A Core Calculus for Scala Type Checking

Vincent Cremet[1], François Garillot[2], Sergueï Lenglet[3], Martin Odersky[1]

[1] École Polytechnique Fédérale de Lausanne
INR Ecublens, 1015 Lausanne, Switzerland
[2] École Normale Supérieure
45 rue d'Ulm, 75230 Paris, France
[3] École Normale Supérieure de Lyon
46 allée d'Italie, 69364 Lyon, France

Abstract. We present a minimal core calculus that captures interesting constructs of the Scala programming language: nested classes, abstract types, mixin composition, and path dependent types. We show that the problems of type assignment and subtyping in this calculus are decidable.

1 Introduction

The programming language Scala proposes a new model for component systems [28]. Components in this model are classes, which can be combined using nesting and mixin composition. Classes can contain abstract types which may be instantiated in subclasses. The Scala component model thus provides a single framework for the construction of objects and modules. Modules are identified with objects, functors with classes, and signatures with traits.

The advantage of this approach is that a single fairly small set of language constructs is sufficient for core programming as well as the definition of components and their composition. Furthermore, the identification of modules and objects provides new ways to formulate standard programming tasks such as the expression problem [14,33,27] and family polymorphism [12,28].

Scala's approach to component modeling is based on three programming language constructs: modular mixin composition, abstract type members, and explicit self-types. All three have been studied in the νObj calculus [25]. A key concept of the νObj calculus, path-dependent types, is also present in Scala. However, some other constructions of νObj do not correspond to Scala language constructs. In particular, νObj has first-class classes which can be passed around as values, but Scala has not.

First-class classes were essential in establishing an encoding of $F_{<:}$ in νObj, which led to a proof of undecidability of νObj by reduction to the same property in $F_{<:}$ [29]. However, since Scala lacks first-class classes, the undecidability result for the calculus does not imply that type checking for the programming language is undecidable.

In this paper, we study the problem of decidability of Scala type checking. We construct (algorithmic) Featherweight Scala, abbreviated FS_{alg}, a minimal core calculus of classes that captures an essential set of features of Scala's type system. Classes can have types, values, methods and other classes as members. Types, methods, and values can be abstract. The calculus is designed to be syntactically a subset of Scala (with the deviation

of explicit self-names, explained below). Its typing rules correspond closely to the ones implemented in the Scala compiler.

An important aim in developping Featherweight Scala was to show that Scala's core type-checking rules are decidable. One particular problem in this respect are cyclic definitions that relate members of different classes. Featherweight Scala allows cyclic references between members of different mixin classes. Because cycles cannot be ruled out by construction, they have to be detected by the type checker. The typing rules achieve this by keeping track of the sets of definitions that have already been visited in a typing proof. This gives the calculus an algorithmic flavor, hence the name FS_{alg}.

All presented deduction rules are syntax-directed and thus lead directly to procedures for subtyping and type assignment. The central result of this paper is that these procedures are algorithms, i.e. that they terminate in each case.

Related Work: Scala's component constructions provide a middle ground between the worlds of object-oriented programming and functional module systems [15,21]. Many of the concepts in both worlds are unified. Mixin composition in Scala borrows from mixin-modules [3,4,16], as well as from the more linear object-oriented mixin composition [6,5]. Components in Scala can be mutually recursive, a property which is also addressed by work on recursive modules [9,23]. Abstract types in Scala are also present in SML style signatures [21,15], and correspond almost exactly to virtual classes in Beta [22].

Later work on virtual classes [11,13] is more general in that classes, and not just types, can be abstract. However, references to abstract types in [13] can only refer to members of an enclosing "self", not to members of an arbitrary path. Several other variations on virtual types [32,17,24] and some alternative proposals [8,31,7,20] have also been researched. Typed class-based calculi for describing Scala's static analysis are described in [2] and [1] but the authors do not address the problem of their decidability.

The focus in the paper is on a minimal calculus that captures the essential features of an existing programming language. In this motivation it follows the work on Featherweight Java [19]. Both calculi model a simple, purely functional core language of objects with fields and methods in a nominal class-based type system. They also take some similar shortcuts in the interest of conciseness. For instance, both assume call-by-name evaluation in order not to have to deal with the thorny initialization issues of their underlying languages. However, the set of more advanced language constructs that are modeled are different in each case. Featherweight Java models type casts, and has extensions that model generics as well as inner classes [18]. Featherweight Scala models inner classes, member type abstraction, as well as path-dependent types. It can also model most of the generic constructs in Featherweight Generic Java [19] via encodings.

The rest of this paper is structured as follows. Section 2 explains Scala's model of abstract types and path dependent types from a programmer's perspective. Section 3 introduces the Featherweight Scala calculus FS_{alg}. Section 4 shows that subtyping and type-assignment are decidable in this calculus. Section 5 relates the obtained results to the situation in the Scala language. Section 6 concludes.

```
trait Any extends { this0 | }
trait Nat extends Any { this0 |
  def isZero(): Boolean
  def pred(): Nat
  trait Succ extends Nat { this1 |
    def isZero(): Boolean = false
    def pred(): Nat = this0
  }
  def succ(): Nat = ( val result = new this0.Succ; result )
  def add(other: Nat): Nat = (
    if (this0.isZero()) other else this0.pred().add(other.succ())
  )
  def subtract(other: Nat): Nat = (
    if (other.isZero()) this0 else this0.pred().subtract(other.pred())
  )
}
val zero = new Nat { this0 |
  def isZero(): Boolean = true
  def pred(): Nat = error("zero.pred")
}
```

Fig. 1. Definition of Peano numbers

2 Programming in Featherweight Scala

Featherweight Scala is a fairly small subset of Scala, but it is expressive enough for one to write meaningful programs in it. In the following, we show how some common classes and programming idioms can be encoded in the calculus.

Peano Numbers. We start with an encoding of Peano numbers, shown in Figure 1. This encoding presents a trait *Nat* with five member methods. Methods *isZero* and *pred* are *abstract*, that is, they lack an implementation in trait *Nat*. By contrast, methods *succ*, *add* and *subtract* are *concrete*. A *trait* in Scala is an abstract class which may be combined with other traits using mixin composition.

References from one member of a *Nat* object to another always go via the "self" reference of the class. The name of the self-reference is given after the opening brace of the class body. In the example above it is *this0*. The name can be freely chosen, but in the examples in this paper we always use *thisN*, where *N* is the nesting level of the enclosing class. As a shorthand notation we sometimes omit the definition of a self-name which is never used in the class body that follows.

Class *Nat* also contains a nested class *Succ* which defines the successor value of the current object *this0*. This class is an extension of *Nat*, which gives concrete definitions for the two abstract members of *Nat*: *isZero* returns always ***false*** and the predecessor method *pred* always returns the self-reference of the outer *Nat* class.

The successor method *succ* simply creates a new instance of the *Succ* class and returns it. Note that Featherweight Scala allows instance creation expressions such as ***new*** *this0.Succ* only as right-hand sides of value definitions; that's why we were forced to define in the body of *succ* an intermediate value *result*. Regular Scala does not have this restriction, and also provides many other shorthands that would make the example more pleasant to write and read.

The final two methods, *add* and *subtract*, define addition and subtraction in terms of the previous three methods. Their implementation uses standard syntax for field selection, method calls, and recursion.

Figure 1 also gives a definition of the *zero* value for Peano numbers. This value is defined as a direct specialization of class *Nat*, which also defines the natural implementations of *Nat*'s abstract methods *isZero* and *pred*. The right-hand side of *zero*'s definition combines a definition of a new anonymous class and a creation of an instance of this class in a single syntactic construct. Alternatively, one could also proceed in two steps, by defining a subclass *Zero* of *Nat*, and then creating an instance value ***val*** *zero* = ***new*** *Zero*.

The preceding example used exclusively the constructs of the Featherweight Scala calculus, with two exceptions: First, we assumed a type *Boolean* with values ***true*** and ***false*** and a *if-then-else* construct. Second, we assumed an *error* function which aborts a program with a given error message.

In the example we have also taken some liberty in presenting top-level definitions for *Nat* and *zero*. By contrast, a Featherweight Scala program is simply an expression, which typically contains embedded definitions for classes and values. To get a complete program which computes *2+2* one could combine the program fragments in Figure 1 as follows:

```
val universe = new { global |
  trait Nat { ... }
}
val zero = new universe.Nat { ... };
val two: Nat = zero.succ().succ();
two.add(two)
```

In this program, references from one top-level definition in *universe* to another would go via the top-level self-reference *global*. Note that according to Featherweight Scala's syntax, the above definition of *two* is invalid, because a local variable can only be initialized with a new instance expression. However, a similar behavior can be obtain with the following simple programming scheme.

```
val two = new { val local: Nat = zero.succ().succ() };
two.local.add(two.local)
```

The syntax used in the examples in this section is also regular Scala, with one exception: Scala does not have a clause { *thisN* | ... } which names the self-reference *thisN* of the enclosing class. Instead, one always uses the reserved word ***this***. Self-references of an outer class *C* can be denoted by prefixing ***this*** with the name of the outer class, e.g., *C.**this***.

```
trait List extends Any { this0 |
  type Elem
  type ListOfElem = List { this1 | type Elem = this0.Elem }
  def isEmpty(): Boolean
  def head(): this0.Elem
  def tail(): this0.ListOfElem
}
trait Nil extends List { this0 |
  def isEmpty(): Boolean = true
  def head(): this0.Elem = error("Nil.head")
  def tail(): this0.ListOfElem = error("Nil.tail")
}
trait Cons extends List { this0 |
  val hd: this0.Elem
  val tl: this0.ListOfElem
  def isEmpty(): Boolean = false
  def head(): this0.Elem = hd
  def tail(): this0.ListOfElem = tl
}
```

Fig. 2. Definition of the List class hierarchy

Lists. As a second example, Figure 2 presents a class hierarchy for lists in Featherweight Scala. There are three classes: A base class *List* and two subclasses *Cons* and *Nil* that define non-empty and empty lists, respectively.

Lists can have arbitrary element types. In the standard Scala library, this is expressed by a parameterized type, but the featherweight version does not have type parameters. Instead, we use *abstract types* to express the genericity of the list abstraction.

The element type of a given list is represented by the type member *Elem* of class *List*. The member is defined as an abstract type in class *List*. Hence, when a *List* object is created, a concrete implementation of this type has to be provided. For instance, the definition

```
val nilOfNat = new Nil { type Elem = Nat }
```

defines a value *nilOfNat* as an empty list with element type *Nat*. The *type alias* **type** *Elem* = *Nat* is used to "fill in" the abstract type member *Elem* that *Nil* inherits from *List*.

The *List* class also contains a type alias which defines *ListOfElem* as a type name for lists whose element type is the same as the element type of the list in question. This alias does not implement an abstract type member in a parent class; it is there only for convenience.

The *List* class also defines a test method *isEmpty*, as well as methods which return the head and tail of a list. Method *head* returns values of type *Elem* whereas *tail* returns values of type *ListOfElem*. All three methods are abstract in class List.

The subclass *Nil* of *List* represents empty lists. It defines method *isEmpty* to return ***true***. Selecting the head or tail of an empty list always results in an error.

The subclass *Cons* of *List* represents non-empty lists. The head and tail of a non-empty list are kept in the fields *hd* and *tl* of class *Cons*. Scala uses ***val*** for a definition of a local value or a field of a class, whereas ***def*** is used for a method definition. Classes in Featherweight Scala do not have constructors; however, one can use member-redefinition to initialize the values of an object. For instance, the following code defines two lists of element type *Nat* which contain the values (*2*) and (*1*, *2*), respectively.

```
val list2 = new Cons { this0 |
    type Elem = Nat
    val hd: Nat = zero.succ().succ()
    val tl: this0.ListOfElem = nilOfNat
}
val list12 = new Cons { this0 |
    type Elem = Nat
    val hd: Nat = zero.succ()
    val tl: this0.ListOfElem = list2
}
```

The *List* example showed how genericity can be encoded using abstract types. In fact, there is a general encoding that lets one encode all forms of parameterized types in Scala into types with abstract members. Details are found in [1].

```
trait Function extends Any { this0 |
    type Dom
    type Range
    def apply(x: this0.Dom): this0.Range
}
val inc = new Function { this0 |
    type Dom = Nat
    type Range = Nat
    def apply(x: this0.Dom): this0.Range = x.succ()
}
```

Fig. 3. Definition of first-class functions

Higher-Order Functions. Featherweight Scala has methods, i.e. function-valued class-members, but it has no function-valued parameters or results. However, it is possible to encode first-class functional values as instances of a standard *Function* class, which is presented in Figure 3. The current Scala implementation uses a similar encoding to map functional values to the JVM.

Class *Function* gives an interface for functions with arbitrary domain and range types. The interface specifies two abstract types *Dom* and *Range* as well as an abstract method *apply* that takes arguments of type *Dom* and that yields results of type *Range*.

A first-class functional value is then a concrete implementation of class *Function*, which gives types for *Dom* and *Range* as well as an implementation for method *apply*. Figure 3 shows as an example a first-class incrementer function *inc* over Peano numbers.

```
trait Mapper extends Any { this0 |
  type A
  type B
  def map(f: Function { type Dom = this0.A; type Range = this0.B },
          xs: List { type Elem = this0.A }): List { type Elem = this0.B } =
    if (xs.isEmpty()) (
      val result = new Nil {
        type Elem = this0.B
      };
      result
    ) else (
      val result = new Cons {
        type Elem = this0.B
        val hd: this0.B = f.apply(xs.head())
        val tl: List { type Elem = this0.B } = this0.map(f, xs.tail())
      };
      result
    )
}
```

Fig. 4. Encoding of the higher-order *map* function

Since first-class functions are objects, they can be passed around like any other value. To apply a first-class function, one simply invokes its apply method (regular Scala defines syntactic sugar so this is done automatically whenever a first-class function value appears in function position in an application).

As an example, Figure 4 presents a *map* function which applies a given argument function to all elements of a given list and returns a list consisting of all the results of these applications. In regular Scala, this function would be defined as follows:

```
def map[A, B](f: A ⇒ B, xs: List[A]): List[B] =
  if (xs.isEmpty) Nil else f(x.head) :: map(f, xs.tail)
```

Since *map* is conceptually a polymorphic method, its encoding in Featherweight Scala makes use of a wrapper class *Mapper* which defines two abstract types A and B, representing the element types of the argument and result lists, respectively.

The *map* method in *Mapper* takes as arguments a function f from type A to type B, and a list xs of element type A. It returns a list of element type B. An application of *map* would be written as follows:

```
val list23: List { type Elem = Nat } = (
    val mapper = new Mapper { type A = Nat; type B = Nat };
    mapper.map(inc, list12)
)
```

This instantiates the *Mapper* class with type *Nat* as the element type of the argument and result list, and invokes the *map* method of the instantiation with *inc* and *list12* as arguments. The expression would return the encoding of the list (*2, 3*).

The example shows that monomorphic functions such as *inc* can be first-class values. However, the construction cannot be generalized to polymorphic functions. The reason is that polymorphic functions like *map* have to be encoded using wrapper classes. Such wrapper classes are first-class values neither in Featherweight nor in regular Scala. By contrast, the νObj calculus has classes as first class values, and therefore can encode polymorphic functional values.

3 The Algorithmic Featherweight Scala Calculus

The FS_{alg} calculus aims at describing some central aspects of the SCALA type system in a simple and formal way. The features whose study has been privileged in this work are: method overriding, mixins, inner classes, virtual types, singleton types and types with member refinements. The calculus does not model objects with state and has no concept of type parameters in classes or methods. Parameterized classes and parameterized class types can be encoded using virtual types and class types with member refinements.

3.1 Syntax

The abstract syntax of FS_{alg} is given in Figure 5. Amongst the names occurring in a program, we distinguish the *variables*, that are used as binders for objects and can be α-conversed, the *value labels*, that designate the members defining a field or a method, and the *type labels*, that designate the members defining a class or a virtual type.

A *member* can be a value field, a method, a type field or a class. A value field is immutable and refers to an object. The types of value fields, method parameters and method results must be given explicitly. Value fields and methods can be either concrete or abstract. A declaration is called abstract if the right-hand side is absent. For instance, a declaration like $\mathtt{val}_n a : T = t$ defines a concrete field a with value t, whereas the field declaration $\mathtt{val}_n a : T$ is abstract. In order to factorize abstract and concrete declarations we use the notation $\mathtt{val}_n a : T\,(= t)^?$. Type fields are also either concrete or abstract, a concrete type field being sometimes called a *type alias*. A class member $\mathtt{trait}_n A\,\mathtt{extends}\,(\overline{T})\,\{\varphi \mid \overline{M}\}$ declares a class A with parents $\overline{T}$ and members $\overline{M}$, where the variable φ denotes the current instance of the class. Class members cannot be abstract.

Every occurrence of a declaration in a program is tagged with a unique integer n. This integer has no computational meaning, it is simply used for detecting cycles during the static analysis.

The *terms* of the calculus are standard. A variable x can represent the current instance of a class, a method parameter, or the name of an object which has been created locally.

Syntax

x, y, z, φ	Variable
a	Value label
A	Type label
$P ::=$	Program
$\quad \{x \mid \overline{M}\ t\}$	
$M, N ::=$	Member decl
$\quad \mathtt{val}_n a : T\ (= t)^?$	Field decl
$\quad \mathtt{def}_n a\ (\overline{y : S}) : T\ (= t)^?$	Method decl
$\quad \mathtt{type}_n A\ (= T)^?$	Type decl
$\quad \mathtt{trait}_n A\ \mathtt{extends}\ (\overline{T})\ \{\varphi \mid \overline{M}\}$	Class decl
$s, t, u ::=$	Term
$\quad x$	Variable
$\quad t.a$	Field selection
$\quad s.a\ (\overline{t})$	Method call
$\quad \mathtt{val}\ x = \mathtt{new}\ T; t$	Object creation

$p ::=$	Path
$\quad x$	Variable
$\quad p.a$	Field selection
$S, T, U ::=$	Type
$\quad p.A$	Type selection
$\quad p.\mathbf{type}$	Singleton type
$\quad (\overline{T})\ \{\varphi \mid \overline{M}\}$	Type signature

Reduction

$$\frac{\mathtt{val}_n a : T = t \in \Sigma(x)}{\Sigma\ ;\ x.a \to \Sigma\ ;\ t} \quad (\textsc{red-value})$$

$$\frac{\mathtt{def}_n a\ (\overline{z : S}) : T = t \in \Sigma(x)}{\Sigma\ ;\ x.a(\overline{y}) \to \Sigma\ ;\ [\overline{y}/\overline{z}]t} \quad (\textsc{red-method})$$

$$\frac{\Sigma \vdash T \prec_x \overline{M}}{\Sigma\ ;\ \mathtt{val}\ x = \mathtt{new}\ T; t \to \Sigma, x : \overline{M}\ ;\ t} \quad (\textsc{red-new})$$

$$\frac{\Sigma\ ;\ t \to \Sigma'\ ;\ t'}{\Sigma\ ;\ e[t] \to \Sigma'\ ;\ e[t']} \quad (\textsc{red-context})$$

Lookup

$$\frac{\forall\, i,\ \Sigma \vdash T_i \prec_\varphi \overline{N_i}}{\Sigma \vdash (\overline{T})\ \{\varphi \mid \overline{M}\} \prec_\varphi (\biguplus_i \overline{N_i}) \uplus \overline{M}} \quad (\textsc{lookup-sig})$$

$$\frac{\mathtt{trait}_n A\ \mathtt{extends}\ (\overline{T})\ \{\varphi \mid \overline{M}\} \in \Sigma(y) \qquad \Sigma \vdash (\overline{T})\ \{\varphi \mid \overline{M}\} \prec_\varphi \overline{N}}{\Sigma \vdash y.A \prec_\varphi \overline{N}} \quad (\textsc{lookup-class})$$

$$\frac{\mathtt{type}_n A = T \in \Sigma(y) \qquad \Sigma \vdash T \prec_\varphi \overline{M}}{\Sigma \vdash y.A \prec_\varphi \overline{M}} \quad (\textsc{lookup-alias})$$

where

$e ::=$ **(term evaluation context)**
- $\langle\rangle$
- $e.a$
- $e.a\ (t)$
- $x.a\ (\overline{s}, e, \overline{u})$
- $\mathtt{val}\ x = \mathtt{new}\ E; t$

$E ::=$ **(type evaluation context)**
- $e.A$
- $(\overline{T}, E, \overline{U})\ \{\varphi \mid \overline{M}\}$

Fig. 5. The FS_{alg} Calculus : Syntax & Reduction

Path Typing

$$\frac{x : T \in \Gamma}{\mathcal{S}, \Gamma \vdash_{path} x : T} \ (\textsc{path-var}) \qquad \frac{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \mathtt{val}_n a : T\,(= t)^?}{\mathcal{S}, \Gamma \vdash_{path} p.a : T} \ (\textsc{path-select})$$

Type Assignment

$$\frac{\mathcal{S}, \Gamma \vdash_{path} p : T}{\mathcal{S}, \Gamma \vdash p : p.\mathbf{type}} \ (\textsc{path}) \qquad \frac{\begin{array}{c}\mathcal{S}, \Gamma \vdash s : S \\ \mathcal{S}, \Gamma \vdash \overline{t} : \overline{T'} \qquad \mathcal{S}, \Gamma \vdash \overline{T'} <: \overline{T} \\ \mathcal{S}, \Gamma \vdash S \ni \mathtt{def}_n a\,(\overline{x : T}) : U\,(= u)^?\end{array}}{\mathcal{S}, \Gamma \vdash s.a\,(\overline{t}) : U} \ (\textsc{method})$$

$$\frac{\mathcal{S}, \Gamma \vdash t : S \qquad t \text{ is not a path} \qquad \mathcal{S}, \Gamma \vdash S \ni \mathtt{val}_n a : T\,(= u)^?}{\mathcal{S}, \Gamma \vdash t.a : T} \ (\textsc{select}) \qquad \frac{\mathcal{S}, \Gamma, x : T \vdash t : S \qquad x \notin \mathrm{fn}(S) \qquad \mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M_c} \qquad \mathcal{S}, \Gamma \vdash T \text{ WF}}{\mathcal{S}, \Gamma \vdash \mathtt{val}\ x = \mathtt{new}\ T; t : S} \ (\textsc{new})$$

Expansion

$$\frac{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \mathtt{trait}_n A\,\mathtt{extends}\,(\overline{T})\,\{\varphi \mid \overline{M}\} \qquad \{n\} \cup \mathcal{S}, \Gamma \vdash (\overline{T})\,\{\varphi \mid \overline{M}\} \prec_\varphi \overline{N} \qquad n \notin \mathcal{S}}{\mathcal{S}, \Gamma \vdash p.A \prec_\varphi \overline{N}} \ (\prec\textsc{-class})$$

$$\frac{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \mathtt{type}_n A = T \qquad \{n\} \cup \mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M} \qquad n \notin \mathcal{S}}{\mathcal{S}, \Gamma \vdash p.A \prec_\varphi \overline{M}} \ (\prec\textsc{-type})$$

$$\frac{\forall i,\ \mathcal{S}, \Gamma \vdash T_i \prec_\varphi \overline{N_i}}{\mathcal{S}, \Gamma \vdash (\overline{T})\,\{\varphi \mid \overline{M}\} \prec_\varphi (\biguplus_i \overline{N_i}) \uplus \overline{M}} \ (\prec\textsc{-signature})$$

Membership

$$\frac{\mathcal{S}, \Gamma \vdash p \simeq q \qquad \mathcal{S}, \Gamma \vdash_{path} q : T \qquad \psi(p) \cup \mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M} \qquad \psi(p) \not\subseteq \mathcal{S}}{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni [p/\varphi]M_i} \ (\ni\textsc{-singleton})$$

$$\frac{T \text{ is not a singleton type} \qquad \mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M} \qquad \varphi \notin \mathrm{fn}(M_i)}{\mathcal{S}, \Gamma \vdash T \ni M_i} \ (\ni\textsc{-other})$$

Fig. 6. The FS_{alg} Calculus : Type Assignment, Expansion & Membership

Field selections and method calls have the same syntax as in JAVA. The construct `val` $x =$ `new` $T; t$ allows the user to define a new instance of the type T, with a name x whose scope is limited to the term t.

A FS_{alg} *program* is simply a term, which is usually of the form

$$\texttt{val}\ z = \texttt{new}\ \left\{\varphi \,|\, \overline{M}\right\}; t\ .$$

It consists of a list of member declarations $\overline{M}$ that together make up a *universe* object z and a main term t to be evaluated in the context of z. The variable φ is an alias of z; it represents the self reference of the universe object which contains all top-level declarations.

Well-Formedness

$$\frac{\mathcal{S}, \Gamma \vdash_{path} p : T \qquad \psi(p) \not\subseteq \mathcal{S} \qquad \psi(p) \cup \mathcal{S}, \Gamma \vdash T \text{ WF}}{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \text{ WF}} \text{ (WF-SINGLETON)}$$

$$\frac{\mathcal{S}, \Gamma, \varphi : \left(\overline{T}\right) \left\{\varphi \,|\, \overline{M}\right\} \vdash \left(\overline{T}\right) \left\{\varphi \,|\, \overline{M}\right\} \text{ WF}_{\varphi}}{\mathcal{S}, \Gamma \vdash \left(\overline{T}\right) \left\{\varphi \,|\, \overline{M}\right\} \text{ WF}} \text{ (WF-SIGNATURE)}$$

$$\frac{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \texttt{trait}_n A \,\texttt{extends}\, \left(\overline{T}\right) \left\{\varphi \,|\, \overline{M}\right\}}{\mathcal{S}, \Gamma \vdash p.A \text{ WF}} \text{ (WF-CLASS)}$$

$$\frac{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \texttt{type}_n A \left(= T\right)^{?} \qquad \left(\{n\} \cup \mathcal{S}, \Gamma \vdash T \text{ WF} \qquad n \notin \mathcal{S}\right)^{?}}{\mathcal{S}, \Gamma \vdash p.A \text{ WF}} \text{ (WF-TYPE)}$$

MemberWell-Formedness

$$\frac{\left(\mathcal{S}, \Gamma \vdash T \text{ WF}\right)^{?}}{\mathcal{S}, \Gamma \vdash \texttt{type}_n A \left(= T\right)^{?} \text{ WF}_x} \text{ (WF-X-TYPE)}$$

$$\frac{\mathcal{S}, \Gamma, \varphi : x.A \vdash \left(\overline{T}\right) \left\{\varphi \,|\, \overline{M}\right\} \text{ WF}_{\varphi}}{\mathcal{S}, \Gamma \vdash \texttt{trait}_n A \,\texttt{extends}\, \left(\overline{T}\right) \left\{\varphi \,|\, \overline{M}\right\} \text{ WF}_x} \text{ (WF-X-CLASS)}$$

$$\frac{\mathcal{S}, \Gamma \vdash T \text{ WF} \qquad \left(\mathcal{S}, \Gamma \vdash t : T' \qquad \mathcal{S}, \Gamma \vdash T' <: T\right)^{?}}{\mathcal{S}, \Gamma \vdash \texttt{val}_n a : T \left(= t\right)^{?} \text{ WF}_x} \text{ (WF-X-FIELD)}$$

$$\frac{\mathcal{S}, \Gamma \vdash \overline{S}, T \text{ WF} \qquad \overline{S} \text{ does not contain singleton types} \qquad \left(\mathcal{S}, \Gamma, \overline{x : S} \vdash t : T' \qquad \mathcal{S}, \Gamma \vdash T' <: T\right)^{?}}{\mathcal{S}, \Gamma \vdash \texttt{def}_n a \left(\overline{x : S}\right) : T \left(= t\right)^{?} \text{ WF}_x} \text{ (WF-X-METHOD)}$$

$$\frac{\forall i,\ \mathcal{S}, \Gamma \vdash T_i \prec_{\varphi} \overline{N_i} \qquad \mathcal{S}, \Gamma \vdash \overline{M} \text{ WF}_{\varphi} \qquad \mathcal{S}, \Gamma \vdash \overline{T} \text{ WF} \qquad \forall\ (i,j),\ \mathcal{S}, \Gamma \vdash (\overline{N_{i+j}}, \overline{M}) \ll \overline{N_i}}{\mathcal{S}, \Gamma \vdash \left(\overline{T}\right) \left\{\varphi \,|\, \overline{M}\right\} \text{ WF}_{\varphi}} \text{ (WF-X-SIGNATURE)}$$

Path Alias Expansion

$$\frac{\mathcal{S}, \Gamma \vdash_{path} p : q.\mathbf{type} \qquad \psi(p) \cup \mathcal{S}, \Gamma \vdash q \simeq q' \qquad \psi(p) \not\subseteq \mathcal{S}}{\mathcal{S}, \Gamma \vdash p \simeq q'} \text{ ($\simeq$-Step)}$$

$$\frac{\mathcal{S}, \Gamma \vdash_{path} p : T \qquad T \text{ is not a singleton type}}{\mathcal{S}, \Gamma \vdash p \simeq p} \text{ ($\simeq$-Refl)}$$

Fig. 7. The FS_{alg} Calculus : Well-Formedness and Path Alias Expansion

Type Alias Expansion

$$\frac{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \mathtt{type}_n A = T \qquad \{n\} \cup \mathcal{S}, \Gamma \vdash T \simeq U \qquad n \notin \mathcal{S}}{\mathcal{S}, \Gamma \vdash p.A \simeq U} \ (\simeq\text{-TYPE})$$

$$\frac{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \mathtt{trait}_n A \,\mathtt{extends}\, (\overline{T})\ \{\varphi \mid \overline{M}\}}{\mathcal{S}, \Gamma \vdash p.A \simeq p.A} \ (\simeq\text{-CLASS})$$

$$\mathcal{S}, \Gamma \vdash (\overline{T})\ \{\varphi \mid \overline{M}\} \simeq (\overline{T})\ \{\varphi \mid \overline{M}\} \ (\simeq\text{-SIGNATURE})$$

$$\frac{\mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \mathtt{type}_n A}{\mathcal{S}, \Gamma \vdash p.A \simeq p.A} \ (\simeq\text{-ABSTYPE})$$

$$\mathcal{S}, \Gamma \vdash p.\mathbf{type} \simeq p.\mathbf{type} \ (\simeq\text{-SINGLETON})$$

Algorithmic Subtyping

$$\frac{\mathcal{S}, \Gamma \vdash T \simeq T' \qquad \mathcal{S}, \Gamma \vdash U \simeq U' \qquad \mathcal{S}, \Gamma \vdash_* T' <: U'}{\mathcal{S}, \Gamma \vdash T <: U} \ (<:\text{-UNALIAS})$$

$$\frac{A \neq A' \qquad \{n\} \cup \mathcal{S}, \Gamma \vdash T_i <: p'.A' \qquad n \notin \mathcal{S} \qquad \mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \mathtt{trait}_n A \,\mathtt{extends}\, (\overline{T})\ \{\varphi \mid \overline{M}\}}{\mathcal{S}, \Gamma \vdash_* p.A <: p'.A'} \ (<:\text{-CLASS})$$

$$\frac{\mathcal{S}, \Gamma \vdash p \simeq p' \qquad \mathcal{S}, \Gamma \vdash q \simeq p'}{\mathcal{S}, \Gamma \vdash_* p.\mathbf{type} <: q.\mathbf{type}} \ (<:\text{-SINGLETON-RIGHT})$$

$$\frac{\mathcal{S}, \Gamma \vdash T_i <: p.A}{\mathcal{S}, \Gamma \vdash_* (\overline{T})\ \{\varphi \mid \overline{M}\} <: p.A} \ (<:\text{-SIG-LEFT})$$

$$\frac{U \text{ is not a singleton type} \qquad \mathcal{S}, \Gamma \vdash p \simeq q \qquad \mathcal{S}, \Gamma \vdash_{path} q : T \qquad \mathcal{S}, \Gamma \vdash T <: U}{\mathcal{S}, \Gamma \vdash_* p.\mathbf{type} <: U} \ (<:\text{-SINGLETON-LEFT})$$

$$\frac{\mathcal{S}, \Gamma \vdash p \simeq p' \qquad \mathcal{S}, \Gamma \vdash q \simeq p'}{\mathcal{S}, \Gamma \vdash_* p.A <: q.A} \ (<:\text{-PATHS})$$

$$\frac{T \text{ is not a singleton type} \qquad \forall i,\ \mathcal{S}, \Gamma \vdash T <: T_i \qquad \mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{N} \qquad \mathrm{dom}(\overline{M}) \subseteq \mathrm{dom}(\overline{N}) \qquad \mathcal{S}, \Gamma, \varphi : (\overline{T})\ \{\varphi \mid \overline{M}\} \vdash \overline{N} \ll \overline{M}}{\mathcal{S}, \Gamma \vdash_* T <: (\overline{T})\ \{\varphi \mid \overline{M}\}} \ (<:\text{-SIG-RIGHT})$$

Member Subtyping

$$\frac{\mathcal{S}, \Gamma \vdash T <: T'}{\mathcal{S}, \Gamma \vdash \mathtt{val}_n a : T\,(= t)^? <: \mathtt{val}_m a : T'\,(= t')^?} \ (<:\text{-MEMBER-FIELD})$$

$$\mathcal{S}, \Gamma \vdash \mathtt{type}_n A = T <: \mathtt{type}_n A\,(= T)^? \ (<:\text{-MEMBER-TYPE})$$

$$\mathcal{S}, \Gamma \vdash \mathtt{trait}_n A \,\mathtt{extends}\, (\overline{T})\ \{\varphi \mid \overline{M}\} <: \mathtt{trait}_n A \,\mathtt{extends}\, (\overline{T})\ \{\varphi \mid \overline{M}\} \ (<:\text{-MEMBER-CLASS})$$

$$\frac{\mathcal{S}, \Gamma \vdash \overline{S'} <: \overline{S} \qquad \mathcal{S}, \Gamma \vdash T <: T'}{\mathcal{S}, \Gamma \vdash \mathtt{def}_n a\,(\overline{x : S}) : T\,(= t)^? <: \mathtt{def}_m a\,(\overline{x : S'}) : T'\,(= t')^?} \ (<:\text{-MEMBER-METHOD})$$

Fig. 8. The FS_{alg} Calculus : Subtyping

We distinguish a subcategory of terms that can be used inside types, and that we call *paths*. A path is either a variable or the selection of a field on a term that is itself a path. The introduction of paths is motivated by their property of always evaluating to the same object value and of being strongly normalizable. Both properties are needed if we want type soundness to hold. However, this goes beyond the scope of the present paper.

Our calculus has a rich syntax of *types*. A type selection $p.A$ is either a *class type* if A is a class label, a *virtual type* if A is an abstract type label, or an *alias type* if A is a concrete type label. A class type $p.A$ has as values all instances of class A whose enclosing instance associated with A is the object denoted by p. Virtual and alias types $p.A$ have a rather different meaning: they represent the type held by the type field A in the object p. A singleton type $p.\mathbf{type}$ represents the type of which p is the unique element. Finally, a type signature $\left(\overline{T}\right)\left\{\varphi \mid \overline{M}\right\}$ combines the concepts of intersection types and member refinements: it represents the intersection of types $\overline{T}$ with additional constraints on members expressed by declarations $\overline{M}$.

3.2 Operational semantics

Figure 5 contains the inference rules that define a small-step operational semantics for our calculus. It is composed of a *reduction* relation and a *lookup* relation. Both relations use the concept of *evaluation environment* Σ, which is a list of bindings $x : \overline{M}$ that associates an object name x with its set of members. The reduction relation $\Sigma \ ; \ t \to \Sigma' \ ; \ t'$ reduces t to t' in the environment Σ. The reduction of a term can imply the creation of new objects that are added to the environment, leading to a new environment Σ'. The lookup relation $\Sigma \vdash T \prec_\varphi \overline{M}$ collects all declarations $\overline{M}$ in a type T. Together with the concept of evaluation context for terms and types, the rule RED-CONTEXT lets us reduce inside a term. The evaluation of a program $\texttt{val}\ z = \texttt{new}\ \left\{\varphi \mid \overline{M}\right\}; t$ consists in repeatedly reducing the term t in the environment context $z : [z/\varphi]\overline{M}$ until reaching a term that is a value, i.e. a variable y. Such a semantics is needed if we want to state and prove a theorem of type safety. Note that in this semantics the value attached to a field member is re-evaluated each time the field is selected, which corresponds to a call-by-name semantics.

3.3 Type system

The type system of FS_{alg} is described by an algorithmic system of inference rules. In such a system, any judgment is matched by the conclusion of at most one rule, which means that the application of rules is completely deterministic. Typing FS_{alg} requires the definition of several auxiliary judgments about types in addition to the classical judgment that assigns a type T to a term t: membership ($\mathcal{S}, \Gamma \vdash T \ni M$), subtyping ($\mathcal{S}, \Gamma \vdash T <: U$), expansion ($\mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M}$) and type well-formedness ($\mathcal{S}, \Gamma \vdash T$ WF). Most of the judgments are parameterized by a typing context $\mathcal{S}, \Gamma$, where $\mathcal{S}$ is a set of indices representing locked declarations, and Γ is a set of bindings $x : T$ between variables and types such that all variables x are pairwise distinct.

Type assignment. The first two boxes of Figure 6 present the judgments that assign types to terms. The judgment $\mathcal{S}, \Gamma \vdash t : T$ always assigns the most precise type to a term, in particular a path p always receives the type $p.\mathbf{type}$ according to this judgment, provided it can be assigned a bound T (rule PATH). A field selection $t.a$ is typable if it is possible to type t and to establish, using the *membership* judgment, that the type S of t contains a field declaration $\mathtt{val}_n a : T\,(= u)^?$; in this case type T is assigned to the selection $t.a$ (rule SELECT). Note that this rule is only applicable if t is *not* a path. In case t is a path p, we fall back to rule PATH. Rule METHOD allows to type method calls $s.a\,(\bar{t})$: the type of s must contain a declaration for the method a, and each argument must have a type which is compatible with the one required by the method declaration, i.e. which is a *subtype* of the expected type. If these conditions are fulfilled, the type announced by the method declaration is given to the term $s.a\,(\bar{t})$. Finally, rule NEW allows the typing of a local object creation $\mathtt{val}\ x = \mathtt{new}\ T; t$. Such a term gets the type of t if several conditions are satisfied. The members declared inside type T must be concrete: the *expansion* judgment $\mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M}$ returns all the member declarations $\overline{M}$ of a type T such that x represents the self reference. The type T must be *well-formed*. The term t must be typable in the environment Γ extended with the binding $x : T$. Finally, because the scope of x is limited to t, the variable x must not appear in the type S of t.

The judgment $\mathcal{S}, \Gamma \vdash_{path} p : T$ is a typing judgment specialized for paths. Contrary to the previous judgment, it does not always return the singleton type $p.\mathbf{type}$ for a path p. Rather, it returns the less precise but more informative type associated with it, called its *bound*. If p is a variable x, its bound is the type associated with x in the environment (rule PATH-VAR), if p is a selection $p'.a$, its bound is the declared type T of a as seen from p' (rule PATH-SELECT).

Membership and expansion. In Featherweight Java [19] (FJ), there is a lookup relation that computes the most precise signature of a method visible from a given class. In FS_{alg} the member judgment $\mathcal{S}, \Gamma \vdash T \ni M$ presented at the bottom of Figure 6 generalizes this relation to the computation of any kind of declaration (not just methods) visible from any kind of type (not just class types). In FJ the lookup relation is quite simple, it returns the signature of the method as it appears in the program; in Featherweight Generic Java (FGJ), since classes can have type parameters and since a member signature can refer to some type parameters of its enclosing class, the lookup relation requires also the computation of type values for such type parameters. In FS_{alg}, things are more complicated: because types depend on paths, the signature of a member can depend on the self reference x of its enclosing class, or more generally on the self reference of any enclosing class, direct or indirect. Thus, the result of the lookup must replace x with the actual value of the enclosing instance. To illustrate this, suppose we have a method declaration $\mathtt{def}_n a\,() : x.A$ in a class C where x is the self reference. If the starting type T of the membership judgment is a singleton type $p.\mathbf{type}$, then it is possible to replace x with p and obtain $p.A$. But if the starting type T is a class type $q.C$, then the lookup fails because there is no available instance of C with which to replace x. However, if the signature of a does not depend on x (for instance if the return type of a is an external type $root.Int$), then the lookup succeeds, even from $q.C$ (because in $root.Int$ there is no self reference to be replaced). Rules $\ni$-SINGLETON and $\ni$-OTHER respectively implement

the situation where the starting type T is a singleton type and where it is not. In the first case, the path p is first expanded into q with the *path alias expansion* judgment. Then, we take the type T of q which, by construction, is not a singleton type. Using the *type expansion judgment* we collect all declarations $\overline{M}$ of T and we substitute p for the self reference φ in the declaration we are interested in. In case the starting type T is not a singleton type, we can immediately collect its declarations, but we have to check that the declaration we are looking for does not contain the self reference φ.

The type expansion judgment $\mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M}$ (third box of Figure 6) collects all declarations $\overline{M}$ of a type T where φ is used to represent the self reference inside declarations $\overline{M}$. The expansion of a class type $p.A$ is the expansion of the type signature $\left(\overline{T}\right) \left\{\varphi \mid \overline{M}\right\}$ composed of its parents $\overline{T}$ and its direct members $\overline{M}$ (rule $\prec$-CLASS). The index n of the class is added to the set $\mathcal{S}$ of locks in order to avoid falling into an infinite expansion (for instance if a class extends itself). Rule $\prec$-TYPE is completely analogous. It expands a type alias while performing the same actions on locks in order to prevent infinite alias expansion. Finally, rule $\prec$-SIGNATURE expands a type signature: it starts by expanding all parents $\overline{T}$ and then merges all collected declarations $\overline{N}$ with the direct members $\overline{M}$ of the type signature. The *concatenation with rewriting of common members* $\uplus$ of several sets of declarations is defined by $\overline{M} \uplus \overline{N} = \overline{M}|_{\mathrm{dom}(\overline{M}) \setminus \mathrm{dom}(\overline{N})}, \overline{N}$, where the domain $dom(\overline{M})$ of a sequence of declarations is the set of labels it defines and the restriction $\overline{M}|_{\mathcal{L}}$ of declarations $\overline{M}$ to a set of labels $\mathcal{L}$ consists of all of those declarations in $\overline{M}$ that define labels in $\mathcal{L}$.

Type and path alias expansion. We introduce here two auxiliary judgments, *type alias expansion* and *path alias expansion* that will be used when defining the subtyping judgment. The idea of the type alias expansion judgment $\mathcal{S}, \Gamma \vdash T \simeq U$ is very simple: we take a type T and if T is a type alias $p.A$ for another type T', we recursively expand T' until we reach a type that is no longer a type alias. This simple behavior is formalized by the five rules in the first box of Figure 8.

There exists also a relation of aliasing between paths. For instance, with a field declaration $\mathtt{val}_n a : p.\mathbf{type}$ in a class where x is the self reference, the path $x.a$ has type $p.\mathbf{type}$. Because $p.\mathbf{type}$ is a singleton type and $x.a$ belongs to this type, this really means that $x.a$ and p represent the same object, or equivalently that $x.a$ is an *alias* for p. In this reasoning, we have performed a one-step alias expansion going from $x.a$ to p. The judgment defined at the bottom of Figure 7 implements the complete alias expansion of a path by repeating the operation we have performed in this example. Once again, we prevent falling into a loop by adding an index to the set $\mathcal{S}$ of locks (rule $\simeq$-STEP). This index is the one of the last selected field in the considered path p. It is computed by the function $\psi(p)$. This function takes as implicit arguments the environments $\mathcal{S}$ and Γ and is defined as follows.

$$\begin{array}{ll} \psi(p.a) = n & \text{if } \mathcal{S}, \Gamma \vdash p.\mathbf{type} \ni \mathtt{val}_n a : T\,(= t)^? \\ \psi(x) \quad = x & \end{array}$$

The expansion terminates when we reach a path that cannot be given a singleton type (rule $\simeq$-REFL).

Subtyping. The subtyping judgment $\mathcal{S}, \Gamma \vdash T <: U$ (central box of Figure 8) is used to compare two types T and U. In theory, such a relation must be defined by considering all possible kinds of types for T and all possible kinds of types for U. But in practice it is possible to factorize and eliminate a great number of cases. First we expand both types into types T' and U' (rule <:-UNALIAS). This simple operation allows to quickly eliminate all cases where T' or U' is an abstract type $p.A$, because if a type is still abstract after alias expansion nothing can be said about it. As a consequence, in the auxiliary judgment $\mathcal{S}, \Gamma \vdash_* T <: U$ we assume that T and U are not abstract types. If U is a singleton type $q.\mathbf{type}$, then only another singleton type $p.\mathbf{type}$ can be a subtype of it (rule <:-SINGLETON-RIGHT), and in this case paths p and q must be equivalent, i.e. they must be aliases for the same path p'. If the left-hand side is a singleton type $p.\mathbf{type}$ and the right-hand side U is not, then we just take the bound T of p and recursively compare it with U (rule <:-SINGLETON-LEFT). If both types are a selection of the same type label A (rule <:-SINGLETON-LEFT), then their prefixes must be equivalent. Suppose now that the right-hand side is a class type. We just have to consider the cases where the left-hand side is a class type or a type signature. If the left hand-side is a class type $p.A$ then we recursively check that there exists one parent of the class that is a subtype of $p'.A'$ (rule <:-CLASS). If the left-hand side is a type signature $\left(\overline{T}\right)\left\{\varphi \,|\, \overline{M}\right\}$ the procedure is analogous. Finally, we are left with the case where the right-hand side is a type signature $\left(\overline{T}\right)\left\{\varphi \,|\, \overline{M}\right\}$. There are then two things to check: first that the type T is a subtype of all parents $\overline{T}$ in the type signature, which expresses the fact that a type signature represents the intersection of its parents. And secondly, that type T satisfies the constraints expressed by the declarations $\overline{M}$. This is the case if T expands to a set of declarations $\overline{N}$, with a greater domain than $\overline{M}$, and if the declarations that are common to $\overline{N}$ and $\overline{M}$ are more precise in $\overline{N}$, which is expressed by the subtyping test between members $\mathcal{S}, \Gamma \vdash \overline{N} \ll \overline{M}$.

The subtyping between members is standard (bottom of Figure 8): it is covariant for the types of field declarations and for the result types of methods, contravariant for the method parameter types, and invariant for type and class declarations. The invariance for type aliases is crucial since an alias conceptually represents an equality between types. The member subtyping relation is lifted to sequences of members using the following definitions:

$$\overline{N} \ll \overline{N'} \Leftrightarrow \left(\forall (N, N') \in \overline{N} \times \overline{N'},\ \mathrm{dom}(N) = \mathrm{dom}(N') \Rightarrow N <: N'\right)$$

Well-formedness. There are two well-formedness judgments: one for types $\mathcal{S}, \Gamma \vdash T$ WF (top of Figure 7), and one for members $\mathcal{S}, \Gamma \vdash M$ WF_φ (top of Figure 7)

For a singleton type $p.\mathbf{type}$ to be well-formed, the path p must be typable (rule WF-SINGLETON). In order to avoid a cyclic dependence between a path and its type, we also check the well-formedness of the bound T of p. Because we do not want to fall into a loop, we extend the set of locked indices with $\psi(p)$. For a class type $p.A$ to be well-formed (rule WF-CLASS), it is sufficient to check that a class named A is accessible from p, which is expressed by a membership judgment starting from the type $p.\mathbf{type}$. For an abstract type $p.A$, a declaration of the type A must also be visible from p. In addition, if the type is an alias for a type T we also check that T is well-formed. This is needed since we do not

want type aliases to let us define recursive types. Such a test serves for detecting illegal cycles, as for the rule WF-SINGLETON. Finally, the well-formedness of a type signature is checked by first putting a binding with the type signature in the environment (rule WF-TYPE).

The well-formedness of fields and methods is standard. For a field declaration the type T' of the optional value t must conform to the declared bound T (rule WF-X-FIELD). For a method, the types of parameters $\overline{S}$ and the method type T must be well-formed in the current environment, which means that the judgment excludes the possibility for the type S_i of a parameter, or for the result type T, to depend on parameters $\overline{x}$. The body t of the method must be typable in an environment extended with the parameters, and its type T' must conform to the return type (rule WF-X-METHOD). Note that we require parameter types not to contain singleton types. This restriction has almost no impact on expressiveness and it has the advantage of simplifying the termination proof of path alias expansion. The well-formedness of a type declaration is equivalent to the well-formedness of the type T it is an alias for (rule WF-X-TYPE). Eventual cycles in this definition are detected indirectly by the well-formedness of T. For typing a class declaration (rule WF-X-CLASS) we check the well-formedness of its associated type signature $\left(\overline{T}\right)\left\{\varphi \,|\, \overline{M}\right\}$ composed of its parents and direct members, in an environment extended with a binding for the class self reference φ. It might be surprising that here we do not check for the absence of cycles in the class hierarchy, but such cycles are actually detected by the well-formedness of the type signature: if a class inherits, directly or indirectly, of itself, there cannot exist an expansion of the parents $\overline{T}$. A type signature $\left(\overline{T}\right)\left\{\varphi \,|\, \overline{M}\right\}$ (rule WF-X-SIGNATURE) is well-formed w.r.t. a self reference φ if the $\overline{T}$ can be expanded, which forbids singleton types and abstract types in the parents of a signature, and if all new members of the type, directly present in $\overline{M}$, are compatible. The compatibility of a list of groups of members is defined in such a way that a member declaration is always more precise than another declaration defined in a previous group in this list, which is expressed by the formula $\forall\ (i,j),\ \mathcal{S}, \Gamma \vdash (\overline{N_{i+j}}, \overline{M}) \ll \overline{N_i}$.

4 Decidability of the Algorithmic Type System

Lemma 4.1 If a term t can be assigned a type T by the **Path Typing** judgment, then it is unique.

Proof. It is easy to see that a variable only has a single type assignment in the context at any time, so all that remains to prove is that field declarations of the form $\mathtt{val}_n a : T\,(= t)^?$ for a given a are unique in a given p.**type**. We do it by induction on the **Expansion** judgment and PATHSELECT, using the semantics of $\uplus$, a method that we are going to detail in the following.

Lemma 4.2 The calculus defines a deterministic algorithm.

Proof. The rules are syntax-directed (the form of the input determines the rule that must be used, and all the parameters of any recursive calls), except for $\prec$-CLASS, $\prec$-TYPE and $\simeq$-STEP.

$\prec$-CLASS and $\prec$-TYPE seem to create an ambiguity, but they are in fact algorithmically equivalent, since they only differ on the member that will be sought in the expansion of the type of the path that is in the conclusion.

The path in the premises of $\simeq$-STEP is uniquely defined by the **Path Typing** judgment using Lemma 4.1.

Lemma 4.3 The **Path Typing**, **Expansion**, **Membership**, and **Path Alias Expansion** judgments terminate on all inputs.

Proof. We start by inlining the membership rule $\ni$-SINGLETON in the recursive calls of the three other judgments, leading to the following rules:

$$\frac{\mathcal{S}, \Gamma \vdash p \simeq q \qquad \mathcal{S}, \Gamma \vdash_{path} q : T \qquad \psi(p) \cup \mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M}, [\varphi/p](\mathtt{val}_n a : T' \, (= t)^?), \overline{M'} \qquad \psi(p) \not\subseteq \mathcal{S}}{\mathcal{S}, \Gamma \vdash_{path} p.a : T'} \text{(PATH-SELECT)}$$

$$\frac{\begin{array}{c}\mathcal{S}, \Gamma \vdash p \simeq q \qquad \mathcal{S}, \Gamma \vdash_{path} q : T \\ \psi(p) \cup \mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M}, [\varphi/p](\mathtt{trait}_n A \, \mathtt{extends} \, (\overline{S}) \, \{x \mid \overline{N}\}, \overline{M'}) \qquad \psi(p) \not\subseteq \mathcal{S} \\ \{n\} \cup \mathcal{S}, \Gamma \vdash (\overline{S}) \, \{x \mid \overline{N}\} \prec_x \overline{N'} \qquad n \notin \mathcal{S}\end{array}}{\mathcal{S}, \Gamma \vdash p.A \prec_x \overline{N'}} \text{($\prec$-CLASS)}$$

$$\frac{\begin{array}{c}\mathcal{S}, \Gamma \vdash p \simeq q \qquad \mathcal{S}, \Gamma \vdash_{path} q : T \\ \psi(p) \cup \mathcal{S}, \Gamma \vdash T \prec_\varphi \overline{M}, [\varphi/p](\mathtt{type}_n A = S), \overline{M'} \qquad \psi(p) \not\subseteq \mathcal{S} \\ \{n\} \cup \mathcal{S}, \Gamma \vdash S \prec_x \overline{N'} \qquad n \notin \mathcal{S}\end{array}}{\mathcal{S}, \Gamma \vdash p.A \prec_x \overline{N'}} \text{($\prec$-TYPE)}$$

The other rules are left unchanged, and the system with those updated rules is trivially equivalent to the previous one. We prove the termination of the **Expansion**, **Path Typing** and **Path Alias Expansion** judgments by mutual induction, reasoning by case on the last rule of the derivation. The result is then easily extended to the **Membership** judgmement by inspection.

Cases $\simeq$-REFL and PATHVAR are easy. For the others, we consider that several rules in those judgments make all their recursive calls on a strictly larger set of locked symbols, and can therefore appear at most a finite number of times in any derivation. We can then concentrate on the remaining case, $\prec$-SIGNATURE. Let us define the *size* of a type by the lexicographical pair (N, L) where N is the number of its members and L its textual size. Then this size is finite and positive, and $\prec$-SIGNATURE only makes recursive calls on strictly smaller types. Those recursive calls conclude when reduced to a type signature with no inherited types, in which case the conclusion is true by vacuity.

Corollary 1. *The **Type Alias Expansion** judgment terminates on all inputs.*

Proof. Easy induction, considering that the rule $\simeq$-TYPE can only be applied a finite number of times.

Lemma 4.4 The **Algorithmic Subtyping** and **Member Subtyping** judgments terminate on all inputs.

Proof. We proceed by mutual induction on those two judgments, and then by case on the last step of the derivation. Using Corollary 1, we can concentrate on the subtyping of unaliased terms ($\vdash_*$) using rule <:-UNALIAS.

*Case 1 (**Algorithmic Subtyping**).* Cases <:-SINGLETON-RIGHT and <:-PATHS are easy using Lemma 4.3. Moreover, it is easy to show using the **Path Alias Expansion** judgment, that the type T in the premises of <:-SINGLETON-LEFT is not a singleton type. This rule therefore allows us to *unfold* a subtyping problem referring to a singleton type to a subtyping problem between non-singleton types in a single step. Since our subtyping rules do not contain singleton types in their premises, the only singleton types we can encounter in a subtyping derivation are those explicitly mentioned in the program, and their number is finite. We can therefore consider that we work modulo this *unfolding*.

Using the definition of the size of a type given in the proof of Lemma 4.3, we can show that all the recursive calls to the **Algorithmic Subtyping** judgment are made on a strictly smaller type, noticing in the case of <:-CLASS that a given `trait` can not extend itself.

The remaining case is <:-SIG-RIGHT, where the call to the **Member Subtyping** judgment could potentially create a cycle. Let us proceed by contradiction: if there is such a cycle, it means that we have found a type signature $\left(\overline{T}\right)\left\{x \mid \overline{M}\right\}$ such that one of its member declarations contains a declared type that is is either $\left(\overline{T}\right)\left\{x \mid \overline{M}\right\}$ or some $\left(\overline{S}\right)\left\{y \mid \overline{N}\right\}$ such that $\exists\ i,\ S_i = \left(\overline{T}\right)\left\{x \mid \overline{M}\right\}$. Then the textual length of this type is larger than the textual length of the type signature that contains it, which is absurd.

*Case 2 (**Member Subtyping**).* Easy with the previous case.

Lemma 4.5 The **Type Assignement**, **Well-Formedness** and **Member Well-Formedness** judgments terminate on all inputs.

Proof. We proceed by mutual induction on those judgments, then by case on the last rule of the derivation.

*Case 3 (**Member Well-Formedness**).* Considering WF-X-SIGNATURE, we notice that, in a similar way to the problem we encountered with rule <:-SIG-RIGHT in Case 1 of Lemma 4.4, we are in presence of a judgement that makes several potentially cyclic recursive calls to the **Well-Formedness** judgment. However, since the judgment is syntax-directed, we notice that such a cycle would require us to find a type signature $\left(\overline{T}\right)\left\{x \mid \overline{M}\right\}$ directly containing a member whose declared type lexically contains $\left(\overline{T}\right)\left\{x \mid \overline{M}\right\}$ itself, which would give this type signature an infinite textual length.

The remaining interesting cases are WF-X-FIELD and WF-X-METHOD. in both cases, making a derivation involving those rules cyclic requires defining a type signature that directly contains a field or method whose value contains an instantiation of this very type signature:

$$\overline{M} \ni \texttt{val}_n a : T = \texttt{val}\ \ x = \texttt{new}\ \ \left(\overline{T}\right)\left\{x \mid \overline{M}\right\}; t$$

This would again give an infinite textual length to the term $\left(\overline{T}\right)\left\{x \mid \overline{M}\right\}$.

*Case 4 (**Well-Formedness**).* We start by noticing that WF-SINGLETON and WF-TYPE can only occur a finite number of times, since they make their recursive calls on a strictly larger set of recursive types. We then conclude using the previous case and the termination lemma for the **Membership** judgment (Lemma 4.3).

*Case 5 (**Type Assignement**).* We conclude remarking that we make recursive calls to the **Type Assignement** judgment on structurally smaller terms, and using the previous termination lemmas.

5 Type Checking and Type Inference in Scala

The previous section showed that type-checking in FS_{alg} is decidable. Does the same hold for full Scala? It is at present too hard to give a definite answer since full Scala is too complicated to admit a formalization of its type system which is complete yet still manageable enough to admit a proof of decidability. But one can conjecture. To do this, we need to compare full Scala with Featherweight Scala. Most of the additional syntactic constructs in full Scala do not cause particular problems for type-checking. However, unlike Featherweight Scala, full Scala has local type inference [30,26].

Local type inference needs to construct least upper bounds and greatest lower bounds (*wrt* the subtype ordering) of sets of types. The decidability of these *lub* and *glb* operations in FS_{alg} is currently an open question. To see the problem, consider the following three class definitions.

```
trait A { this0 |
  type T
  def fromT(x: T): A
}
trait B { this0 |
  type T
  def fromT(x: T): B
}
trait C extends A with B { this0 |
  def fromT(x: T): C
}
```

Now assume that we want to find the greatest lower bound of A and B. Clearly, C is a lower bound of A and B, but it is not the greatest one. A greater lower bound is represented by the following refinement type:

```
A with B { this0 | fromT(x: T): C }
```

One can apply the same step to the result type of *fromT* to obtain a still greater lower bound. Repeating this step infinitely often one obtains the following limit of an ascending chain of lower bounds:

```
A with B { this0 |
  fromT(x: T): A with B { this1 |
    fromT(x: T): A with B { this2 |
      fromT(x: T): A with B { this3 |
        ...
      }
    }
  }
}
```

This limit does not exist as a finite type in FS_{alg}, but the natural algorithm for computing lower bounds is likely to try to construct it, and this would result in non-termination. A similar infinite approximation can be constructed for the *lub* operation by using the contravariance of method parameters. An example is the *lub* of the two refinements

$$\{\ \textbf{def}\ f(x\colon A)\colon Int\ \} \quad \textit{and} \quad \{\ \textbf{def}\ f(x\colon B)\colon Int\ \}\ .$$

The problem of infinite approximations of *lub*'s and *glb*'s also occurs when type-checking Java 1.5 programs with generics and wildcards [34]. The decidability of the latter is currently open [35].

The Scala compiler addresses this problem by imposing a maximum size on the types computed by its *lub* and *glb* operations. It is currently set at 10 levels of parameterizations or refinements. If a type computed by *lub* or *glb* exceeds this limit the system will reply with an error such as the one below:

```
error: failure to compute least upper bound of types
(A) ⇒ scala.Int and (B) ⇒ scala.Int;
an approximation is:
(A with B{
  def fromT(this.T): (A with B{
    def fromT(this.T): (A with B{
      def fromT(this.T): (A with B{...})})})}) ⇒ Int
additional type annotations are needed
  if (cond) (x: A) ⇒ 1 else (x: B) ⇒ 1
  ^
```

The Scala compiler thus turns the potential problem of undecidability of type inference into a completeness problem: local type inference might now fail to give a solution even if a best type would exist. However, in practice such complicated types arise very rarely. Moreover, it is always possible to guide the type inference process by adding more type annotations, so that infinite approximations are avoided.

In the failed example above, the problem would have been avoided by giving an explicit annotation of the desired type of the problematic conditional. For instance, the following compiles without error.

$$(\textbf{if}\ (cond)\ (x\colon A) \Rightarrow 1\ \textbf{else}\ (x\colon B) \Rightarrow 1)\colon (C \Rightarrow Int)$$

To summarize, the results on type-checking Featherweight Scala give some degree of confidence that type-checking regular Scala is also decidable. Furthermore, the formalization of locks in FS_{alg} corresponds closely to the present implementation in the Scala compiler,

so that there is hope that this implementation does in fact represent an algorithm for type checking Scala programs. Type-inference, on the other hand, needs to compute *lub*'s and *glb*'s of types and is believed to be undecidable. The Scala compiler avoids potential non-termination at the price of incompleteness by imposing an upper limit on the size of the types computed by a *lub* or *glb*.

Note that we have classified here the typing of an if-then-else expression as a type-inference problem, not a type checking problem. The justification of this classification is that it is possible (and, at rare occasions, necessary) to provide a type for the expression with an explicit annotation.

6 Conclusion

We have presented a calculus for type-checking core Scala programs. Featherweight Scala describes the central constructs for programming components in Scala: nested classes, modular mixin composition, abstract types, type aliases, and path-dependent types. Unlike previous work on foundations of Scala [25], this calculus is decidable and admits a straight-forward type-checking algorithm.

Featherweight Scala programs are essentially a syntactic subset of regular Scala programs. The subset is kept minimal, so that one can concentrate on a small set of typing issues. In future work it would be interesting to extend the calculus to a larger fragment of Scala. Among the most interesting extensions are a call-by-value semantics, polymorphic methods, and mutable state.

The correctness of the calculus also remains to be verified. The operational semantics of FS_{alg} is defined by a small-step reduction semantics. We intend to show in future work that it satisfies the subject-reduction and type-soundness properties. Judging from our experience with previous calculi [25,10] this looks plausible, but a formal proof still needs to be completed.

Acknowledgement. We thank Rachele Fuzzati for proofreading the formal description of the calculus.

References

1. P. Altherr. *A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings*. PhD thesis, EPFL, March 2006. No. 3509.
2. P. Altherr and V. Cremet. Inner Classes and Virtual Types. EPFL Technical Report IC/2005/013, March 2005.
3. D. Ancona and E. Zucca. A primitive calculus for module systems. In *Principles and Practice of Declarative Programming*, LNCS 1702, 1999.
4. D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 2002.
5. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
6. G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, 1992. IEEE Computer Society.

7. K. Bruce. Some challenging typing issues in object-oriented languages. In *Electronic notes in Theoretical Computer Science, volume 82(8).*, 2003.
8. K. B. Bruce, M. Odersky, and P. Wadler. A statical safe alternative to virtual types. In *Proceedings of the 5th International Workshop on Foundations of Object-Oriented Languages*, San Diego, USA, 1998.
9. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.
10. V. Cremet. *Foundations for Scala: Semantics and Proof of Virtual Types.* PhD thesis, EPFL, May 2006. No. 3556.
11. E. Ernst. *gBeta: A language with virtual attributes, block structure and propagating, dynamic inheritance.* PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.
12. E. Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, Budapest, Hungary, 2001.
13. E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. In *ACM Symposium on Principles of Programming Languages (POPL'06)*, Jan. 2006.
14. J. Garrigue. Code reuse through polymorphic variants. In *In Workshop on Foundations of Software Engineering, Sasaguri, Japan, November 2000.*, 2000.
15. R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, January 1994.
16. T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, pages 6–20, 2002.
17. A. Igarashi and B. C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34–49, 2002.
18. A. Igarashi and B. C. Pierce. On inner classes. *Inf. Comput.*, 177(1):56–89, 2002.
19. A. Igarishi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. OOPSLA*, Nov. 1999.
20. P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann. Simple dependent types: Concord. In *Proc. FTfJP*, 2004.
21. X. Leroy. A syntactic theory of type generativity and sharing. In *ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon*, 1994.
22. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, June 1993. ISBN 0-201-62430-3.
23. K. Nakata, A. Ito, and J. Garrigue. Recursive object-oriented modules. In *Proc. FOOL 12*, Jan. 2005.
24. N. Nystrom, S. Chong, and A. Myers. Scalable extensibility via nested inheritance. In *Proc. OOPSLA*, pages 99–115, 2005.
25. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.
26. M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 41–53, January 2001.
27. M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.
28. M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. OOPSLA*, 2005.
29. B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.
30. B. C. Pierce and D. N. Turner. Local type inference. In *Proc. POPL*, 1998.
31. D. Rémy and J. Vuillon. On the (un)reality of virtual types. available from `http://pauillac.inria.fr/remy/work/virtual`, Mar. 2000.

32. M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, San Diego, CA, USA, January 1998.
33. M. Torgersen. The expression problem revisited: Four new solutions using generics. In *Proc. ECOOP 2004*, volume 3086 of *Springer LNCS*, pages 123–143. Springer-Verlag, July 2004.
34. M. Torgersen, C. P. Hansen, E. Ernst, P. vod der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *Proceedings SAC'04*, pages 1289–1296, Nicosia, Cyprus, Mar. 2004. ACM Press.
35. S. Zdancewic. Type inference for Java 5: Wildcards, F-bounds, and undecidability. `http://www.cis.upenn.edu/~stevez/note.html`, 2006.