus : Declaration Lattice

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 **val** y = **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 judgement $\Gamma \vdash T \prec_z \overline{D}$ "flattens" all the declarations of a type: it relates a type T to a set of declarations that describe the type structurally. Expansion is precise and unique, though it doesn't always exist. See section 3.2 for examples.

The expansion relation $\prec$ is needed to type-check 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 ($\perp$-$\prec$) states that the bottom type $\perp$ expands to the bottom element $\overline{D_\perp}$ 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. Subtyping 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).

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

$$\frac{\Gamma \vdash p : T \ , \ T \prec_z \overline{D}}{\Gamma \vdash p \ni [p/z]D_i} \quad (\textsc{Path-}\ni)$$

$$\frac{z \notin \mathit{fn}(D_i) \qquad \Gamma \vdash t : T \ , \ T \prec_z \overline{D}}{\Gamma \vdash t \ni D_i} \quad (\textsc{Term-}\ni)$$

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

$$\frac{\Gamma \vdash T \prec_z \overline{D'}}{\Gamma \vdash T \left\{z \Rightarrow \overline{D}\right\} \prec_z \overline{D'} \wedge \overline{D}} \quad (\textsc{Rfn-}\prec)$$

$$\frac{\Gamma \vdash p \ni L : S..U \ , \ U \prec_z \overline{D}}{\Gamma \vdash p.L \prec_z \overline{D}} \quad (\textsc{Tsel-}\prec)$$

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

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

$$\Gamma \vdash \top \prec_z \{\} \quad (\top\text{-}\prec)$$

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

Figure 3. The DOT Calculus : Membership and Expansion

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash T \left\{z \Rightarrow \overline{D}\right\} \ \mathbf{wfe} \ , \ S <: T \ , \ S \prec_z \overline{D'} \\ \Gamma, z : S \vdash \overline{D' <: D}\end{array}}{\Gamma \vdash S <: T \left\{z \Rightarrow \overline{D}\right\}} \quad (<:\text{-}\textsc{Rfn})$$

$$\frac{\Gamma \vdash T \left\{z \Rightarrow \overline{D}\right\} \ \mathbf{wfe} \ , \ T <: T'}{\Gamma \vdash T \left\{z \Rightarrow \overline{D}\right\} <: T'} \quad (\textsc{Rfn-}<:)$$

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

$$\frac{\Gamma \vdash p \ni L : S..U \ , \ S <: U \ , \ U <: U'}{\Gamma \vdash p.L <: U'} \quad (\textsc{Tsel-}<:)$$

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash S' <: S \ , \ T <: T'}{\Gamma \vdash (L : S..T) <: (L : S'..T')} \quad (\textsc{TDecl-}<:)$$

$$\frac{\Gamma \vdash S' <: S \ , \ T <: T'}{\Gamma \vdash (m : S \rightarrow T) <: (m : S' \rightarrow T')} \quad (\textsc{MDecl-}<:)$$

$$\frac{\Gamma \vdash T <: T'}{\Gamma \vdash (l : T) <: (l : T')} \quad (\textsc{VDecl-}<:)$$

Figure 4. The DOT Calculus : Subtyping and Declaration Subsumption

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

$$\dfrac{\Gamma \vdash T\ \mathbf{wf} \qquad \Gamma, z : T\left\{z \Rightarrow \overline{D}\right\} \vdash \overline{D\ \mathbf{wf}}}{\Gamma \vdash T\left\{z \Rightarrow \overline{D}\right\}\ \mathbf{wf}} \quad (\text{RFN-WF})$$

$$\Gamma \vdash \top\ \mathbf{wf} \quad (\top\text{-WF})$$

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

$$\dfrac{\Gamma \vdash p \ni L : S..U\ ,\ S\ \mathbf{wf}\ ,\ U\ \mathbf{wf}}{\Gamma \vdash p.L\ \mathbf{wf}} \quad (\text{TSEL-WF}_1)$$

$$\dfrac{\Gamma \vdash p \ni L : \bot..U}{\Gamma \vdash p.L\ \mathbf{wf}} \quad (\text{TSEL-WF}_2)$$

$$\dfrac{\Gamma \vdash T\ \mathbf{wf}\ ,\ T'\ \mathbf{wf}}{\Gamma \vdash T \wedge T'\ \mathbf{wf}} \quad (\wedge\text{-WF})$$

$$\dfrac{\Gamma \vdash T\ \mathbf{wf}\ ,\ T'\ \mathbf{wf}}{\Gamma \vdash T \vee T'\ \mathbf{wf}} \quad (\vee\text{-WF})$$

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

$$\dfrac{\Gamma \vdash S\ \mathbf{wf}\ ,\ U\ \mathbf{wf}}{\Gamma \vdash L : S..U\ \mathbf{wf}} \quad (\text{TDECL-WF})$$

$$\dfrac{\Gamma \vdash S\ \mathbf{wf}\ ,\ U\ \mathbf{wf}}{\Gamma \vdash m : S \to U\ \mathbf{wf}} \quad (\text{MDECL-WF})$$

$$\dfrac{\Gamma \vdash T\ \mathbf{wf}}{\Gamma \vdash l : T\ \mathbf{wf}} \quad (\text{VDECL-WF})$$

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

$$\dfrac{\Gamma \vdash T\ \mathbf{wf}\ ,\ T \prec_z \overline{D}}{\Gamma \vdash T\ \mathbf{wfe}} \quad (\text{WFE})$$

Figure 5. The DOT Calculus : Well-Formedness

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 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 $\Gamma \vdash D\ \mathbf{wf}$. All declarations are well-formed if their constituent types are well-formed.

3. Examples

3.1 Greatest Lower Bounds and Least Upper Bounds

In DOT, the greatest lower bound of two types is their intersection and their least upper bound, their union. Recall the introductory example which was problematic in Scala:

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

We represent it as follows in DOT, wrapping the types in a namespace p:

$$\begin{array}{l} \mathbf{val}\ p = \mathbf{new}\ \top\{p \Rightarrow \\ \qquad A_c : \bot..\top\,\{z \Rightarrow T : \bot..p.A_c\} \\ \qquad B_c : \bot..\top\,\{z \Rightarrow T : \bot..p.B_c\} \\ \quad \}\,\{\}\,; \end{array}$$

The greatest lower bound of $p.A_c$ and $p.B_c$ is $p.A_c \wedge p.B_c$ by definition.

3.2 Expansion

The expansion of $p.A_c \wedge p.B_c$ is the set $\{T : \bot..p.A_c \wedge p.B_c\}$ by $\wedge$-$\prec$ rule:

$$\dfrac{p.A_c \prec_z \{T : \bot..p.A_c\} \quad p.B_c \prec_z \{T : \bot..p.B_c\}}{p.A_c \wedge p.B_c \prec_z \{T : \bot..p.A_c \wedge p.B_c\}}\ \wedge\text{-}\prec$$

In turn, $p.A_c \prec_z \{T : \bot..p.A_c\}$ is derived as follows:

$$\dfrac{\dfrac{\cdots}{p \ni A_c : \bot..\top\,\{z \Rightarrow T : \bot..p.A_c\}}\ \text{PATH-}\ni \qquad \dfrac{\dfrac{}{\top \prec_z \{\}}\ \top\text{-}\prec}{\top\,\{z \Rightarrow T : \bot..p.A_c\} \prec_z \{T : \bot..p.A_c\}}\ \text{RFN-}\prec}{p.A_c \prec_z \{T : \bot..p.A_c\}}\ \text{TSEL-}\prec$$

Note that expansions do not always exist; see section 4.2 for illustration.

3.3 Functions as Sugar

Like in Scala, we can encode functions as objects with a special method. Note that the variable z must be fresh.

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

We will freely use the following sugar in the remaining of this paper. We will also sometimes use $\lambda x : S.t$ for $\textbf{fun}\ (x : S)\ \_\ t$, omitting the return type for convenience and brevity.

3.4 Class Hierarchies

A class hierarchy such as

```
object pets {
  trait Pet
  trait Cat extends Pet
  trait Dog extends Pet
  trait Poodle extends Dog
  trait Dalmatian extends Dog
}
```

can be easily represented by concrete type members, setting the upper bounds appropriately:

$$\begin{aligned} &\textbf{val}\; \mathit{pets} = \textbf{new}\; \top\{z \Rightarrow \\ &\qquad \mathit{Pet}_c : \bot..\top \\ &\qquad \mathit{Cat}_c : \bot..z.\mathit{Pet}_c \\ &\qquad \mathit{Dog}_c : \bot..z.\mathit{Pet}_c \\ &\qquad \mathit{Poodle}_c : \bot..z.\mathit{Dog}_c \\ &\qquad \mathit{Dalmatian}_c : \bot..z.\mathit{Dog}_c \\ &\;\}\{\}\,; \end{aligned}$$

The lower of bounds of $\bot$ ensures that these concrete types are nominal as the <:-TSEL rule cannot be meaningfully applied.

3.5 Abstract Type Members

The `choices.Alt` trait takes three abstract type members: `C`, `A`, `B`. `A` and `B` are upper bounded by `C`. The intention is that the `choose` function takes an `A` and a `B` and returns one or the other.

```
object choices {
  trait Alt {
    type C
    type A <: C
    type B <: C
    val choose : A ⇒ B ⇒ C
  }
}
```

In DOT, we can state more precisely the return type of *choose*, thanks to union types:

$$\begin{aligned} &\textbf{val}\; \mathit{choices} = \textbf{new}\; \top\{z \Rightarrow \\ &\quad \mathit{Alt}_c : \bot..\top\{a \Rightarrow \\ &\qquad C : \bot..\top \\ &\qquad A : \bot..a.C \\ &\qquad B : \bot..a.C \\ &\qquad \mathit{choose} : a.A \to a.B \to_s a.A \vee a.B \\ &\quad \} \\ &\}\{\}\,; \end{aligned}$$

Using lower bounds of $\bot$ for the abstract type members means they can vary covariantly. However, we wouldn't want to pass in any $\mathit{pets}.\mathit{Pet}_c$ to a *choose* method that expects only a $\mathit{pets}.\mathit{Dog}_c$. Therefore,

$$\begin{aligned} & \mathit{choices}.\mathit{Alt}_c\,\{a \Rightarrow C : \bot..\mathit{pets}.\mathit{Dog}_c\} \\ <: \quad & \mathit{choices}.\mathit{Alt}_c\,\{a \Rightarrow C : \bot..\mathit{pets}.\mathit{Pet}_c\} \end{aligned}$$

but

$$\begin{aligned} & \mathit{choices}.\mathit{Alt}_c\,\{a \Rightarrow C : \mathit{pets}.\mathit{Dog}_c..\mathit{pets}.\mathit{Dog}_c\} \\ \not<: \quad & \mathit{choices}.\mathit{Alt}_c\,\{a \Rightarrow C : \mathit{pets}.\mathit{Pet}_c..\mathit{pets}.\mathit{Pet}_c\} \end{aligned}$$

As expected, we cannot invoke *choose* meaningfully, unless A and B have realizable lower bounds. For example, assuming we refine the types above so that $A : C..C$ and $B : C..C$, we cannot invoke *choose* when C has a lower bound of $\bot$ but only when it has a realizable lower bound such as $\mathit{pets}.\mathit{Dog}_c$.

3.6 Polymorphic Operators as Sugar

In Scala, we can implement a polymorphic operator `pickLast` which takes concrete types for `C`, `A` and `B` and implements a `choices.Alt` instance where the `choose` function picks the `B` element – note the precision of the `choose` function which has been refined to return an element of exactly type `B`.

```
def pickLast[Cp,Ap <: Cp,Bp <: Cp] = new Alt {
  type C = Cp
  type A = Ap
  type B = Bp
  val choose: A ⇒ B ⇒ B = a ⇒ b ⇒ b
}

val potty = new Poodle {}
val dotty = new Dalmatian {}
val picker = pickLast[Dog,Poodle,Dalmatian]
val p: picker.A = potty
val r: picker.B = picker.choose(potty)(dotty)
```

In DOT, we can implement such a polymorphic operator as sugar. Here, it is not convenient to just return a complex term like we did for functions as sugar because then invoking *choose* falls under the TERM-∋ restriction, which doesn't allow the "self" type to occur in the result of the method invocation. So the translation involves explicitly binding an object to the result of the polymorphic operator. We translate

$$\textbf{val}\; x^a = \textbf{pickLast}(T^C, T^A, T^B);\, e^a$$

to

$$\begin{aligned} &\textbf{val}\; x^a = \textbf{new}\; \mathit{choices}.\mathit{Alt}_c\{x^a \Rightarrow \\ &\qquad C : T^C..T^C \\ &\qquad A : T^A..T^A \\ &\qquad B : T^B..T^B \\ &\qquad \mathit{choose} : x^a.A \to x^a.B \to_s x^a.B \\ &\}\,\{\mathit{choose}(a) = \textbf{fun}\; (b : x^a.B)\; x^a.B\; b\}\,; \\ &e^a \end{aligned}$$

Now, given

$$\begin{aligned} &\textbf{val}\; p = \textbf{new}\; \mathit{pets}.\mathit{Poodle}_c; \\ &\textbf{val}\; d = \textbf{new}\; \mathit{pets}.\mathit{Dalmatian}_c; \\ &\textbf{val}\; a = \textbf{pickLast}(\mathit{pets}.\mathit{Dog}_c, \mathit{pets}.\mathit{Poodle}_c, \mathit{pets}.\mathit{Dalmatian}_c); \end{aligned}$$

The type of a is a subtype of $\mathit{choices}.\mathit{Alt}_c$:

$$\begin{aligned}
&(\textbf{cast}\ \top\\
&(\textbf{cast}\ \mathit{choices}.\mathit{Alt}_c\\
&(\textbf{cast}\ \mathit{choices}.\mathit{Alt}_c\ \{a \Rightarrow C : \bot..\mathit{pets}.\mathit{Dog}_c\}\\
&a)))
\end{aligned}$$

The type of p is a subtype of $a.A$:

$$(\textbf{cast}\ \top\ (\textbf{cast}\ a.A\ p))$$

a chooses a $\mathit{pets}.\mathit{Dalmatian}_c$:

$$(\textbf{cast}\ \top\ (\textbf{cast}\ \mathit{pets}.\mathit{Dalmatian}_c\ (\textbf{app}\ a.\mathit{choose}(p)\ d)))$$

a enforces that its first argument be a $\mathit{pets}.\mathit{Poodle}_c$ and its second a $\mathit{pets}.\mathit{Dalmatian}_c$. The following does not type-check for this reason:

$$(\textbf{cast}\ \top\ (\textbf{app}\ a.\mathit{choose}(d)\ p))$$

3.7 F-bounded Quantification

F-bounded quantification describes an upper bound that itself contains the type being constrained: for example, `Int extends Ordered [Int]`. Here, we define `MetaAlt` to extend `choices.Alt` with `C` as an alias for `MetaAlt`.

```
trait MetaAlt extends choices.Alt {
  type C = MetaAlt
  type A = C
  type B = C
}
```

Now, we can define some `MetaAlt` instances:

```
val first = new MetaAlt {
  val choose: C ⇒ C ⇒ C = a ⇒ b ⇒ a
}
val last = new MetaAlt {
  val choose: C ⇒ C ⇒ C = a ⇒ b ⇒ b
}
val recfirst = new MetaAlt {
  val choose: C ⇒ C ⇒ C = a ⇒ b ⇒ a.choose(a)(b)
}
val reclast = new MetaAlt {
  val choose: C ⇒ C ⇒ C = a ⇒ b ⇒ b.choose(a)(b)
}
```

The equivalent in DOT is straightforward. We wrap $\mathit{MetaAlt}$ in a namespace, so that we can refer to it.

$$\begin{aligned}
&\textbf{val}\ m = \textbf{new}\ \top\{m \Rightarrow\\
&\quad \mathit{MetaAlt}_c : \bot..\mathit{choices}.\mathit{Alt}_c\{a \Rightarrow\\
&\quad\quad C : m.\mathit{MetaAlt}_c..m.\mathit{MetaAlt}_c\\
&\quad\quad A : a.C..a.C\\
&\quad\quad B : a.C..a.C\\
&\quad \}\\
&\}\{\};
\end{aligned}$$

Now, we can create the equivalent of `first` (f), `last` (l), `recfirst` (rf) and `reclast` (rl):

$$\begin{aligned}
&\textbf{val}\ f = \textbf{new}\ m.\mathit{MetaAlt}_c\{\\
&\quad \mathit{choose}(a) = \textbf{fun}\ (b : m.\mathit{MetaAlt}_c)\ m.\mathit{MetaAlt}_c\ a\};\\
&\textbf{val}\ l = \textbf{new}\ m.\mathit{MetaAlt}_c\{\\
&\quad \mathit{choose}(a) = \textbf{fun}\ (b : m.\mathit{MetaAlt}_c)\ m.\mathit{MetaAlt}_c\ b\};\\
&\textbf{val}\ \mathit{rf} = \textbf{new}\ m.\mathit{MetaAlt}_c\{\\
&\quad \mathit{choose}(a) = \textbf{fun}\ (b : m.\mathit{MetaAlt}_c)\ m.\mathit{MetaAlt}_c\\
&\quad\quad (\textbf{app}\ a.\mathit{choose}(a)\ b)\};\\
&\textbf{val}\ \mathit{rf} = \textbf{new}\ m.\mathit{MetaAlt}_c\{\\
&\quad \mathit{choose}(a) = \textbf{fun}\ (b : m.\mathit{MetaAlt}_c)\ m.\mathit{MetaAlt}_c\\
&\quad\quad (\textbf{app}\ b.\mathit{choose}(a)\ b)\};
\end{aligned}$$

Given these definitions, here is a valid expression, which evaluates to f: $(\textbf{cast}\ \top\ (\textbf{app}\ \mathit{rf}.\mathit{choose}(f)\ l))$.

4. Counterexamples to Preservation

We first tried to prove the calculus type-safe using the standard theorems of preservation and progress [15, 16]. 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 [12].

Most of these counterexamples are related to narrowing, the phenomenon that a type can become more precise after substitution: if a method takes a parameter x of type U, then when it is invoked, any argument v of type $S <: U$ can be substituted – this is the narrowing effect: it is as if the context was changed from $x : U$ to $x : S$. Sections 4.1, 4.2 and 4.3 each present a counterexample related to narrowing.

The last counterexample, presented in section 4.4, illustrates the need to relate path-dependent types after reduction. This need for path-equality provisions in order for preservation to hold is well-known from other calculi such as Tribe [4] with path-dependent types and a small-step operational semantics.

More generally, these counterexamples illustrate that preservation doesn't hold because a term that type-checks can step to a term that does not type-check. However, these counterexamples to preservation are not counterexamples to type-safety: i.e. these programs don't get stuck – they eventually step to a value.

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:

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

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

First, let's illustrate why expansion does not always exist.

Here is the simplest such example:

$$\begin{array}{l}\textbf{val}\ z = \textbf{new}\ \top\{z \Rightarrow \\ \quad L : \bot..z.L \\ \}\,\{\}\,;\end{array}$$

The type $z.L$ is **wf** but not **wfe**. Indeed, there is no finite derivation of an expansion for $z.L$, because by the TSEL-$\prec$ rule, in order to expand $z.L$, we need to expand its upper bound, which is $z.L$! However, not that the object creation expression for z would not type-check because subtyping is regular wrt **wfe**, and so the type of a constructor in an object creation expression must have **wfe** type members, since we check that the lower bound is a subtype of the upper bound for each type member.

The example above relies on the TSEL-WF$_2$, in order for the upper bound to refer to the type member being declared. Here is an example that does not rely on this rule, and that we will use below to create a counterexample to preservation. Let

$$\begin{array}{lll}T_1 = \top\{z \Rightarrow & & \\ & A : \bot..z.B & \\ & B : \bot..\top & \} \\ T_2 = \top\{z \Rightarrow & & \\ & A : \bot..\top & \\ & B : \bot..z.A & \}\end{array}$$

Now, consider $T_1 \wedge T_2$. The type is **wf** and **wfe**, but its members A and B are not **wfe**, because the expansion of $T_1 \wedge T_2$ with self-type z has the set of declarations $\{A : \bot..z.B,\ B : \bot..z.A\}$ – thus, to expand $z.A$, we need to expand $z.B$, and to expand $z.B$, we need to expand $z.A$! There is no finite derivation of expansions for the type members A and B.

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:

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

$a.i.l$ 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 believe that the DOT calculus is type-safe, and are developing a proof of type-safety using step-indexed logical relations [1, 2, 10]. In this section, we briefly summarize the main theorem of type-safety, and the strategy based on step-indexed logical relations that we are using to prove it. All our development, including models in Coq [13] and PLT Redex [12], is available from `http://dot.namin.net`.

5.1 Type-Safety

Type-safety states that a well-typed program doesn't get stuck. More formally: If $\emptyset \vdash t : T$ and $t \mid \emptyset \to^* t' \mid s'$ then either t' is a value or $\exists t'', s''. t' \mid s' \to t'' \mid s''$. Note that this is stronger than the standard theorem of progress, which states that a well-type term can take *a* step or is a value.

5.2 Strategy based on Logical Relations

Our strategy follows the standard technique of proving type-safety using logical relations. We define a logical relation $\Gamma \vDash t : T$, such that $\Gamma \vdash t : T$ implies $\Gamma \vDash t : T$ implies type-safety.

The main logical relation $\Gamma \vDash t : T$ is based on a set of mutually recursive logical relations that are step-indexed in order to ensure their well-foundedness: $\mathcal{E}_{k;\Gamma;s}[\![T]\!]$ defines the set of terms that appear to have type T when taking at most k steps, and $\mathcal{V}_{k;\Gamma;s}[\![T]\!]$ defines the set of *values* that appear to have type T when taking at most k steps. There are also two other logical relations, one for completing the store to match the context, and one for completing the context to match the store resulting from taking some number of steps.

$\Gamma \vDash t : T$ is defined as $t \in \mathcal{E}_{k;\Gamma;\emptyset}[\![T]\!], \forall k.\ t \in \mathcal{E}_{k;\Gamma;s}[\![T]\!]$ roughly as follows: after reducing t a number of steps $< k$, if the resulting term is irreducible, then it must be in $\mathcal{V}_{j';\Gamma';s'}[\![T]\!]$ for appropriate j', Γ', s'. By definition, $\Gamma \vDash t : T$ implies that t cannot be stuck.

Then, to prove type-safety, all that needs to be proved is the fundamental theorem – or completeness – of the logical relation: $\Gamma \vdash t : T$ to $\Gamma \vDash t : T$. Type-safety is a straightforward corollary of this theorem, since $\Gamma \vDash t : T$ implies by definition that t cannot be stuck. The proof of the fundamental theorem is on induction on the derivation of $\Gamma \vdash t : T$. The logical relation approach enables us to have strong enough induction hypotheses to carry the proof through, without requiring us to strictly relate intermediate terms by types like preservation.

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:

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

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:

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

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 type-check 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:

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

Now, f was a subtype of $\top \to \top$, but $(\textbf{app}\ f\ \ (\lambda x : \top.x))$ was stuck (and, rightfully, didn't type-check). 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 label that either prevents instantiation or requires an implementation for the function.

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.

Here is a summary of one attempt, which is further detailed below. 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.

In any case, this attempt resulted in a patched calculus that was not as elegant as the original one, in addition to being unsound. We're open to ideas!

In the DOT calculus as presented, the APP and NEW typing rules have implicit relaxations. For instance, in APP, the argument type may be a subtype of the declared parameter type. In order to deal with all the preservation counterexamples due to narrowing, we tried making widening an explicit operation and changing those rules to be strict by replacing those relaxed subtyping judgments with equality judgments. Two types S and T are judged to be equal if $S <: T$ and $T <: S$.

Syntactically, we add a widening term: $t : T$, and extend values with a case for widening: $v : T$. The typing rule for widening, WID, is then the only one admitting a subtyping relaxation: $\Gamma \vdash (t : T) : T$ if $\Gamma \vdash t : T'$ and $\Gamma \vdash T' <: T$.

The reduction rules become more complicated because the type information in the widening needed to be propagated correctly. We will motivate this informally with examples.

$$\begin{aligned}
&\textbf{val}\; v = \textbf{new}\; \top\,\{z \Rightarrow L_a : \bot..\top, l : \top\,\{z \Rightarrow L_a : \bot..\top\}\} \\
&\qquad\qquad \{l = v : \top\,\{z \Rightarrow L_a : \bot..\top\}\}\,; \\
&(\textbf{app}\; (\lambda x\!:\!\top.x)\; (v : \top\,\{z \Rightarrow l : \top\}).l)
\end{aligned}$$

The term $(v : \top\,\{z \Rightarrow l : \top\}).l$ first widens v so that the label has type $\top$ instead of $\top\,\{z \Rightarrow L_a : \bot..\top\}$.

How should reduction proceed? We cannot just strip the widening and then reduce, because then the strict function application would not accept the reduced term. In short, we need to do some type manipulations during reduction, by using the membership and expansion judgments. This is a bit unfortunate, because it means that reduction now needs to know about typing.

Next, we look at path equality provisions. These are even more essential now in the presence of explicit widening. Consider this example:

$$\begin{aligned}
&\textbf{val}\; b = \textbf{new}\; \top\,\{z \Rightarrow X : \top..\top, l : z.X\}\,\{l = b : b.X\}\,; \\
&\textbf{val}\; a = \textbf{new}\; \top\,\{z \Rightarrow i : \top\,\{z \Rightarrow X : \bot..\top, l : z.X\}\} \\
&\qquad\qquad \{i = b : \top\,\{z \Rightarrow X : \bot..\top, l : z.X\}\}\,; \\
&a.i.l : \top
\end{aligned}$$

$a.i.l$ reduces to $b : \top\,\{z \Rightarrow X : \bot..\top, l : z.X\}$. Now, how can we continue? $b.l$ reduces to $b\ :\ b.X$ which has bounds $\top..\top$, but $(b\ :\ \top\,\{z \Rightarrow X : \bot..Top, l : z.X\}).l$ has bounds $\bot..\top$, so without some provision for path equality, we cannot widen $b.l$ to $(b : \top\,\{z \Rightarrow X : \bot..\top, l : z.X\}).l$.

We add the path equality provisions to the subtyping rules.

Let's first ignore the extension of the calculus requiring explicit widenings. Then, we need to add one intuitive rule to the subtyping judgment: <:-PATH. If p (path-)reduces to q, and $T <: q.L$, then $T <: p.L$. Path reduction is a simplified form of reduction involving only paths. However, this means that the subtyping judgment, and indirectly, all the typing-related judgments, now need to carry the store in addition to the context so that path reductions can be calculated.

Now, let's see how path equality provisions and explicit widening can fit together.

First, path reduction is not isomorphic to reduction anymore, since we want to actually skip over widenings, as motivated by the example above.

In addition, we now also need a dual rule, PATH-<:: if p (path-dually)-reduces to q, and $q.L <: T$ then $p.L <: T$. This is because when we have a widening on an object on which a method is called, we have to upcast the argument to the parameter type expected by the original method. Here is a motivating example.

Let T_c be a shorthand for the type:

$$\begin{aligned}
&\top\{z \Rightarrow \\
&\quad A : \top\,\{z \Rightarrow m : \bot \to \top\}\,..\top \\
&\quad B : \top..\top \\
&\quad m : z.A \to \top \\
&\}
\end{aligned}$$

Let T be a shorthand for the type:

$$\begin{aligned}
&\top\{z \Rightarrow \\
&\quad A : \top\,\{z \Rightarrow m : \bot \to \top\}\,..\top \\
&\quad B : \top..\top \\
&\quad m : z.A\,\{z \Rightarrow B : \top..\top\} \to \top \\
&\}
\end{aligned}$$

Now, consider the term:

$$\begin{aligned}
&\textbf{val}\; v = \textbf{new}\; T_c\,\{m(x) = x : \top\}\,; \\
&(v : T).m(v : ((v : T).A\,\{z \Rightarrow B : \top..\top\}))
\end{aligned}$$

When we evaluate the method invocation, we need to cast v : $((v\ :\ T).A\,\{z \Rightarrow B : \top..\top\})$ to $v.A$, and for this, we need the newly introduced PATH-<: rule.

Note that the path dual reduction can be a bit stricter with casts than the path reduction. In any case, introducing this PATH-<: rule into the subtyping judgment is problematic: it is now possible to say $p.L <: T$, even though T can do more than what $p.L$ is defined to do. Here is an example, where we construct an object, with $T = \bot$. (The convolution in the example is due to the requirement that concrete types be only mentioned once.)

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

Notice that d has type $\bot$ if you ignore the cast on b. This example doesn't type-check initially because PATH-<: only applies when objects are in the store, so the application is not well-typed. But if we start preservation in a store which has a, b, c and d then the application type-checks, because, through PATH-<:, we can find that the type of d is a subtype of $\bot$. Now, of course, when we get to $d.\mathit{foo}$, reduction fails.

So the preservation theorem as defined on a small-step semantics (where we start with an arbitrary well-formed environment) fails when we add the PATH-<: rule.

6.3 Related Work

In addition to Scala's previous models [5, 14], 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), because a well-typed term can step to an intermediate term that is not well-typed. In any case, the standard theorems of preservation and progress are just one way to prove type safety, which states that a well-typed term cannot be stuck – a weaker statement that does not require type-checking intermediate steps. We are developing a proof of type-safety using 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 and Viktor Kuncak 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] A. Igarashi and B. C. Pierce. Foundations for virtual types. *Inf. Comput.*, 175(1):34–49, 2002.

[12] 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.

[13] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2010. URL `http://coq.inria.fr`. Version 8.3.

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

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

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