Connecting Scala to DOT

Semester Project Presentation

Samuel Grütter

EPFL

June 21, 2016

Scala vs DOT

	Scala	DOT
type assignment for object constructions	nominal	structural
classes, inheritance	yes	no
mixins, super, lazy val, null, and many more features . . .	yes	no

Scala vs DOT

	Scala	DOT
type assignment for object constructions	nominal **new** C	structural **new** $\{z \Rightarrow \overline{d}\}$
classes, inheritance	yes	no
mixins, super, lazy val, null, and many more features ...	yes	no

Scala vs DOT

	Scala	DOT
type assignment for object constructions	nominal	structural
classes, inheritance	yes	no
mixins, super, lazy val, null, and many more features . . .	yes	no

Outline

1. Scala vs DOT
2. Miniscala
   - A small Scala class calculus
3. DOT-dependent Miniscala typechecking
   - Defining Miniscala typechecking in terms of DOT typechecking
4. DOT-independent Miniscala typechecking
   - Defining Miniscala typechecking on Miniscala
5. Practical expressivity checking
   - Implementing a Scala-to-DOT translation pipeline
6. Conclusion

Miniscala: A small Scala class calculus

- Identifiers (used as variables and labels)
  x, y, z, l, m
- Paths (only used to refer to classes)
  $p \quad ::= \quad x \mid x.l \mid \texttt{AnyRef}$
- Terms
  $t \quad ::= \quad x \mid t.m(t) \mid \textbf{new}\ p \mid d; t \mid <\!abstract\!>$
- Definitions
  $$\begin{array}{lll} d & ::= & \textbf{val}\ l : T = t \\ & & \textbf{def}\ m(x : S) : T = t \\ & & \textbf{class}\ l\ \textbf{extends}\ p\ \{z \Rightarrow \overline{d}\} \end{array}$$
- Types
  $T \quad ::= \quad p$

Compile and run Miniscala with scalac/dotty

Given Miniscala term `t`:

```
object Main {
  def main(args: Array[String]): Unit = println({
    t
  })
}
```

Miniscala Example

```
class Nats extends AnyRef{nats ⇒
  class Nat extends AnyRef{n ⇒
    def succ(d : Unit) : nats.Nat = {
      class S extends nats.Succ{s ⇒
        def pred(d : Unit) : nats.Nat = n
      };
      new S
    }
    def plus(other : nats.Nat) : nats.Nat = <abstract>
  }
  class Zero extends nats.Nat{z ⇒
    def plus(other : nats.Nat) : nats.Nat = other
  }
  class Succ extends nats.Nat{n ⇒
    def pred(d : Unit) : nats.Nat = <abstract>
    def plus(other : nats.Nat) : nats.Nat = n.pred(unit).plus(other).succ(unit)
  }
};
val nats = new Nats;  val zero = new nats.Zero;  val one = zero.succ(unit);  . . .
```

Miniscala Typechecking Rules

Two approaches to define typechecking:

3. DOT-dependent Miniscala typechecking
   - Defining Miniscala typechecking in terms of DOT typechecking
4. DOT-independent Miniscala typechecking
   - Defining Miniscala typechecking on Miniscala

Defining Miniscala typechecking in terms of DOT typechecking

Define f: Miniscala term $\longrightarrow$ DOT term

Definition of Miniscala typechecking:
The Miniscala term t is said to typecheck iff the DOT term $f(t)$ typechecks according to the DOT rules.

Trying to define a syntactic transformation from Miniscala to DOT

We need to collect inherited members.
Example:

```
class A extends AnyRef { z =>
  class I extends AnyRef { i =>
    def fun1(x: AnyRef): AnyRef = x
  }
};
class B extends A { z => };
val b = new B;
class C extends b.I { z =>
  def fun2(x: AnyRef): AnyRef = x
};
new C
```

It's not just a syntactic transformation

- We need semantic information, no simple syntactic transformation
- DOT already deals with semantics
- So let DOT do the "collecting inherited members" part

νDOT

$$\begin{tikzcd}
\nu \mathit{Obj} \arrow[dr] & & \mathsf{DOT} \arrow[dl] \\
& \nu\mathsf{DOT} &
\end{tikzcd}$$

νDOT

Grammar:

- Terms
  $t \quad ::= \quad x \;\mid\; t.m(t) \;\mid\; \textbf{new}\ t \;\mid\; [x : S|\overline{d}] \;\mid\; t \,\&\, t$
- Definitions
  $d \quad ::= \quad \textbf{def}\ m(x : S) : T = t \;\mid\; \textbf{type}\ A = T$
- Types
  $S, T, U \quad ::= \quad \ldots \;\mid\; [x : S|T]$
- Values
  $v \quad ::= \quad \{z \Rightarrow \overline{d}\} \;\mid\; [x : S|\overline{d}]$

Evaluation rules:

- $[z : S_1|\overline{d_1}] \,\&\, [z : S_2|\overline{d_2}] \;\rightarrow\; [z : S_1 \wedge S_2|\overline{d_1} \triangleleft \overline{d_2}]$
  where $\triangleleft$ is overriding concatenation
- plus what you'd expect

Typechecking νDOT

$$\frac{\Gamma, x : S \vdash \overline{d} : T}{\Gamma \vdash [x : S | \overline{d}] : [x : S | T]} \tag{T-TPL}$$

$$\frac{\Gamma \vdash t_i : [x : S_i | T_i] \quad \text{for } i = 1, 2}{\Gamma \vdash t_1 \ \& \ t_2 : [x : S_1 \wedge S_2 | T_1 \triangleleft T_2]} \tag{T-MIX}$$

$$\frac{\Gamma \vdash t : [x : S | T] \qquad \Gamma, x : T \vdash T <: S}{\Gamma \vdash \textbf{new } t : \{x \Rightarrow T\}} \tag{T-NEW}$$

T-TPL puts an arbitrary self type S into Γ: Bad bounds? ⚠ ⚠

T-NEW to the rescue: It guarantees the following desirable properties:

- The self type T put into Γ has good bounds.
- All self types which were put into Γ to typecheck the definitions of the class template t have good bounds.
- The actual self type of the object being created, i.e. T, complies to the self types which were assumed when typechecking the definitions of the class template t.

Type safety for νDOT

- Proven in Coq
- Helper lemma: *Behavior of* $\triangleleft$ *with respect to typing*
  If $\Gamma \vdash \overline{d_1} : T_1$ and $\Gamma \vdash \overline{d_2} : T_2$, then $\Gamma \vdash \overline{d_1} \triangleleft \overline{d_2} : T_1 \triangleleft T_2$.
- Theorem: *Type safety*
  If $s\ \emptyset \vdash t : T$, then either there exists a variable x and a value v such that $t = x$ and $(x = v) \in s$, or there exists an s' extending s and a t' such that $s|t \ \rightarrow\ s'|t'$ and $s'\ \emptyset \vdash t' : T$.

Translating from Miniscala to νDOT

Translating from Miniscala to νDOT

- Method to create class template:

```
def tpl_Succ(pr: nats.Nat): nats.Tpl_Succ =
  nats.tpl_Nat(unit) & [ n: nats.Inst_Succ |
    def pred(dummy: AnyRef): nats.Nat = pr
    def plus(other: nats.Nat): nats.Nat = ...
    def id(dummy: AnyRef): nats.Nat = ...
  ]
```

- Instantiate objects:

```
new tpl_Succ(someNat)
```

Details:
https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-to-nuDOT-rules.md
https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/NumbersWithCtor.scala
https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/NumbersWithCtor-in-nuDOT.txt

Ok, but ...

...what about nominality?

```
class Company {
  def name: String = ...
  def address: String = ...
};
class Employee {
  def name: String = ...
  def address: String = ...
};
val e: Employee = new Company
```

Remember the definition we want to make:
The Miniscala term t is said to typecheck iff the DOT term $f(t)$ typechecks according to the DOT rules.

Trying to encode nominality

nominality of constructors
(can we do it in DOT?)

collecting inherited members
(requires νDOT)

Trying to encode nominality

nominality of constructors
(requires POT)

collecting inherited members
(requires νDOT)

Miniscala Typechecking Rules

Two approaches to define typechecking:

3. DOT-dependent Miniscala typechecking
   - Defining Miniscala typechecking in terms of DOT typechecking
4. DOT-independent Miniscala typechecking
   - Defining Miniscala typechecking on Miniscala

DOT-independent Miniscala typechecking

Drawbacks of defining Miniscala typechecking in terms of DOT:

- DOT typechecking is undecidable (because of System $F_{<:}$)
- The typecheckers used in the Scala and dotty compilers operate on Scala code, not on DOT

More motivation:

- If translation (to POT) needs Γ anyways, can also typecheck

So we define:

- Type checking with translation: $\Gamma \vdash t : T \rightsquigarrow t'$

TODO:

- Prove soundness: If Miniscala typechecking of a term succeeds, then the term translated to DOT should also typecheck in DOT

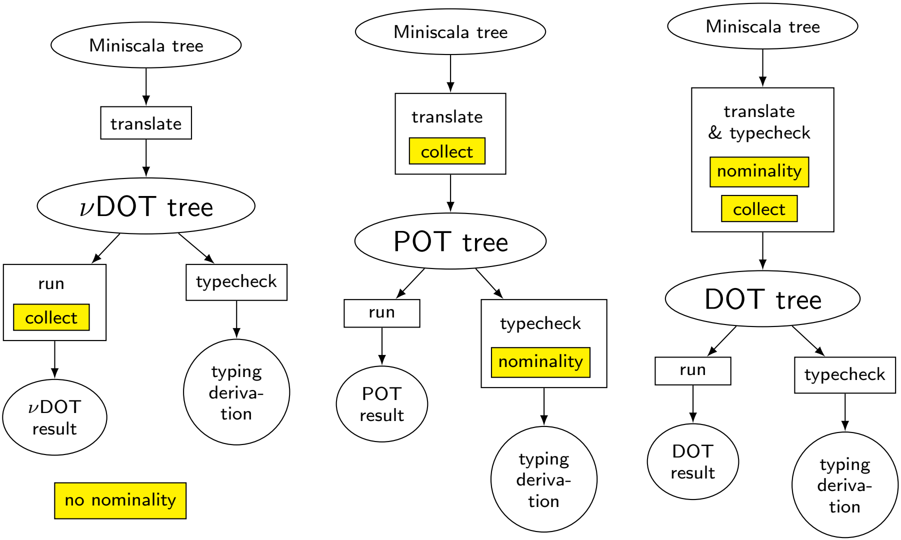
Details:
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-to-DOT-rules.md`
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/Booleans.scala`
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/Booleans-in-DOT.txt`

Miniscala Typechecking Rules

Two approaches to define typechecking:

3. DOT-dependent Miniscala typechecking
   - Defining Miniscala typechecking in terms of DOT typechecking
4. DOT-independent Miniscala typechecking
   - Defining Miniscala typechecking on Miniscala

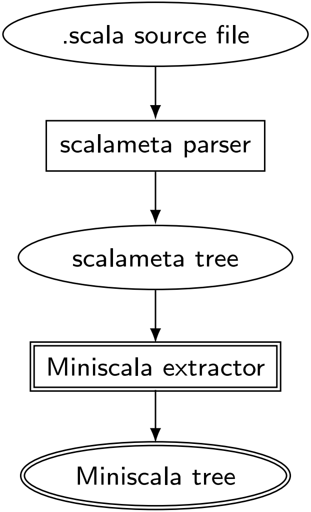
Summary of the three translations



Outline

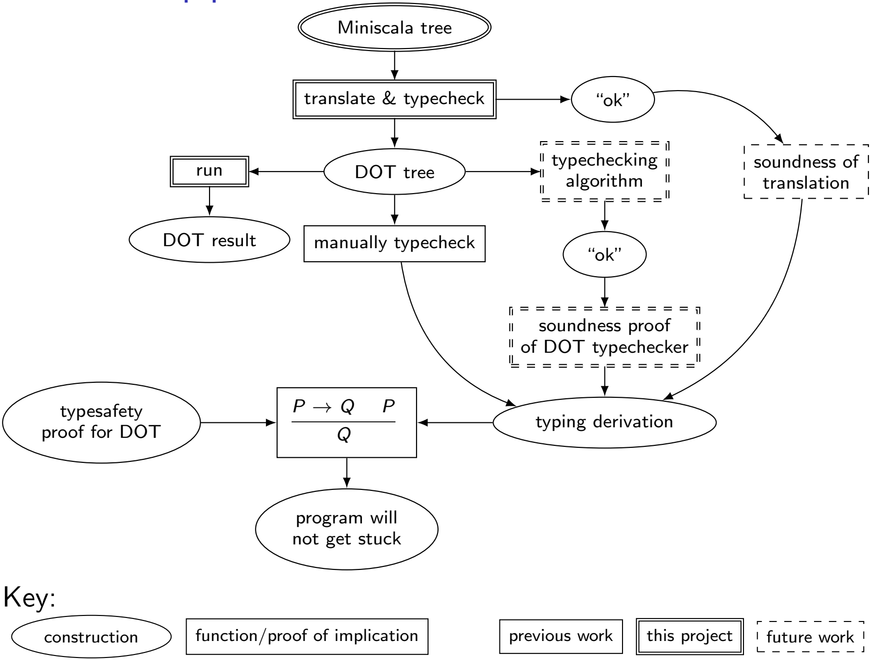
1. Scala vs DOT
2. Miniscala
   - A small Scala class calculus
3. DOT-dependent Miniscala typechecking
   - Defining Miniscala typechecking in terms of DOT typechecking
4. DOT-independent Miniscala typechecking
   - Defining Miniscala typechecking on Miniscala
5. Practical expressivity checking
   - Implementing a Scala-to-DOT translation pipeline
6. Conclusion

Translation pipeline



TODO: add inheritance

Translation pipeline



Conclusion

- Many dimensions to explore
  - Typechecking: only in DOT / also in Miniscala
  - Collecting inherited members: in translation / in target calculus
  - Guarantees: only soundness / also nominality
  - Target Calculus: νDOT/ POT / plain DOT
  - Formalization: on paper / in Scala / in Coq
- Directions for future work:
  - More implementations and proofs
  - Develop POT
  - More features of Scala
  - Ideally: Translation of Scala to DOT could serve as a formal specification of Scala

Questions?

Thank you ☺

additional slides

Report

Trying to encode nominality in DOT

First approach: One module per class

```
let Company : { c =>
  type Inst <: {
    def name(dummy: Top): String
    def address(dummy: Top): String
  }
  def create(dummy: Top): c.Inst
} = new { c =>
  type Inst = {
    def name(dummy: Top): String
    def address(dummy: Top): String
  }
  def create(dummy: Top): c.Inst = new {
    def name(dummy: Top): String = ...
    def address(dummy: Top): String = ...
  }
} in ...
```

Trying to encode nominality in DOT

First approach: One module per class

```
let Company : { c =>
  type Inst <: {
    def name(dummy: Top): String
    def address(dummy: Top): String
  }
  def create(dummy: Top): c.Inst
} = new { c =>
  type Inst = {
    def name(dummy: Top): String
    def address(dummy: Top): String
  }
  def create(dummy: Top): c.Inst = new {
    def name(dummy: Top): String = ...
    def address(dummy: Top): String = ...
  }
} in ...
```

Works fine, except:

- Inside implementation of `Company`, we can accidentally assign an `Employee` to a `Company` field

Trying to encode nominality in DOT

Second approach: Separate “branding” and implementation

```
let branding_Company : { b =>
  type R = {
    def name(dummy: Top): String
    def address(dummy: Top): String
  }
  type C <: b.R
  def brand(x: b.R): b.C
} = new { b =>
  type R = {
    def name(dummy: Top): String
    def address(dummy: Top): String
  }
  type C = b.R
  def brand(x: b.R): b.C = x
} in
let impl_Company = new {
  def create(dummy: Top): branding_Company.C =
    branding_Company.brand(new {
      def name(dummy: Top): String = ...
      def address(dummy: Top): String = ...
    })
} in ...
```

Trying to encode nominality in DOT

Problem: Mutually recursive classes

- Mutually recursive classes have to be members of an outer class with self reference `z`
- References to class become e.g. `z.branding_Company.C`
- Need paths of length > 1
- Let's create POT!

POT: A target calculus with paths of length > 1

Extension of DOT

- Paths:
  $p \quad ::= \quad x \;\mid\; p.l$
- Types:
  $T \quad ::= \quad \dots \;\mid\; p.A$
- Deeply nested object creations by allowing `val` defs

POT is not sound!

```
let a = new { a =>
  val f: { type C: Top..Bot } = new { type C = Top }
  def bad(x1: Top): Bot = let x2: a.f.C = x1 in x2
} in ...
```

Culprit: Subsumption between ascribed type of `val` and actual type

But that's exactly what we need to encode nominality!

Quick fix

Only allow so-called "safe" types to be ascribed to `val` defs in objects

$$\frac{}{\textbf{safe } \top} \qquad \frac{\textbf{safe } R}{\textbf{safe } (\{\textbf{type } A = T\} \wedge R)}$$

$$\frac{\textbf{safe } R}{\textbf{safe } (\{\textbf{def } m(x : T_1) : T_2\} \wedge R)} \qquad \frac{\textbf{safe } R}{\textbf{safe } (\{\textbf{type } A <: T\} \wedge R)}$$

$$\frac{\textbf{safe } R \qquad \textbf{safe } T}{\textbf{safe } (\{\textbf{val } l : T\} \wedge R)} \qquad \frac{\textbf{safe } R}{\textbf{safe } (\{\textbf{type } A >: T\} \wedge R)}$$

The rule for typechecking object creations would then include this check:

$$\frac{\Gamma, x : T \vdash \overline{d} : T \qquad \textbf{safe } T \qquad (\text{labels of } \overline{d} \text{ distinct})}{\Gamma \vdash \textbf{new } \{x \Rightarrow \overline{d}\} : \{x \Rightarrow T\}}$$

TODO: Prove soundness

Translation from Miniscala to POT

Translation from Miniscala to POT

Collecting inherited class members

- νDOT could do it
- POT cannot
- Could create νPOT
- Or just add a Γ to the translation function and do it in the translation

Translation from Miniscala to POT

Collecting inherited class members

- *class expansion* judgment: $\Gamma \vdash p \prec_z \overline{d}$ means that the path p refers to a class whose set of members, including inherited ones, is $\overline{d}$, z being the self reference
- *class lookup* judgment: $\Gamma \vdash \textbf{class } p \textbf{ extends } q\{z \Rightarrow \overline{d}\}$ means that the path p refers to a class whose parent class is the class referred to by path q and that its class body is $\{z \Rightarrow \overline{d}\}$

$$\frac{}{\Gamma \vdash \texttt{AnyRef} \prec_z (\mathit{empty})} \qquad \frac{(x : \textbf{extends } q\{z \Rightarrow \overline{d}\}) \in \Gamma}{\Gamma \vdash \textbf{class } x \textbf{ extends } q\{z \Rightarrow \overline{d}\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \textbf{class } p \textbf{ extends } q\{z \Rightarrow \overline{d_2}\} \\ \Gamma \vdash q \prec_z \overline{d_1}\end{array}}{\Gamma \vdash p \prec_z (\overline{d_1} \triangleleft \overline{d_2})} \qquad \frac{\Gamma \vdash x \prec_x (\overline{d_0}; \textbf{class } l \textbf{ extends } q\{z \Rightarrow \overline{d}\}; \overline{d_1})}{\Gamma \vdash \textbf{class } x.l \textbf{ extends } q\{z \Rightarrow \overline{d}\}}$$

Translation from Miniscala to POT

- Can collect inherited members in translation
- Can enforce nominality in target calculus (POT)
- TODO: Need to insert calls to `brand` where *double vision problem* occurs, as pointed out by Nada
- Otherwise: Goal achieved:
  Given translation $g(\Gamma, t)$, define $f(t) = g(\emptyset, t)$
  Define:
  The Miniscala term t is said to typecheck iff the POT term $f(t)$ typechecks according to the POT rules.

Details:
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-to-POT-rules.md`
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/MutRecInh.scala`
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/MutRecInh-in-POT.txt`