Dependent Object Types

Towards a foundation for Scala's type system

Nada Amin, Tiark Rompf, Adriaan Moors, Martin Odersky

Microsoft Research

July 2, 2013

DOT: Dependent Object Types

The DOT calculus proposes a new *type-theoretic foundation* for Scala and languages like it. It models

- path-dependent types
- abstract type members
- mixture of nominal and structural typing via refinement types

It does not model

- inheritance and mixin composition
- what's currently in Scala

DOT normalizes Scala's type system by

- unifying the constructs for type members
- providing classical intersection and union types

DOT: Syntax

- terms

  variables x, y, z

  selections $t.l$

  method invocations $t.m(t)$

  object creations $\mathbf{val}\ y = \mathbf{new}\ c;\ t'$

  c is a constructor $T_c \left\{ \overline{l = v}\ \overline{m(x) = t} \right\}$

- types

  type selections $p.L$

  refinement types $T \left\{ z \Rightarrow \overline{D} \right\}$

  type intersections $T \wedge T'$

  type unions $T \vee T'$

  a top type $\top$

  a bottom type $\bot$

- declarations

  type $L : S..U$

  value $l : T$

  method $m : S \rightarrow U$

Classical Intersection and Union Types

- form a lattice wrt subtyping
- simplify glb and lub computations

```
trait A { type T <: A }
trait B { type T <: B }
trait C extends A with B { type T <: C }
trait D extends A with B { type T <: D }
// in Scala, lub(C, D) is an infinite sequence
A with B { type T <: A with B { type T <: A with B {
  type T <: ...
}}}
// type inference needs to compute glbs and lubs
if (cond) ((a: A) => c: C) else ((b: B) => d: D)
// lub(A => C, B => D) <: glb(A, B) => lub(C, D)
```

Constructs for Type Members

```
trait Food
trait Animal {
  // in DOT, abstract Meal: Bot .. Food
  type Meal <: Food
  def eat(meal: Meal) {}
}
// in Dot, concrete Grass: Bot .. Food
trait Grass extends Food
trait Cow extends Animal {
  // in DOT, abstract Meal: Grass .. Grass
  type Meal = Grass
}
val a = new Animal {}
val c = new Cow {}
val g = new Grass {}
a.eat(???) // ???.type <: a.Meal so ???.type <: Bot
c.eat(g) // g.type <: c.Meal so g.type <: Grass
```

DOT: Judgments

Typing Judgments

- type assignment
  $\Gamma \vdash t : T$
- subtyping
  $\Gamma \vdash S <: T$
- well-formedness
  $\Gamma \vdash T \ \mathbf{wf}$
- membership
  $\Gamma \vdash t \ni D$
- expansion
  $\Gamma \vdash T \prec_z \overline{D}$

Small-Step Operational Semantics

- reduction
  $t \mid s \ \rightarrow \ t' \mid s'$

Revisiting LUB Computation

- Suppose f has type $T_f = (A \rightarrow_s C) \vee (B \rightarrow_s D)$
- $T_f = \top \{z \Rightarrow apply : A \rightarrow C\} \vee \top \{z \Rightarrow apply : B \rightarrow D\}$
- Let's type-check $y = (\textbf{app}\ f\ x) = f.apply(x)$
- $T_f \prec_f \{apply : A \wedge B \rightarrow C \vee D\}$
- $f \ni apply : A \wedge B \rightarrow C \vee D$
- $T_x <: A \wedge B$
- $T_y = C \vee D$

Revisiting Refined Type Members

```
trait Food
trait Animal {
  // in DOT, abstract Meal: Bot .. Food
  type Meal <: Food
  def eat(meal: Meal) {}
}
// in Dot, concrete Grass: Bot .. Food
trait Grass extends Food
trait Cow extends Animal {
  // in DOT, abstract Meal: Grass .. Grass
  type Meal = Grass
}
```

$$Cow \prec_c \{Meal : Grass \vee Bot..Grass \wedge Food, eat : c.Meal \rightarrow Unit\}$$

$$Cow \prec_c \{Meal : Grass..Grass, eat : c.Meal \rightarrow Unit\}$$

Why not alias Meal to Food in Animal?

```
trait Food
trait Animal {
  // in DOT, abstract Meal: Food .. Food
  type Meal = Food
  def eat(meal: Meal) {}
}

trait Grass extends Food
trait Cow extends Animal {
  // in DOT, abstract Meal: Grass .. Grass
  type Meal = Grass
}
```

$$Cow \prec_c \{Meal : Grass \vee Food..Grass \wedge Food, eat : c.Meal \to Unit\}$$

$$Cow \prec_c \{Meal : Food..Grass, eat : c.Meal \to Unit\}$$

Example: Nominal Class Hierarchies

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

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

Counterexample: TERM-$\ni$ Restriction

Let X be a shorthand for the type:

$$\top\{z \Rightarrow\ L_a : \top..\top\ l : z.L_a\}$$

Let Y be a shorthand for the type:

$$\top\{z \Rightarrow\ l : \top\}$$

Now, consider the term

$$\textbf{val}\ u = \textbf{new}\ X\,\{l = u\}\,;$$
$$(\textbf{app}\ (\textbf{fun}\ (y : \top \rightarrow_s Y)\ Y\ (\textbf{app}\ y\ u))\ (\textbf{fun}\ (d : \top)\ Y\ (\textbf{cast}\ X\ u))).l$$

- How to type (**cast** X u).l?

Counterexample: Path Equality

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

- $a.i.l$ reduces to $b.l$.
- $b.l$ has type $b.X$, so we need $b.X <: a.i.X$.

Lemma: Subtyping Inversion?

$$\frac{\Gamma \vdash p : T \;,\; p' : T' \;,\; T' <: T \;,\; p \ni D}{\exists D', \Gamma \vdash p' : D' \;,\; D' <: D}$$

- Take $p = a.b$ and $p' = b$.
- Need to show $b.X <: a.i.X$?
- Need p reduces to p'!

Counterexample: (Expansion and) Well-Formedness Lost

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

DOT: Dependent Object Types

- DOT is a core calculus for path-dependent types.
- DOT aims to normalize Scala's type system.
- Still tweaking the design to prove type safety!
  - Preservation is tricky... any alternatives?
  - Logical relations?
  - Big-step semantics?