The Curse of DOT

Adriaan Moors, EPFL

Penn PLClub, 15 July 2011

DOTh unto others

- Martin: (for) now at Typesafe
- Geoff: now at LogicBlox
- Donna: now at Microsoft
- [most authors of previous Scala calculi]: now at Google
(it's not you, DOT, it's me)

Why DO iT?

- Need clean semantics for core Scala
  - no definite one for now
- Driver for change in Scala (3.0?)
  - we'll prove soundness first (promise!)

Design Goals

1. capture essence of Scala
   - structural refinements (with self variable)
     - value members
     - type members: *lower* & upper bounds
   - path-dependent types
   - new, selection, λ abstraction&application

Design Goals

2. study way forward
   - true intersection types, union types
     - get rid of lub approximation
   - commutative type composition
     - composition accumulates member info, rather than linearisation picking a winner among subtypes

Design Goals

3. leave out

   - inheritance (model it on top)
   - methods (have functions already)
   - mutable state
   - everything else

.intro

Sca.la

```
trait List { self =>
  type Element
  val hd: self.Element
  val tl: List{type Element = self.Element}
}
```

- DOT in Scala syntax:
  - Scala: trait = abstract class
    - DOT: abstract type member with unique name
  - `self`: self variable

Path-dependent types

```
trait List { type Element }

val xs: List = ... ; val ys: List = ....

// => xs.Element, ys.Element incompatible
```

- to get equal types, select equal type members on equal paths (the target of the selection)
- scope of unpacking is to an existential type as target is to an abstract type member selection

Refinements

```
trait List { type Element }

val xs: List{ type Element = Int } = ...
val ys: List{ type Element = Int } = ...
```

- `xs.Element`, `ys.Element` are both `Int`

A propos, in Scala:

```
trait List { type Element }
val xs: List = ... ; val zs: xs.type = xs
// => xs.Element same type as zs.Element
```

- DOT does not have singleton types (`xs.type`)
- only really need to select a type on a path: `p.L`
  - (in Scala, select type on a type: `p.type#L`)

.Scala

- post-DOT Scala will have:
  - union types, true intersection types
  - type composition pushed down to members

Union types as lazy lubs

- in theory, least upper bounds and f-bounded polymorphism do not mix (easily)
- in practice, real Scala programs regularly give rise to imprecise (truncated) lubs

F-Bounded Lubbing

```
scala> class F[T <: F[T]]

scala> class A extends F[A]

scala> class B extends F[B]

scala> List(new A, new B)

res0: List[F[_ >: B with A
              <: F[_ >: B with A
                     <: ScalaObject]]]
    = List(A@b83621e, B@5e9ea579)
```

True Intersection

- (Scala) mixin composition: `A with B`
  - not commutative: linearisation picks a winning contribution
- (DOT) true intersection: $A \wedge B$
  - constituents contribute equally to the members of the resulting type

In terms of members

- members of T $\wedge$ T'
  - union of members of T and T'
  - synonymous members' types are $\wedge$'ed
- members of T $\vee$ T'
  - intersection of members of T and T'
  - members' types are $\vee$'ed

Example

```
class Coll { type El }

trait OrderedColl extends Coll {
  type El <: Ordered }

trait GPUColl extends Coll {
  type El <: GPUAble }
```

// Current Scala: ~~OrderedColl with GPUColl~~

```
// DOT: OrderedColl ∧ GPUColl has member
//   type El <: Ordered ∧ GPUAble
```

Gr.eek

$$\frac{t : \mathrm{Top}\{..\ \mathrm{val}\ l: T\ ..\}}{t.l : T^{t}}$$

$$\frac{p : \mathrm{Top}\{..\ \mathrm{type}\ L >: S <: U\ ..\}}{S^{p} <: p.L \ \text{and}\ p.L <: U^{p}}$$

$$\frac{\begin{array}{c} T <: T' \quad T <: Tp\{D\} \quad Tp <: \mathrm{Top}\{D_{p}\} \\ G,\ z: T \ |\!- \ (D_{p}^{z} \wedge D^{z}) <: D'^{z} \end{array}}{T <: T'\{D'\}}$$

Buffet of Sneakiness

- path equality
- checking type well-formedness
- transitivity of subtyping

Path Equality

```
x: {z => type T; val l: z.T} |- x.l : x.T

imagine `x` evaluates to the object ref `a`

x.l --> a.l

preservation must thus relate these types:

x.T --> a.T
```

Path Equality

- common problem in virtual class calculi
- easy solution:
  - embed store in typing context
    - reference → (object type, constructor args)
  - equate types modulo path equality
    - only used in preservation

Well-Formed T

- `WF T` checked when typing `new T`
  - must wait since WF is context-dependent
    - no global class table as in FJ
  - must ensure we check all of T's members

Well-Formed T

- For *all* members, check:
  - for each `type L: T..U`, `T <: U`
  - for each `val l: T`, supplied argument `: T`

All members of T

- members of p.L ?
  - first subsume to *least* structural supertype
  - usually called “exposition”

Quality of Derivations

- DOT collapses subtyping and exposition
  - rules are extremely similar
- track when member info was subsumed (width/depth)
- cf. exact types, without complicating the language of types

Gr.eek @ q

irrelevant

core idea:
track quality flow

```
H1 @ q1   H2 @ q2   H3 @ q3
---------------------------
        H @ q1 ∧ 2
```

q = precise
  | subsumed

simplify:
quality subsumption

```
H1 @ q   H2 @ q   H3 @ q'
---------------------------
          H @ q
```

```
H @ precise
---------------
H @ subsumed
```

simplify:
notation

```
     H1   H2   H3
---------------------------
          H
```

Gr.eek

$$\frac{t : \mathrm{Top}\{..\ \mathrm{val}\ l\!: T\ ..\}}{t.l : T^{t}}$$

$$\frac{p : \mathrm{Top}\{..\ \mathrm{type}\ L >: S <: U\ ..\}}{S^{p} <: p.L \text{ and } p.L <: U^{p}}$$

$$\frac{T <: T' \quad T <: Tp\{D\} \quad Tp <: \mathrm{Top}\{D_{p}\} \quad G,\ z\!: T \mid\!- (D_{p}^{z} \wedge D^{z}) <: D'^{z}}{T <: T'\{D'\}}$$

quality Gr.eek

$$\frac{t : \mathtt{Top\{..\ val\ l{:}\ T\ ..\}}}{t.l : T^{t}}$$

$$\frac{p : \mathtt{Top\{..\ type\ L >{:}\ S <{:}\ U\ ..\}}}{S^{p} <: p.L \text{ and } p.L <: U^{p}}$$

$$\frac{T <: T' \quad T <: Tp\{D\} \quad Tp <: Top\{D_{p}\} \quad G,\ z{:}\ T \mid\!- (D_{p}^{z} \wedge D^{z}) <: D'^{z}}{T <: T'\{D'\}}$$

$$\frac{T <: S \quad T <: U}{T <: S \wedge U}$$

$$\frac{}{S \wedge T <: S \ @\ \text{subsumed}}$$

Transitivity

- when does `T <: p.L <: U` imply `T <: U` ?
  - when p's type is well-formed*
- WF is not an obvious property
  - e.g., not monotone wrt intersection
    - {L : Bot..Bot} WF and {L : Top..Top} WF
    - {L : Bot $\vee$ Top .. Bot $\wedge$ Top } not WF

Sneaky Middleman

```
             p : Top{L : Top..Bot}    p : Top{L : Top..Bot}
[Sub-Sel-R]---------------------    ---------------------[Sub-Sel-L]
                    Top <: p.L    p.L <: Bot
                   -------------------------[Sub-Trans]
                             Top <: Bot
```

Sneaky Middlemen

- intuition: only need transitivity during preservation; no paths in sight that have type members with vacuous bounds (dealing with values)
- how about lambda/self-bound variables?
  - when preservation “goes under the binder”, it must have a value, of well-formed type, in hand for the abstraction

Punted on proving transitivity

- Given the right side conditions, what could possibly go wrong?
- I have not yet figured out how to make the (mutual) induction go through
  - Must relate subtyping (immediate + inverted from deeper WF) and typing

Plan B

- Prove properties of algorithmic system
- Preservation stated modulo `<:*`

Questions! Thank you!

Besides., working on

- language virtualisation
  - = MOP + lightweight, type-directed staging
  - "virtualising" pattern matching (specify its zero-plus monad)
- type-level computation
  - implicits are poor man's type-level prolog
  - agree with Haskell: stick to type *functions*

One failed attempt

```
  expose p.L S U     expose p.L S' U'
  --------------     ----------------
        S <: p.L     p.L <: U'

NTS: S <: U'

splice: expose p.L S U -> expose p.L S' U' -> expose p.L S U'
decompose into HTyp + HSub + {HSubDecl1, HSubDecl2}
recompose HTyp + HSub + HSubDecl12

invert_expose: expose p.L S U' -> S <: U'
(under suitable side conditions of well-formedness of the
context)
```

Nominality & Soundness

- Axiom “L may label a class if it occurs once in whole program”, seems awkward to me.
- But it is necessary for soundness.
- Alternative?

Nominality & Soundness

- Finding the least structural supertype of `p.L` is only part of the challenge
- Without the nominality axiom, a given `L` may be bound to incompatible types
- Other attack angle: dynamic vs static type of `p`

Rebinding class labels

- virtual class calculi face the same challenge
  - how to check `new p.L` statically?
    - p's dynamic type must be allowed to tighten members, otherwise what's the point of subclassing?
    - but that may render L's bounds vacuous

Exact'ish types

- Easy, radical, solution: `new p.L` only allowed if `p`'s dynamic type is known statically.
- Masked types?
  - `new p.L` allowed if p's static type indicates subtypes may not change `L`'s type
- Virtual classes need something like this