DOT

(**D**ependent **O**bject **T**ypes)

Nada Amin

ECOOP PC Workshop

February 28, 2016

DOT: Dependent Object Types

- DOT is a core calculus for path-dependent types.
- Goals
  - simplify Scala's type system by desugaring to DOT
  - simplify Scala's type inference by relying on DOT
  - prove soundness
- Non-Goals
  - directly model "code sharing" mechanisms such as class inheritance and trait mixins
  - model higher-kinded types and existentials, though partly encodable

Type Members, Path-Dependent Types

```
trait Keys {
  type Key
  def key(data: String): Key
}

object hashKeys extends Keys {
  type Key = Int
  def key(s: String) = s.hashCode
}

def mapKeys(k: Keys, ss: List[String]): List[k.Key] =
  ss.map(k.key)
```

Translucency

```
val abstracted: Keys = hashKeys
val transparent: Keys { type Key = Int } = haskKeys
val upperBounded: Keys { type Key <: Int } = hashKeys
val lowerBounded: Keys { type Key >: Int } = hashKeys

(1: lowerBounded.Key)
(upperBounded.key("a"): Int)
```

Covariant Lists

```
trait List[+E] {
  def isEmpty: Boolean; def head: E;
  def tail: List[E]
}
object List {
  def nil: List[Nothing] = new List[Nothing] {
    def isEmpty = true; def head = head; def tail = tail
  }
  def cons[E](hd: A, tl: List[E]) = new List[E] {
    def isEmpty = false; def head = hd; def tail = tl
  }
}
```

Variance

```
trait List { z =>
  type E
  def isEmpty: Boolean; def head: E;
  def tail: List { type E <: z.E}
}
object List {
  def nil = new List {
    type E = Nothing
    def isEmpty = true; def head = head; def tail = tail
  }
  def cons(x: { type E })(hd: x.E, tl: List { E <: x.E }) =
    new { type E = x.E
    def isEmpty = false; def head = hd; def tail = tl
  }
}
```

Structural Records

```
val pkgList = { p =>
  type List = { z =>
    type E
    def isEmpty: Boolean; def head: z.E;
    def tail: p.List { type E <: z.E}
  }
  def nil: p.List { E = Nothing } = new {
    type E = Nothing
    def isEmpty = true; def head = head; def tail = tail
  }
  def cons(x: { type E })(hd: x.E, tl: p.List { E <: x.E }):
    p.List { type E = x.E } = new {
    type E = x.E
    def isEmpty = false; def head = hd; def tail = tl
  }
}
```

Structural Records

```
val pkgList = { p =>
  type List = { z =>
    type E
    def isEmpty: Boolean; def head: z.E;
    def tail: p.List { type E <: z.E}
  }
  def nil: p.List { E = Nothing } = new { l =>
    type E = Nothing
    def isEmpty = true; def head = l.head; def tail = l.tail
  }
  def cons(x: { type E })(hd: x.E, tl: p.List { E <: x.E }):
    p.List { type E = x.E } = new {
    type E = x.E
    def isEmpty = false; def head = hd; def tail = tl
  }
}
```

Nominality

```
pkgList: { p =>
  type List <: { z =>
    type E
    def isEmpty: Boolean; def head: E;
    def tail: List { type E <: z.E}
  }
  def nil: p.List { E = Nothing }



  def cons(x: { type E })(hd: x.E, tl: List { E <: x.E }):
    p.List { type E = x.E }



}
```

DOT: Syntax

x, y, z	**Variable**	$v ::=$	**Value**
a, b, c	**Term member**	$\quad \nu(x\colon T^x)d^x$	object
A, B, C	**Type member**	$\quad \lambda(x\colon T)t^x$	lambda
$S, T, U ::=$	**Type**	$s, t, u ::=$	**Term**
$\quad \top$	top type	$\quad x$	variable
$\quad \bot$	bot type	$\quad v$	value
$\quad S \wedge T$	intersection	$\quad x.a$	selection
$\quad S \vee T$	union	$\quad x\ y$	application
$\quad \{a : T\}$	field declaration	$\quad \mathbf{let}\ x = t\ \mathbf{in}\ u^x$	let
$\quad \{A : S..T\}$	type declaration	$d ::=$	**Definition**
$\quad x.A$	type selection	$\quad \{a = t\}$	field def.
$\quad \mu(x\colon T^x)$	recursive type	$\quad \{A = T\}$	type def.
$\quad \forall(x\colon S)T^x$	dependent function	$\quad d_1 \wedge d_2$	aggregate def.

Type Members, Path-Dependent Types (in DOT)

```
let p = ν(p) {

  Keys = μ(s: { Key } ∧ { key: String → s.Key })

  hashKeys = ν(s) { Key = Int; key = λ(s: String)s.hashCode }

  mapKeys = λ(k: p.Keys)λ(ss: List[String])ss.map(k.key)

} in
...
p.hashKeys : μ(s: { Key = Int } ∧ { key: String → s.Key })
...
p.hashKeys : μ(s: { Key } ∧ { key: String → s.Key })
...
p.hashKeys : p.Keys
```

Type Assignment $\Gamma \vdash t : T$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \tag{\textsc{Var}}$$

$$\frac{\Gamma,\ x : T \vdash t : U}{\Gamma \vdash \lambda(x : T)t : \forall(x : T)U} \tag{\textsc{All-I}}$$

$$\frac{\Gamma \vdash x : \forall(z : S)T \quad \Gamma \vdash y : S}{\Gamma \vdash x\,y : [z := y]T} \tag{\textsc{All-E}}$$

$$\frac{\Gamma,\ x : T \vdash d : T}{\Gamma \vdash \nu(x : T)d : \mu(x : T)} \tag{\{\}-I}$$

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \tag{\{\}-E}$$

Type Assignment (2)

$$\frac{\Gamma \vdash t : T \quad \Gamma,\ x : T \vdash u : U \quad x \notin fv(U)}{\Gamma \vdash \textbf{let}\ x = t\ \textbf{in}\ u : U} \quad (\textsc{Let})$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x : T)} \quad (\textsc{Rec-I})$$

$$\frac{\Gamma \vdash x : \mu(x : T)}{\Gamma \vdash x : T} \quad (\textsc{Rec-E})$$

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \quad (\textsc{And-I})$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: U}{\Gamma \vdash t : U} \quad (\textsc{Sub})$$

Definition Type Assignment $\Gamma \vdash d : T$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} : \{a : T\}} \qquad (\textsc{Fld-I})$$

$$\Gamma \vdash \{A = T\} : \{A : T..T\} \qquad (\textsc{Typ-I})$$

$$\frac{\Gamma \vdash d_1 : T_1 \quad \Gamma \vdash d_2 : T_2 \\ dom(d_1), dom(d_2) \text{ disjoint}}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \qquad (\textsc{AndDef-I})$$

Note that there is no subsumption rule for definition type assignment.

Subtyping $\Gamma \vdash T <: U$

$$\Gamma \vdash T <: \top \qquad (\textsc{Top})$$

$$\Gamma \vdash \bot <: T \qquad (\textsc{Bot})$$

$$\Gamma \vdash T <: T \qquad (\textsc{Refl})$$

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \qquad (\textsc{Trans})$$

$$\Gamma \vdash T \wedge U <: T \qquad (\textsc{And}_1\text{-}{<:})$$

$$\Gamma \vdash T \wedge U <: U \qquad (\textsc{And}_2\text{-}{<:})$$

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \qquad ({<:}\text{-}\textsc{And})$$

Subtyping (2)

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \qquad (\textsc{Sel-<:})$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \qquad (\textsc{<:-Sel})$$

$$\frac{\begin{array}{c}\Gamma \vdash S_2 <: S_1 \\ \Gamma,\ x : S_2 \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \forall(x : S_1)\, T_1 <: \forall(x : S_2)\, T_2} \qquad (\textsc{All-<:-All})$$

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a : T\} <: \{a : U\}} \qquad (\textsc{Fld-<:-Fld})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}} \qquad (\textsc{Typ-<:-Typ})$$

Subtyping of Recursive Types?

$$\frac{\Gamma,\ x : T \vdash\ T <: U}{\Gamma \vdash\ \mu(x\!: T) <: \mu(x\!: U)} \qquad \text{(Rec-<:-Rec)}$$

Preservation Challenge: Branding

```
trait Brand {
  type Name
  def id(x: Any): Name
}
// in REPL
val brand: Brand = new Brand {
  type Name = Any
  def id(x: Any): Name = x
}
brand.id("hi"): brand.Name // ok
"hi": brand.Name // error but probably sound
val brandi: Brand = new Brand {
  type Name = Int
  def id(x: Any): Name = 0
}
brandi.id("hi"): brandi.Name // ok
"hi": brandi.Name // error and probably unsound
```

Why It's Difficult

We always need some form of inversion.
E.g.:

- If $\Gamma \vdash x : \forall(x : S)\,T$
  then x is bound to some lambda value $\lambda(x\!:\!S')t$,
  where $S <: S'$ and $\Gamma \vdash t : T$.

This looks straightforward to show.
But it isn't.

User-Definable Theories

In DOT, the subtyping relation is given in part by user-definable definitions

```
type T >: S <: U
```

This makes `T` a supertype of `S` and a subtype of `U`.
By transitivity, `S <: U`.
So the type definition above proves a subtype relationship which was potentially not provable before.

Bad Bounds

What if the bounds are non-sensical?

```
type T >: Any <: Nothing
```

By the same argument as before, this implies that

```
Any <: Nothing
```

Once we have that, again by transitivity we get $S <: T$ for arbitrary S and T.
That is the subtyping relations collapses to a point.

Can we Exclude Bad Bounds Statically?

Type $\bot$ is a subtype of all other types, including

```
{ type E = Top }
```

and

```
{ type E = Bot }
```

.

So if p: $\bot$ we have Top $<:$ p.E and p.E $<:$ Bot.
Transitivity would give us Top $<:$ p.E $<:$ Bot!
Subtyping lattice collapses.

Adding intersection types is equivalent to bottom in terms of bad bounds!
Try p:

```
{ type E = Top } & { type E = Bot }
```

.

But maybe we can verify all intersections in the program? No, because types can get more specific during reduction. Requiring good bounds breaks monotonicity.

Dealing With It: A False Start

Bad bounds make problems by combining the selection subtyping rules with transitivity.

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \quad (\textsc{Sel-}<:)$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \quad (<:\textsc{-Sel})$$

Can we "tame" these rules so that bad bounds cannot be exploited? E.g.

Dealing With It: A False Start

$$\frac{\Gamma \vdash x : \{A : S..T\} \qquad \Gamma \vdash S <: T}{\Gamma \vdash x.A <: T} \qquad (\textsc{Sel-}{<:})$$

$$\frac{\Gamma \vdash x : \{A : S..T\} \qquad \Gamma \vdash S <: T}{\Gamma \vdash S <: x.A} \qquad ({<:}\textsc{-Sel})$$

Problem: we lose monotonicity.
Tighter assumptions may yield worse results.

Transitivity and Narrowing

$$\frac{\begin{array}{c}\Gamma^a, (x : U), \Gamma^b \vdash\ T <: T' \\ \Gamma^a \vdash\ S <: U\end{array}}{\Gamma^a, (x : S), \Gamma^b \vdash\ T <: T'} \qquad (<:\text{-NARROW})$$

$$\frac{\Gamma \vdash\ S <: T\ ,\ \ T <: U}{\Gamma \vdash\ S <: U} \qquad (<:\text{-TRANS})$$

Observations and Ideas

- Bottom types do not occur at runtime!
- Is it enough to have transitivity in "realizable" environments?
- Yes, though there are some subtleties for subtyping recursive types.

DOT: Some Unsound Variations

- Add subsumption to definition type assignment.

$$\frac{\Gamma \vdash d : T \quad \Gamma \vdash T <: U}{\Gamma \vdash d : U} \qquad (\textsc{Def-Sub})$$

$$\nu(x : \{X : \top..\bot\})\{X = \top\} : \mu(x : \{X : \top..\bot\})$$

- Change type definition from $\{A = T\}$ to $\{A : S..U\}$.

$$\Gamma \vdash \{A : S..U\} : \{A : S..U\} \qquad (\textsc{Typ-I})$$

$$\nu(x : \{X : \top..\bot\})\{X : \top..\bot\} : \mu(x : \{X : \top..\bot\})$$

Retrospective on Proving Soundness

A good proof is one that makes us wiser. – Yuri Manin

- Static semantics should be monotonic. All attempts to prevent bad bounds broke it.
- Embrace subsumption, don't requires precise calculations in arbitrary contexts.
- Create recursive objects concretely, enforcing good bounds and shape syntactically not semantically. Then abstract, if desired.
- Inversion lemmas need only hold for realizable environments.
- Tension between preservation and abstraction. Rely on a precise static environment that corresponds to the runtime.

Unsoundness in Scala (fits in a Tweet)

```scala
trait A { type L >: Any}
def id1(a: A, x: Any): a.L = x
val p: A { type L <: Nothing } = null
def id2(x: Any): Nothing = id1(p, x)
id2("oh")
```

Unsoundess in Java (thanks Ross Tate!)

```
class Unsound {
  static class Bound<A, B extends A> {}
  static class Bind<A> {
    <B extends A> A bad(Bound<A,B> bound, B b) {
      return b;
    }
  }
  public static <T,U> U coerce(T t) {
    Bound<U,? super T> bound = null;
    Bind<U> bind = new Bind<U>();
    return bind.bad(bound, t);
  }
}
```

Thanks!

- Dependent Object Types, FOOL'12
  (Nada Amin, Adriaan Moors, Martin Odersky)
- Foundations of Path-Dependent Types, OOPSLA'14
  (Nada Amin, Tiark Rompf, Martin Odersky)
- From F to DOT: Type Soundness Proofs with Definitional Interpreters
  (Tiark Rompf and Nada Amin)
- The Essence of Dependent Object Types, WadlerFest'16
  (Nada Amin, Samuel Grütter, Martin Odersky, Sandro Stucki, Tiark Rompf)

Preservation Challenge: Path equality

```
trait B { type X; val l: X }
val b1: B = new B { type X = String; val l: X = "hi" }
val b2: B = new B { type X = Int;    val l: X = 0    }
trait A { val i: B }
val a = new A { val i: B = b1 }
println(a.i.l : a.i.X) // ok
println(b1.l  : b1.X)  // ok
println(b1.l  : a.i.X) // error: type mismatch;
// found   : b1.l.type (with underlying type b1.X)
// required: a.i.X
// abstractly, would need to show
Any  <: Nothing // lattice collapse!
// to show
b1.X <: a.i.X
// because
U of b1.X = Any <: Nothing = S of a.i.X
println(b2.l  : a.i.X) // error and probably unsound
```

Path-Dependent Types (Recap Example)

```scala
trait Animal { type Food; def gets: Food
               def eats(food: Food) {}; }
trait Grass; trait Meat
trait Cow extends Animal with Meat {
  type Food = Grass; def gets = new Grass {} }
trait Lion extends Animal {
  type Food = Meat;  def gets = new Meat {} }
val leo = new Lion {}
val milka = new Cow {}
leo.eats(milka) // ok
val lambda: Animal = milka
lambda.eats(milka) // type mismatch
// found : Cow
// required: lambda.Food
lambda.eats(lambda.gets) // ok
```

Path-Dependent Types (Recap Example Continued)

```
def share(a1: Animal)(a2: Animal)
         (bite: a1.Food with a2.Food) {
  a1.eats(bite)
  a2.eats(bite)
}
val simba = new Lion {}
share(leo)(simba)(leo.gets) // ok
share(leo)(lambda)(leo.gets) // error: type mismatch
// found : Meat
// required: leo.Food with lambda.Food

// Observation:
// We don't know whether the parameter type of
share(lambda1)(lambda2)_
// is realizable until run-time.
```

Realizability of Intersection Types at Run-Time

```
val lambda1: Animal = new Lion {}
val lambda2: Animal = new Cow  {}
lazy val p: lambda1.Food & lambda2.Food = ???
// for illustration, say we re-defined the following:
trait Food { type T }
trait Meat extends Food  { type T = Nothing }
trait Grass extends Food { type T = Any }
// statically
p.T /*has fully abstract bounds*/
// at runtime
lambda1.Food /*is*/ Meat /*&*/ lambda2.Food /*is*/ Grass
p /*has type*/ Meat & Grass
// lower bound is union of lower bounds
p.T /*has lower bound*/ Nothing | Any /*is*/ Any
// upper bound is intersection of upper bounds
p.T /*has upper bound*/ Nothing & Any /*is*/ Nothing
p.T /*has bad bounds at run-time!*/
```

Operational Semantics $t \longrightarrow t'$

$$
\begin{array}{rcll}
e[t] & \longrightarrow & e[t'] & \textbf{if } t \longrightarrow t' \\
\textbf{let } x = v \textbf{ in } e[x\ y] & \longrightarrow & \textbf{let } x = v \textbf{ in } e[[z := y]t] & \textbf{if } v = \lambda(z\colon T)t \\
\textbf{let } x = v \textbf{ in } e[x.a] & \longrightarrow & \textbf{let } x = v \textbf{ in } e[t] & \textbf{if } v = \nu(x\colon T)\ldots\{a = t\}\,. \\
\textbf{let } x = y \textbf{ in } t & \longrightarrow & [x := y]t & \\
\textbf{let } x = \textbf{let } y = s \textbf{ in } t \textbf{ in } u & \longrightarrow & \textbf{let } y = s \textbf{ in let } x = t \textbf{ in } u &
\end{array}
$$

where the *evaluation context* e is defined as follows:

$$
e ::= [\,] \mid \textbf{let } x = [\,] \textbf{ in } t \mid \textbf{let } x = v \textbf{ in } e
$$

Note that evaluation uses only *variable renaming*, not full substitution.

Introducing Explicit Stores

We have seen that the `let` prefix of a term acts like a store.
For the proofs of progress and preservation it turns out to be easier to model the store explicitly.
A store is a set of bindings $x = v$ or variables to values.
The evaluation relation now relates terms and stores.

$$s \mid t \;\longrightarrow\; s' \mid t'$$

Operational Semantics $s \mid t \longrightarrow s' \mid t'$

$$
\begin{array}{rcll}
s \mid x.a & \longrightarrow & s \mid t & \textbf{if } s(x) = \nu(x\colon T)\ldots\{a = t\}\ldots \\
s \mid x\,y & \longrightarrow & s \mid [z := y]t & \textbf{if } s(x) = \lambda(z\colon T)t \\
s \mid \textbf{let } x = y \textbf{ in } t & \longrightarrow & s \mid [x := y]t & \\
s \mid \textbf{let } x = v \textbf{ in } t & \longrightarrow & s, x = v \mid t & \\
s \mid \textbf{let } x = t \textbf{ in } u & \longrightarrow & s' \mid \textbf{let } x = t' \textbf{ in } u & \textbf{if } s \mid t \longrightarrow s' \mid t'
\end{array}
$$

Relationship between Stores and Environments

For the theorems and proofs of progress and preservation, we need to relate environment and store.

Definition: An environment Γ *corresponds* to a store s, written $\Gamma \sim s$, if for every binding $x = v$ in s there is an entry $\Gamma \vdash x : T$ where $\Gamma \vdash_! v : T$.

$\Gamma \vdash_! v : T$ is an exact typing relation.

We define $\Gamma \vdash_! x : T$ iff $\Gamma \vdash x : T$ by a typing derivation which ends in a (All-I) or (-I) rule

(i.e. no subsumption or substructural rules are allowed at the toplevel).

Progress and Preservation, Finally

Theorem (Preservation)
If $\Gamma \vdash t : T$ and $\Gamma \sim s$ and $s \mid t \longrightarrow s' \mid t'$, then there exists an environment $\Gamma' \supset \Gamma$ such that, one has $\Gamma' \vdash t' : T$ and $\Gamma' \sim s'$.

Theorem (Progress)
If $\Gamma \vdash t : T$ and $\Gamma \sim s$ then either t is an answer, or $s \mid t \longrightarrow s' \mid t'$, for some store s', term t'.