Scala Step-by-Step

Soundness for DOT with Step-Indexed Logical Relations in Iris

Paolo G. Giarrusso,[1,2] with Léo Stefanesco,[3] Amin Timany,[4] Lars Birkedal,[4] Robbert Krebbers[2]

[1]BedRock Systems, Inc. [2]Delft University of Technology, The Netherlands [3]IRIF, Université de Paris & CNRS, France
[4]Aarhus University, Denmark

ICFP 2020

Scala Step-by-Step
Soundness for DOT with Step-Indexed Logical Relations in Iris
Paolo G. Giarrusso,[1,2] with Léo Stefanesco,[3] Amin Timany,[4] Lars Birkedal,[4] Robbert Krebbers[2]
[1]BedRock Systems, Inc. [2]Delft University of Technology, The Netherlands [3]IRIF, Université de Paris & CNRS, France [4]Aarhus University, Denmark
ICFP 2020

1. Hi everybody and thanks for tuning in.
2. Today I'll present joint work with Léo Stefanesco, Amin Timany, Lars Birkedal and Robbert Krebbers: a new approach to soundness proofs for the DOT calculus, the core of the Scala programming language.

Why study Scala and DOT?

- Scala “unifies FP and OOP?”
- **Expressive**:
  ML-like software modules $\Rightarrow$ 1st-class objects
  - Unlike typeclasses and ML modules
- Objects gain **impredicative type members** (!)
  - Relatives of Type : Type
  - Challenging to prove sound

Why study Scala and DOT?
- Scala “unifies FP and OOP?”
- **Expressive**:
  ML-like software modules $\Rightarrow$ 1st-class objects
  - Unlike typeclasses and ML modules
- Objects gain **impredicative type members** (!)
  - Relatives of Type : Type
  - Challenging to prove sound

1. Before we dive into this talk, you might ask: what’s interesting about the Scala type system? One might answer that Scala “unifies functional and object-oriented programming” or “has a very expressive module system”, but what does *that* mean?
2. In other functional programming languages, software modules are expressed through special constructs such as typeclasses or ML modules, which are not 1st-class and require special abstraction mechanisms. Instead, Scala extends regular objects with abstract type members, allowing them to serve as 1st-class modules. Hence, you can instantiate modules at runtime and abstract code across modules using regular Scala abstraction mechanisms, such as plain functions or even mixin inheritance.
3. While abstract types appear in other sound type systems, Scala abstract type members add a significant challenge to our type soundness proof.
4. In a few slides, we show a very small example.

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

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

1. While Scala is interesting, its type soundness has been an open question since Scala's introduction in 2003.
2. Soundness proofs have been machine-checked for significant Scala fragments, the DOT calculi, an amazing success.
3. However, we have no proofs that abstract types are indeed abstract. And DOT calculi lag behind Scala even on core features, and are not catching up: DOT is hard to extend, while Scala's evolution is not slowing down.

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
✓ Retrofit DOT over model ⟹ guarded DOT (gDOT):

1. To create a more extensible DOT calculus, our approach avoids syntactic proofs that use preservation and progress.
2. Instead, we built a logical relations model to prove type soundness and data abstraction.
3. Then, we have retrofitted a version of DOT over this model, obtaining a more extensible DOT variant, called guarded DOT or gDOT.
4. Unfortunately, gDOT adds certain guardedness restrictions, but in our evaluation they seem acceptable. In exchange, gDOT is more extensible, as we show through some additional features that we'll mention 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):

1. To create a more extensible DOT calculus, our approach avoids syntactic proofs that use preservation and progress.
2. Instead, we built a logical relations model to prove type soundness and data abstraction.
3. Then, we have retrofitted a version of DOT over this model, obtaining a more extensible DOT variant, called guarded DOT or gDOT.
4. Unfortunately, gDOT adds certain guardedness restrictions, but in our evaluation they seem acceptable. In exchange, gDOT is more extensible, as we show through some additional features that we'll mention 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)

1. To create a more extensible DOT calculus, our approach avoids syntactic proofs that use preservation and progress.
2. Instead, we built a logical relations model to prove type soundness and data abstraction.
3. Then, we have retrofitted a version of DOT over this model, obtaining a more extensible DOT variant, called guarded DOT or gDOT.
4. Unfortunately, gDOT adds certain guardedness restrictions, but in our evaluation they seem acceptable. In exchange, gDOT is more extensible, as we show through some additional features that we'll mention 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)

1. To create a more extensible DOT calculus, our approach avoids syntactic proofs that use preservation and progress.
2. Instead, we built a logical relations model to prove type soundness and data abstraction.
3. Then, we have retrofitted a version of DOT over this model, obtaining a more extensible DOT variant, called guarded DOT or gDOT.
4. Unfortunately, gDOT adds certain guardedness restrictions, but in our evaluation they seem acceptable. In exchange, gDOT is more extensible, as we show through some additional features that we'll mention later.

A Scala Example

A Scala Example

1. To make things concrete, let's now look at an example of Scala code.

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 Example: 1st-class Validators
We want Validators that:
✓ Validate Inputs 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

1. In this example, we create **Validator** components that validate **Input**s from users.
2. They provide an abstract type **Valid** of valid **Input**, and a corresponding smart constructor make that validates its input and returns either valid data or nothing.
3. Up to this point, nothing too fancy.
4. However, new validators can be created at runtime, each with a distinct abstract type **Valid**.
5. To simplify the code, we hardcode **Input** to `Int`, and an input n is valid if it is greater than k.
6. The solution is in next slide.

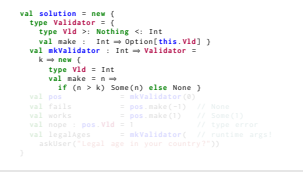
```scala
val solution = new {
  type Validator = {
    type Vld              <: Int
    val make :  Int ⇒ Option[this.Vld] }
  val mkValidator : Int ⇒ Validator =
    k ⇒ new {
      type Vld = Int
      val make = n ⇒
        if (n > k) Some(n) else None }
  val pos              = mkValidator(0)
  val 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?"))
}
```


1. Our **solution** is an object that defines a type of **Validator**s. Inhabitants of type **Validator** contain two members: the type of valid inputs, **Valid**, and the smart constructor, make. **Valid** is declared to be a subtype of `Int`, the type of all inputs. That is, type **Valid** has `Int` as upper bound,
2. and implicitly the empty type **Nothing** as lower bound. These bounds encode that **Valid** is abstract, hence inhabitants of **Valid** can only be constructed through make.
3. Function **makeValidator** maps input k to a **Validator** that accepts only integers greater than k.
4. For instance, here by calling **makeValidator**`(0)` we create a **Validator** called **pos**, which only accepts positive integers, as shown by `fails` and `works`. The type system rejects nope because **pos**.**Valid** is abstract, even tho it is implemented by `Int`, preventing users from bypassing our smart constructor.
5. Thanks to 1st-class modules, we can also choose k at runtime, as done when creating `legalAges`. As promised, type `legalAges`.**Valid** is a distinct abstract type.

```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?"))
}
```

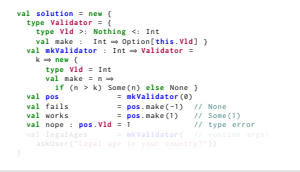

1. Our **solution** is an object that defines a type of **Validator**s. Inhabitants of type **Validator** contain two members: the type of valid inputs, **Valid**, and the smart constructor, make. **Valid** is declared to be a subtype of `Int`, the type of all inputs. That is, type **Valid** has `Int` as upper bound,
2. and implicitly the empty type **Nothing** as lower bound. These bounds encode that **Valid** is abstract, hence inhabitants of **Valid** can only be constructed through make.
3. Function **makeValidator** maps input k to a **Validator** that accepts only integers greater than k.
4. For instance, here by calling **makeValidator**`(0)` we create a **Validator** called **pos**, which only accepts positive integers, as shown by `fails` and `works`. The type system rejects nope because **pos**.**Valid** is abstract, even tho it is implemented by `Int`, preventing users from bypassing our smart constructor.
5. Thanks to 1st-class modules, we can also choose k at runtime, as done when creating `legalAges`. As promised, type `legalAges.`**Valid** is a distinct abstract type.

```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?"))
}
```

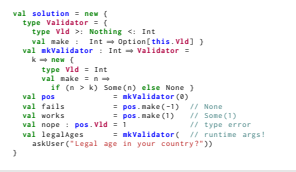
1. Our **solution** is an object that defines a type of **Validator**s. Inhabitants of type **Validator** contain two members: the type of valid inputs, **Valid**, and the smart constructor, make. **Valid** is declared to be a subtype of `Int`, the type of all inputs. That is, type **Valid** has `Int` as upper bound,
2. and implicitly the empty type **Nothing** as lower bound. These bounds encode that **Valid** is abstract, hence inhabitants of **Valid** can only be constructed through make.
3. Function **makeValidator** maps input k to a **Validator** that accepts only integers greater than k.
4. For instance, here by calling **makeValidator**`(0)` we create a **Validator** called **pos**, which only accepts positive integers, as shown by `fails` and `works`. The type system rejects nope because **pos**.**Valid** is abstract, even tho it is implemented by `Int`, preventing users from bypassing our smart constructor.
5. Thanks to 1st-class modules, we can also choose k at runtime, as done when creating `legalAges`. As promised, type `legalAges`.**Valid** is a distinct abstract type.

```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?"))
}
```

1. Our **solution** is an object that defines a type of **Validator**s. Inhabitants of type **Validator** contain two members: the type of valid inputs, **Valid**, and the smart constructor, make. **Valid** is declared to be a subtype of Int, the type of all inputs. That is, type **Valid** has Int as upper bound,
2. and implicitly the empty type **Nothing** as lower bound. These bounds encode that **Valid** is abstract, hence inhabitants of **Valid** can only be constructed through make.
3. Function **makeValidator** maps input k to a **Validator** that accepts only integers greater than k.
4. For instance, here by calling **makeValidator**(0) we create a **Validator** called **pos**, which only accepts positive integers, as shown by fails and works. The type system rejects nope because **pos**.**Valid** is abstract, even tho it is implemented by Int, preventing users from bypassing our smart constructor.
5. Thanks to 1st-class modules, we can also choose k at runtime, as done when creating legalAges. As promised, type legalAges.**Valid** is a distinct abstract type.

```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?"))
}
```

1. Our **solution** is an object that defines a type of **Validator**s. Inhabitants of type **Validator** contain two members: the type of valid inputs, **Valid**, and the smart constructor, make. **Valid** is declared to be a subtype of Int, the type of all inputs. That is, type **Valid** has Int as upper bound,
2. and implicitly the empty type **Nothing** as lower bound. These bounds encode that **Valid** is abstract, hence inhabitants of **Valid** can only be constructed through make.
3. Function **makeValidator** maps input k to a **Validator** that accepts only integers greater than k.
4. For instance, here by calling **makeValidator**(0) we create a **Validator** called **pos**, which only accepts positive integers, as shown by fails and works. The type system rejects nope because **pos**.**Valid** is abstract, even tho it is implemented by Int, preventing users from bypassing our smart constructor.
5. Thanks to 1st-class modules, we can also choose k at runtime, as done when creating legalAges. As promised, type legalAges.**Valid** is a distinct abstract type.

Example Summary

- 1st-class modules with abstract types ↦
  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.

Example Summary

- 1st-class modules with abstract types ↦
  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.

1. This small example already uses many Scala features: in Scala 1st-class modules with abstract types are encoded as objects with bounded type members.
2. We've used the typing rule for type member introduction: a type definition $\{\texttt{type A} = T\}$ inhabits a bounded type declarations $\{\texttt{type A} >: L <: U\})$ if T is between bounds L and U.
3. We've even used impredicative type members! Types like `Validator`, with nested type members like `Valid`, are still regular types, not "large" types, and are subject to regular abstractions; so for instance, `Validator` can in turn be a type member.

Sketching Our Soundness Proof

1. Next, we sketch how we prove type soundness. We'll oversimplify in this talk and leave the rest to the paper.

Logical relation models

$$\begin{array}{rcl}
\text{Type } T & \mapsto & \text{set of values } \mathcal{V}[\![T]\!]: \\
\mathcal{V}[\![S \wedge T]\!] & \triangleq & \mathcal{V}[\![S]\!] \cap \mathcal{V}[\![T]\!]
\end{array}$$

$$\begin{array}{rcl}
\text{Syn. typing judgment } \vdash J & \mapsto & \textit{sem. typing judgment } \vDash J: \\
\vDash S <: T & \triangleq & \mathcal{V}[\![S]\!] \subseteq \mathcal{V}[\![T]\!] \\
\vDash e : T & \triangleq & e \textbf{ runs safely} \text{ with result in } \mathcal{V}[\![T]\!]
\end{array}$$

$$\begin{array}{rcl}
\text{Typing rule} & \mapsto & \textit{typing lemma}: \\
\vDash S \wedge T <: S & \Leftrightarrow & \mathcal{V}[\![S \wedge T]\!] \subseteq \mathcal{V}[\![S]\!]
\end{array}$$

Result: extensible type soundness!

Logical relation models

Type T ↦ set of values $\mathcal{V}[\![T]\!]$:
$\mathcal{V}[\![S \wedge T]\!]$ ≜ $\mathcal{V}[\![S]\!] \cap \mathcal{V}[\![T]\!]$
Syn. typing judgment $\vdash J$ ↦ *sem. typing judgment* $\vDash J$:
$\vDash S <: T$ ≜ $\mathcal{V}[\![S]\!] \subseteq \mathcal{V}[\![T]\!]$
$\vDash e : T$ ≜ e **runs safely** with result in $\mathcal{V}[\![T]\!]$
Typing rule ↦ *typing lemma*:
$\vDash S \wedge T <: S$ ⇔ $\mathcal{V}[\![S \wedge T]\!] \subseteq \mathcal{V}[\![S]\!]$
Result: extensible type soundness!

1. Our model uses a logical relation. We map each type T to a set "V of T" of values that behave as required by type T.
2. For instance, intersection types are interpreted using set intersection. The set "V of S and T" is the intersection of V of S and V of T.
3. Then, we map syntactic typing judgments to semantic typing judgments. For instance, S is a subtype of T if $\mathcal{V}[\![S]\!]$ is a subset of $\mathcal{V}[\![T]\!]$. Crucially, an expression e has semantic type T if e runs safely and any result is in $\mathcal{V}[\![T]\!]$; hence semantically typed expressions are type-safe **by definition**.
4. Then, we map each typing rule to a typing lemma about semantic typing judgments that we must prove. For instance, type S and T is a subtype of S because "V of S and T" is a subset of "V of S".
5. We now have a type soundness proof that is extensible: proving new typing lemmas cannot invalidate old ones, because each lemma is proved independently.

Types Members, Naively

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

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

1. Here's a naive model of impredicative type members. An object v has type member A in bounds L and U if $v.A$ contains type φ between bounds L and U.
2. Metavariables φ and v range over sets of semantic types and values, defined as follows: Types are sets of values, represented as membership predicates. And values can be (among other things) objects, that is, finite maps from field labels to values or types.
3. However, this definition is illegal: inlining SemType shows we're defining SemVal by negative recursion, which is illegal.
4. This problem is exclusive to impredicative type members.

Types Members, Naively

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

$$\text{SemType} \triangleq \text{SemVal} \to \text{Prop}$$

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

Types Members, Naively
$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!] \triangleq \{v \mid \exists \varphi.\ v.\texttt{A} \searrow \varphi \wedge \mathcal{V}[\![\,L\,]\!] \subseteq \varphi \subseteq \mathcal{V}[\![\,U\,]\!]\}$
$\text{SemType} \triangleq \text{SemVal} \to \text{Prop}$
$\text{SemVal} \cong \ldots + (\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + \text{SemType}))$

1. Here's a naive model of impredicative type members. An object v has type member A in bounds L and U if $v.A$ contains type φ between bounds L and U.
2. Metavariables φ and v range over sets of semantic types and values, defined as follows: Types are sets of values, represented as membership predicates. And values can be (among other things) objects, that is, finite maps from field labels to values or types.
3. However, this definition is illegal: inlining SemType shows we're defining SemVal by negative recursion, which is illegal.
4. This problem is exclusive to impredicative type members.

Types Members, Naively

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

$$\boxed{\text{SemType}} \triangleq \boxed{\text{SemVal}} \to \text{Prop}$$

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

Types Members, Naively

1. Here's a naive model of impredicative type members. An object v has type member A in bounds L and U if $v.A$ contains type φ between bounds L and U.
2. Metavariables φ and v range over sets of semantic types and values, defined as follows: Types are sets of values, represented as membership predicates. And values can be (among other things) objects, that is, finite maps from field labels to values or types.
3. However, this definition is illegal: inlining SemType shows we're defining SemVal by negative recursion, which is illegal.
4. This problem is exclusive to impredicative type members.

Types Members, Naively

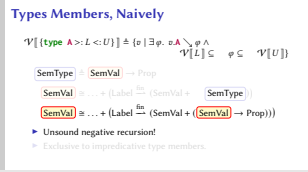
$$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!] \triangleq \{v \mid \exists \varphi.\ v.\texttt{A} \searrow \varphi \wedge \mathcal{V}[\![L]\!] \subseteq \varphi \subseteq \mathcal{V}[\![U]\!]\}$$

$$\text{SemType} \triangleq \text{SemVal} \to \text{Prop}$$

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

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

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



1. Here's a naive model of impredicative type members. An object v has type member A in bounds L and U if $v.A$ contains type φ between bounds L and U.
2. Metavariables φ and v range over sets of semantic types and values, defined as follows: Types are sets of values, represented as membership predicates. And values can be (among other things) objects, that is, finite maps from field labels to values or types.
3. However, this definition is illegal: inlining SemType shows we're defining SemVal by negative recursion, which is illegal.
4. This problem is exclusive to impredicative type members.

Types Members, Naively

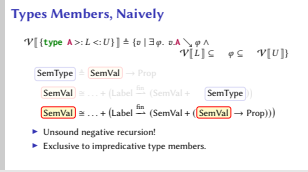
$$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!] \triangleq \{v \mid \exists \varphi.\ v.\texttt{A} \searrow \varphi \wedge \mathcal{V}[\![\,L\,]\!] \subseteq \varphi \subseteq \mathcal{V}[\![\,U\,]\!]\}$$

$$\text{SemType} \triangleq \text{SemVal} \to \text{Prop}$$

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

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

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



1. Here's a naive model of impredicative type members. An object v has type member A in bounds L and U if $v.A$ contains type φ between bounds L and U.
2. Metavariables φ and v range over sets of semantic types and values, defined as follows: Types are sets of values, represented as membership predicates. And values can be (among other things) objects, that is, finite maps from field labels to values or types.
3. However, this definition is illegal: inlining SemType shows we're defining SemVal by negative recursion, which is illegal.
4. This problem is exclusive to impredicative type members.

Type Members, Soundly with Iris

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

$$\text{SemType} \triangleq \text{SemVal} \to \text{iProp}$$

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

+ 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
$\mathcal{V}[\![\{\texttt{type}\ \texttt{A} >: L <: U\}]\!] \triangleq \{v \mid \exists \varphi.\ v.\texttt{A} \searrow \varphi \wedge \triangleright \mathcal{V}[\![L]\!] \subseteq \triangleright \varphi \subseteq \triangleright \mathcal{V}[\![U]\!]\}$
SemType ≜ SemVal → iProp
SemVal ≅ … + (Label $\xrightarrow{\text{fin}}$ (SemVal + ▶ SemType))
+ Solution: Guard recursion, *i.e.,* "truncate" SemTypes with the later functor ▶ from Iris.
+ Reason about solution using Iris logic, ignoring details of construction.

1. Thankfully, abstract step-indexing allows us to construct SemVal with a small change to our recursive equation. We must guard the recursion, that is, storing types into values must truncate types using Iris's later functor.
2. I won't explain what exactly that means. We can ignore details and reason about the solution to this equation using the Iris logic.
3. Most of the logical relation is identical to the naive model, except for type members. Since values only contain **truncated** type members, assertions about type members must be weakened, using the later modality.

Type Members, Soundly with Iris

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

$$\text{SemType} \triangleq \text{SemVal} \to \text{iProp}$$

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

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

Type Members, Soundly with Iris
$\mathcal{V}[\![\,\{\texttt{type A} >: L <: U\}\,]\!] \triangleq \{v \mid \exists \varphi.\ v.\texttt{A} \searrow \varphi \wedge \triangleright\mathcal{V}[\![\,L\,]\!] \subseteq \triangleright\varphi \subseteq \triangleright\mathcal{V}[\![\,U\,]\!]\}$
$\text{SemType} \triangleq \text{SemVal} \to \text{iProp}$
$\text{SemVal} \cong \ldots + (\text{Label} \xrightarrow{\text{fin}} (\text{SemVal} + \blacktriangleright \text{SemType}))$
– Assertions about φ are weakened through later modality $\triangleright$

1. Thankfully, abstract step-indexing allows us to construct SemVal with a small change to our recursive equation. We must guard the recursion, that is, storing types into values must truncate types using Iris's later functor.
2. I won't explain what exactly that means. We can ignore details and reason about the solution to this equation using the Iris logic.
3. Most of the logical relation is identical to the naive model, except for type members. Since values only contain **truncated** type members, assertions about type members must be weakened, using the later modality.

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

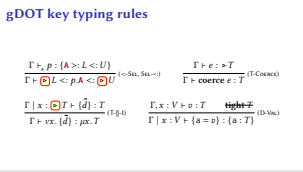
Retrofitting DOT over Model: gDOT
- Turn rules from pDOT/OOPSLA DOT into typing lemmas appropriate to the model; each proof is around 2-10 lines of Coq.
- Add type $\triangleright T$ with $\mathcal{V}[\![\triangleright T]\!] \triangleq \triangleright \mathcal{V}[\![T]\!]$ and associated typing rules (!)
+ 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

1. Adapt rules from past DOT calculi to match the model.
2. Add type $\triangleright T$, reflecting the later modality, and (quite a few) associated typing rules.
3. Some rules become stronger; in other cases, restrictions become more principled,
4. We obtain guarded DOT (gDOT), *i.e.,* DOT with guardedness restrictions.

gDOT key typing rules

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

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



1. Rules (<:-Sel, Sel-<:) reflect the guardedness restrictions on type members. Rule (T-Coerce) shows how to eliminate ▷ in many cases, by adding no-op coercions to terms.
2. Rule (T-{}-I) allows introducing recursive objects. To prevent certain circular typing derivations, we avoid the ad-hoc fixes of other DOT calculi, and guard against recursive self-references by adding ▷ to the type of self-variable x.
3. Rule (D-Val) allows to nest objects in objects. Unlike pDOT, in gDOT nested objects support abstract type members, and this is possible because of the restriction we added to the rule for object introduction.

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: ≈ 9200 LoC; examples: ≈ 5600 LoC)

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: ≈ 9200 LoC; examples: ≈ 5600 LoC)

1. In the paper:
2. We give motivating examples for some of our novel features.
3. We scale this model to full gDOT, with mu types, singleton types, path-dependent functions, paths, and so on.
4. We demonstrate expressivity despite the restrictions to type members
5. We give proofs about data abstraction.
6. Mechanize everything in Coq using Iris.

Future work

- Type projections
- Higher-kinds
- Elaboration from calculi closer to Scala, and ▹-inference
- Applications to other type systems with impredicative type members/virtual classes

Future work
- Type projections
- Higher-kinds
- Elaboration from calculi closer to Scala, and ▹-inference
- Applications to other type systems with impredicative type members/virtual classes

1. We are extending our model to type projections and higher kinds.
2. Scala users don't write ▹, but can they be inferred during elaboration?
3. Finally we hope these techniques can be extended to other object-oriented type systems with impredicative type members or virtual classes.

Conclusions

- Scala needs extensible type-soundness ⇒ semantics-first
- Challenge: impredicative type members
- Iris enabled machine-checking solution conveniently in Coq

Conclusions

- Scala needs extensible type-soundness ⇒ semantics-first
- Challenge: impredicative type members
- Iris enabled machine-checking solution conveniently in Coq

1. Scala's type system needs extensible type-soundness proofs, so we use design our type-system semantics-first.
2. The challenge was that impredicative type members are crucial for Scala's expressivity, but hard to support.
3. Iris was essential to design and machine-check our model.