Scala Step-by-Step

Soundness for DOT with Step-Indexed Logical Relations in Iris

Paolo G. Giarrusso, **Léo Stefanesco**, Amin Timany,
Lars Birkedal, Robbert Krebbers

Séminaire Cambium — 28 septembre 2020

Slides adapted from Paolo Giarrusso's

Why study Scala and DOT?

- Scala "unifies FP and OOP?"
- **Expressive**:
  ML-like software modules $\Longrightarrow$ 1^{st}-class objects
  - Unlike typeclasses and ML modules
- Objects gain **impredicative type members**
  - Relatives of Type : Type
  - Challenging to prove sound

Scala's Open Problem: Type Soundness

- First Scala version: 2003 [Odersky *et al.*]
- ✓ Soundness proven for DOT calculi, including:
  - WadlerFest DOT [2016, Amin, Grütter, Odersky, Rompf & Stucki]
  - OOPSLA DOT [2016, Rompf & Amin]
  - pDOT [2019, Rapoport & Lhoták]
- × abstract types / data abstraction / parametricity?
- × DOT lags behind Scala

Our Approach: Semantics-first Design

- × Preservation & progress (syntactic)
- ✓ Logical relations model
  - \+ Type soundness
  - \+ Data abstraction
- ✓ Retrofit DOT over model ⟹ guarded DOT (gDOT):
  - − Guardedness restrictions (acceptable in our evaluation)
  - \+ More extensible
  - \+ Extra features (see later)

Our Approach: Semantics-first Design

- ✗ Preservation & progress (syntactic)
- ✓ Logical relations model
  - \+ Type soundness
  - \+ Data abstraction
- ✓ Retrofit DOT over model ⟹ guarded DOT (gDOT):
  - – Guardedness restrictions (acceptable in our evaluation)
  - \+ More extensible
  - \+ Extra features (see later)

Our Approach: Semantics-first Design

- ✗ Preservation & progress (syntactic)
- ✓ Logical relations model
  - \+ Type soundness
  - \+ Data abstraction
- ✓ Retrofit DOT over model ⇒ guarded DOT (gDOT):
  - – Guardedness restrictions (acceptable in our evaluation)
  - \+ More extensible
  - \+ Extra features (see later)

Our Approach: Semantics-first Design

- × Preservation & progress (syntactic)
- ✓ Logical relations model
  - \+ Type soundness
  - \+ Data abstraction
- ✓ Retrofit DOT over model ⇒ guarded DOT (gDOT):
  - – Guardedness restrictions (acceptable in our evaluation)
  - \+ More extensible
  - \+ Extra features (see later)

A Scala Example

Scala Example: 1st-class Validators

We want `Validator`s that:

- ✓ Validate `Input`s from users
- ✓ Provide:
  - Abstract type `Vld` of valid `Input`
  - Smart constructor `make : Input ⇒ Option[Vld]`
- New validators can be created at runtime
- Each with a distinct **abstract type `Vld`**
- Simplifications:
  - `Input = Int`
  - Input n is valid if greater than k

```scala
val solution = new {
  type Validator = {
    type Vld             <: Int
    val make :  Int ⇒ Option[this.Vld] }
  val mkValidator : Int ⇒ Validator =
    k ⇒ new {
      type Vld = Int
      val make = n ⇒
        if (n > k) Some(n) else None }
  val pos              = mkValidator(0)
  val fails            = pos.make(-1)   // None
  val works            = pos.make(1)    // Some(1)
  val nope : pos.Vld = 1                // type error
  val legalAges        = mkValidator(   // runtime args!
    askUser("Legal age in your country?"))
}
```

```scala
val solution = new {
  type Validator = {
    type Vld >: Nothing <: Int
    val make : Int ⇒ Option[this.Vld] }
  val mkValidator : Int ⇒ Validator =
    k ⇒ new {
      type Vld = Int
      val make = n ⇒
        if (n > k) Some(n) else None }
  val pos             = mkValidator(0)
  val fails           = pos.make(-1)   // None
  val works           = pos.make(1)    // Some(1)
  val nope : pos.Vld = 1               // type error
  val legalAges       = mkValidator(   // runtime args!
    askUser("Legal age in your country?"))
}
```

```scala
val solution = new {
  type Validator = {
    type Vld >: Nothing <: Int
    val make : Int ⇒ Option[this.Vld] }
  val mkValidator : Int ⇒ Validator =
    k ⇒ new {
      type Vld = Int
      val make = n ⇒
        if (n > k) Some(n) else None }
  val pos            = mkValidator(0)
  val fails          = pos.make(-1)   // None
  val works          = pos.make(1)    // Some(1)
  val nope : pos.Vld = 1              // type error
  val legalAges      = mkValidator(   // runtime args!
    askUser("Legal age in your country?"))
}
```

```scala
val solution = new {
  type Validator = {
    type Vld >: Nothing <: Int
    val make : Int ⇒ Option[this.Vld] }
  val mkValidator : Int ⇒ Validator =
    k ⇒ new {
      type Vld = Int
      val make = n ⇒
        if (n > k) Some(n) else None }
  val pos              = mkValidator(0)
  val fails            = pos.make(-1)   // None
  val works            = pos.make(1)    // Some(1)
  val nope : pos.Vld = 1                // type error
  val legalAges        = mkValidator(   // runtime args!
    askUser("Legal age in your country?"))
}
```

```scala
val solution = new {
  type Validator = {
    type Vld >: Nothing <: Int
    val make : Int ⇒ Option[this.Vld] }
  val mkValidator : Int ⇒ Validator =
    k ⇒ new {
      type Vld = Int
      val make = n ⇒
        if (n > k) Some(n) else None }
  val pos             = mkValidator(0)
  val fails           = pos.make(-1)  // None
  val works           = pos.make(1)   // Some(1)
  val nope : pos.Vld = 1              // type error
  val legalAges       = mkValidator(  // runtime args!
    askUser("Legal age in your country?"))
}
```

Example Summary

- 1^{st}-class modules with abstract types $\mapsto$
  Scala objects with (bounded) abstract type members:

$$\frac{\Gamma \vdash L <: T <: U}{\Gamma \vdash \{\texttt{type A} = T\} : \{\texttt{type A} >: L <: U\}}$$

- **impredicative** type members
  - types (`Validator`) with nested type members (`Vld`) are regular types, not "large" types; *e.g.*, `Validator` can be a type member.

Syntax and semantics

$$
\begin{aligned}
v &::= x \mid \lambda x.\ e \mid \nu x.\ \{\bar{d}\} \mid (v_1, v_2) \mid n \in \mathbb{N} \\
d &::= a = v \mid A = T \\
e &::= v \mid e\, e' \mid e.a \mid \pi_i e \mid (e_1, e_2) \\
T &::= \forall\,(x : T_1).\ T_2 \mid p.A \mid \mu x.\ T \mid \{a : T\} \mid \{A >: L <: U\} \\
&\quad \mid \bot \mid \top \mid T_1 \wedge T_2 \mid T_1 \vee T_2 \mid \mathrm{Nat} \\
p &::= v \mid p.a
\end{aligned}
$$

Reductions: $(\lambda x.\ e)v \longrightarrow e[x := v]$ $\qquad$ $v.a \longrightarrow p$ if $v.a \searrow p$

where $v.l \searrow d \Leftrightarrow \exists\, x, \bar{d}, v = \nu x.\ \{\bar{d}\} \wedge \mathrm{lookup}(l, \bar{d}[x := v]) = d.$

Example

A module type: $T = \mu x.\ \{A >: \bot <: \top,\ f : x.A \to x.A\}$

with an implementation $I = \nu x.\ \{A = \text{Nat},\ f = \lambda y.\ y + y\}$

and a module transformer of type $T \to T$:
$F = \lambda M.\ \nu x.\ \{A = M.A \times M.A,\ f = \lambda y.\ (M.f(\pi_1 y), M.f(\pi_2 y))\}$

Note that when we β-reduce in $F\ I$, we substitute in the type component A of a term.

Sketching Our Soundness Proof

Logical relation models (a.k.a. Realizability)

Each type T is interpreted as a set $[\![T]\!]$ of terms.

The logical relation should be *sound*:

Proposition (Soundness)

For any type T, and any term t, if $t \in [\![T]\!]$, then t is *safe*.

Usually, this is trivial to prove; subtler is proving that well typed terms are realizers:

Proposition (Fundamental lemma)

If $\vdash t : T$, then $t \in [\![T]\!]$.

Typical call-by-value logical relations in Iris

The value relation is an Iris predicate on *values*:

$$\mathcal{V}[\![T]\!] : \text{SemType} \qquad \text{where SemType} := \text{Val} \to \text{iProp}$$

defined by induction on the type for a λ-calculus as:

$$\begin{aligned}
\mathcal{V}[\![\text{Nat}]\!]\, v &= v \in \mathbb{N} \\
\mathcal{V}[\![T_1 \times T_2]\!]\, v &= \exists\, v_1, v_2, v = (v_1, v_2) \land \forall\, i, \mathcal{V}[\![T_i]\!] v_i \\
\mathcal{V}[\![T_1 \to T_2]\!]\, v &= \exists\, e, v = (\lambda x.\ e) \land \forall\, v', \mathcal{V}[\![T_1]\!]\, v' \Rightarrow \mathcal{E}[\![T_2]\!]\, (e[x := v'])
\end{aligned}$$

and the predicate on terms is defined as:

$$\mathcal{E}[\![T]\!]\, e := \text{wp}\ e\ \{v.\, \mathcal{V}[\![T]\!]\, v\}$$

Logical relation for open terms

We can extend the logical relation to open terms in a context:

$$x_1 : T_1, \ldots, x_n : T_n \vDash e : T \quad = \quad \forall \vec{v_i} \in \overrightarrow{\mathcal{V}[\![T_i]\!]}, e[\vec{x_i} := \vec{v_i}] \in \mathcal{E}[\![T]\!]$$

One proves the fundamental lemma by proving the *compatibility lemmas*, which state that one can replace $\vdash$ by $\vDash$ in each typing rule.

The difficulty with DOT

Since DOT values contain types, the logical relation must be a predicate on *semantic values* SemVal, which contain semantic types.

And since types contain *value variables* we must parameterize the relation with an envionment:

$$\text{SemType} = (\text{Var} \to \text{SemVal}) \to \text{SemVal} \to \text{Prop}$$

The difficulty with DOT

$$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!]\,\rho\,v \triangleq \exists\,\varphi.\ v.\texttt{A} \searrow \varphi \wedge \mathcal{V}[\![\,L\,]\!]\,\rho \subseteq \varphi \subseteq \mathcal{V}[\![\,U\,]\!]\,\rho$$

The difficulty with DOT

$$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!]\,\rho\,v \triangleq \exists\,\varphi.\ v.\texttt{A} \searrow \varphi \wedge \mathcal{V}[\![\,L\,]\!]\,\rho \subseteq \quad \varphi \subseteq \quad \mathcal{V}[\![\,U\,]\!]\,\rho$$

$$\text{SemType} \triangleq (\text{Val} \to \text{SemVal}) \to \text{SemVal} \to \text{Prop}$$

$$\text{SemVal} \cong \ldots + \left(\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + \quad \text{SemType})\right)$$

The difficulty with DOT

$$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!]\,\rho\,v \triangleq \exists\,\varphi.\ v.\texttt{A} \searrow \varphi \wedge \mathcal{V}[\![\,L\,]\!]\,\rho \subseteq \varphi \subseteq \mathcal{V}[\![\,U\,]\!]\,\rho$$

$$\text{SemType} \triangleq (\text{Val} \to \text{SemVal}) \to \text{SemVal} \to \text{Prop}$$

$$\text{SemVal} \cong \ldots + \left(\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + \text{SemType})\right)$$

The difficulty with DOT

$$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!]\,\rho\,v \triangleq \exists\,\varphi.\ v.\texttt{A} \searrow \varphi \wedge \mathcal{V}[\![\,L\,]\!]\,\rho \subseteq \quad \varphi \subseteq \quad \mathcal{V}[\![\,U\,]\!]\,\rho$$

$$\text{SemType} \triangleq (\text{Val} \to \text{SemVal}) \to \text{SemVal} \to \text{Prop}$$

$$\text{SemVal} \cong \ldots + \left(\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + \quad \text{SemType})\right)$$

$$\text{SemVal} \cong \ldots + \left(\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + (\text{SemVal} \to \text{Prop}))\right)$$

- Unsound negative recursion!
- Exclusive to impredicative type members.

The difficulty with DOT

$$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!]\,\rho\,v \triangleq \exists\,\varphi.\ v.\texttt{A} \searrow \varphi \wedge \mathcal{V}[\![\,L\,]\!]\,\rho \subseteq \quad \varphi \subseteq \quad \mathcal{V}[\![\,U\,]\!]\,\rho$$

$$\text{SemType} \triangleq (\text{Val} \to \text{SemVal}) \to \text{SemVal} \to \text{Prop}$$

$$\text{SemVal} \cong \ldots + \left(\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + \quad \text{SemType})\right)$$

$$\text{SemVal} \cong \ldots + \left(\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + (\text{SemVal} \to \text{Prop}))\right)$$

- Unsound negative recursion!
- Exclusive to impredicative type members.

Type Members, Soundly with Iris

$$\mathcal{V}[\![\{\texttt{type A} >: L <: U\}]\!]\,\rho\,v \triangleq \exists \varphi.\ v.\texttt{A} \searrow \varphi \wedge \triangleright\,\mathcal{V}[\![L]\!]\,\rho \subseteq \triangleright\,\varphi \subseteq \triangleright\,\mathcal{V}[\![U]\!]\,\rho$$

$$\text{SemType} \triangleq (\text{Val} \to \text{SemVal}) \to \text{SemVal} \to \text{iProp}$$

$$\text{SemVal} \cong \ldots + \left(\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + \blacktriangleright\,\text{SemType})\right)$$

- Solution: Guard recursion, *i.e.,* "truncate" SemTypes with the later functor ▶ from Iris.
- Reason about solution using Iris logic, ignoring details of construction.

Type Members, Soundly with Iris

$$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!]\,\rho\,v \triangleq \exists\,\varphi.\ v.\texttt{A} \searrow \varphi \wedge \triangleright\,\mathcal{V}[\![\,L\,]\!]\,\rho \subseteq \triangleright\,\varphi \subseteq \triangleright\,\mathcal{V}[\![\,U\,]\!]\,\rho$$

$$\text{SemType} \triangleq (\text{Val} \to \text{SemVal}) \to \text{SemVal} \to \text{iProp}$$

$$\text{SemVal} \cong \ldots + \left(\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + \blacktriangleright\,\text{SemType})\right)$$

– Assertions about φ are weakened through later modality $\triangleright$

Solving this equation using Iris

Roughly, an Iris proposition is a step-indexed predicate on *worlds*:

$$\text{iProp} \cong \mathbb{N} \to \text{World} \to \text{Prop}$$

We use *saved predicates* to have semantic types (Iris predicates) inside the worlds: $\text{World} \supset \mathbb{I} \rightharpoonup \text{SemType}$.

In the logical side, this gives us a predicate $s \rightsquigarrow \phi$, where s is an identifier and ϕ is a semantic type $\text{SemVal} \to \text{iProp}$.

$$\frac{}{\Rrightarrow \exists s, s \rightsquigarrow \phi} \qquad \frac{s \rightsquigarrow \phi * s \rightsquigarrow \phi'}{\triangleright(\phi \equiv \phi')}$$

Stamped terms

Semantic terms contain an identifier s such that $s \rightsquigarrow \phi$ instead of the actual predicate.

We define a predicate $\text{stamped}(e, e')$ if e' is the stamped version of e, meaning that all types T in e are replaced with s with $s \rightsquigarrow \mathcal{V}[\![T]\!]$.

Lemma

$$\forall\, e, \models\!\!\Rrightarrow \exists\, s', \text{stamped}(e, e')$$

To handle substitutions during β-reduction, stamped terms also contain substitutions $\sigma : \text{Val} \to \text{SemVal}$ for each identifier.

Lemma

Since types do not affect the dynamics, e is safe iff e' is safe.

The logical relation

For type members:

$$\mathcal{V}[\![\ \{\mathsf{A}{>:}L{<:}U\}\ ]\!]_\rho(v) \triangleq \exists\, \sigma, s, \psi.\ \mathsf{lookup}(\mathsf{A}, v) = (\sigma, s) \wedge (s \rightsquigarrow_\sigma \psi) \wedge \big(\forall\, v.\ \triangleright\, \mathcal{V}[\![\ L\ ]\!]_\rho(v) \Rightarrow \triangleright\, \psi(v)\big) \wedge \big(\forall\, v.\ \triangleright\, \psi(v) \Rightarrow \triangleright\, \mathcal{V}[\![\ U\ ]\!]_\rho(v)\big)$$

where $s \rightsquigarrow_\sigma \psi \triangleq \exists\, \varphi.\ (s \rightsquigarrow \varphi) \wedge \triangleright(\psi = \varphi(\sigma))$

For functions:

$$\mathcal{V}[\![\ \forall\, x : S.\, T\ ]\!]_\rho(v) \triangleq \exists\, e.\ (v =_\alpha \lambda x.\, e) \wedge \forall\, w.\ \triangleright\, \mathcal{V}[\![\ S\ ]\!]_\rho(w) \Rightarrow \triangleright\, \mathcal{E}[\![\ T\ ]\!]_{(\rho, x:=w)}(e[x := w])$$

Retrofitting DOT over Model: gDOT

- Turn rules from pDOT/OOPSLA DOT into typing lemmas appropriate to the model; each proof is around 2-10 lines of Coq.
- Add type $\triangleright T$ with $\mathcal{V}[\![\triangleright T]\!] \triangleq \triangleright \mathcal{V}[\![T]\!]$ and associated typing rules
+ Stronger/additional rules
  + Abstract types in nested objects (*mutual information hiding*), as in example
  + Distributivity of $\wedge$, $\vee$, ...
  + Subtyping for recursive types (beyond OOPSLA DOT)
+ (Arguably) more principled restrictions

gDOT key typing rules

$$\frac{\Gamma \vdash_{\mathsf{p}} p : \{\mathsf{A} >: L <: U\}}{\Gamma \vdash \triangleright L <: p.\mathsf{A} <: \triangleright U}\ (<:\text{-Sel, Sel-}<:) \qquad \frac{\Gamma \vdash e : \triangleright T}{\Gamma \vdash \mathbf{coerce}\ e : T}\ (\text{T-Coerce})$$

$$\frac{\Gamma \mid x : \triangleright T \vdash \{\bar{d}\} : T}{\Gamma \vdash \nu x.\{\bar{d}\} : \mu x.T}\ (\text{T-}\{\}\text{-I}) \qquad \frac{\Gamma, x : V \vdash v : T \qquad \sout{\textbf{tight}\ T}}{\Gamma \mid x : V \vdash \{\mathsf{a} = v\} : \{\mathsf{a} : T\}}\ (\text{D-Val})$$

Contributions/In the paper

- Motivating examples for novel features
- Scale model to gDOT
  - μ-types, singleton types, path-dependent functions, paths, …
- Demonstrate expressivity despite guardedness restriction
- Data abstraction proofs
- Coq mechanization using Iris (soundness: $\approx$ 9200 LoC; examples: $\approx$ 5600 LoC)