DOT

(Dependent Object Types)

Nada Amin

with

Samuel Grütter Martin Odersky Sandro Stucki Tiark Rompf

LAMP

May 17, 2016

DOT

(Dependent Object Types)

Nada Amin

with

Samuel Grütter Martin Odersky Sandro Stucki Tiark Rompf

LAMP

May 17, 2016

Abstract: In this talk, I'll sketch our soundness results for DOT. I'll go beyond Wadlerfest by discussing subtyping of recursive types, non-ANF & store-less variants, and paths beyond variables.

Why DOT?

- DOT as a type-theoretic foundation:
  - few yet powerful concepts,
    with uniform means of abstraction and combination
    e.g. quantification only over term, yet supports polymorphism
  - "user-extensible" subtyping
  - mixture of nominal and structural
  - nominality is "scoped"
    e.g. no global class table – nice for static analysis?
  - no imposed notion of code sharing
    such as prototype vs class inheritance, mixins, ...
- Impact on Scala/Dotty:
  - characterizing soundness issues,
    e.g. type selection on `Null` or $\bot$ paths
  - suggesting simplifications,
    e.g. a core type system based on DOT
  - lifting ad-hoc restrictions,
    e.g. recursive structural types are more powerful in DOT than in Scala.

Why DOT?

- DOT as a type-theoretic foundation:
  - few yet powerful concepts,
    with uniform means of abstraction and combination
    e.g. quantification only over term, yet supports polymorphism
  - "user-extensible" subtyping
  - mixture of nominal and structural
  - nominality is "scoped"
    e.g. no global class table – nice for static analysis?
  - no imposed notion of code sharing
    such as prototype vs class inheritance, mixins, ...
- Impact on Scala/Dotty:
  - characterizing soundness issues,
    e.g. type selection on `Null` or $\bot$ paths
  - suggesting simplifications,
    e.g. a core type system based on DOT
  - lifting ad-hoc restrictions,
    e.g. recursive structural types are more powerful in DOT than in Scala.

The key aim behind DOT is to build a solid foundation for Scala and similar languages from first principles.

Though the soundness issues were usually known, the work on DOT helps characterizing the flaws and evaluating fixes (ad-hoc vs principled).

Of course, lifting of restrictions should be done cautiously, since one needs to evaluate not just the core calculus but also feature interaction.

DOT: The Essence of Scala

What do you get if you boil Scala on a slow flame and wait until all incidental features evaporate and only the most concentrated essence remains? After doing this for 8 years we believe we have the answer: it's DOT, the calculus of dependent object types, that underlies Scala.

– Martin Odersky

`http://www.scala-lang.org/blog/2016/02/03/essence-of-scala.html`

DOT (Syntax)

$t ::=$		**terms**:
	x	variable
	$\{x \Rightarrow \overline{d}\}$	object
	$t.l$	field. sel.
	$t.m(t)$	meth. app.
$d ::=$		**init.**:
	$l = p$	field mem.
	$m(x : T) = t$	meth. mem.
	$L = T$	type mem.
$v ::=$		**values**:
	$\{x \Rightarrow \overline{d}\}$	object
$p ::=$		**paths**:
	x	variable
	v	value
	$p.l$	field. sel.

$S, T, U ::=$		**types**:
	$\top$	top
	$\bot$	bottom
	$T \wedge T$	intersection
	$T \vee T$	union
	$l : U$	field mem.
	$m(x : S) : U$	meth. mem.
	$L : S..U$	type mem.
	$p.L$	type sel.
	$\{x \Rightarrow T\}$	rec. self

DOT (Syntax)

$t ::=$		**terms**:
	x	variable
	$\{x \Rightarrow \overline{d}\}$	object
	$t.l$	field. sel.
	$t.m(t)$	meth. app.
$d ::=$		**init.**:
	$l = p$	field mem.
	$m(x : T) = t$	meth. mem.
	$L = T$	type mem.
$v ::=$		**values**:
	$\{x \Rightarrow \overline{d}\}$	object
$p ::=$		**paths**:
	x	variable
	v	value
	$p.l$	field. sel.

$S, T, U ::=$		**types**:
	$\top$	top
	$\bot$	bottom
	$T \wedge T$	intersection
	$T \vee T$	union
	$l : U$	field mem.
	$m(x : S) : U$	meth. mem.
	$L : S..U$	type mem.
	$p.L$	type sel.
	$\{x \Rightarrow T\}$	rec. self

In DOT, an object can hold types, as well as fields and methods. These type members can be selected through path-dependent types. In DOT, an object closes over a self, and introduces a recursive type which also closes over a self term. DOT has a full subtyping lattice. Intersection types are also used to type an object with multiple members. Subtyping determines membership.

Deriving DOT

From $F_{<:}$ to DOT

1. lower bound
2. type member and selection
3. subtyping lattice
4. records
5. recursion over self
6. normalization in paths

From $F_{<:}$ to DOT

1. lower bound
2. type member and selection
3. subtyping lattice
4. records
5. recursion over self
6. normalization in paths

Bottom-up exploration of the landscape, survival of the fittest designs and soundness proofs, that scale to variations. Big ideas: invertible value typing in empty context, lenient well-formedness for uniformity and monotonicity, subsumption (almost) everywhere, OK for lattice to collapse in unrealizable context, syntactic rather than semantic checks.

System $F_{<:}$

$$t ::= x \mid \lambda x : T.t \mid t\ t \mid \lambda X <: T.t \mid t\ [T]$$

$$T ::= T \to T \mid \top \mid X \mid \forall X <: T.T$$

- combines System F (polymorphic lambda-calculus) and subtyping
- generalizes universal quantification to upper-bounded quantification
  - example of universal quantification

    `id = λX<:⊤.λx:X.x`
    - `id : ∀X<:⊤.X→X`
  - example of upper-bounded quantification

    `p = λX<:{a:Nat}.λx:X.{orig_x=x, s=succ(x.a)};`
    - `p : ∀X<:{a:Nat}.X → {orig_x:X, s:Nat}`

      `n = (p [{a:Nat,b:Nat}] (a=0,b=0)).orig_x.b`
    - `n: Nat`

System $F_{<:}$

$$t ::= x \mid \lambda x : T.t \mid t\ t \mid \lambda X <: T.t \mid t\ [T]$$

$$T ::= T \to T \mid \top \mid X \mid \forall X <: T.T$$

- combines System F (polymorphic lambda-calculus) and subtyping
- generalizes universal quantification to upper-bounded quantification
  - example of universal quantification

    ```
    id = λX<:⊤.λx:X.x
    ```
    - ```
      id : ∀X<:⊤.X→X
      ```
  - example of upper-bounded quantification

    ```
    p = λX<:{a:Nat}.λx:X.{orig_x=x, s=succ(x.a)};
    ```
    - ```
      p : ∀X<:{a:Nat}.X → {orig_x:X, s:Nat}
      n = (p [{a:Nat,b:Nat}] (a=0,b=0)).orig_x.b
      ```
    - ```
      n: Nat
      ```

System $F_{<:}$ combines System F and subtyping. Polymorphism and subtyping interact via upper-bounded quantification, which generalizes universal quantification. Upper-bounded quantification makes it possible to constrain polymorphism, exploiting the constraints afforded by subtyping (e.g. `succ(x.a)`) and the precision afforded by polymorphism (e.g. `.orig_x.b`).
      ```

Soundness

Theorem (Type-Safety)

If t is a closed well-typed term, $\emptyset \vdash t : T$, then either t is a value or else there is some t' with $t \longrightarrow t'$ and $\emptyset \vdash t' : T$.

Theorem (Preservation)

If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

Theorem (Progress)

If t is a closed well-typed term, then either t is a value or else there is some t' with $t \longrightarrow t'$.

Properties

Narrowing

Substitution

Inversion of Subtyping

Inversion of Value Typing

1. If $\Gamma \vdash \lambda x : S_1.t_2 : T$ and $\Gamma \vdash T <: U_1 \to U_2$,
   then $\Gamma \vdash U_1 <: S_1$ and there is some S_2 such that $\Gamma, x : S_1 \vdash t_2 : S_2$
   and $\Gamma \vdash S_2 <: U_2$.
2. If $\Gamma \vdash \lambda X <: S_1.t_2 : T$ and $\Gamma \vdash T <: \forall X <: U_1.U_2$,
   then $\Gamma \vdash U_1 <: S_1$ and there is some S_2 such that
   $\Gamma, x <: S_1 \vdash t_2 : S_2$ and $\Gamma, X <: U_1 \vdash S_2 <: U_2$.

Canonical Forms

1. If v is a closed value of type $T_1 \to T_2$,
   then v has the form $\lambda x : S_1.t_2$.
2. If v is a closed value of type $\forall X <: T_1.T_2$,
   then v has the form $\lambda X <: S_1.t_2$.

Properties

Narrowing

Substitution

Inversion of Subtyping

Inversion of Value Typing

1. If $\Gamma \vdash \lambda x : S_1.t_2 : T$ and $\Gamma \vdash T <: U_1 \to U_2$,
   then $\Gamma \vdash U_1 <: S_1$ and there is some S_2 such that $\Gamma, x : S_1 \vdash t_2 : S_2$
   and $\Gamma \vdash S_2 <: U_2$.
2. If $\Gamma \vdash \lambda X <: S_1.t_2 : T$ and $\Gamma \vdash T <: \forall X <: U_1.U_2$,
   then $\Gamma \vdash U_1 <: S_1$ and there is some S_2 such that
   $\Gamma, x <: S_1 \vdash t_2 : S_2$ and $\Gamma, X <: U_1 \vdash S_2 <: U_2$.

Canonical Forms

1. If v is a closed value of type $T_1 \to T_2$,
   then v has the form $\lambda x : S_1.t_2$.
2. If v is a closed value of type $\forall X <: T_1.T_2$,
   then v has the form $\lambda X <: S_1.t_2$.

- Narrowing: a subtyping or typing judgment still holds in a context where a type variable is narrowed, i.e. by tightening its upper bound.
- Substitution preserves typing; type substitution preserves subtyping and typing.
- Inversion of subtyping: essentially, we need to pushback any topmost use of transitivity to make it possible to relate the left and right types.
- Inversion of value typing: used in preservation in the substitution cases for term and type applications.
- Canonical forms: used in progress in the substitution cases for term and type applications.

1. Lower Bound

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad \text{(S-TVar)}$$

vs

$$\frac{X : S..U \in \Gamma}{\Gamma \vdash S <: X <: U} \quad \text{(S-TVar)}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad \text{(S-TVar)}$$

vs

$$\frac{X : S..U \in \Gamma}{\Gamma \vdash S <: X <: U} \quad \text{(S-TVar)}$$

Challenge: inversion of subtyping. Tension between narrowing and transitivity pushback. Dilemma: enforce "good" bounds to prevent subtyping collapse? How can we get away with "bad" bounds and subtyping collapse in unrealizable contexts?

System $F_{<:>}$

$$\lambda X <: T.t \text{ becomes } \lambda X : S..U.t$$

$$\forall X <: T.T \text{ becomes } \forall X : S..U.T$$

$\bot$ to recover upper-bounded quantification

- example of lower-bounded quantification:

```
p = λX:{a:Nat,b:Nat}..⊤.λf:X→⊤.{orig=f, r=(f {a=0,b=0})};
```

- `p : ∀X:{a:Nat,b:Nat}..⊤.(X→⊤)→{orig:X→⊤, r:⊤}`

```
pa = p [{a:Nat}] (λx:{a:Nat}. x.a);
```

- `pa : {orig:{a:Nat}→⊤, r:⊤}`

- example of "translucent" quantification:

```
p = λX:{a:Nat,b:Nat}..{a:Nat}.λf:X→X.(f {a=0,b=0}).a
```

- `p : ∀X:{a:Nat,b:Nat}..{a:Nat}.(X → X) → Nat`

```
n = p [{a:Nat}] (λx:{a:Nat}. {a=succ(x.a)})
```

- `n : Nat`

System $F_{<:>}$

$\lambda X <: T.t$ becomes $\lambda X : S..U.t$

$\forall X <: T.T$ becomes $\forall X : S..U.T$

$\bot$ to recover upper-bounded quantification

- example of lower-bounded quantification:

```
p = λX:{a:Nat,b:Nat}..⊤.λf:X→⊤.{orig=f, r=(f {a=0,b=0})};
▸ p : ∀X:{a:Nat,b:Nat}..⊤.(X→⊤)→{orig:X→⊤, r:⊤}
pa = p [{a:Nat}] (λx:{a:Nat}. x.a);
▸ pa : {orig:{a:Nat}→⊤, r:⊤}
```

- example of "translucent" quantification:

```
p = λX:{a:Nat,b:Nat}..{a:Nat}.λf:X→X.(f {a=0,b=0}).a
▸ p : ∀X:{a:Nat,b:Nat}..{a:Nat}.(X → X) → Nat
n = p [{a:Nat}] (λx:{a:Nat}. {a=succ(x.a)})
▸ n : Nat
```

Translucency means that we can constrain polymorphism, and exploit those constraints in the implementation. Through existential types, translucency means that we can choose how much implementation details to reveal. Thanks to the LB on the type variable `X`, it is possible to call `f {a=0,b=0}`. (1) Thanks to polymorphism, the `orig` field keeps the more specific type of `f`, i.e. a function type with fewer requirements on the parameter type. So `pa.orig` can be applied to records that do not have a `b` field. (2) Thanks to the UB on `X`, we can select the field `a` on the result of `X`, because we know that the type `X` has at least such a field.

Dealing with "bad" bounds

- Restrict Preservation to $\Gamma = \emptyset$.
  If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.
- Define invertible value typing, aka "possible types": $v :: T$.
  1. $v :: \top$.
  2. If $x : S_1 \vdash t_1 : T_1$ and $\emptyset \vdash S_2 <: S_1, T_1 <: T_2$ then $\lambda x : S_1.t_1 :: S_2 \to T_2$.
  3. If $X : S_1..U_1 \vdash t_1 : T_1$ and $\emptyset \vdash S_1 <: S_2, U_2 <: U_1$ and $X : S_2..U_2 \vdash T_1 <: T_2$ then $\lambda X : S_1..U_1.t_1 :: \forall X : S_2..U_2.T_2$.
- Prove subtyping closure aka widening of possible types.
  If $v :: T$ and $\emptyset \vdash T <: U$ then $v :: U$.
- Prove value typing implies possible types.
  If $\emptyset \vdash v : T$ then $v :: T$.
- Prove inversion of value typing and canonical forms via (direct) inversion of possible types.

Dealing with "bad" bounds

- Restrict Preservation to $\Gamma = \emptyset$.
  If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.
- Define invertible value typing, aka "possible types": $v :: T$.
  1. $v :: \top$.
  2. If $x : S_1 \vdash t_1 : T_1$ and $\emptyset \vdash S_2 <: S_1, T_1 <: T_2$ then $\lambda x : S_1.t_1 :: S_2 \to T_2$.
  3. If $X : S_1..U_1 \vdash t_1 : T_1$ and $\emptyset \vdash S_1 <: S_2, U_2 <: U_1$ and $X : S_2..U_2 \vdash T_1 <: T_2$ then $\lambda X : S_1..U_1.t_1 :: \forall X : S_2..U_2.T_2$.
- Prove subtyping closure aka widening of possible types.
  If $v :: T$ and $\emptyset \vdash T <: U$ then $v :: U$.
- Prove value typing implies possible types.
  If $\emptyset \vdash v : T$ then $v :: T$.
- Prove inversion of value typing and canonical forms via (direct) inversion of possible types.

How can we avoid inverting subtyping, and restricting inversion to empty context? Solution: directly invertible value typing. One case per "possible type".

Note (contra-)variance for $\forall$ possible type. Supertype has *tighter* bounds, since it's the subtype that can do more, i.e. be applied to more types.

Note that there is no case for type variable as a type in possible types, because it does not occur in empty context. Similarly, closure is easy.

2. System D: $D_{<:}$, $D_{<:>}$

$$t ::= x \mid \lambda x : T.t \mid t\ t \mid \{L = T\}$$

$$T ::= \top \mid \bot \mid \forall x : S.T \mid \{L : S..U\} \mid p.L$$

- System D unifies term and type abstraction.
- A term can hold a type: a term $\{L = T\}$ introduces a type $\{L : S..U\}$.
- Path-dependent type: $p.L$ is a type that depends on some term p.
- What terms are paths p?
  Here, only normal forms (variables or values).

$$p ::= x \mid v$$

$$v ::= \lambda x : T.t \mid \{L = T\}$$

- λ-values are paths???

2. System D: $D_{<:}$, $D_{<:>}$

$$t ::= x \mid \lambda x : T.t \mid t\ t \mid \{L = T\}$$

$$T ::= \top \mid \bot \mid \forall x : S.T \mid \{L : S..U\} \mid p.L$$

- System D unifies term and type abstraction.
- A term can hold a type: a term $\{L = T\}$ introduces a type $\{L : S..U\}$.
- Path-dependent type: $p.L$ is a type that depends on some term p.
- What terms are paths p?
  Here, only normal forms (variables or values).

$$p ::= x \mid v$$

$$v ::= \lambda x : T.t \mid \{L = T\}$$

- λ-values are paths???

Can start with $F_{<:}$ or $F_{<:>}$. In any case, needs at least type aliases, so nicer to state in terms of $F_{<:>}$.
The elimination of $\{L : S..U\}$ is at the type level, via a type selection or path-dependent type.
$D_{<:}/D_{<:>}$ encode $F_{<:}/F_{<:>}$. Function types are now dependent.
Substitution must preserve syntactic validity of types, hence, it's more uniform to just allow any value as a path. Of course, it's reasonable to have a surface type checker that is more strict than the typing used for soundness.

Subtyping of Type Selections aka Path-Dependent Types

$$\frac{\Gamma \vdash p : \{L : S..U\}}{\Gamma \vdash S <: p.L <: U} \quad \text{(S-TSel)}$$

$$\Gamma \vdash T <: \{L = T\}.L <: T \quad \text{(S-TSel-Tight)}$$

- Define subtyping generally (non-tight), so that substitution is easier: no need for narrowing while substituting a value.
- Define tight subtyping for "possible types". Prove widening of "possible types". For lambda values, delegate to regular subtyping for non-empty context and also define "shallow" variant that does not care about lambda values beyond shape.
- Prove tight $\equiv$ general subtyping in empty context. Use shallow variant of possible types for inverting path typing in subtyping type selections.
- Now adjusted back to $\mathsf{F}_{<:>}$ proof strategy.

Subtyping of Type Selections aka Path-Dependent Types

$$\frac{\Gamma \vdash p : \{L : S..U\}}{\Gamma \vdash S <: p.L <: U} \quad \text{(S-TSel)}$$

$$\Gamma \vdash T <: \{L = T\}.L <: T \quad \text{(S-TSel-Tight)}$$

- Define subtyping generally (non-tight), so that substitution is easier: no need for narrowing while substituting a value.
- Define tight subtyping for "possible types". Prove widening of "possible types". For lambda values, delegate to regular subtyping for non-empty context and also define "shallow" variant that does not care about lambda values beyond shape.
- Prove tight $\equiv$ general subtyping in empty context. Use shallow variant of possible types for inverting path typing in subtyping type selections.
- Now adjusted back to $F_{<:>}$ proof strategy.

Tight selection is natural for "type tag" values, and it should suffice thanks to subtyping transitivity. However, it complicates substitution because for variables, it's natural to allow subsumption on the path, even just to get to the right type shape (a type of "type tag").
Solution: define subtyping generally, and also a tight version for "possible types". The shallow "possible types" does not deeply check abstraction, because we don't care so much about abstraction in type selections.

Back to $F_{<:>}$: Prove subtyping closure of, and value typing implies, and inversions using, "possible types".

3. Full Subtyping Lattice

$$\Gamma \vdash \bot <: T \quad \text{(Bot)} \qquad \Gamma \vdash T <: \top \quad \text{(Top)}$$

$$\frac{\Gamma \vdash T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{(And11)} \qquad \frac{\Gamma \vdash T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2} \text{(Or21)}$$

$$\frac{\Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{(And12)} \qquad \frac{\Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2} \text{(Or22)}$$

$$\frac{\Gamma \vdash T <: T_1 \ , \ T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2} \text{(And2)} \qquad \frac{\Gamma \vdash T_1 <: T \ , \ T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T} \text{(Or1)}$$

$$\Gamma \vdash \bot <: T \quad \text{(Bot)} \qquad \Gamma \vdash T <: \top \quad \text{(Top)}$$

$$\frac{\Gamma \vdash T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{(And11)} \qquad \frac{\Gamma \vdash T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2} \text{(Or21)}$$

$$\frac{\Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{(And12)} \qquad \frac{\Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2} \text{(Or22)}$$

$$\frac{\Gamma \vdash T <: T_1 \ , \ T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2} \text{(And2)} \qquad \frac{\Gamma \vdash T_1 <: T \ , \ T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T} \text{(Or1)}$$

Lattice is easy, because we're lenient about "bad" bounds.

4. Records, typed via Intersection Types

$t ::=$		**terms**:
	x	variable
	$\{\overline{d}\}$	record
	$t.m(t)$	meth. app.
$d ::=$		**init.**:
	$m(x : T) = t$	meth. mem.
	$L = T$	type mem.

$m(x : S) : U$	meth. mem.
$L : S..U$	type mem.

- A record $\{d_1, \ldots, d_n\}$ has type $T_1 \wedge \ldots \wedge T_n$.

$t ::=$		**terms**:	
	x	variable	
	$\{\overline{d}\}$	record	
	$t.m(t)$	meth. app.	
$d ::=$		**init.**:	
	$m(x : T) = t$	meth. mem.	$m(x : S) : U$ meth. mem.
	$L = T$	type mem.	$L : S..U$ type mem.

- A record $\{d_1, \ldots, d_n\}$ has type $T_1 \wedge \ldots \wedge T_n$.

Now, that we have intersection types, we can use them to type records. Economy of concept! We also just use subtyping to judge membership. In this case, we also consolidate values (abstraction or type tags) into method and type members.

Of course, that's not so interesting without recursion!

5. Recursion: From Records to Objects

- An object is a record which closes over a self variable z: $\{z \Rightarrow \overline{d}\}$.
- An object introduces a recursive type: $\{z \Rightarrow T\}$.

$$\frac{\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash d_i : T_i \qquad \forall i, 1 \leq i \leq n}{\Gamma \vdash \{x \Rightarrow d_1 \ldots d_n\} : \{x \Rightarrow T_1 \wedge \ldots \wedge T_n\}} \text{ (TNew)}$$

(labels disjoint)

- Store to keep track of object identities?
- Recursive types bring lots of power: F-bounded abstraction and beyond, non-termination, nominality through type abstraction, etc.

- An object is a record which closes over a self variable z: $\{z \Rightarrow \overline{d}\}$.
- An object introduces a recursive type: $\{z \Rightarrow T\}$.

$$\frac{\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash d_i : T_i \quad \overset{\text{(labels disjoint)}}{\forall i, 1 \leq i \leq n}}{\Gamma \vdash \{x \Rightarrow d_1 \ldots d_n\} : \{x \Rightarrow T_1 \wedge \ldots \wedge T_n\}} \quad \text{(TNew)}$$

- Store to keep track of object identities?
- Recursive types bring lots of power: F-bounded abstraction and beyond, non-termination, nominality through type abstraction, etc.

A recursive type also closes over a self *term*. Once more, in DOT, all quantification is over terms, not types. We type an object *without* subsumption, then close over the recursive type. This is important for soundness.

Typing and Subtyping of Recursive Types

Type assignment $\boxed{\Gamma \vdash t :_{(!)} T}$

$$\frac{\Gamma \vdash p : [z \mapsto p]T}{\Gamma \vdash p : \{z \Rightarrow T\}} \quad \text{(Pack)}$$

$$\frac{\Gamma \vdash p :_{(!)} \{z \Rightarrow T\}}{\Gamma \vdash p :_{(!)} [z \mapsto p]T} \quad \text{(Unpack)}$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\frac{\Gamma, z : T_1 \vdash T_1 <: T_2}{\Gamma \vdash \{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\}} \quad \text{(Bind)}$$

$$\frac{\Gamma, z : T_1 \vdash T_1 <: T_2 \qquad z \notin fv(T_2)}{\Gamma \vdash \{z \Rightarrow T_1\} <: T_2} \quad \text{(Bind1)}$$

Typing and Subtyping of Recursive Types

Type assignment $\boxed{\Gamma \vdash t :_{(!)} T}$

$$\frac{\Gamma \vdash p : [z \mapsto p]T}{\Gamma \vdash p : \{z \Rightarrow T\}} \quad \text{(Pack)}$$

$$\frac{\Gamma \vdash p :_{(!)} \{z \Rightarrow T\}}{\Gamma \vdash p :_{(!)} [z \mapsto p]T} \quad \text{(Unpack)}$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\frac{\Gamma, z : T_1 \vdash T_1 <: T_2}{\Gamma \vdash \{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\}} \quad \text{(Bind)}$$

$$\frac{\Gamma, z : T_1 \vdash T_1 <: T_2 \quad z \notin fv(T_2)}{\Gamma \vdash \{z \Rightarrow T_1\} <: T_2} \quad \text{(Bind1)}$$

In typing, we have packing and unpacking, to introduce and eliminate a recursive self type.

In subtyping, we can compare two recursive types, but also compare a recursive type with another type without a self.

Notice that the typing rule comes in two variant $:$ and $:_!$. The second variant is used in typing paths during subtyping of type selections. In particular, this variant for subtyping disallows packing.

Restrictions in Type Selections of Abstract Variables

$$\frac{\Gamma_{[x]} \vdash x :_! (L : T..\top)}{\Gamma \vdash T <: x.L} \quad \text{(Sel2)}$$

$$\frac{\Gamma_{[x]} \vdash x :_! (L : \bot..T)}{\Gamma \vdash x.L <: T} \quad \text{(Sel1)}$$

- To prove tight $\equiv$ non-tight subtyping in empty context, we need substitution because of type selection on recursive types. But if we use substitution, we cannot use the IH.
- With the restriction on Γ, we can use tight subtyping – on values only – from the outset.
- Restrict substitution so that substituted variable is first in context, i.e. $\Gamma' = \emptyset$.
  If $\Gamma', x : U, \Gamma \vdash t : T$ and $\Gamma' \vdash v : U$, then
  $\Gamma', [x \mapsto v]\Gamma \vdash [x \mapsto v]t : [x \mapsto v]T$

Restrictions in Type Selections of Abstract Variables

$$\frac{\Gamma_{[x]} \vdash x :_! (L : T..\top)}{\Gamma \vdash T <: x.L} \quad \text{(Sel2)}$$

$$\frac{\Gamma_{[x]} \vdash x :_! (L : \bot..T)}{\Gamma \vdash x.L <: T} \quad \text{(Sel1)}$$

- To prove tight $\equiv$ non-tight subtyping in empty context, we need substitution because of type selection on recursive types. But if we use substitution, we cannot use the IH.
- With the restriction on Γ, we can use tight subtyping – on values only – from the outset.
- Restrict substitution so that substituted variable is first in context, i.e. $\Gamma' = \emptyset$.
  If $\Gamma', x : U, \Gamma \vdash t : T$ and $\Gamma' \vdash v : U$, then
  $\Gamma', [x \mapsto v]\Gamma \vdash [x \mapsto v]t : [x \mapsto v]T$

In fact, there is another restriction in type selections. This is due to a cycle in the proofs, though it's not clear yet whether it is necessary for soundness. Probably not, because it seems like we can always use subtyping transitivity afterwards to get more or less what we could get directly without the restriction.

Possible Types (Base Cases)

$$v :: \top \qquad \text{(V-Top)}$$

$$\frac{(L = T) \in \overline{[x \mapsto \{x \Rightarrow \overline{d}\}]d} \qquad \emptyset \vdash S <: T \ , \ T <: U}{\{x \Rightarrow \overline{d}\} :: (L : S..U)} \qquad \text{(V-Typ)}$$

$$\frac{\begin{array}{c}\text{(labels disjoint)} \qquad \forall i, 1 \leq i \leq n \\ \emptyset, (x : T_1 \wedge \ldots \wedge T_n) \vdash d_i : T_i \\ \exists j, \ [x \mapsto \{x \Rightarrow \overline{d}\}]d_j = (m(z : S) = t) \\ [x \mapsto \{x \Rightarrow \overline{d}\}]T_j = (m(z : S) : U) \\ \emptyset \vdash S' <: S \qquad \emptyset, (z : S') \vdash U <: U'\end{array}}{\{x \Rightarrow \overline{d}\} :: (m(x : S') : U')} \qquad \text{(V-Fun)}$$

Possible Types (Inductive Cases)

$$\frac{v :: T \qquad (L = T) \in \overline{[x \mapsto \{x \Rightarrow \overline{d}\}]d}}{v :: (\{x \Rightarrow \overline{d}\}.L)} \quad \text{(V-Sel)}$$

$$\frac{v :: [x \mapsto v]T}{v :: \{x \Rightarrow T\}} \quad \text{(V-Bind)}$$

$$\frac{v :: T_1 \qquad v :: T_2}{v :: T_1 \wedge T_2} \quad \text{(V-And)}$$

$$\frac{v :: T_1}{v :: T_1 \vee T_2} \quad \text{(V-Or1)}$$

$$\frac{v :: T_2}{v :: T_1 \vee T_2} \quad \text{(V-Or2)}$$

Proof Sketch

- Let $v ::_m T$ denote a derivation of $v :: T$ with no more than m uses of (V-Bind).
- Let Widen_m denote the assumption that $v ::_m T$ can be widened: If $v ::_m T$ and $\emptyset \vdash T <: U$ then $v ::_m U$.
- Prove some substitution lemmas assuming widening.
  1. If $v ::_m T$ and Widen_m and $x : T, \Gamma \vdash S <: U$, then $[x \mapsto v]\Gamma \vdash [x \mapsto v]S <: [x \mapsto v]U$.
  2. If $v ::_m T$ and Widen_m and $x \neq z$ and $x : T, \Gamma \vdash z :_! T$, then $[x \mapsto v]\Gamma \vdash z :_! [x \mapsto v]T$.
  3. If $v ::_m T$ and Widen_m and $x : T, \Gamma \vdash x :_! U$, then $v ::_m [x \mapsto v]U$.
- Prove widening: $\forall m.\text{Widen}_m$.
- Prove empty-context value typing implies "possible types": If $\emptyset \vdash v : T$ then $v :: T$.

Proof Sketch

- Let $v ::_m T$ denote a derivation of $v :: T$ with no more than m uses of (V-Bind).
- Let Widen_m denote the assumption that $v ::_m T$ can be widened: If $v ::_m T$ and $\emptyset \vdash T <: U$ then $v ::_m U$.
- Prove some substitution lemmas assuming widening.
  1. If $v ::_m T$ and Widen_m and $x : T, \Gamma \vdash S <: U$, then $[x \mapsto v]\Gamma \vdash [x \mapsto v]S <: [x \mapsto v]U$.
  2. If $v ::_m T$ and Widen_m and $x \neq z$ and $x : T, \Gamma \vdash z :_! T$, then $[x \mapsto v]\Gamma \vdash z :_! [x \mapsto v]T$.
  3. If $v ::_m T$ and Widen_m and $x : T, \Gamma \vdash x :_! U$, then $v ::_m [x \mapsto v]U$.
- Prove widening: $\forall m.\text{Widen}_m$.
- Prove empty-context value typing implies "possible types": If $\emptyset \vdash v : T$ then $v :: T$.

The main challenge is to orchestrate the induction metrics. Bootstrap mutual induction between substitution and widening of value typing. Outer induction on number of packing uses, inner induction on size of derivation. Note that in type selections, we use tight typing for values and only need $\Gamma \vdash x :: T$ for type selections involving abstract variables. Substitution lemmas: (1) for subtyping, (2) for abstract paths, (3) for turning abstract path into concrete value typing.

6. Beyond Normal Paths

- Relating paths across reduction steps?
- Use evaluation/normalization of paths to just relate values.
- Properties:

uniqueness	If $p \Downarrow v_1$ and $p \Downarrow v_2$ then $v_1 = v_2$.
confluence	If $p \longrightarrow p'$ and $p \Downarrow v$ then $p' \Downarrow v$.
strong normalization	If $\emptyset \vdash p :_! T$ then there is some v with $p \Downarrow v$.
preservation	If $\emptyset \vdash p :_! T$ and $p \Downarrow v$ then $v :: T$.

- The approach works well for simple paths, i.e. immutable fields of chain selections.
- For *application* in paths, work-in-progress. Once more, the issue is bootstrapping the lemmas given *substitution* in paths.

6. Beyond Normal Paths

- Relating paths across reduction steps?
- Use evaluation/normalization of paths to just relate values.
- Properties:

uniqueness	If $p \Downarrow v_1$ and $p \Downarrow v_2$ then $v_1 = v_2$.
confluence	If $p \longrightarrow p'$ and $p \Downarrow v$ then $p' \Downarrow v$.
strong normalization	If $\emptyset \vdash p :_! T$ then there is some v with $p \Downarrow v$.
preservation	If $\emptyset \vdash p :_! T$ and $p \Downarrow v$ then $v :: T$.

- The approach works well for simple paths, i.e. immutable fields of chain selections.
- For *application* in paths, work-in-progress. Once more, the issue is bootstrapping the lemmas given *substitution* in paths.

This was mechanized in big-step setting. However, with the storeless variant of DOT is should work fine in a small-step setting as well, since values are comparable. Open question: how to modify "possible types"? Properties required are intuitive; however, they cannot be taken for granted given non-termination & mutation.

Need to show mini-safety theorem for path evaluation.

Conclusion

- Deriving DOT: $F_{<:}$, $F_{<:>}$, $D_{<:}$, $D_{<:>}$, Lattice, Records, Objects, ...
- A bottom-up exploration:
  - + interesting intermediary points in the landscape
  - survival of the fittest designs and proofs
    - + survival bias means design and proof are quite robust to variations...
    - – ...but also stuck in local sweet spots
    - = lots of time spent on dead ends
- Sound DOT design "discovered" rather than invented.

Conclusion

- Deriving DOT: $F_{<:}$, $F_{<:>}$, $D_{<:}$, $D_{<:>}$, Lattice, Records, Objects, ...
- A bottom-up exploration:
  - + interesting intermediary points in the landscape
  - survival of the fittest designs and proofs
    - + survival bias means design and proof are quite robust to variations...
    - – ...but also stuck in local sweet spots
    - = lots of time spent on dead ends
- Sound DOT design "discovered" rather than invented.

Convey not just that a calculus is sound in isolation, but also what assumptions the soundness proof relies.
Our proof relies on runtime values having only type members with good bounds, which the syntax enforces. Because of recursive types, such a property would be difficult to enforce semantically. It also relies on call-by-value semantics, in that it expects all variables that can partake in types to point to runtime values when a method body is evaluated.
The process of designing the calculus and proving it sound have been intertwined. As we understood the landscape better, we have been able to make the model more uniform yet powerful.

Bonus

DOT: Some Unsound Variations

- Add subsumption to member initialization.

$$\frac{\Gamma \vdash d : T \quad \Gamma \vdash T <: U}{\Gamma \vdash d : U} \qquad \text{(DSub)}$$

$$\{x \Rightarrow L = \top\} : \{x \Rightarrow L : \top..\bot\}$$

- Change type member initialization from $\{L = T\}$ to $\{L : S..U\}$.

$$\frac{\Gamma \vdash S <: U}{\{L : S..U\} : \{L : S..U\}} \qquad \text{(DTyp)}$$

$$\{x \Rightarrow L : \top..\bot\} : \{x \Rightarrow L : \top..\bot\}$$

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 subsume/abstract, if desired.
- Inversion lemmas need only hold in empty abstract environment.
- Tension between preservation and abstraction. Rely on precise types for runtime values.

Unsoundness in Scala (fits in a Tweet)

```
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);
  }
}
```

Formal Model

Formal Model

This is a formal model for DOT at step 5 (including recursive subtyping, but excluding fields and full paths).

DOT Syntax

$t ::=$		**terms**:
	x	variable
	$\{z \Rightarrow \overline{d}\}$	object
	$t.m(t)$	meth. app.
$d ::=$		**init.**:
	$L = T$	type mem.
	$m(x : T) = t$	meth. mem.
$v ::=$		**values**:
	$\{z \Rightarrow \overline{d}\}$	object
$p ::=$		**paths**:
	x	variable
	v	value

$S, T, U ::=$		**types**:
	$\top$	top
	$\bot$	bot.
	$T \wedge T$	inter.
	$T \vee T$	union
	$L : S..U$	type mem.
	$m(x : S) : U$	meth. mem.
	$p.L$	sel.
	$\{z \Rightarrow T\}$	rec. sel.
$\Gamma ::=$		**contexts**:
	$\emptyset \mid \Gamma, x : T$	var. bind.

DOT Subtyping $\Gamma \vdash S <: U$

Lattice structure

$$\Gamma \vdash \bot <: T \quad \text{(Bot)} \qquad \Gamma \vdash T <: \top \quad \text{(Top)}$$

$$\frac{\Gamma \vdash T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{(And11)} \qquad \frac{\Gamma \vdash T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2} \text{(Or21)}$$

$$\frac{\Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{(And12)} \qquad \frac{\Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2} \text{(Or22)}$$

$$\frac{\Gamma \vdash T <: T_1 \ , \ T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2} \text{(And2)} \qquad \frac{\Gamma \vdash T_1 <: T \ , \ T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T} \text{(Or1)}$$

Properties

$$\Gamma \vdash T <: T \quad \text{(Refl)} \qquad \frac{\Gamma \vdash T_1 <: T_2 \ , \ T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \text{(Trans)}$$

DOT Subtyping $\boxed{\Gamma \vdash S <: U}$

Method and type members

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash U_1 <: U_2}{\Gamma \vdash m(x : S_1) : U_1 <: m(x : S_2) : U_2} \text{(Fun)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \ , \ U_1 <: U_2}{\Gamma \vdash L : S_1..U_1 <: L : S_2..U_2} \text{(Typ)}$$

Type selections

$$\frac{\Gamma_{[x]} \vdash x :_! (L : T..\top)}{\Gamma \vdash T <: x.L} \text{(Sel2)} \qquad \frac{[z \mapsto \overline{d}]\overline{d} \ni L = T}{\Gamma \vdash T <: \{z \Rightarrow \overline{d}\}.L} \text{(SSel2)}$$

$$\frac{\Gamma_{[x]} \vdash x :_! (L : \bot..T)}{\Gamma \vdash x.L <: T} \text{(Sel1)} \qquad \frac{[z \mapsto \overline{d}]\overline{d} \ni L = T}{\Gamma \vdash \{z \Rightarrow \overline{d}\}.L <: T} \text{(SSel1)}$$

DOT Subtyping $\boxed{\Gamma \vdash S <: U}$

Recursive self types

$$\frac{\Gamma, z : T_1 \vdash T_1 <: T_2}{\Gamma \vdash \{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\}} \quad \text{(BindX)}$$

$$\frac{\Gamma, z : T_1 \vdash T_1 <: T_2 \qquad z \notin fv(T_2)}{\Gamma \vdash \{z \Rightarrow T_1\} <: T_2} \quad \text{(Bind1)}$$

DOT Typing $\boxed{\Gamma \vdash t :_{(!)} T}$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x :_{(!)} T} \quad \text{(Var)} \qquad \frac{\Gamma \vdash t :_{(!)} T_1 \ , \ T_1 <: T_2}{\Gamma \vdash t :_{(!)} T_2} \quad \text{(Sub)}$$

$$\frac{\Gamma \vdash p : [z \mapsto p]T}{\Gamma \vdash p : \{z \Rightarrow T\}} \quad \text{(Pack)} \qquad \frac{\Gamma \vdash p :_{(!)} \{z \Rightarrow T\}}{\Gamma \vdash p :_{(!)} [z \mapsto p]T} \quad \text{(Unpack)}$$

DOT Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t : (m(x : T_1) : T_2)\ ,\ t_2 : T_1 \qquad x \notin fv(T_2)}{\Gamma \vdash t.m(t_2) : T_2} \quad \text{(TApp)}$$

$$\frac{\Gamma \vdash t : (m(x : T_1) : T_2)\ ,\ p : T_1}{\Gamma \vdash t.m(p) : [x \mapsto p]\,T_2} \quad \text{(TAppDep)}$$

(labels disjoint)

$$\frac{\Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash d_i : T_i \qquad \forall i, 1 \leq i \leq n}{\Gamma \vdash \{x \Rightarrow d_1 \ldots d_n\} : [x \mapsto \{x \Rightarrow d_1 \ldots d_n\}](T_1 \wedge \ldots \wedge T_n)} \quad \text{(TObj)}$$

DOT Member Initialization $\boxed{\Gamma \vdash d : T}$

$$\frac{\Gamma \vdash T <: T}{\Gamma \vdash (L = T) : (L : T..T)} \quad \text{(DTyp)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (m(x) = t) : (m(x : T_1) : T_2)} \quad \text{(DFun)}$$

DOT Small-Step Operational Semantics $\boxed{t \longrightarrow t'}$

$$\frac{[z \mapsto \overline{d}]\overline{d} \ni m(x : T_{11}) = t_{12}}{\{z \Rightarrow \overline{d}\}.m(v_2) \longrightarrow [x \mapsto v_2]t_{12}} \quad \text{(E-App)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.m(t_2) \longrightarrow t_1'.m(t_2)} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1.m(t_2) \longrightarrow v_1.m(t_2')} \quad \text{(E-App2)}$$