Dependent Object Types

Spring 2012 Semester Project

Nada Amin
EPFL
nada.amin@epfl.ch

Abstract

In my semester project, I have studied a new proposed 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. My ultimate goal was to prove DOT sound, but instead, I found lots of counterexamples to soundness, and have explored patches to the calculus.

1. Introduction

We briefly summarize the salient features of the original DOT calculus. The reader is encouraged to consult the appendices for the full description. We will often refer to the rules of the calculus in the remainder of this report.

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 mixing composition. The calculus does not model what's currently in Scala: it is more normative than descriptive.

The terms in DOT consists 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 $T_c\ \{\overline{l = v}\}$.

The types in DOT consists of type selections $p.L$, refinement types $T\ \{z \Rightarrow \overline{D}\}$, function types $T\ \rightarrow\ T'$, type intersections $T \wedge T'$, type unions $T \vee T'$, a top type $\top$, and a bottom type $\bot$.

The DOT calculus is expressed using a small-step operational semantics. The typing judgment directly or indirectly uses the following other judgments: membership, expansion, well-formedness and subtyping. Expansion "flattens" all the declarations of a type, and membership selects a particular flattened declaration by label. Expansion is precise. The NEW typing judgment relies on the preciseness of expansion to check that all bounds of types declared within the type of the constructor are consistent (i.e. that the lower bound is a subtype of the upper bound), and that all values declared within the type of the constructor are initialized.

In the rest of this introduction, we present some program examples. We summarize our findings of counterexamples to soundness in section 2. We explore some patches to the calculus in section 3. We discuss opportunities for future work and conclude in section 4.

1.1 Program Examples

1.1.1 Basics: Booleans, Error, ...

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

$$
\begin{array}{l}
\mathbf{val}\ root = \mathbf{new}\ \top\{r \Rightarrow \\
\quad Unit : \bot..\top \\
\quad unit : \top \rightarrow r.Unit \\
\quad Boolean : \bot..\top\{z \Rightarrow \\
\qquad ifNat : (r.Unit \rightarrow r.Nat) \rightarrow (r.Unit \rightarrow r.Nat) \rightarrow r.Nat \\
\qquad \} \\
\quad false : r.Unit \rightarrow r.Boolean \\
\quad true : r.Unit \rightarrow r.Boolean \\
\quad \ldots \\
\quad error : r.Unit \rightarrow \bot \\
\}\{ \\
\quad unit = \lambda x : \top.\ \ \mathbf{val}\ u = \mathbf{new}\ root.Unit;\ \ u \\
\quad false = \lambda u : root.Unit. \\
\qquad \mathbf{val}\ ff\ = \mathbf{new}\ root.Boolean\{ \\
\qquad\quad ifNat = \lambda t : root.Unit \rightarrow root.Nat. \\
\qquad\qquad \lambda e : root.Unit \rightarrow root.Nat. \\
\qquad\qquad e\ root.unit \\
\qquad \}; \\
\qquad ff \\
\quad true = \ldots \\
\quad error = \lambda u : root.Unit.\ \ root.error\ u \\
\quad \ldots \\
\}; \\
\ldots
\end{array}
$$

1.1.2 Polymorphic 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. We can choose to make the lists invariant or covariant in the element

type by fixing the lower bound of the $Elem$ re-declaration to the upper bound or to $\bot$, respectively.

$$
\begin{array}{l}
\textbf{val}\ genLists = \textbf{new}\ \top\{g \Rightarrow ListPackage : \bot..\top\{p \Rightarrow \\
\ Elem : \bot..\top \\
\ ListOfElement : g.List\,\{z \Rightarrow Elem : p.Elem..p.Elem\}\,.. \\
\qquad\qquad g.List\,\{z \Rightarrow Elem : p.Elem..p.Elem\} \\
\ List : \bot..\top\{z \Rightarrow \\
\qquad isEmpty : root.Unit \to root.Boolean \\
\qquad head : root.Unit \to g.Elem \\
\qquad tail : root.Unit \to g.ListOfElem \\
\qquad\} \\
\ nil : root.Unit \to g.ListOfElem \\
\ cons : g.Elem \to g.ListOfElem \\
\}\}; \\
\textbf{val}\ natLists = \textbf{new}\ genLists.ListPackage\{p \Rightarrow Elem : Nat..Nat\}\{ \\
\ nil = \ldots \\
\ cons = \ldots \\
\}; \\
\ldots
\end{array}
$$

2. Counterexamples

2.1 Subtyping Transitivity and Preservation

Some subtyping transitivity is essential for soundness. Indeed, we show how to construct a counterexample to preservation from any counterexample to subtyping transitivity where the three involved types S, T, U are expressible within a realizable universe, though the types themselves don't need to be realizable:

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

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 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 $S <: U$. Thus, we need subtyping transitivity: $S <: T$ and $T <: U$ implies $S <: U$.

2.2 Non-Expanding Types and Subtyping Transitivity

The well-formedness rule TSEL-WF$_2$ allow a type selection $p.L$ to refer back to itself in its upper bound. This is useful for expressing recursive class types and F-bounded abstract types.

Consider the simplest possible type that refers to itself: $p : T = \top\,\{z \Rightarrow L : \bot..z.L\}$. Now, $p.L$ is well-formed but non-expanding. Indeed, the only expansion rule that applies to $p.L$ is TSEL-$\prec$, but it requires expanding the upper bound... $p.L$! Thus, there is no finite derivation for the expansion of $p.L$.

Non-expanding types are problematic for subtyping transitivity. Consider: $p.L <: \top$ by $<:$-$\top$ and $\top <: \top\,\{z \Rightarrow\}$ by $<:$-RFN. By transitivity on $\top$, we expect $p.L <: \top\,\{z \Rightarrow\}$, but we cannot infer this from the rules. The only rules that apply are $<:$-RFN and TSEL-$<:$. $<:$-RFN gets stuck expanding $p.L$. TSEL-$<:$ gets stuck because the conclusion is also a premise as $p \ni L : \bot..p.L$.

For this counterexample, it seems enough to just add transitivity on $\top$ as a built-in rule. But this doesn't solve the general issue that non-expanding types don't play well with subtyping. The problem is deep, as it's possible to construct a realizable universe with three non-trivial types that fail subtyping transitivity.

Consider an environment where u is bound to:

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

Note that each lower bound is a subtype of its upper bound, so u is realizable. Hence, if we find a counterexample to subtyping transitivity with types expressible within u, we can apply the trick from section 2.1 to create a counterexample to preservation.

Indeed, here is such a counterexample to subtyping transitivity:

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

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

2.3 Narrowing

2.3.1 Functions as Objects

The original DOT calculus has both object and function types. It doesn't have built-in methods, but they can be expressed as functional labels. On the other hand, Scala doesn't have a built-in function type. Instead, it has both methods and fields, and encodes functions as objects with an apply method. Here, we show that the DOT model as originally designed is problematic.

A concrete object can 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 is a subtype of $\top \to \top$, but $f\ (\lambda x{:}\top.\ x)$ is stuck (and, rightfully, doesn't typecheck). But we can use narrowing to create a counterexample to preservation: $(\lambda g{:}\top \to \top.\ g\ (\lambda x{:}\top.\ x))\ f$.

2.3.2 TERM-$\ni$ 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-$\ni$ 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:

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

Let Y be a shorthand for the type:

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

Now, consider the term

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

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

2.3.3 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.

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

2.3.4 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.

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

2.4 Path Equality

We need to be able to relate path-dependent types after reduction. The original DOT calculus doesn't have any rules for dealing with path-equality. Here is an example which motivates the need for path-equality provisions.

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

$a.i.l$ 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, the need for acknowledging path equality.

3. Patches

We have explored several patches to the calculus to deal with the counterexamples presented above.

3.1 Well-Formed and Expanding Types

We introduce a new judgment form for whether a type is well-formed and expanding: $\Gamma \vdash T$ **wfe** if and only if $\Gamma \vdash T$ **wf** and $\exists \overline{D}$ such that $\Gamma \vdash T \prec_z \overline{D}$. Then, we replace all other uses of the **wf** judgment, including those within the **wf** judgment inference rules, with uses of the **wfe** judgment.

We limit subtyping to **wfe** types. In fact, we make subtyping regular with respect to **wfe**: i.e., if $\Gamma \vdash S <: T$, then $\Gamma \vdash S$ **wfe** and $\Gamma \vdash T$ **wfe**. Assuming a few lemmas, we are able to formally prove that this modified subtyping judgment is transitive. The most notable lemma we assume is the "Galois connection" between subtyping and expansion: if $\Gamma \vdash S <: T$, $S \prec_z \overline{D_S}$, $T \prec_z \overline{D_T}$, then $\Gamma, z : S \vdash \overline{D_s} <: \overline{D_T}$.

3.2 Functions as Sugar

We adopt Scala's model of treating functions as syntactic sugar for an object with a special method. In order to do so, we introduce a new kind of label for methods with one parameter: $m : S \to U$.

A difference in expressivity between this model and the original one is that we now have to explicitly provide a return type of the method, while the return type was inferred for functions. In practice, this is cumbersome but not fundamentally limiting.

3.3 Explicit Widening

In the original DOT calculus, 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 soundness problems due to narrowing, we make widening an explicit operation and change 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 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 needs to be propagated correctly. We will motivate this informally with examples.

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

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:

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

$a.i.l$ 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 $(b : \top\{z \Rightarrow X : \bot..\top, l : z.X\}).l$.

3.4 Path Equality Provisions

We add the path equality provisions to the subtyping rules.

Let's first ignore the extension of the calculus requiring explicit widenings introduced in 3.3. 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 last example in 3.3.

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:

$$\begin{array}{l}\top\{z \Rightarrow \\ \quad A : \top\{z \Rightarrow m : \bot \to \top\}..\top \\ \quad B : \top..\top \\ \quad m : z.A \to \top \\ \}\end{array}$$

Let T be a shorthand for the type:

$$\begin{array}{l}\top\{z \Rightarrow \\ \quad A : \top\{z \Rightarrow m : \bot \to \top\}..\top \\ \quad B : \top..\top \\ \quad m : z.A\{z \Rightarrow B : \top..\top\} \to \top \\ \}\end{array}$$

Now, consider the term:

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

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.)

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

Notice that d has type $\bot$ if you ignore the cast on b. This example doesn't typecheck 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 $d.foo$, 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.

4. Conclusion and Future Work

This semester's work was instructive in pointing out the problematic aspects of the original DOT calculus, and exploring some possible fixes. However, I haven't been able to reach a revised calculus without soundness hole. Furthermore, I believe that the revised calculus with all the patches is not as elegant as the original calculus. Going forward, I want to take a broader view by by studying and incorporating ideas from other calculi which model dependent types, and path-dependent types in particular.

Acknowledgments

I thank Adriaan Moors and Martin Odersky for sharing previous work, fruitful discussions and guidance.