Semester Project Report: Connecting Scala to DOT

Samuel Grütter Supervisor: Nada Amin

EPFL, June 2016

Abstract

Recently, the Dependent Object Calculus (DOT), serving as a foundation of Scala's type system, has been proven sound [1], [2]. However, compared to Scala, it does not have classes nor inheritance, and object creations are typed in a structural instead of a nominal way, which leads to a considerable gap between actual Scala code and its formal model DOT.

This project tries to bridge that gap by first defining *Miniscala*, a formal model of a small subset of Scala featuring classes, inheritance and subtyping, and then defining a translation function from Miniscala to DOT. The main challenges are to collect the inherited members of classes, to encode nominality, and to support mutually recursive classes. To address these challenges, various versions of the translation function are presented and discussed, some of which require a modified version of the target calculus, which leads to two new variants of the DOT calculus: νDOT, an extension which can compose objects, and POT, which allows paths of length greater than 1.

Contents

1 Overview **2**

2 Miniscala: A small Scala class calculus **2**

3 Defining Miniscala typechecking in terms of DOT typechecking **4**
- 3.1 Trying to define a syntactic transformation from Miniscala to DOT 4
- 3.2 νDOT: A target calculus capable of collecting inherited members 5
- 3.3 A syntactic transformation from Miniscala to νDOT 8
- 3.4 Trying to encode nominality in DOT 12
- 3.5 POT: A target calculus with paths of length > 1 14
- 3.6 A context-aware transformation from Miniscala to POT 15

4 DOT-independent Miniscala typechecking **16**
- 4.1 Soundness of the translation 17

5 Practical expressivity checking **17**
- 5.1 Implementing the Miniscala to DOT translation in Scala 17
- 5.2 Typechecking DOT terms in Coq 17

6 Conclusion **18**

1 Overview

Miniscala is a subset of Scala featuring classes, inheritance and subtyping. Section 2 presents the grammar of the Miniscala calculus, and gives an informal presentation of its semantics.

To define when a Miniscala term typechecks, there are two approaches: The first is to translate the Miniscala term to DOT, and to say that the Miniscala term typechecks if and only if its translation to DOT typechecks. This approach is developed in section 3.

The second approach is to define separate typechecking rules in Miniscala, and to relate them to the typechecking rules of DOT. This approach is pursued in section 4.

Section 5 presents an implementation of a pipeline starting with actual Scala code, translating it to the formal DOT model in Coq, and allowing to typecheck and run the code in Coq.

Section 6 reviews and compares the various translations, and concludes.

2 Miniscala: A small Scala class calculus

Miniscala is a subset of Scala featuring classes, inheritance and subtyping. It has three kinds of definitions (**class**, **def** and **val**), which may occur as members in class bodies, but also as statements inside terms.

Its grammar is defined as follows:

x, y, z, l, m	identifier (used as variables and labels)
$p ::=$	path (only used to refer to classes)
$\quad x$	variable
$\quad x.l$	selection (prefix has to be a variable, not a path)
$\quad$ `AnyRef`	root of the class hierarchy
$t ::=$	term
$\quad x$	variable
$\quad t.m(t)$	method call
$\quad$ **new** p	create new instance of p
$\quad d; t$	a definition followed by a term
$\quad$ *<abstract>*	placeholder for the body of abstract methods
$d ::=$	definition
$\quad$ **val** $l : T = t$	value definition
$\quad$ **def** $m(x : S) : T = t$	method definition
$\quad$ **class** l **extends** p $\{z \Rightarrow \overline{d}\}$	class definition
$S, T, U ::=$	type
$\quad p$	path referring to class (the only type user programs can contain)
$\quad$ **extends** p $\{z \Rightarrow \overline{d}\}$	type of a path referring to a class definition

A Miniscala program is just a term t. If we want to compile and run a miniscala program t with the Scala compiler, we can surround it with the following code: [1]

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

This will print the result of evaluating t. Note that t cannot access `args`, nor can it invoke `println` itself.

[1] We will also have to remove the "= *<abstract>*" body of abstract methods, and add the **abstract** modifier to the classes containing such methods.

We will use the following example, which encodes natural numbers in an object oriented way, to explain some details about the design of Miniscala.

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

The sequence operator ";" is right-associative, that is, $d_1; d_2; t$ is interpreted as $(d_1; (d_2; t))$. The name defined by d_1 is visible in the definition of d_2 and in t, but the name defined in d_2 is *not* visible in the definition of d_1. This is a restriction compared to full Scala, where **def**s and **class**es defined in the same expression block all can see each other.

Or, to illustrate it in terms of the example, the (empty) body of **class** *Unit* cannot see the **class** *Nats*, but the body of **class** *Nats* can see the **class** *Unit*.

If we want to make mutually recursive definitions, we have to put them into the body of a class, and then they can refer to each other via the self reference z. Like this, **class** *Nat* and **class** *Succ* can refer to each other.

In order to use *Nat*, *Zero*, and *Succ*, we have to create an instance of their surrounding **class** *Nats* (that we call *nats*), and then we can refer to the classes using eg *nats.Zero*.

Note that all methods have to take exactly one argument. If we do not want to provide any argument, we can pass any dummy value, or we can create a **class** *Unit* and pass instances of it.

To pass several arguments to a method, we have to use currying, or we have to define and use tuples.

Note that we can define classes inside every expression. For instance, the *succ* method of **class** Nat defines a local class S, and thanks to subtyping, the **new** S can be upcast to $nats.Succ$ and then to $nats.Nat$ to match the desired return type.

The typechecking rules and the evaluation rules of Miniscala are not yet given here: Their definition is the subject of the two following sections.

3 Defining Miniscala typechecking in terms of DOT typechecking

The DOT calculus has an expressive and well-understood type system, and it is claimed to be "sufficiently expressive to encode standard abstraction patterns we expect in programming languages today" [1].

So, it should be possible to define a function f which takes a Miniscala term and translates it to a DOT term, and to define Miniscala typechecking as follows: The Miniscala term t is said to typecheck iff the DOT term $f(t)$ typechecks according to the DOT rules.

Defining such a translation function f is the subject of this section.

3.1 Trying to define a syntactic transformation from Miniscala to DOT

If we try to define a recursive translation function whose only argument is a Miniscala term, we have to collect all inherited members of a class at some point, because DOT does not have any notion of inheritance. That is, when the translation function emits a **new** in DOT, it must know the list of all members to put into the newly allocated object, because there is no way to compose objects in DOT.

Obtaining such a list of all inherited members can be suprisingly complex, as the following example illustrates:

```
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
```

When we want to create an instance of class `C`, we first find its member `fun2`. But to find the inherited members, we have to resolve its parent class `b.I`, so we first have to know that `b` refers to an instance of class `B`, and then know that `B` has an inner class `I` because it inherited this inner class from its parent class `A`, and only then we find that `fun1` has also to be a member of the `C` that we are going to create.

All this logic of collecting the members of a class would have to be implemented in the translation function, so it seems that this would make the translation function very complex.

Or to say it differently: The translation function cannot just be a syntactic transformation, but it has to be aware of some semantics: For instance, if it sees a path, it has to know what class the path refers to.

Now, since DOT already does deal with semantics, it seems more appropriate to modify DOT such that all everything related to semantics is done in DOT, and the translation is a purely syntactic transformation. This modified version of DOT is called νDOT, and is presented in the next subsection.

3.2 νDOT: A target calculus capable of collecting inherited members

The νDOT calculus is an extension of the DOT calculus as presented in [2]. Its goal is to be able to model the composition of objects in a way which is flexible enough so that Miniscala's inheritance can be encoded in it.

This is achieved by adding *class templates* to DOT, inspired by the class templates of the *νObj* caluclus [3].

3.2.1 Syntax

The νDOT grammar is defined as follows:

x, y, z	variable
m	method label
A, B	type label
$t ::=$	term
$\quad x$	$\quad$ variable
$\quad t.m(t)$	$\quad$ method call
$\quad \textbf{new } t$	$\quad$ create new instance (t must evaluate to a class template)
$\quad [x : S \mid \overline{d}]$	$\quad$ class template with self reference x of type S
$\quad t \ \& \ t$	$\quad$ the *mix* operator
$d ::=$	definition
$\quad \textbf{def } m(x : S) : T = t$	$\quad$ method definition
$\quad \textbf{type } A = T$	$\quad$ type alias definition
$S, T, U ::=$	type
$\quad \top$	$\quad$ top type
$\quad \bot$	$\quad$ bottom type
$\quad T \wedge T$	$\quad$ intersection type
$\quad T \vee T$	$\quad$ union type
$\quad \{\textbf{def } m(x : S) : T\}$	$\quad$ record type with one method
$\quad \{A : S..U\}$	$\quad$ record type with one type member with lower and upper bound
$\quad x.A$	$\quad$ selection type
$\quad \{z \Rightarrow T\}$	$\quad$ bind type
$\quad [x : S \mid T]$	$\quad$ type of a class template
$v ::=$	value
$\quad \{z \Rightarrow \overline{d}\}$	$\quad$ object
$\quad [x : S \mid \overline{d}]$	$\quad$ class template
$s ::= \overline{x = v}$	store
$\Gamma ::= \overline{x : T}$	typing context

Compared to DOT, there are four main differences in the syntax:

Class templates The term $[x : S \mid \overline{d}]$ defines a class template with members $\overline{d}$. They may contain the self reference x, and assume it has type S. The type S has to be an intersection of record types,

and it should mention all members that $\overline{d}$ need to typecheck. Note that this might include members which do not occur in $\overline{d}$, and that not all members in $\overline{d}$ need to be mentioned in S. Class templates are first-class constructs, that is, they can be passed as arguments to methods. This feature is not strictly necessary, but it just turned out that the calculus is simpler and more regular if we have it. Another way to look at class templates is to see them as "partially constructed objects", which can be turned into "real objects" using the **new** operator.

New object creation To construct a new object, we have to provide a term t which evaluates to a class template, and the **new** operator will create an object containing all members defined in the class template.

The *mix* operator The term t_1 & t_2 takes two terms evaluating to a class template, and combines them into one class template whose self type is the intersection of the two self types, and whose members are those of t_1, overridden with those of t_2.

The type of class templates Since class templates are terms, we must be able to assign them a type, so we have to introduce the type $[x : S|T]$, which is the type of the class template $[x : S|\overline{d}]$, if $\overline{d}$ has type T. As we can see, T will always be an intersection of record types.

3.2.2 Evaluation rules

We give small-step evaluation rules mapping a store s and a term t to a new store s' and a new term t', written as $s|t \;\rightarrow\; s'|t'$ (note that the meaning of "|" is overloaded: It is used to separate the self type from definitions inside class templates, but also to separate the store from the term):

$$
\begin{array}{rcll}
s|\textbf{new } x & \rightarrow & s, y = \{z \Rightarrow \overline{d}\}|y & \text{where } y \text{ is fresh, and } (x = [z : S|\overline{d}]) \in s \\
s|[z : S|\overline{d}] & \rightarrow & s, y = [z : S|\overline{d}]|y & \text{where } y \text{ is fresh} \\
s|x_1 \;\&\; x_2 & \rightarrow & s|[z : S_1 \wedge S_2|\overline{d_1} \triangleleft \overline{d_2}] & \text{where } (x_i = [z : S_i|\overline{d_i}]) \in s \text{ for } i = 1, 2 \\
s|y.m(x) & \rightarrow & s|[x/z]t & \text{where } (y = \{y \Rightarrow \overline{d}; \textbf{def } m(z : S) : T = t\}) \in s \\
s|e[t] & \rightarrow & s'|e[t'] & \text{if } s|t \;\rightarrow\; s'|t'
\end{array}
$$

The last rule uses an evaluation context e, which is defined by the grammar

$$e ::= [] \mid e.m(t) \mid x.m(e) \mid \textbf{new } e \mid e \;\&\; t \mid x \;\&\; e$$

The $\triangleleft$ operator is used to do overriding concatenation: $\overline{d_1} \triangleleft \overline{d_2}$ appends all definitions of $\overline{d_2}$ to $\overline{d_1}$, removing those in $\overline{d_1}$ whose label also occures in $\overline{d_2}$.

Note that the syntactic form v is only used to describe what can be put into the store, and is not part of the syntax for terms. So the actual result of a computation will not be a v, but a variable x referring to some result in the store.

3.2.3 Typechecking rules

We only give the typechecking rules which are different or new compared to DOT.

First, notice that the subtyping rules are not changed at all compared to DOT, so there is no subtyping for the type of class templates. The only changed rules are those for type assignment.

Typechecking class templates is straightforward:

$$\frac{\Gamma, x : S \vdash \overline{d} : T}{\Gamma \vdash [x : S|\overline{d}] : [x : S|T]} \quad \text{(T-TPL)}$$

Note that we do not perform any checks on the self type S. In particular, it might contain a type member $\{A : \top..\bot\}$, so when typechecking $\overline{d}$, we can use subtyping transitivity to derive $\top <: \bot$ from $\top <: x.A$ and $x.A <: \bot$, so we can typecheck any arbitrary term.

Interestingly, though, this is not a soundness problem, because we will check the self type whenever we create a new object, so no objects can be created from such bad class templates, so the code which was typechecked under the assumption $\top <: \bot$ will never be executed.

The rule for the mix operator is the following:

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

T_1 and T_2 are intersections of record types, and the $\triangleleft$ operator is defined the same way as it is defined for sequences of definitions: $T_1 \triangleleft T_2$ is the type with all members of T_1 and T_2, but we remove those from T_1 whose label also occurs in T_2.

To typecheck the creation of new objects, we have the following rule:

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

Note that the type T is the *precise* type of the declarations in the class template t, because there is no occasion for subsumption to be applied: t might be the mixing of several class templates, but all of these were originally typechecked with T-TPL, and T-TPL invokes declaration typing, which does not have any subsumption either, and there's no subtyping for the type of class templates. So we can conclude that the type T is an intersection of record types, and that all its type members are type aliases. So, putting T into the environment to do the subtyping check is safe in the sense that we will not put any "bad bounds" (abstract type members whose lower bound is not a subtype of the upper bound) into the environment.

Moreover, S is the intersection of all self types which were put into the typing context to typecheck the definitions of the class template t, and by checking that $T <: S$, we ensure that all these self types have good bounds, because if a type alias $\{A : S_1..S_1\}$ is a subtype of an abstract type member $\{A : S_0..S_2\}$, we know that this abstract type member has good bounds (i.e. $S_0 <: S_2$), because $S_0 <: S_1 <: S_2$.

And finally, checking that $T <: S$ also ensures that the self types that were assumed when typechecking the definitions in the class template t are satisfied by T.

So, in summary, the T-NEW rule 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.

3.2.4 Type safety for νDOT

We present a type safety proof for νDOT mechanized in Coq. Its source can be found at
`https://github.com/TiarkRompf/minidot/blob/nuDOT/dev2016/nuDOT.v`.
It is based on the typesafety proof for DOT, which can be found at
`https://github.com/TiarkRompf/minidot/blob/a8fa8f5/dev2016/dot.v`.

The proof follows the DOT proof closely. Some lemmas about the behavior of the overriding concatenation operator had to be proven. The central one is the following:

Lemma 3.1 (Behavior of $\triangleleft$ with respect to typing)**.** *If* $\Gamma \vdash \overline{d_1} : T_1$ *and* $\Gamma \vdash \overline{d_2} : T_2$*, then* $\Gamma \vdash \overline{d_1} \triangleleft \overline{d_2} : T_1 \triangleleft T_2$*.*

Proof. By induction on the maximum number of members in $\overline{d_1}$ and $\overline{d_2}$. □

On paper, this is an easy proof, but in Coq, it required a well chosen definition of the $\triangleleft$ operator and some well chosen helper lemmas.

Next, we prove that the evaluation relation is unique, i.e. that for each $s|t$ pair, there is at most one $s'|t'$ pair such that $s|t \rightarrow s'|t'$:

Lemma 3.2 (Uniqueness of evaluation)**.** *If* $s|t \rightarrow s_1|t_1$ *and* $s|t \rightarrow s_2|t_2$*, then* $s_1 = s_2$ *and* $t_1 = t_2$*.*

Proof. By induction on the $s|t \rightarrow s_1|t_1$ hypothesis. □

This simplifies the type safety statement: Instead of stating that there exists a term t' to which t can step and that moreover, typing is preserved for all t' to which t can step, it suffices to state that there exists a t' to which t can step and for which typing is preserved. To state the type safety theorem, we extend the typing and subtyping judgment to include a store s in the same manner[2] as in [2].

Theorem 3.3 (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* $s'\ \emptyset \vdash t' : T$*.*

Proof. By induction on the $\Gamma \vdash t : T$ hypothesis. □

3.3 A syntactic transformation from Miniscala to νDOT

Now that we have a calculus which can compose objects (using the *mix* operator), we can embed Miniscala in it with a purely syntactic transformation.
The full transformation is defined at
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-to-nuDOT-rules.md`,
and a complete sample Miniscala program is at
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/NumbersWithCtor.scala`,
whose translation into νDOT can be found at
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/NumbersWithCtor-in-nuDOT.txt`.

In this report, instead of printing the precise but lengthy transformation rules, we rather explain them by a sample program snippet.

[2]Note, though, that we use the letter s instead of ρ to denote the store.

3.3.1 Constructor arguments

For each class (even for abstract classes), the translation will create a constructor method returning a νDOT class template. Since all methods in νDOT take exactly one argument, we'd have to pass some ignored dummy object to these constructors. But it turns out that we can also pass it a constructor argument, and add constructor arguments to Miniscala, to obtain a slightly more powerful calculus. So each constructor in this extended Miniscala calculus takes exactly one argument, *except* the constructor for `AnyRef`. Having at least one class which can be instantiated without providing a constructor argument is crucial, because otherwise we could not create any objects, since each object creation would require the existence of another object created earlier, but we would not have any object to start with, so we would have a calculus which cannot create any objects.

The precise definition of Miniscala with constructor arguments is given at the beginning of the translation description file referenced above.

3.3.2 Translating classes which are members of another class

Let us first consider the case where a class is a member of another class, and see how such an inner class is translated: For each class member with label l, the translation will create three members: A type alias $Inst\_l$ for the type of instances of that class, a type alias Tpl_l for the type of the class template, and a constructor method tpl_l returning a class template.

To avoid conflicts between these generated label names and the existing labels in the source Miniscala program, we suppose that the source program does not have any names containing underscores.

To have a running example, suppose we want to translate the class `Succ`, which represents natural numbers ≥ 1, and has an unnecessarily complicated `id` function returning an instance representing the same value as itself:

```
class Succ(pr: nats.Nat) extends nats.Nat(unit) { n ⇒
  def pred(dummy: AnyRef): nats.Nat = pr
  def plus(other: nats.Nat): nats.Nat = pr.plus(other).succ(unit)
  def id(dummy: AnyRef): nats.Nat = n.succ(unit).pred(unit)
}
```

It is defined as a member of an outer class `Nats`, which defines a library for natural numbers, and whose self reference is `nats`. `Succ` extends an abstract class `Nat`, which has one abstract and one concrete method:

```
abstract class Nat(dummy: AnyRef) extends AnyRef { n ⇒
  def plus(other: nats.Nat): nats.Nat
  def succ(dummy: AnyRef): nats.Succ = new nats.Succ(n)
}
```

For the class `Nat`, the following three νDOT definitions are created:

A type alias for the type of instances The type alias for instances of this class is called `Inst_Succ`, and it lists all members defined in the class `Succ`, and intersects them with the inherited members, i.e. with `nats.Inst_Nat`:

```
type Inst_Succ = { n ⇒ nats.Inst_Nat /\
  { def pred(dummy: AnyRef): nats.Nat } /\
  { def plus(other: nats.Nat): nats.Nat } /\
```

```
  { def id(dummy: AnyRef): nats.Nat }
}
```

A type alias for the type of class templates Class templates for this type will have the following type:

```
type Tpl_Succ = [ n : nats.Inst_Succ |
  { def pred(dummy: AnyRef): nats.Nat } /\
  { def plus(other: nats.Nat): nats.Nat } /\
  { def id(dummy: AnyRef): nats.Nat }
]
```

Its self type is the previously defined `nats.Inst_Succ`.

A constructor method returning a class template To construct an instance of type `nats.Succ`, we have to call the constructor defined below using `new nats.tpl_Succ(somePred)`. This constructor returns a template, so we can use it to either mix it with another object created by a subclass, or we can use `new` to turn the class template into a real object.

```
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 = pr.plus(other).succ(unit)
    def id(dummy: AnyRef): nats.Nat = n.succ(unit).pred(unit)
  ]
```

The `tpl_Succ` constructor invokes the parent constructor `nats.tpl_Nat`, and uses the mix operator to combine the partially constructed `Nat` (containing only the concrete `succ` method) with the additional methods `pred`, `plus`, and `id`.

Note that since we specify `nats.Inst_Succ` as self type, which is a subtype of `nats.Nat`, we know that the self reference `n` has a method called `succ`, so the `id` method can call that method, even though it is not mentioned in the body of this `tpl_Succ` constructor.

3.3.3 Translating classes which are defined inside terms

Let us now consider the case where the class is defined inside a term, as in the following example:

```
class Nats(dummy: AnyRef) extends AnyRef { nats ⇒
  ...
};
val nats: Nats = new Nats(unit);
(new nats.Zero(unit)).succ(unit).succ(unit)
```

We also have to create the three definitions described in the previous subsection, but we have to put them into a wrapper object that we name `Nats`, because in νDOT, type members cannot be assigned to let-bound variables.

```
let Nats = new
  [ Nats: { Nats ⇒
    type Inst = ...
    type Tpl = ...
    def tpl(dummy: Top): Nats.Tpl }
```

```
    | type Inst = ...
      type Tpl = ...
      def tpl(dummy: Top): Nats.Tpl = ...
    ]
  in ...
```

Since they are the only members of this wrapper object, we do not need to add a suffix to the names `Inst`, `Tpl`, and `tpl`.

One might wonder why we do not always put these three members into a wrapper object, i.e. also if the original class definition appears inside another class, since this would be more regular. The reason is that we would then need paths of length 2 to refer to them. For instance, if we have an instance of the `Nats` library called `nats`, we would have to write `nats.Succ.Inst` to refer to the instance type, and such path types are not supported by νDOT (nor by DOT); they only support paths of the form `x.L`, where `x` is a variable.

3.3.4 Translating types

Recall that Miniscala types are just paths (of length ≤ 1) or `AnyRef`. So, translating a Miniscala type to νDOT can be done as follows:

- `AnyRef` is translated to $\top$
- Types which are just a variable `x` (referring to a class defined inside a term) are translated to $x.Inst$.
- Types which are a path of the form `x.l` (referring to a class defined inside another class, of which an instance has been stored into the variable `x`) are translated to $x.Inst\_l$.

3.3.5 Translating constructor invocations

Translating constructor calls works similarly:

- `new AnyRef` is translated to **new** $[z : \top|]$
- `new x(t)` is translated to **new** $x.tpl(t')$, where t' is the translation of the term `t`.
- `new x.l(t)` is translated to **new** $x.tpl\_l(t')$, where t' is the translation of the term `t`.

3.3.6 Translating terms

To translate terms, they are traversed recursively, and all the translations described above are applied. We refer to the file referenced above for the full specification of the function.

3.3.7 Discussion

We now have defined a translation function f, as described in the beginning of section 3, which can be used to define Miniscala typechecking as follows: The Miniscala term t is said to typecheck iff the νDOT term $f(t)$ typechecks according to the νDOT rules.

So, it seems that we have achieved our goal stated at the beginning of section 3. However, there's something unsatisfying about it: We do not enforce nominality.

For instance, consider the following program: [3]

[3]Note that it is not strictly in Minsicala syntax to provide better readability, but it could easily be turned into a Miniscala program

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

In Scala, this does not typecheck, because a `Company` cannot be assigned to a variable of type `Employee`. However, if we translate this program to νDOT, the types `Company.Inst` and `Employee.Inst` are subtypes of each other, because they have exactly the same members, so this program would typecheck. So, our above definition of Miniscala typechecking is not satisfying, because it accepts some programs that the Scala compiler does not accept. [4]

So we will investigate in the next section how we can enforce nominality.

3.4 Trying to encode nominality in DOT

Nominality is orthogonal to the object composition capability achieved by νDOT, so let us first study nominality in the simpler calculus DOT, and only later port the results to νDOT.

3.4.1 First approach: One module per class

Let us try to encode the class `Company` from section 3.3.7 in such a way that new instances of class `Company` can only be created with some designated constructor, and not "by accident":

```
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 ...
```

To create instances of type `Company`, we have to use the `Company.create` method, which will return an instance of type `Company.Inst`. There is no other way to create objects of type `Company.Inst`, because the lower bound of `Company.Inst` is $\bot$. The implementation of the constructor `create`, however, can see a more precise type for `Inst`, namely a type alias, which is then upcasted

[4] Note, though, that this is not a soundness issue, because all the required members are present.

by the type ascribed to the variable `Company`, so that from the outside, we do not see an alias any more, but only an abstract type member whose lower bound is $\bot$.

This encoding of nominality seems to work fine, except for one problem: Suppose that in the implementation of the method `address`, we call a method taking as argument a `Company`, i.e., a `c.Inst`. If we pass this method an `Employee` instance instead, it will still typecheck, because we see `c.Inst` as a type alias.

3.4.2 Second approach: Separate "branding" and implementation

To fix this problem, we should separate the "branding" part (i.e. assigning an instance a nominal type, that is, a type whose lower bound is $\bot$) from the implementation of the class.

We can do that as follows:

```
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 ...
```

We use a type alias `R` for a record type alias, and an abstract type member `C` which is lower bounded by $\bot$ when seen from outside the "branding module" `branding_Company`, but which is a type alias when seen from inside, so the method `brand` can just be implemented as the identity function.

The constructor `create` inside `impl_Company` can then invoke this branding method, to turn a newly created object whose type is a structural record type into an object of type `branding_Company.C`, which is lower-bounded by $\bot$ and thus cannot be "faked".

3.4.3 Problem: Mutually recursive classes

The above approach seems to work, except for mutually recursive classes: Suppose we want to add a class `Employee`, and suppose that the `Company` and `Employee` refer to each other. Then, we cannot

create one module after the other as above, but we have to put them inside a common wrapper object, so that they can refer to each other via the self reference of the wrapper object. But then, these references are paths of length 2: If the self reference of the wrapper object is `z`, the references are e.g. `z.branding_Company.C`. Such paths are not supported by DOT, so we have a problem.

A lot of effort has been spent on finding a different encoding which only needs paths of length 1, but no solution was found; there was always a point where a situation similar to the one described above occurred.

This is the motivation to add paths of length > 1 to DOT, which is done in the next section.

3.5 POT: A target calculus with paths of length > 1

The goal of this calculus is to allow path types of length > 1, which is needed to encode mutually recursive nominal classes.

POT stands for "DOT with real paths", and is an extension of DOT obtained by allowing not only types of the form $x.A$, but also $p.A$, where the grammar for paths p is given by $p ::= x \mid p.l$.

Moreover, to make sure such path types can refer to something, we also have to allow the construction of deeply nested objects, which is achieved by allowing `val` definitions in objects.

The upcasting trick used to achieve nominality (section 3.4) requires that we can ascribe a less precise type to `val` definitions, so we have to allow this in POT.

But it turns out that if we combine all these features, the calculus is not sound any more. Consider the following example:

```
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 ...
```

The method `bad` assign the type `Bot` to any object, by first ascribing it as `a.f.C` (whose lower bound it sees is `Top`), and then returning it as a `Bot`, which works because the upper bound of `a.f.C` it sees is `Bot`.

Obviously, the problem is that the ascribed type of `a.f` has bad bounds. So, we should check that all ascribed types of `val` definitions have good bounds. But that does not help, because when checking the body of the object `a` being created, we have to put a hypothetical instance of it into the typechecking environment, so when we check the bounds of `type C`, we can prove `Top` $<:$ `Bot` by transitivity: `Top` $<:$ `a.f.C` $<:$ `Bot`.

One way to regain soundness is to allow only so-called "safe" types to be ascribed to `val` definitions inside objects. We can inductively define a judgment "**safe** T" saying that it is "safe" to have the type T as the self type which is put into the typechecking environment to typecheck object creations, as follows:

$$\frac{}{\textbf{safe } \top} \text{ (SAFE-TOP)} \qquad \frac{\textbf{safe } R}{\textbf{safe } (\{\textbf{type } A = T\} \wedge R)} \text{ (SAFE-ALIAS)}$$

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

$$\frac{\textbf{safe } R \quad \textbf{safe } T}{\textbf{safe } (\{\textbf{val } l : T\} \wedge R)} \text{ (SAFE-FLD)} \qquad \frac{\textbf{safe } R}{\textbf{safe } (\{\textbf{type } A >: T\} \wedge R)} \text{ (SAFE-LB)}$$

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

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

The **safe** judgment only accepts intersections of record types, but since all $\overline{d}$ have an intersection of record types as their types, this is not a restriction. But the interesting restriction is in SAFE-FLD: There, T is the type we ascribed to the right-hand side of the **val** definition, and it is ensured that the type members of this type are all either an alias, or only upper-bounded, or only lower-bounded. This is a simple way to exclude "bad bounds" and to reject the bad example from above.

It seems that with this restriction, the POT calculus should be sound, but this has not (yet) been verified formally.

3.6 A context-aware transformation from Miniscala to POT

Now that we have a calculus in which we can encode nominality, let us use it to encode Miniscala. However, we still have to solve the problem of collecting inherited class members: The POT calculus cannot do that, so we have to do it in the translation function.

Contrary to what was feared in section 3.1, this is quite easy if we allow the translation function to take a typechecking environment Γ as an additional argument. [5]

The translation basically works the same way as described in section 3.4.2, by separating the "branding" component from the implementation component.
For a complete description of the transformation, we refer to
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-to-POT-rules.md`.
A sample Miniscala program is at
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/MutRecInh.scala`,
whose translation into POT can be found at
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/MutRecInh-in-POT.txt`.

In this report, we will focus on explaining how collecting inherited class members works.

3.6.1 Collecting inherited class members

To create constructors, we have to be able to find all members of a given class. To do so, we define a *class expansion* judgment $\Gamma \vdash p \prec_z \overline{d}$, which means that the path p refers to a class whose set of members, including inherited ones, is $\overline{d}$.

It is mutually dependent on a *class lookup* judgment $\Gamma \vdash \textbf{class } p \textbf{ extends } q\{z \Rightarrow \overline{d}\}$, which says 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 $\{z \Rightarrow \overline{d}\}$. Note that here, $\overline{d}$ does *not* include inherited members.

The two judgments can be defined by the following four rules:

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

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

Note that the $\triangleleft$ operator is the same as defined in section 3.2.2.

[5] Note that we do *not* do any typechecking, we just need Γ to collect inherited members.

If we consider Γ and p as an input to these two judgments, and the other arguments as an output, we can use the class expansion judgment as a partial function to collect all members (including inherited ones) of a class during the translation.

3.6.2 Discussion

Now if we call the translation function described in this section $g(\Gamma, t)$, and further define $f(t) = g((empty), t)$, this translation function f satisfies the goals that we have stated in the beginning of section 3: We can use it to define Miniscala typechecking as follows: The Miniscala term t is said to typecheck iff the POT term $f(t)$ typechecks according to the POT rules.

Another approach which could be explored is the following: If we do not like the complexity of collecting inherited class members to be in the translation function, we could combine νDOT and POT into a new calculus called νPOT, and then define a syntactic transformation from Miniscala to νPOT.

Both of these approaches are equally powerful: They can encode nominality, inheritance, and mutually recursive classes, and the complexity of the definitions is similar, with the only difference that one collects inherited class members in the translation function, whereas the other does it in the target calculus.

4 DOT-independent Miniscala typechecking

Defining Miniscala typechecking in terms of DOT typechecking, as we did in section 3, has some drawbacks: First, DOT typechecking is undecidable, because we can embed System $F_{<:}$ in DOT, as shown in [1], and second, the actual typecheckers used in the Scala and dotty compilers operate on Scala code, not on DOT, so it is difficult to compare them to the typechecking rules of DOT.

To address these shortcomings, it makes sense to define typechecking directly on Miniscala terms, because we might be able to restrict Miniscala in such a way that typechecking becomes decidable, and because Miniscala is much closer to actual Scala than DOT.

If the translation needs an environment Γ anyways, it is quite simple to extend it to do also the typechecking. Moreover, if we do typechecking on Miniscala, we do not need to enfore nominality in the target calculus, so the translation becomes even simpler.

So we can define a translation from Miniscala to DOT, which does type assignment and type checking at the same time. We design the rules in bidirectional typechecking style, so that they define an algorithm. We define the following two statements:

- Type assignment with translation: $\Gamma \vdash t \Rightarrow T \rightsquigarrow t'$ meaning that under context Γ, the Miniscala term t is *assigned* the type Miniscala type T, and is translated to the DOT term t'. So Γ and t are inputs, and T and t' are the outputs.
- Type checking with translation: $\Gamma \vdash t \Leftarrow T \rightsquigarrow t'$ meaning that under context Γ, the Miniscala term t *satisfies* the given Miniscala type T, and is translated to the DOT term t'. So Γ, t, and T are inputs, and t' is the output.

The translation rules are similar to the those for the translation to POT (see section 3.6), but we do not need to encode nominality, and we do typechecking additionally.
The complete rules are given in the file
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-to-DOT-rules.md`,
and a sample Miniscala program can be found at
`https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/Booleans.scala`,

whose translation to DOT is at
https://github.com/samuelgruetter/dot-calculus/blob/master/scala/miniscala-examples/Booleans-in-DOT.txt.

4.1 Soundness of the translation

An interesting aspect of this approach is that we can prove soundess of the translation function, that is, if Miniscala typechecking of a term succeeds, then the term translated to DOT should also typecheck in DOT.

Due to time constraints, no formal proof of this statement was developed, but this is definitely an interesting topic for future research.

5 Practical expressivity checking

In this section, we present two implementations which together form a pipeline from actual Scala code to DOT code in Coq, which can be proven to be runnable without getting stuck.

5.1 Implementing the Miniscala to DOT translation in Scala

All translation functions presented in this report so far were described on paper only. So, applying them to an example to see if they work is tedious manual work, since the translations can become quite verbose, as one can see in the examples at
https://github.com/samuelgruetter/dot-calculus/tree/master/scala/miniscala-examples.

Therefore, it makes sense to write down the translations in a runnable format. This was done for a simple version of the Miniscala to DOT translation, without inheritance. The source of the Scala implementation can be found at
https://github.com/samuelgruetter/dot-calculus/tree/master/scala/miniscala1

It works as follows: First, it parses a `.scala` file using the *scalameta* parser. The Miniscala term t to be parsed has to be surrounded by the following code in order to make it a valid Scala program:

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

Next, the scalameta abstract syntax tree (AST) is translated into a Miniscala AST, where variable names are represented as strings. Then, the actual typechecking and translation takes place, producing a DOT AST (defined as Scala case classes), where variable names are still represented as strings. Then, the representation of variables is changed to the *locally nameless* style, and finally, the trees are printed as Coq code, compatible with the DOT version defined in
https://github.com/TiarkRompf/minidot/blob/nuDOT/dev2016/dot_storeless_tidy.v.

5.2 Typechecking DOT terms in Coq

Once the Scala implementation of the previous section gave us a DOT term in Coq, we might want to verify if it typechecks according to the typechecking rules defined in Coq. Doing so by manually writing a proof tree in Coq is very tedious, so it makes sense to automate this. One way would be to do so using Coq's tactic language *Ltac*, but it's an untyped language which is hard to debug and hard to reason about. Another, preferrable way, is to implement a typechecking algorithm in Coq's term language *Gallina*, and to prove that the algorithm is sound with respect to the typing rules.
This was done in the file https://github.com/TiarkRompf/minidot/blob/nuDOT/dev2016/dot_typechecker.v.
Note that at the time of writing, the soundness proof of the typechecking algorithm is not yet completed, but there seem to be no serious obstacles to finish it.

Now, given the translation into Coq of a Scala file, produced by the implementation presented in section 5.1, we can prove that the program will not get stuck before even running it: We just feed it to the typechecking algorithm in Coq, and if it succeeds, we can apply the correctness theorem of the typechecking algorithm to conclude that it typechecks according to the DOT rules, and by applying the DOT soundness theorem, we finally can conclude that the program will not get stuck during execution.

6 Conclusion

We have presented Miniscala, a subset of Scala featuring classes, inheritance and subtyping. It has three kinds of definitions (`class`, `def` and `val`), which may occur as members in class bodies, but also as statements inside expressions.

The following three ways of encoding Miniscala in (some variant of) DOT, referred to as *the target calculus*, were presented:

Name	Description
DOT	"plain" DOT, as presented in `https://github.com/TiarkRompf/minidot/blob/master/dev2016/dot_storeless_tidy.v`
νDOT	DOT with first-order class templates $[x : S\|\overline{d}]$ inspired by the νObj calculus [3] `https://github.com/TiarkRompf/minidot/blob/nuDOT/dev2016/nuDOT.v`
POT	DOT with paths of length > 1, not yet fully proved

The goal of the translation can be two different ones:

- *Goal 1*: To show that all of the typechecking can be done in (some variant of) DOT (section 3).
- *Goal 2*: To design separate typechecking rules for Miniscala, and to relate them to the typechecking rules of DOT (section 4).

The following table summarizes the features, advantages and disadvantages of the three translations:

	to νDOT	to POT	to DOT
Goal	Goal 1	Goal 1	Goal 2
Inheritance	✓	✓	✓
Mutual recursion	✓	✓	✓
Nominality	✗	✓	✓
Constructors can take arguments and use them in the expression passed to the parent constructor and in the class body	✓	(✗)	(✗)
No typechecking context is needed for the translation	✓	✗	✗
Target calculus is an already published DOT version and does not need any adaptations	✗	✗	✓

It is interesting to note that to achieve *Goal 1*, we cannot translate to plain DOT, but we have to translate to a specialized version of DOT.

Moreover, we have presented a Scala implementation of the translation from Scala code to DOT ASTs in Coq code, and a typechecking algorithm for DOT in Coq.

There is a lot of room for future work in these areas: The translations could be formalized in Coq, and their soundness could be proven formally. Moreover, the Miniscala calculus could be extended to contain on one hand features that DOT already contains, such as intersection types and abstract type members, and, on the other hand, to contain more Scala features, such as generics, traits and mixins, access to `super`, or higher kinded types.

Ideally, Miniscala could be extended to contain (almost) all features of Scala, and the translation into DOT could serve as a formal specification of Scala.

References

[1] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. *The Essence of Dependent Object Types*, pages 249–272. Springer International Publishing, Cham, 2016.

[2] N. Amin and T. Rompf. Type soundness for dependent object types (DOT). In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '16, to appear.

[3] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. *ECOOP 2003 – Object-Oriented Programming: 17th European Conference, Darmstadt, Germany, July 21-25, 2003. Proceedings*, chapter A Nominal Theory of Objects with Dependent Types, pages 201–224. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.