Type Soundness Proofs with Definitional Interpreters: From $F_{<:}$ to Scala

Tiark Rompf (Purdue), Nada Amin (EPFL)

October 3, 2015

IBM

twitter

LinkedIn

SONY make.believe

imageworks

SIEMENS

xerox

Novell

outside.in

EDF

NASA

foursquare

TOMTOM

amazon

UBS

Office DEPOT Taking Care of Business

guardian.co.uk

OPOWER

Bank of America

yammer

Thatcham

Autodesk

CREDIT SUISSE

- Formal description of the Scala type system?
- Type soundness proof?

People have tried ...

- 2003: νObj
- 2006: Featherweight Scala
- 2008: Scalina
- 2008-2010: ScalaClassic
- 2012: DOT (Dependent Object Types)

No mechanized soundness proof.

People have tried ...

- 2003: νObj
- 2006: Featherweight Scala
- 2008: Scalina
- 2008-2010: ScalaClassic
- 2012: DOT (Dependent Object Types)

- 2014: μDOT
- 2015: D2

Why is (was) it so hard?

- Scala is a rich language
- We haven't fully understood some of its core features
- Maybe we aren't doing this right ...

DOT (FOOL'12)

Syntax

x, y, z	Variable	$L ::=$	Type label
l	Value label	L_c	class label
m	Method label	L_a	abstract type label
$v ::=$	Value	$S, T, U, V, W ::=$	Type
x	variable	$p.L$	type selection
$t ::=$	Term	$T\{z \Rightarrow \overline{D}\}$	refinement
v	value	$T \wedge T$	intersection type
$\textbf{val}\ x = \textbf{new}\ c;\ t$	new instance	$T \vee T$	union type
$t.l$	field selection	$\top$	top type
$t.m(t)$	method invocation	$\bot$	bottom type
$p ::=$	Path	$S_c, T_c ::=$	Concrete type
x	variable	$p.L_c \mid T_c\{z \Rightarrow \overline{D}\} \mid T_c \wedge T_c \mid \top$	
$p.l$	selection	$D ::=$	Declaration
$c ::= T_c\{\overline{d}\}$	Constructor	$L : S..U$	type declaration
$d ::=$	Initialization	$l : T$	value declaration
$l = v$	field initialization	$m : S \to U$	method declaration
$m(x) = t$	method initialization		
$s ::= \overline{x \mapsto c}$	Store	$\Gamma ::= \overline{x : T}$	Environment

Reduction $\boxed{t \mid s \ \to\ t' \mid s'}$

$$\frac{y \mapsto T_c\left\{\overline{l = v'}\ \overline{m(x) = t}\right\} \in s}{y.m_i(v) \mid s \ \to\ [v/x_i]t_i \mid s}\ \text{(MSEL)} \qquad \textbf{val}\ x = \textbf{new}\ c;\ t \mid s \ \to\ t \mid s, x \mapsto c\ \text{(NEW)}$$

$$\frac{y \mapsto T_c\left\{\overline{l = v}\ \overline{m(x) = t}\right\} \in s}{y.l_i \mid s \ \to\ v_i \mid s}\ \text{(SEL)} \qquad \frac{t \mid s \ \to\ t' \mid s'}{e[t] \mid s \ \to\ e[t'] \mid s'}\ \text{(CONTEXT)}$$

where evaluation context $e ::= [\,] \mid e.m(t) \mid v.m(e) \mid e.l$

Type Assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(VAR)} \qquad \frac{\Gamma \vdash t \ni l : T'}{\Gamma \vdash t.l : T'}\ \text{(SEL)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t \ni m : S \to T \\ \Gamma \vdash t' : T' ,\ T' <: S\end{array}}{\Gamma \vdash t.m(t') : T}\ \text{(MSEL)} \qquad \frac{\begin{array}{c} y \notin \mathsf{fn}(T') \\ \Gamma \vdash T_c\ \textbf{wfe} ,\ T_c \prec_y \overline{L : S..U}, \overline{D} \\ \Gamma, y : T_c \vdash \overline{S <: U} ,\ \overline{d} : \overline{D} ,\ t' : T'\end{array}}{\Gamma \vdash \textbf{val}\ y = \textbf{new}\ T_c\{\overline{d}\};\ t' : T'}\ \text{(NEW)}$$

Declaration Assignment $\boxed{\Gamma \vdash d : D}$

$$\frac{\Gamma \vdash v : V' ,\ V' <: V}{\Gamma \vdash (l = v) : (l : V)}\ \text{(VDECL)} \qquad \frac{\begin{array}{c}\Gamma \vdash S\ \textbf{wfe} \\ \Gamma, x : S \vdash t : T' ,\ T' <: T\end{array}}{\Gamma \vdash (m(x) = t) : (m : S \to T)}\ \text{(MDECL)}$$

Figure 1. The DOT Calculus : Syntax, Reduction, Type / Declaration Assignment

Well-formed types $\boxed{\Gamma \vdash T\ \textbf{wf}}$

$$\frac{\begin{array}{c}\Gamma \vdash T\ \textbf{wf} \\ \Gamma, z : T\{z \Rightarrow \overline{D}\} \vdash \overline{D\ \textbf{wf}}\end{array}}{\Gamma \vdash T\{z \Rightarrow \overline{D}\}\ \textbf{wf}}\ \text{(RFN-WF)} \qquad \Gamma \vdash \top\ \textbf{wf}\ (\top\text{-WF}) \qquad \Gamma \vdash \bot\ \textbf{wf}\ (\bot\text{-WF})$$

$$\frac{\Gamma \vdash p \ni L : S..U ,\ S\ \textbf{wf} ,\ U\ \textbf{wf}}{\Gamma \vdash p.L\ \textbf{wf}}\ (\text{TSEL-WF}_1) \qquad \frac{\Gamma \vdash p \ni L : \bot..U}{\Gamma \vdash p.L\ \textbf{wf}}\ (\text{TSEL-WF}_2)$$

$$\frac{\Gamma \vdash T\ \textbf{wf} ,\ T'\ \textbf{wf}}{\Gamma \vdash T \wedge T'\ \textbf{wf}}\ (\wedge\text{-WF}) \qquad \frac{\Gamma \vdash T\ \textbf{wf} ,\ T'\ \textbf{wf}}{\Gamma \vdash T \vee T'\ \textbf{wf}}\ (\vee\text{-WF})$$

Well-formed declarations $\boxed{\Gamma \vdash D\ \textbf{wf}}$

$$\frac{\Gamma \vdash S\ \textbf{wf} ,\ U\ \textbf{wf}}{\Gamma \vdash L : S..U\ \textbf{wf}}\ \text{(TDECL-WF)} \qquad \frac{\Gamma \vdash S\ \textbf{wf} ,\ U\ \textbf{wf}}{\Gamma \vdash m : S \to U\ \textbf{wf}}\ \text{(MDECL-WF)}$$

$$\frac{\Gamma \vdash T\ \textbf{wf}}{\Gamma \vdash l : T\ \textbf{wf}}\ \text{(VDECL-WF)}$$

$$\begin{array}{lcll}
\mathrm{dom}(\overline{D} \wedge \overline{D'}) & = & \mathrm{dom}(\overline{D}) \cup \mathrm{dom}(\overline{D'}) & \\
\mathrm{dom}(\overline{D} \vee \overline{D'}) & = & \mathrm{dom}(\overline{D}) \cap \mathrm{dom}(\overline{D'}) & \\
(D \wedge D')(L) & = & L : (S \vee S')..(U \wedge U') & \text{if } (L : S..U) \in \overline{D} \text{ and } (L : S'..U') \in \overline{D'} \\
 & = & D(L) & \text{if } L \notin \mathrm{dom}(\overline{D'}) \\
 & = & D'(L) & \text{if } L \notin \mathrm{dom}(\overline{D}) \\
(D \wedge D')(m) & = & m : (S \vee S') \to (U \wedge U') & \text{if } (m : S \to U) \in \overline{D} \text{ and } (m : S' \to U') \in \overline{D'} \\
 & = & D(m) & \text{if } m \notin \mathrm{dom}(\overline{D'}) \\
 & = & D'(m) & \text{if } m \notin \mathrm{dom}(\overline{D}) \\
(D \wedge D')(l) & = & l : T \wedge T' & \text{if } (l : T) \in \overline{D} \text{ and } (l : T') \in \overline{D'} \\
 & = & D(l) & \text{if } l \notin \mathrm{dom}(\overline{D'}) \\
 & = & D'(l) & \text{if } l \notin \mathrm{dom}(\overline{D}) \\
(D \vee D')(L) & = & L : (S \wedge S')..(U \vee U') & \text{if } (L : S..U) \in \overline{D} \text{ and } (L : S'..U') \in \overline{D'} \\
(D \vee D')(m) & = & m : (S \wedge S') \to (U \vee U') & \text{if } (m : S \to U) \in \overline{D} \text{ and } (m : S' \to U') \in \overline{D'} \\
(D \vee D')(l) & = & l : T \vee T' & \text{if } (l : T) \in \overline{D} \text{ and } (l : T') \in \overline{D'}
\end{array}$$

Sets of declarations form a lattice with the given meet $\wedge$ and join $\vee$, the empty set of declarations as the top element, and the bottom eler $\overline{D_\bot}$, Here $\overline{D_\bot}$ is the set of declarations that contains for every term label l the declaration $l : \bot$, for every type label L the declara $L : \top..\bot$ and for every method label m the declaration $m : \top \to \bot$.

Figure 2. The DOT Calculus : Declaration Lattice

Membership $\boxed{\Gamma \vdash t \ni D}$

$$\frac{\Gamma \vdash p : T ,\ T \prec_z \overline{D}}{\Gamma \vdash p \ni [p/z]D_i}\ (\text{PATH-}\ni) \qquad \frac{z \notin \mathsf{fn}(D_i) \qquad \Gamma \vdash t : T ,\ T \prec_z \overline{D}}{\Gamma \vdash t \ni D_i}\ (\text{TERM-}\ni)$$

Expansion $\boxed{\Gamma \vdash T \prec_z \overline{D}}$

$$\frac{\Gamma \vdash T \prec_z \overline{D'}}{\Gamma \vdash T\{z \Rightarrow \overline{D}\} \prec_z \overline{D'} \wedge \overline{D}}\ (\text{RFN-}\prec) \qquad \frac{\Gamma \vdash p \ni L : S..U ,\ U \prec_z \overline{D}}{\Gamma \vdash p.L \prec_z \overline{D}}\ (\text{TSEL-}\prec)$$

$$\frac{\Gamma \vdash T_1 \prec_z \overline{D}_1 ,\ T_2 \prec_z \overline{D}_2}{\Gamma \vdash T_1 \wedge T_2 \prec_z \overline{D}_1 \wedge \overline{D}_2}\ (\wedge\text{-}\prec) \qquad \frac{\Gamma \vdash T_1 \prec_z \overline{D}_1 ,\ T_2 \prec_z \overline{D}_2}{\Gamma \vdash T_1 \vee T_2 \prec_z \overline{D}_1 \vee \overline{D}_2}\ (\vee\text{-}\prec)$$

$$\Gamma \vdash \top \prec_z \{\}\ (\top\text{-}\prec) \qquad \Gamma \vdash \bot \prec_z \overline{D_\bot}\ (\bot\text{-}\prec)$$

Figure 3. The DOT Calculus : Membership and Expansion

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

$$\frac{\Gamma \vdash T\ \textbf{wfe}}{\Gamma \vdash T <: T}\ \text{(REFL)} \qquad \frac{\Gamma \vdash T\ \textbf{wfe}}{\Gamma \vdash \bot <: T}\ (\bot\text{-}<:)$$

$$\frac{\begin{array}{c}\Gamma \vdash T\{z \Rightarrow \overline{D}\}\ \textbf{wfe} ,\ S <: T ,\ S \prec_z \overline{D'} \\ \Gamma, z : S \vdash \overline{D' <: D}\end{array}}{\Gamma \vdash S <: T\{z \Rightarrow \overline{D}\}}\ (<:\text{-RFN}) \qquad \frac{\Gamma \vdash T\{z \Rightarrow \overline{D}\}\ \textbf{wfe} ,\ T <: T'}{\Gamma \vdash T\{z \Rightarrow \overline{D}\} <: T'}\ (\text{RFN-}<:)$$

$$\frac{\Gamma \vdash p \ni L : S..U ,\ S <: U ,\ S' <: S}{\Gamma \vdash S' <: p.L}\ (<:\text{-TSEL}) \qquad \frac{\Gamma \vdash p \ni L : S..U ,\ S <: U ,\ U <: U'}{\Gamma \vdash p.L <: U'}\ (\text{TSEL-}<:)$$

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

$$\frac{\Gamma \vdash T_2\ \textbf{wfe} ,\ T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2}\ (<:\text{-}\vee_1) \qquad \frac{\Gamma \vdash T_2\ \textbf{wfe} ,\ T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T}\ (\wedge_1\text{-}<:)$$

$$\frac{\Gamma \vdash T_1\ \textbf{wfe} ,\ T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2}\ (<:\text{-}\vee_2) \qquad \frac{\Gamma \vdash T\ \textbf{wfe}}{\Gamma \vdash T <: \top}\ (<:\text{-}\top) \qquad \frac{\Gamma \vdash T_1\ \textbf{wfe} ,\ T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T}\ (\wedge_2\text{-}<:)$$

Declaration subsumption $\boxed{\Gamma \vdash D <: D'}$

What worked:

- Bottom up instead of top down
- Focus on the core:
  path-dependent types and objects with (recursive) type members
- Look beyond rewriting semantics

Types in Scala

Modules, Objects, Functions

```
object listModule {
  trait List[Elem] = {
    def head(): Elem
    def tail(): List[Elem]
  }
  def nil() = new List[Nothing] {
    def head() = error()
    def tail() = error()
  }
  def cons[T](hd: T)(tl: List[T]) = new List[T] {
    def head() = hd
    def tail() = tl
  }
}
```

Types in Scala

'modular'	named type	`scala.collection.BitSet`
	compound type	`Channel with Logged`
	refined type	`Channel { def close(): Unit }`
'functional'	parameterized type	`List[String]`
	existential type	`List[T] forSome { type T }`
	higher-kinded type	`List`

Reducing Functional to Modular

- type parameter to type member
  `class List[Elem] {} /*vs*/ class List { type Elem }`
- parameterized type to refined type
  `List[String] /*vs*/ List { type Elem = String }`
- existential type?
  `List[T] forSome { type T } /*vs*/ List`
- higher-kinded type?
  `List /*vs*/ List`

Modules, Objects, Functions

```
val listModule = new { m =>
  type List = { this =>
    type Elem
    def head(): this.Elem
    def tail(): m.List & { type Elem <: this.Elem }
  }
  def nil() = new { this =>
    type Elem = Nothing
    def head() = error()
    def tail() = error()
  }
  def cons[T](hd:T)(tl:m.List & { type Elem <: T }) = new { this =>
    type Elem <: T
    def head() = hd
    def tail() = tl
  }
}
```

Nominality by Ascription

```
type ListAPI = { m =>
  type List <: { this =>
    type Elem
    def head(): this.Elem
    def tail(): m.List & { type Elem <: this.Elem }
  }
  def nil(): List & { type Elem = Bot }
  def cons[T]: T =>
    m.List & { type Elem <: T } =>
      m.List & { type Elem <: T }
}
```

Types in D2 (inspired by DOT)

types	`S, T, U`
path-dependent type	`p.L`
recursive self type	`{ z => T }`
intersection	`T & T`
union	`T | T`
top	`Any`
bottom	`Nothing`
type declaration	`type L: S .. U`
field declaration	`val l: U`
method declaration	`def m(x: S): U`

Subtyping

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

$$\frac{\Gamma \vdash x : (\textbf{type } L : S..U) \;,\; S' <: S}{\Gamma \vdash S' <: x.L} \quad (<:\text{-}\textsc{TSel})$$

$$\frac{\Gamma \vdash x : (\textbf{type } L : S..U) \;,\; U <: U'}{\Gamma \vdash x.L <: U'} \quad (\textsc{TSel}\text{-}<:)$$

$$\frac{\Gamma, z : T \vdash T <: T'}{\Gamma \vdash \{z \Rightarrow T\} <: \{z \Rightarrow T'\}} \quad (\textsc{Rec}\text{-}<:\text{-}\textsc{Rec})$$

$$\frac{\Gamma \vdash S' <: S \;,\; U <: U'}{\Gamma \vdash (\textbf{type } L : S..U) <: (\textbf{type } L : S'..U')} \quad (\textsc{TMem}\text{-}<:\text{-}\textsc{TMem})$$

Challenge: Type Preservation

Trouble: Type Preservation

```
trait Brand {
  type Hidden
  def pack(x: Int): Hidden
  def unpack(x: Hidden): Int
}
val brand: Brand = new Brand {
  type Hidden = Int
  def pack(x: Int): Hidden = x
  def unpack(x: Hidden): Int = x
}
brand.unpack(brand.pack(7)) // ok
brand.unpack(7) // not ok -- but occurs during reduction!
```

Trouble: $\bot$ and Intersections

Type $\bot$ is a subtype of all other types, including `{ type E = Int }` and `{ type E = String }`.

So if p: $\bot$ we have Int $<:$ p.E and p.E $<:$ String.

Transitivity would give us Int $<:$ p.E $<:$ String!

Subtyping lattice collapses.

Adding intersection types is equivalent to bottom (bad bounds!)

Key Observation

- Bottom types do not occur at runtime!
- It is enough to have transitivity and narrowing on runtime environments
- Have a (restrictive) static type system and a (lenient) one during evaluation

Definitional Interpreters

STLC Static Semantics

Syntax

$$\begin{array}{lcl} T & ::= & X \mid \top \mid T \to T \\ t & ::= & x \mid \lambda x : T.t \mid t\ t \\ \Gamma & ::= & \emptyset \mid \Gamma, x : T \end{array}$$

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

$$\Gamma \vdash S <: \top$$

$$\frac{\Gamma \vdash T_1 <: S_1 \ , \ S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2}$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \ , \ t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}$$

$$\frac{\Gamma \vdash t : S \ , \ S <: T}{\Gamma \vdash t : T}$$

STLC

```
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}:
option (option vl) :=
  DO n1 <== FUEL n;                                 (* totality    *)
  match t with
    | tcst c      => DONE VAL (vcst c)             (* constant    *)
    | tvar x      => DONE (lookup x env)           (* variable    *)
    | tabs y      => DONE VAL (vabs env x ey)      (* lambda      *)
    | tapp ef ex  =>                               (* application *)
      DO vf <== eval n1 env ef;
        DO vx <==  eval n1 env ex;
        match vf with
          | (vabs env2 x ey) =>
            eval n1 ((x,vx)::env2) ey
          | _ => ERROR
        end
  end.
```

Soundness

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash H \qquad \mathsf{eval}\ n\ H\ e = \mathsf{Done}\ r}{r = \mathsf{Val}\ v \qquad v : T}$$

$F_{<:}$ Static Semantics

Syntax

$$
\begin{array}{lcl}
X & ::= & Y \mid Z \\
T & ::= & X \mid \top \mid T \to T \mid \forall Z <: T.T^Z \\
t & ::= & x \mid \lambda x : T.t \mid \Lambda Y <: T.t \mid t\ t \mid t\,[T] \\
\Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, X <: T
\end{array}
$$

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

$$\Gamma \vdash S <: \top$$

$$\Gamma \vdash X <: X$$

$$\frac{\Gamma \ni X <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \ , \ S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 <: S_1 \\ \Gamma, Z <: T_1 \vdash S_2^Z <: T_2^Z\end{array}}{\Gamma \vdash \forall Z <: S_1.S_2^Z <: \forall Z <: T_1.T_2^Z}$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \ , \ t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}$$

$$\frac{\Gamma, Y <: T_1 \vdash t_2 : T_2^Y}{\Gamma \vdash \Lambda Y <: T_1.t_2 : \forall Z <: T_1.T_2^Z}$$

$$\frac{\Gamma \vdash t_1 : \forall Z <: T_{11}.T_{12}^Z \ , \ T_2 <: T_{11}}{\Gamma \vdash t_1[T_2] : T_{12}^{T_2}}$$

$$\frac{\Gamma \vdash t : S \ , \ S <: T}{\Gamma \vdash t : T}$$

$\mathsf{F}_{<:}$ — ?

$$(\Lambda X <: T.t)[T] \longrightarrow t[T/X]$$

$F_{<:}$ — ?

```
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}:
option (option vl) :=
  DO n1 <== FUEL n;                                 (* totality    *)
  match t with
    | tcst c      => DONE VAL (vcst c)              (* constant    *)
    | tvar x      => DONE (lookup x env)            (* variable    *)
    | tabs y      => DONE VAL (vabs env x ey)       (* lambda      *)
    | ttabs y     => DONE VAL (vtabs env x ey)      (* forall      *)
    | tapp ef ex  =>                                (* application *)
      DO vf <== eval n1 env ef;
        DO vx <==  eval n1 env ex;
        match vf with
          | (vabs env2 x ey) =>
            eval n1 ((x,vx)::env2) ey
          | _ => ERROR
        end
    | ttapp ef T =>
      DO vf <== eval n1 env ef;
        match vf with
          | (vtabs env2 x ey) => eval n1 env2 (substitute ey x T)
          | _ => ERROR
        end
  end.
```

$F_{<:}$ — ?

```
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}:
option (option vl) :=
  DO n1 <== FUEL n;                              (* totality    *)
  match t with
    | tcst c      => DONE VAL (vcst c)           (* constant    *)
    | tvar x      => DONE (lookup x env)         (* variable    *)
    | tabs y      => DONE VAL (vabs env x ey)    (* lambda      *)
    | ttabs y     => DONE VAL (vtabs env x ey)   (* forall      *)
    | tapp ef ex  =>                             (* application *)
      DO vf <== eval n1 env ef;
        DO vx <==  eval n1 env ex;
        match vf with
          | (vabs env2 x ey) =>
            eval n1 ((x,vx)::env2) ey
          | _ => ERROR
        end
    | ttapp ef T =>
      DO vf <== eval n1 env ef;
        match vf with
          | (vtabs env2 x ey) => eval n1 ((x,T)::env2) ey
          | _ => ERROR
        end
  end.
```

$F_{<:}$

```
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}:
option (option vl) :=
  DO n1 <== FUEL n;                               (* totality    *)
  match t with
    | tcst c       => DONE VAL (vcst c)           (* constant    *)
    | tvar x       => DONE (lookup x env)         (* variable    *)
    | tabs y       => DONE VAL (vabs env x ey)    (* lambda      *)
    | ttabs y      => DONE VAL (vtabs env x ey)   (* forall      *)
    | tapp ef ex   =>                             (* application *)
      DO vf <== eval n1 env ef;
        DO vx <==  eval n1 env ex;
        match vf with
          | (vabs env2 x ey) =>
            eval n1 ((x,vx)::env2) ey
          | _ => ERROR
        end
    | ttapp ef T =>
      DO vf <== eval n1 env ef;
        match vf with
          | (vtabs env2 x ey) => eval n1 ((x,vty env T)::env2) ey
          | _ => ERROR
        end
  end.
```

$F_{<:}$ Dynamics

Syntax

$$\begin{array}{lll} v & ::= & \langle H, \lambda x : T.t\rangle \mid \langle H, \Lambda Y <: T.t\rangle \\ H & ::= & \emptyset \mid H, x : v \mid H, Y = \langle H, T\rangle \\ J & ::= & \emptyset \mid J, Z <: \langle H, T\rangle \end{array}$$

Runtime subtp. $\boxed{J \vdash H_1\ T_1 <: H_2\ T_2}$

$$J \vdash H_1\ T <: H_2\ \top$$

$$\frac{J \vdash H_2\ T_1 <: H_1\ S_1 \qquad J \vdash H_1\ S_2 <: H_2\ T_2}{J \vdash H_1\ S_1 \to S_2 <: H_2\ T_1 \to T_2}$$

$$\frac{\begin{array}{c} J \vdash H_2\ T_1 <: H_1\ S_1 \\ J, Z <: \langle H_2, T_1\rangle \vdash H_1\ S_2^Z <: H_2\ T_2^Z \end{array}}{J \vdash H_1\ \forall Z <: S_1.S_2^Z <: H_2\ \forall Z <: T_1.T_2^Z}$$

Abstract type variables

$$J \vdash H_1\ Z <: H_2\ Z$$

$$\frac{J \ni Z <: \langle H, U\rangle \qquad J \vdash H\ U <: H_2\ T}{J \vdash H_1\ Z <: H_2\ T}$$

Concrete type variables

$$\frac{H_1 \ni Y_1 = \langle H, T\rangle \qquad H_2 \ni Y_2 = \langle H, T\rangle}{J \vdash H_1\ Y_1 <: H_2\ Y_2}$$

$$\frac{H_1 \ni Y = \langle H, U\rangle \qquad J \vdash H\ U <: H_2\ T}{J \vdash H_1\ Y <: H_2\ T}$$

$$\frac{J \vdash H_1\ T <: H\ L \qquad H_2 \ni Y = \langle H, L\rangle}{J \vdash H_1\ T <: H_2\ Y}$$

Transitivity

$$\frac{J \vdash H_1\ T1 <: H_2\ T2 \qquad H_2\ T_2 <: H_3\ T_3}{J \vdash H_1\ T_1 <: H_3\ T_3}$$

$F_{<:}$ Dynamics

Value type assignment $\boxed{H \vdash v : T}$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma, x : T_1 \vdash t : T_2}{H \vdash \langle H, \lambda x : T_1.t\rangle : T_1 \to T_2}$$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma, Y <: T_1 \vdash t : T_2^Y}{H \vdash \langle H, \Lambda Y <: T_1.t\rangle : \forall Z <: T_1.T_2^Z}$$

$$\frac{H_1 \vdash v : T_1 \qquad \emptyset \vdash H_1\ T_1 <: H_2\ T_2}{H_2 \vdash v : T_2}$$

Consistent environments $\boxed{\Gamma \vDash H\ J}$

$$\emptyset \vDash \emptyset\ \emptyset$$

$$\frac{\Gamma \vDash H\ J \qquad H \vdash v : T}{\Gamma, x : T \vDash (H, x : v)\ J}$$

$$\frac{\Gamma \vDash H\ J \qquad J \vdash H_1\ T_1 <: H\ T}{\Gamma, Y <: T \vDash (H, Y = \langle H_1, T_1\rangle)\ J}$$

$$\frac{\Gamma \vDash H\ J \qquad J \vdash H_1\ T_1 <: H\ T}{\Gamma, Z <: T \vDash H\ (J, Z <: \langle H_1, T_1\rangle)}$$

Challenge: Transitivity

$$\frac{J \vdash H_1\ T_1 <: H\ Y\ ,\ J \vdash H\ Y <: H_2\ T_2}{J \vdash H_1\ T_1 <: H_2\ T_2} \quad (<:\text{-TRANS})$$

Challenge: Transitivity

- Attempt 1: induction on subtyping derivation
  problem: contravariant positions
- Attempt 2: induction on middle type in $T_1 <: T_2 <: T_3$ (like $\mathsf{F}_{<:}$)
  problem: $T_1 <: Y <: T_2$
- Attempt 3: induction on a well-formedness witness
  problem: cyclicity
- Attempt 4: add transitivity axiom
  problem: inversion lemma
- Solution: transitivity axiom plus "push-back": eliminate uses of axiom after the fact

$F_{<:}$ Extended to Path-Dependent Types

Syntax

$$
\begin{array}{lcl}
T & ::= & x.\mathrm{Type} \mid \top \mid (z:T) \to T^z \mid \{\mathrm{Type} = T\} \\
t & ::= & x \mid \lambda x : T.t \mid t\ t \mid \{\mathrm{Type} = T\} \\
v & ::= & \langle H, \lambda x : T.t \rangle \mid \langle H, T \rangle \\
G & ::= & \emptyset \mid G, x : T \\
H & ::= & \emptyset \mid H, x : v \\
J & ::= & \emptyset \mid H, z : \langle H, T \rangle
\end{array}
$$

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

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_1}{\Gamma \vdash \{\mathrm{Type} = T_1\} \ <: \ \{\mathrm{Type} = T_2\}}$$

$$\frac{\Gamma(x) = U \qquad \Gamma \vdash U <: \{\mathrm{Type} = T\}}{\Gamma \vdash x.T <: T}$$

Runtime Subtp. $\boxed{J \vdash H_1\ T_1 <: H_2\ T_2}$

$$\frac{J \vdash H_1\ T_1 <: H_2\ T_2 \qquad J \vdash H_2\ T_2 <: H_1\ T_1}{J \vdash H_1\ \{\mathrm{Type} = T_1\} \ <: \ H_2\ \{\mathrm{Type} = T_2\}}$$

$$\frac{J(z) = \langle H, U \rangle \qquad J \vdash H\ U <: H_2\ \{\mathrm{Type} = T\}}{J \vdash H_1\ z.\mathrm{Type} <: H_2\ T}$$

$$\frac{H_1(x) = v \quad H \vdash v : U \qquad J \vdash H\ U <: H_2\ \{\mathrm{Type}}{J \vdash H_1\ x.\mathrm{Type} <: H_2\ T}$$

$F_{<:}$ Extended to Path-Dependent Types

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \{\mathrm{Type} = T\} : \{\mathrm{Type} = T\}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2^x}{\Gamma \vdash \lambda x : T_1.t_2 : (z : T_1) \to T_2^z}$$

$$\frac{\Gamma \vdash t_1 : (z : T_1) \to T_2^z \ ,\ t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2}$$

Value type assignment $\boxed{H \vdash v : T}$

$$\frac{\Gamma \vDash H\,\emptyset \qquad \Gamma \vdash \{\mathrm{Type} = T\} : \{\mathrm{Type} = T\}}{H \vdash \langle H, T \rangle : \{\mathrm{Type} = T\}}$$

$$\frac{\Gamma \vDash H\emptyset \qquad \Gamma, x : T_1 \vdash t : T_2^x}{H \vdash \langle H, \lambda x : T_1.t \rangle : (z : T_1) \to T_2^z}$$

Further Implemented Extensions

- Subtyping lattice (intersections and unions)
- Objects with multiple type and method members
- Recursive types `{ x => T }`
- Key technique: more intricate transitivity pushback

Thank you!

Type Inference

Type Inference in Scala (Least Upper Bound?)

```
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 <: ... } }

// in Scala REPL
> val o = if (true) (new C{}) else (new D{})
o: A with B{type T <: A with B} = ...
> val o:A with B{type T<:A with B {type T<:A with B}} =
          if (true) (new C{}) else (new D{})
o: A with B{type T <: A with B{type T <: A with B}} = ...
```

Type Inference in Scala (Working too Hard too Soon?)

```
scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.col
lection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.m
utable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.mutable.It
erable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.co
llection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.col
lection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clear
able with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.gener
ic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with scala.collection.mutable.Builder[(Int, scala.collection.mutab
le.Set[Int]) with Int,scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutab
le.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with
 scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Sh
rinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: C
loneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with Int => Any w
ith scala.collection.generic.Shrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with 
Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.gen
eric.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable 
with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.ge
neric.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable
 with Equals}}] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.Iterable[_
 >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.mutable.Iterable[_ >: (Int,
 scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.
Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.S
et[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals]
 with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int
] with scala.collection.generic.Clearable with Equals}] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with
 Int,scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Clon
eable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection
.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] wit
h scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mut
able with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with Int => Any with scala.collect
ion.generic.Shrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.c
ollection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable wi
th Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collec
tion.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[
Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}}] w
ith Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable
.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int
]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collect
ion.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with M
utable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collec
tion.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Cl
earable with Equals}] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.Iter
-:---  lub.scala      Top (1,0)      (Scala Hi Undo-Tree)
Mark set
```

Type Inference in Scala (Working too Hard too Soon?)

```
der[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.colle
ction.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with sca
la.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with Int => Any with scala.collection.generic.S
hrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mut
able.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] wi
th scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.
Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq:
 Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}}] with scala.col
lection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mut
able.Set[Int]) with Int] with scala.collection.mutable.Cloneable[scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[
Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.col
lection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable wi
th Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.co
llection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generi
c.Clearable with Equals}] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.
Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.
collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.c
ollection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Cle
arable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.gen
eric.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with Int => Any with scala.collection.generic.Shrinkable[Int]{de
f seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cl
oneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collecti
on.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] w
ith scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with M
utable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}}] with Int => Any with scala.coll
ection.generic.Shrinkable[Int]{def seq: scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scal
a.collection.mutable.Cloneable[scala.collection.mutable.Iterable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collect
ion.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equ
als] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.g
eneric.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{d
ef seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}] with sca
la.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with Int,scala.collection.mutable.Iterable[_ >: (Int, scala.collecti
on.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.collection.generic.Shrinkable[Int]
 with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.collection.mutable.Set[Int]) with I
nt,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with Int => An
y with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.co
llection.generic.Clearable with Equals}] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: scala.collection.mutable.It
erable[_ >: (Int, scala.collection.mutable.Set[Int]) with Int] with scala.collection.mutable.Cloneable[Cloneable with Mutable with scala.co
llection.generic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals] with scala.collection.mutable.Builder[(Int, scala.col
lection.mutable.Set[Int]) with Int,Cloneable with Mutable with scala.collection.generic.Shrinkable[Int] with scala.collection.generic.Clear
able with Equals] with Int => Any with scala.collection.generic.Shrinkable[Int]{def seq: Cloneable with Mutable with scala.collection.gener
ic.Shrinkable[Int] with scala.collection.generic.Clearable with Equals}}}
-:---  lub.scala      Bot (1,21480)  (Scala Undo-Tree)
```

Type Inference in Scala (Working too Hard too Soon?)

```
import scala.collection.mutable.{Map => MMap, Set => MSet}
val ms: MMap[Int, MSet[Int]] = MMap.empty

// in Scala REPL
> if (!ms.contains(1)) ms += 1 -> MSet(1) else ms(1) += 1
res0: ... (796 characters) ...
> :t res0
    : ... (21481 characters) ...
```

- Inspired by a bug report
  SI-5862: very slow compilation due to humonguous LUB.
- The character lengths reported are for Scala 2.11 (*after* the fix).
- In Dotty, type inference can be lazy thanks to the native unions (for least upper bounds) and intersections (for greatest lower bounds) of the core calculus (DOT).