Dependent Object Types - A foundation for Scala's type system

Draft of December 19, 2011 – Do Not Distrubute

Martin Odersky, Geoffrey Alan Washburn

EPFL

Abstract.

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 modelling 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 (nuObj, FS), we make no attempt to model inheritance or mixin composition. Indeed we will argue that such concepts are better modelled 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 modelled 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.

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 the standard theorems of preservation and progress to establish soundness of the type system. 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 to inheritance that might work better is to model it 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 preservation wrt the translation. This might be easier than to prove type preservation wrt reduction.

2 The DOT Calculus

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

Notation

We use standard notational conventions for sets. The notation $\overline{X}$ denotes a set of elements X. Given a such a set $\overline{X}$ in a typing rule, X_i denotes an arbitrary element of X. The $\uplus$ operator extends a set of bindings. It is required that the added binding does not introduce a variable which is already bound in the base-set. 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 $\Gamma \vdash Q$.

Syntax

There are three alphabets: Variable names x, y, z are freely alpha-renamable. They occur as parameters of lambda abstractions, 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. 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.

Syntax

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

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

$$(\lambda x{:}T.t\ )v \mid s \ \to\ [v/x]t \mid s \quad (\beta_v) \qquad \mathbf{val}\ x = \mathbf{new}\ c;\ t \mid s \ \to\ t \mid s, x \mapsto c\ (\textsc{new})$$

$$\frac{x \mapsto T_c\left\{\overline{l = v}\right\} \in s}{x.l_i \mid s \ \to\ v_i \mid s}\ (\textsc{sel}) \qquad \frac{t \mid s \ \to\ t' \mid s'}{e[t] \mid s \ \to\ e[t'] \mid s'}\ (\textsc{context})$$

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

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (\textsc{var}) \qquad \frac{\Gamma \vdash t \ni l : T'}{\Gamma \vdash t.l : T'}\ (\textsc{sel})$$

$$\frac{\Gamma \vdash t : S \to T\ ,\ t' : T'\ ,\ T' <: S}{\Gamma \vdash t\ t' : T}\ (\textsc{app})$$

$$\frac{x \notin \mathit{fn}(T) \qquad \Gamma \vdash S\ \mathbf{wf}}{\Gamma, x : S \vdash t : T} \over \Gamma \vdash \lambda x{:}S.t : S \to T}\ (\textsc{abs})$$

$$\frac{x \notin \mathit{fn}(T')}{\Gamma \vdash T_c\ \mathbf{wf}\ ,\ T_c \prec_x \overline{L : S..U}, \overline{l : V}} \over \Gamma, x : T_c \vdash \overline{S <: U}\ ,\ \overline{v : V'}\ ,\ \overline{V' <: V}\ ,\ t : T' \over \Gamma \vdash \mathbf{val}\ x = \mathbf{new}\ T_c\left\{\overline{l = v}\right\};\ t : T'}\ (\textsc{new})$$

Fig. 1. The DOT Calculus : Syntax, Reduction, Type Assignment

We assume that the label alphabets l 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, lambda abstractions $\lambda x{:}T.t$, function applications $t\ t'$, field selections $t.l$, and object creation expressions $\mathbf{val}\ x = \mathbf{new}\ c;\ t$ where c is a constructor $T_c\left\{\overline{l = v}\right\}$. The latter binds a variable x to a new instance of type T_c with fields $\overline{l}$ initialized to values $\overline{v}$. The scope of x extends through the term t.

Two subclasses of terms are values v, which consist of just variables and lambda abstractions, and paths v which consist of just variables and selections.

The types in DOT are denoted by letters S, T, U, or V. 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.
- Function types $T \to T'$.
- 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 two forms of declarations in DOT, which are both part of refinement types. A value declaration $l : T$ introduces a field with type T. 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 as follows.

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

Sets of declarations form a lattice with the given meet $\wedge$ and join $\vee$, the empty set of declatations 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 $l : \bot$ and for every type label L the declaration $L : \top..\bot$.

Reduction rules

Reduction rules $t \mid s \rightarrow t' \mid s'$ in DOT rewrite pairs of terms t and stores s, where stores map variables to constructors. There are three main reduction rules: (β_V) is the standard beta reduction of call-by-value lambda calculus. 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. Rule (SEL) rewrites a field selection $x.l$ by retrieving the corresponding value from the store. These reduction rules can be applied anywhere in a term where the hole [] of an evaluation context e can be situated.

Type assignment rules

The last part of Figure 1 presents rules for type assignment. The three rules (VAR), (ABS), and (APP) on the left correspond to simply typed lambda calculus with a subtyping discipline (I believe it is straightforward to extend this to full $F_{<:}$). Rule (ABS) has an additional precondition stating that the bound variable x may not appear in the function's result type T'. It would be an interesting extension of the calculus to drop this precondition, thereby allowing dependent function types. Instead of adding a separate subsumption rule, subtyping is expressed by preconditions in rules (APP) and (NEW). Rule rule (SUB) is the standard subsumption rule for subtyping.

Rule (SEL) types a field selection 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 2 and is further explained below.

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** $x =$ **new** $T_c \left\{\overline{l = v}\right\}$; t, one verifies first that the type T_c is well-formed (see Figure 4 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 2. 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 $x : T_c$. In particular this allows field values that recurse on "self" by referring to the bound variable x. As in the (ABS) rule, it is required that the bound variable does not appear in the expression's result type T'.

Membership

Figure 2 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. There are different rules for paths and general terms. Rule (PATH-$\ni$) applies to paths p that

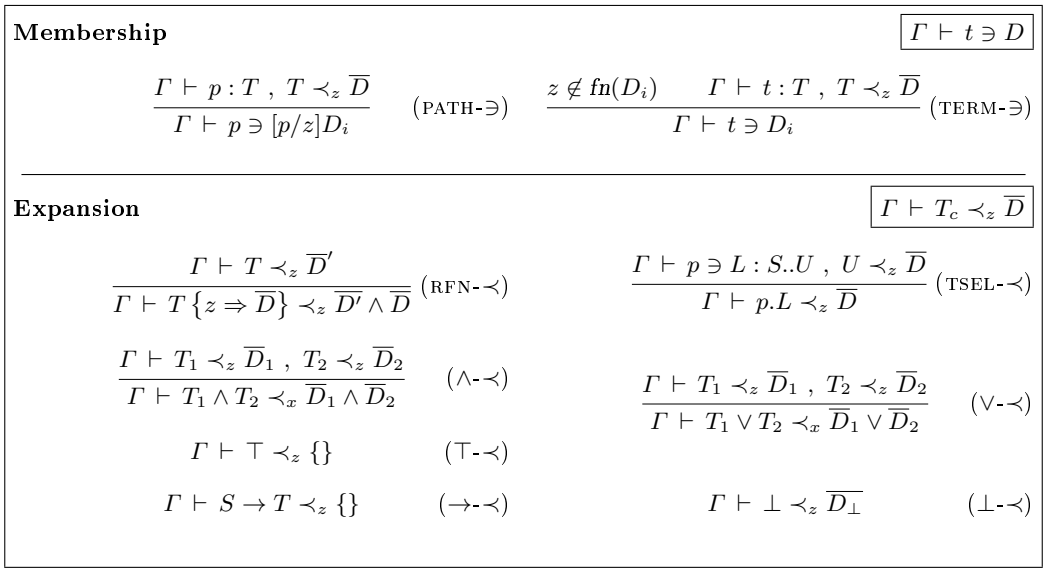

Fig. 2. The DOT Calculus : Membership and Expansion

have a refinement type $T\{z \Rightarrow \overline{D}\}$ as their type. Members of p are then all definitions D in the refinement, where any use of the self reference z is replaced by the path p itself. Rule (AND-∋) allows to merge two member definitions (of the same label) by conjoining them with $\wedge$. Rule (TERM-∋) establishes the members of general terms t of type T by introducing a dummy variable z of type T and then using the rules for path membership on z. It must hold that z itself does not form part of the resulting member D.

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. Since this is the only place where expansion is needed, it is sufficient to define it on concrete types only. The bottom part of Figure 2 gives the typing rules for expansion.

Rule (RFN-≺) states that a refinement type $T_c \prec_z \overline{D}$ expands to the conjunction of the expansion D' of T_c and the newly added declarations D. Rule (TSEL-≺) states that a type selection $p.L$ carries the same declarations as the upper bound U of L in T. Rule (∧-≺) states that expansion distributes through meets. Rule (⊤-≺) states that the top type $\top$ expands to the empty set.

Subtyping

Figure 3 defines the subtyping judgement $\Gamma \vdash S <: T$ which states that in environment Γ type S is a subtype of type T. Any term of type S can then be regarded via the subsumption rule (SUB) as a term of type T.

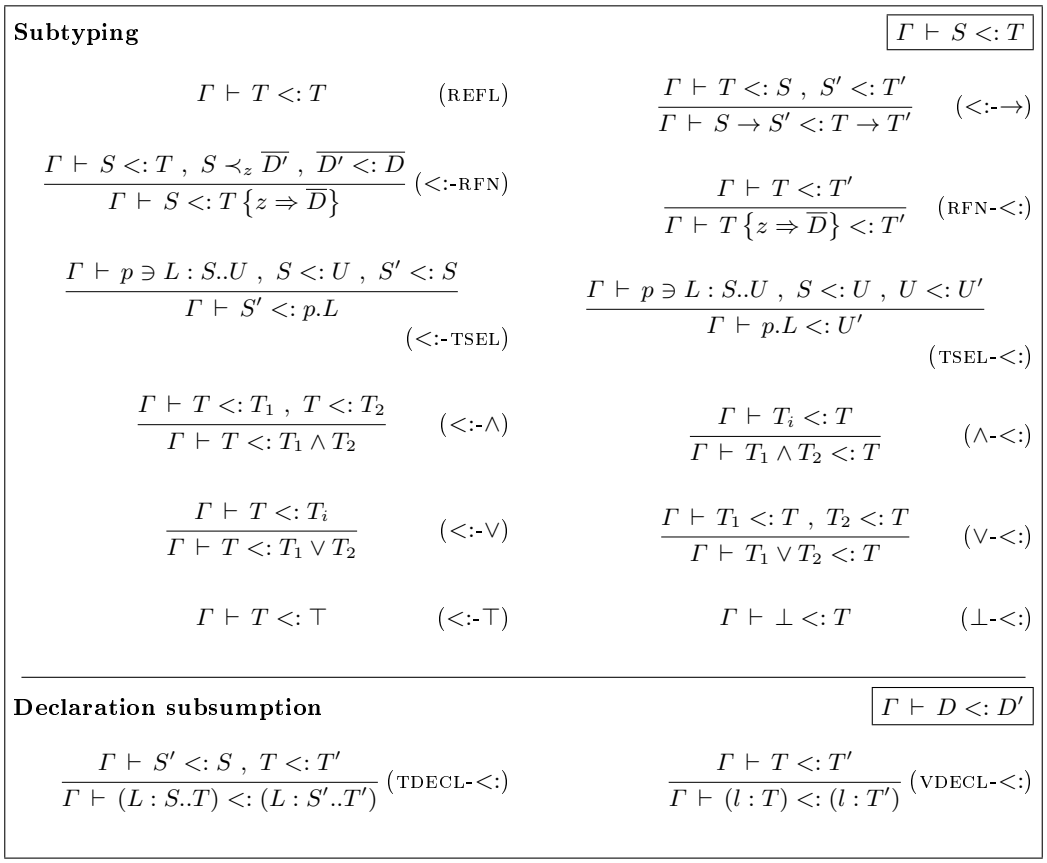

Fig. 3. The DOT Calculus : Subtyping and Declaration Subsumption

As usual, subtyping is reflexive (REFL) but a corresponding rule for transitivity is missing. However, most of the rules for specific types in Figure 3 have transitivity built in. The issue of transitivity is further discussed below.

To prove that a type S is a subtype of a refinement type $T\left\{z \Rightarrow \overline{D}\right\}$, it must hold that S is a subtype of T and that S contains declarations that subsume the declarations $\overline{D}$ ($<:$-RFN). The refinement type itself is a subtype of its parent type T and all its supertypes (RFN-$<:$).

A type selection $p.L$ is a subtype of (all supertypes of) the upper bound U of L in p (TSEL-$<:$). It is a supertype of (all subtypes of) its lower bound S ($<:$-TSEL). Function types are subject to the standard co-/contravariant subtyping rule ($<:$-$\rightarrow$). The final six rules in Figure 3 turn the subtyping relation into a lattice with meets $\wedge$, joins $\vee$, bottom element $\bot$ and top element $\top$.

One interesting aspect of the subtyping relation of DOT is that it is reflexive (REFL) but not transitive. Instead of a separate rule that specifies global transitivity, we find transitivity encoded in most of the inference rules of Figure 3. There is only a single situation where transitivity does not hold: Given a type declaration $L : S..U$ which is a member of some type T, and a variable x of type T, we have that $S <: x.L$ by ($<:$-TSEL) and $x.L <: U$ by (TSEL-$<:$). However, this does not imply that $S <: U$. In fact, T might be an *unrealizable* type, which admits no solution for its L member, precisely because the lower bound of L is not a subtype of its upper bound.

The type assignment rule (NEW) in Figure 1 has as one of its preconditions that the type of the created object must be realizable: each lower bounds must be a subtype of its corresponding upper bound. Subtyping can be shown to be transitive in DOT if the least type of a subtype chain is realizable.

Declaration Subsumption

The declaration subsumption judgement $\Gamma \vdash D <: D'$ in Figure 3 states that in environment Γ the declaration D subsumes the declaration D'. There are two rules, one for type declarations and one for value 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'.

Well-formedness

The well-formedness judgement $\Gamma \vdash T$ **wf** in Figure 4 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). 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,

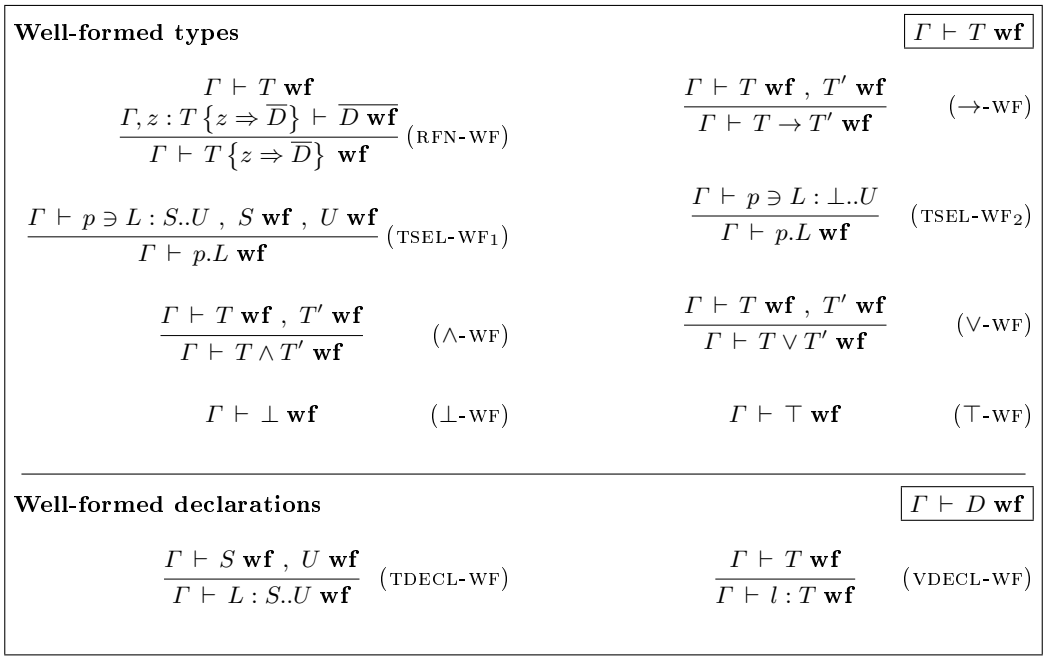

Fig. 4. The DOT Calculus : Well-Formedness

the well-formedness judgement of $p.L$ would not have a finite proof. No such restriction exists for the upper bound of L. 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 $\Gamma \vdash D$ **wf**. All declarations are well-formed if their constituent types are well-formed.