DOT
(Dependent Object Types)

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-kindled types and existentials, though partly encodable
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 } = hashKeys
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 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
  }
}


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
    }
}

Variance
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
}
  type E = x.E
  def isEmpty = false; def head = hd; def tail = tl
}
}
Structural Records

```scala
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
  }

    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

\[ x, \ y, \ z \]
\[ a, \ b, \ c \]
\[ A, \ B, \ C \]
\[ S, \ T, \ U ::= \]
\[ T \]
\[ \bot \]
\[ S \land T \]
\[ S \lor T \]
\[ \{ a : T \} \]
\[ \{ A : S \ldots T \} \]
\[ x.A \]
\[ \mu(x : T^x) \]
\[ \forall(x : S) T^x \]

Variable

Term member

Type member

Type

top type
bot type
intersection
union
field declaration
type declaration
type selection
recursive type
dependent function

Value

object
lambda

Term

variable
value
selection
application
let

Definition

field def.
type def.
aggregate def.
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 $\Gamma \vdash t : T$

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

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

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

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

$$\frac{\Gamma \vdash x : \{ a : T \}}{\Gamma \vdash x . a : T} \quad \text{(\{}-E)}$$
Type Assignment (2)

\[ \Gamma \vdash t : T \quad \Gamma, \; x : T \vdash u : U \]
\[ x \notin \text{fv}(U) \]
\[ \frac{}{\Gamma \vdash \text{let } x = t \text{ in } u : U} \quad \text{(LET)} \]

\[ \Gamma \vdash x : T \]
\[ \frac{}{\Gamma \vdash x : \mu(x : T)} \quad \text{(Rec-I)} \]

\[ \Gamma \vdash x : \mu(x : T) \]
\[ \frac{}{\Gamma \vdash x : T} \quad \text{(Rec-E)} \]

\[ \Gamma \vdash x : T \quad \Gamma \vdash x : U \]
\[ \frac{}{\Gamma \vdash x : T \land U} \quad \text{(And-I)} \]

\[ \Gamma \vdash t : T \quad \Gamma \vdash T <: U \]
\[ \frac{}{\Gamma \vdash t : U} \quad \text{(Sub)} \]
Definition Type Assignment $\Gamma \vdash d : T$

\[
\Gamma \vdash t : T \\
\Gamma \vdash \{a = t\} : \{a : T\} \quad \text{(FLD-I)}
\]

\[
\Gamma \vdash \{A = T\} : \{A : T..T\} \quad \text{(TYP-I)}
\]

\[
\Gamma \vdash d_1 : T_1 \quad \Gamma \vdash d_2 : T_2 \\
\text{dom}(d_1), \text{dom}(d_2) \text{ disjoint} \\
\Gamma \vdash d_1 \land d_2 : T_1 \land T_2 \quad \text{(ANDDEF-I)}
\]

Note that there is no subsumption rule for definition type assignment.
Subtyping $\Gamma \vdash T <: U$

- $\Gamma \vdash T <: \top$ (Top)
- $\Gamma \vdash \bot <: T$ (Bot)
- $\Gamma \vdash T <: T$ (Refl)
- $\Gamma \vdash S <: T \quad \Gamma \vdash T <: U$
  \[ \frac{} {\Gamma \vdash S <: U} \] (Trans)
- $\Gamma \vdash T \land U <: T$ (And$_1$-<:)
- $\Gamma \vdash T \land U <: U$ (And$_2$-<:)
- $\Gamma \vdash S <: T \quad \Gamma \vdash S <: U$
  \[ \frac{} {\Gamma \vdash S <: T \land U} \] (<:-And)
Subtyping (2)

\[ \Gamma \vdash x : \{ A : S..T \} \]
\[ \Gamma \vdash x.A <: T \]  \hspace{2cm} (SEL-<:)

\[ \Gamma \vdash x : \{ A : S..T \} \]
\[ \Gamma \vdash S <: x.A \]  \hspace{2cm} (<:-SEL)

\[ \Gamma \vdash S_2 <: S_1 \]
\[ \Gamma, x : S_2 \vdash T_1 <: T_2 \]
\[ \Gamma \vdash \forall (x:S_1) T_1 <: \forall (x:S_2) T_2 \]  \hspace{2cm} (ALL-<:-ALL)

\[ \Gamma \vdash T <: U \]
\[ \Gamma \vdash \{ a : T \} <: \{ a : U \} \]  \hspace{2cm} (FLD-<:-FLD)

\[ \Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash T_1 <: T_2 \]
\[ \Gamma \vdash \{ A : S_1..T_1 \} <: \{ A : S_2..T_2 \} \]  \hspace{2cm} (TYP-<:-TYP)
Subtyping of Recursive Types?

\[
\Gamma, \; x : T \vdash T <: U \\
\Gamma \vdash \mu(x : T) <: \mu(x : U)
\]  

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

\texttt{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

\[
\{ \text{type } E = \text{Top} \}
\]

and

\[
\{ \text{type } E = \text{Bot} \}
\]

So if $p : \bot$ we have $\text{Top} \prec p.E$ and $p.E \prec \text{Bot}$. Transitivity would give us $\text{Top} \prec p.E \prec \text{Bot}$!

Subtyping lattice collapses.

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

\[
\{ \text{type } E = \text{Top} \} \& \{ \text{type } E = \text{Bot} \}
\]

But maybe we can verify all intersections in the program? No, because types can get more specific during reduction. Requiring good bounds breaks monotonicity.
Bad bounds make problems by combining the selection subtyping rules with transitivity.

\[
\Gamma \vdash x : \{A : S..T\} \\
\Gamma \vdash x.A <: T \\
\Gamma \vdash x : \{A : S..T\} \\
\Gamma \vdash S <: x.A
\]

\((\text{SEL-}:)\) \\
\((<:-\text{SEL})\)

Can we “tame” these rules so that bad bounds cannot be exploited? E.g.
Dealing With It: A False Start

\[
\Gamma \vdash x : \{A : S..T\} \quad \Gamma \vdash S <: T \\
\overline{\quad \Gamma \vdash x.A <: T \quad (\text{SEL-<:})} \\
\]

\[
\Gamma \vdash x : \{A : S..T\} \quad \Gamma \vdash S <: T \\
\overline{\quad \Gamma \vdash S <: x.A \quad (\text{<:-SEL})} \\
\]

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

\[\Gamma^a, (x : U), \Gamma^b \vdash T <: T'\]
\[\Gamma^a \vdash S <: U\]
\[\Gamma^a, (x : S), \Gamma^b \vdash T <: T'\]

\((<:-\text{NARROW})\)

\[\Gamma \vdash S <: T , T <: U\]
\[\Gamma \vdash S <: U\]

\((<:-\text{TRANS})\)
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.

\[
\Gamma \vdash d : T \quad \Gamma \vdash T <: U \\
\therefore \quad \Gamma \vdash d : U \quad \text{(Def-Sub)}
\]

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

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

\[
\Gamma \vdash \{A : S..U\} : \{A : S..U\} \quad \text{(Typ-I)}
\]

\[
\nu(x:\{X : \top..\bot\})\{X : \top..\bot\} : \mu(x : \{X : \top..\bot\})
\]
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 require 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

```scala
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
```
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

```scala
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 $t \rightarrow t'$

\[
\begin{align*}
e[t] & \rightarrow e[t'] & \text{if } t \rightarrow t' \\
\text{let } x = v \text{ in } e[x \ y] & \rightarrow \text{let } x = v \text{ in } e[[z := y]t] & \text{if } v = \lambda(z: T)t \\
\text{let } x = v \text{ in } e[x.a] & \rightarrow \text{let } x = v \text{ in } e[t] & \text{if } v = \nu(x: T) \ldots \{ a = t \}.
\end{align*}
\]

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

\[
e ::= [] \mid \text{let } x = [] \text{ in } t \mid \text{let } x = v \text{ in } e
\]

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.

$$s \mid t \rightarrow s' \mid t'$$
Operational Semantics $s | t \rightarrow s' | t'$

\[
\begin{align*}
s \mid x.a & \rightarrow s \mid t & \text{if } s(x) = v(x: T) \ldots \{a = t\} \ldots \\
s \mid x y & \rightarrow s \mid [z := y]t & \text{if } s(x) = \lambda(z: T)t \\
s \mid \text{let } x = y \text{ in } t & \rightarrow s \mid [x := y]t \\
s \mid \text{let } x = v \text{ in } t & \rightarrow s, x = v \mid t \\
s \mid \text{let } x = t \text{ in } u & \rightarrow s' \mid \text{let } x = t' \text{ in } u & \text{if } s \mid t \rightarrow s' \mid t' 
\end{align*}
\]
Relationship between Stores and Environments

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

Definition: An environment $\Gamma$ 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 $s \mid t \rightarrow s' \mid t'$, 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 $s \mid t \rightarrow s' \mid t'$, for some store $s'$, term $t'$. 