Towards Virtual Traits in Scala

Manuel Weiel Ingo Maier Sebastian Erdweg Michael Eichberg Mira Mezini
TU Darmstadt, Germany

ABSTRACT

Scala is a powerful language that supports a variety of features, but it lacks virtual traits. Virtual traits are class-valued object attributes and can be redefined within subtraits. They support higher-order hierarchies and family polymorphism. This work introduces virtual traits into Scala and explains how to encode virtual traits on top of existing Scala features. We have implemented this encoding using Scala annotation macros and have conducted two small case studies.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Inheritance, Polymorphism, Classes and objects

Keywords

virtual traits, virtual classes, family polymorphism, macros

1. INTRODUCTION

Traits in Scala can be defined as members of other traits. This allows for powerful abstraction mechanisms as described by Odersky and Zenger [1]. In this work, we will call the member traits of an outer trait a *family* of traits. By extending the outer trait, we can add new traits to the family (or shadow existing traits). It is, however, impossible to refine existing inner traits in order to add new functionality to the trait and its subtraits. In this paper, we present our work towards support for this to Scala in the form of *virtual* traits. A trait that is virtual can be refined by overriding it in a subtrait of its outer family trait, which affects not only the inner trait itself but potentially others in the same family (or subfamily) as well.

To motivate virtual traits, let us consider the expression problem [2]. Given a trait hierarchy for arithmetic expressions, we want to be able to independently extend the hierarchy with new data variants and new operations. First, we define an expression model with constants and addition:

Scala '14, July 28–29, 2014, Uppsala, Sweden

http://dx.doi.org/10.1145/2637647.2637654

```
@family trait ExprModel {
  @virtual abstract trait Expr
  @virtual trait Constant(val value: Int) extends Expr
  @virtual trait Add(val l: Expr, val r: Expr) extends Expr
  val zero: Constant = Constant(0) // example expr
}
```

Trait ExprModel represents the family of expressions, which is a simple trait hierarchy in this case. We annotate a family of virtual traits with the annotation **@family**. The mode comprises three traits that are virtual (as denoted by annotation **@virtual**): the abstract base trait Expr, a trait for constant Int values, and a trait for addition. We can extend this expression model with new operations and with new types of expressions as follows. In order to extend our expression model with a variant for multiplication, we can write:

```
@family trait ExprMult extends ExprModel {
  @virtual trait Mult(val l: Expr, val r: Expr) extends Expr
}
```

In this example, we simply add another subtrait Mult of Expr. This is already possible with standard (non-virtual) traits. Besides the additional annotations, the only difference is that virtual traits can have constructor parameters, even though they are not strictly necessary here but helpful to keep the example concise.

The interesting task is to extend a trait family with a new operation. The following code shows how to add an eval operation to the original expression model:

```
@family trait ExprEval extends ExprModel {
  @virtual override abstract trait Expr { def eval: Int }
  @virtual override trait Constant { def eval = value }
  @virtual override trait Add { def eval = l.eval + r.eval }
}
```

We override the abstract virtual trait Expr with a refined implementation that adds an abstract method eval. The inheritance relation between the virtual traits is inherited from the parent family, so it does not have to be repeated. Also we see that l and r in Add already know that Expr introduces eval even though Add does not repeat the constructor parameters. The trait Expr has to be explicitly abstract, as the abstract method eval prohibits instantiation. We also override the two subtraits with versions that implement method eval.

We can instantiate and evaluate a simple expression as follows:

```
val model = ExprEval()
model.Add(model.Constant(17), model.zero).eval // yields 17
```

Virtual traits allow multiple families to be mixed together to support the features of all parent families. The mixing is type-safe, that is, the Scala type system guarantees that the mixed family features all virtual traits with all operations of all parent families. The type check will run after our proposed transformation and we do not need to provide own type checking for virtual traits. In nested virtual traits this mixin has to cascade into all nested virtual traits. This is called deep mixin composition. It is a crucial feature of virtual traits. We can use this feature to compose our extensions for multiplication and evaluation:

```
@family trait ExprMultEval
        extends ExprMult with ExprEval {
  @virtual override trait Mult { def eval = l.eval * r.eval }
}
```

The family ExprMultEval extends both ExprMult and ExprEval. Accordingly, the family must feature a multiplication trait (due to ExprMult) and every subtype of Expr in the family must provide an eval method (due to ExprEval). To this end, we refine the virtual trait Mult and add the required eval method. The Scala type system would reject the family ExprMultEval if we did not refine Mult and add an eval method. We can use the mixed family like the ExprEval family above:

```
val model = ExprMultEval()
import model._
Mult(Constant(7), Add(Constant(3), zero)).eval // yields 21
```

Families of virtual traits always inherit the inheritance relation of the parent families, which enables safe and independent adding of features to an existing hierarchy. Therefore, virtual traits are an effective way to achieve feature-oriented programming [3].

Virtual traits build on virtual classes, for which a formalization of the dynamic and static semantics and a soundness proof exist [4]. Our longterm goal is to add support for virtual traits to Scala, based on the formalization in [4]. In this work, we present steps towards this goal. Specifically, we present the following contributions:

- We describe an encoding of virtual traits in terms of existing Scala features: abstract type members, mixin composition and self type annotations.
- We present a system of annotation macros [5] that automatically rewrites annotated virtual trait families as in the above examples to valid Scala code using our encoding.
- We discuss design decisions and interactions of virtual traits with Scala's trait linearization and type system.

2. TRANSFORMATION

We describe a way to add virtual trait support to Scala using an encoding on top of existing Scala features. Nested traits allow mixin composition but do not provide the ability to override the implementation of a nested trait and refine the implementation. Virtual traits as well as families containing virtual traits therefore transform to a combination of nested traits, virtual types and factories. The factories achieve late binding and family inheritance by statically analyzing the code and determine the right traits that need to be mixed into the class linearization. In the following sections we describe the different steps in the transformation and present why these steps are needed to achieve virtual trait support in Scala.

Feature	Transformation
virtual trait family	• introduce a class for final type bindings • factory to instantiate a family
virtual trait	• trait gets renamed • abstract type with same name and upper bound of the linearization of the virtual trait • concrete type with upper bound of the trait in final binding class • abstract factory method to instantiate the virtual trait • class in final binding class that extends the linearization of the trait • factory method in the final class to call final binding class
constructor parameters	• adds the parameters as **val** with the default value to the trait • adds the parameters to the factory method • adds the parameters to the final binding class
outer calls	• adds a self type to each virtual trait and family with different names • adds a method called **outer** in each virtual trait that is set to the enclosing self type

Table 1: Transformations

2.1 Recipe

Table 1 shows the main aspects of transforming virtual traits and other related elements are to Scala code. In the following paragraphs we show why these transformations are necessary.

Virtual trait families.

Virtual trait families are annotated with **@family**. A virtual trait family may not have constructor parameters. The body of the virtual trait family is kept unchanged unless other rules for transforming apply (e.g. a virtual trait is encountered). Additionally we add a concrete class that extends this trait. This class is necessary to bind the final types to the inner virtual traits, as these may change if a virtual trait family is extended. Therefore we call this class the final binding class. A factory is added to instantiate virtual trait families. It creates an instance of the final binding class. The factory method is called inside the apply method of the companion object of the family.

Virtual traits.

Virtual traits are annotated with **@virtual** and transform into multiple parts. The trait itself contains only partial information about the final type of the virtual trait so the typing information has to be represented as a virtual type. This virtual type will be used inside the code to identify the type of the virtual trait. To avoid name collisions we rename the trait that is annotated with **@virtual**. This allows us to refer to every virtual trait in every family using a unique name. This is needed in building the linearizations in all families. Parents of virtual traits can be other virtual traits as well as normal traits. The parents of the virtual trait have to be modified as well. The linearization of this virtual

trait replaces its parents. Subsection 2.7 will describe in detail how we determine the linearization. The parents of the virtual trait may contain other virtual traits and the name of these virtual trait is bound to the abstract type after the linearization. As abstract types cannot be used as base classes or mixins, the concrete class that mixes all necessary virtual traits together has to search for all needed virtual traits and cannot rely on the defined abstract traits. The trait also defines an explicit self type whose type is set to the newly introduced virtual type. This is needed as the linearization of a virtual trait may change in subfamilies. The upper bound of the virtual type is determined by the linearization of the parent traits. The lower bound is always Null.

Finally we introduce a factory method which is used to instantiate virtual traits. This factory method is abstract as we do not know the final type and the linearization.

Inside of the final class, the abstract type and the abstract factory method are bound to concrete values. The abstract type is bound to the linearization of this virtual trait. The factory method instantiates a class which is created only for the purpose of instantiating virtual traits. The final class therefore establishes the actual type binding for each virtual trait.

As the family itself can be extended, we need the type binding in the final class to have the type of the virtual trait be dependent on the type of the object of its enclosing class.

Constructor parameters.

Virtual traits support constructor parameters, but normal traits do not, so constructor parameters have to receive special treatment. Each constructor parameter adds an abstract **val** declaration to the trait (**val** name: Type;). Also the list of parameters is added to the method signature of the factory method. Finally these constructor parameters are added to the class that is introduced in the final class and this class overrides the **val**s in the trait.

outer *calls.*

To achieve calls to the instance of the enclosing family, the **outer** keyword is introduced. We implement the **outer** keyword by adding a method with the name **outer** to each virtual trait. This method is then bound to the self type of the enclosing class or trait. Therefore the self type has to be added to each virtual trait and family. As nested virtual traits are not covered in this paper we can simplify this by just inserting a self type annotation to the family which is called **outer**.

2.2 Families

To detail the transformation we apply the transformation to small examples.

Allowing late binding of virtual traits inside families requires one to split a family into an abstract part and a concrete implementation that is dependent on the actual family and thus any possible subfamily. Therefore a family has to be transformed into an abstract trait and a concrete implementation of that trait.

```
1  @family trait ExprModel {
2    @virtual trait Expr
3  }
```

Line 1 defines a family which contains a virtual trait (Line 2). We transform this example as follows:

```
1  trait ExprModel extends AnyRef { outer =>
2    @virtual trait Expr
3    ...
4  }
5
6  class VC_FINAL$ExprModel extends ExprModel {
7    ...
8  }
9
10 object ExprModel {
11   def apply() = new VC_FINAL$ExprModel()
12 }
```

As seen in Line 1 and Line 6, we split the implementation of a family into two parts. The first part is an abstract trait (Line 1) which contains the body of the family. The second part is an additional class (Line 6) which extends this abstract trait. The name of this class starts with VC_FINAL. This class is used to facilitate late binding and allow the type of the virtual traits to be bound to the containing class' object. Late binding is achieved as instances of families are always created using the companion object's apply method. Therefore the VC_FINAL class is instantiated which binds the abstract types to concrete values. Even if the instance of the family is cast to a super family type, the instantiation of virtual traits is still managed by the VC_FINAL class. The virtual trait Expr in Line 2 has to be transformed as well, so this transformation is described in the following section.

2.3 Virtual traits

In the last subsection we showed that it is necessary to split families into two parts, an abstract and a concrete implementation part. Virtual traits have to be split as well. The way this is done is more complex for virtual traits though. Therefore transforming virtual traits includes some other steps to allow late binding and to have the right linearization order.

We once again look at a simple example of a family with one virtual trait. To clarify which part of the transformation contains the body of the virtual trait, we add a method declaration inside the virtual trait Expr:

```
1  @family trait ExprModel {
2    @virtual trait Expr {
3      def something: Int = 0
4    }
5  }
```

```
1  trait ExprModel extends AnyRef { outer =>
2    type Expr >: Null <: AnyRef with
         VC_TRAIT$ExprModel$Expr
3    trait VC_TRAIT$ExprModel$Expr { self: Expr =>
4      def something: Int = 0
5    }
6    def Expr(): Expr
7  }
8  class VC_FINAL$ExprModel extends ExprModel {
9    def Expr() = new VC_FIX$ExprModel$Expr()
10   type Expr = AnyRef with VC_TRAIT$ExprModel$Expr
11   class VC_FIX$ExprModel$Expr extends
         VC_TRAIT$ExprModel$Expr
12 }
13 ...
```

Listing 1: Simple virtual trait (transformed)

Listing 1 show that we transform even a simple virtual trait into multiple parts resulting in a lot of boilerplate code. The

virtual trait transforms into a trait (Lines 3-5) with a self type annotation as well as being renamed for an easy way to refer to that special virtual trait implementation in later linearizations.

Additionally to this trait, we define an abstract type with the name of the virtual trait (Line 2). This type is used when the type of a virtual trait is needed. It always has the lower bound of Null. This is due to the fact that all virtual traits are derived by AnyRef and therefore it should be valid to instantiate them with **null**. The upper bound is defined by the inheritance the virtual trait has defined. This abstract type is overridden in the VC_FINAL part (Line 10) by a concrete type that mixes in all necessary traits, which are generated from the virtual traits. Because the type is still abstract in Line 2, it is possible to redefine it in subfamilies. These subfamilies can then redefine the abstract type to accommodate overridden virtual traits.

In addition to this type, a method with the same name as the virtual trait is generated, which is used to instantiate virtual traits. This factory method (Line 6) is also abstract and gets defined in the VC_FINAL part (Line 9). In Line 11 we introduce a concrete class which can be used to create an instance of this virtual trait. Its constructor is called by the factory method (Line 9). This class only exists to mix the right traits in and create an instance of a virtual trait. It is generated in the VC_FINAL part because the actual mixed in traits can vary in other families that subtype ExprModel.

Important to note is the self type annotation in Line 3. It allows the self type to be dynamically refined later in the VC_FINAL part of the enclosing family, because Expr is still abstract at that time. This is needed to allow subfamilies to refine the type of a virtual trait. By defining a self type to the abstract type, we gain access to all inherited methods from the base traits of the virtual trait even if those are defined in subfamilies.

Abstract virtual traits are treated specially. Therefore, the next subsection covers these aspects.

2.4 Abstract virtual traits

Normal traits are always abstract and can therefore not have constructor parameters. As virtual traits are allowed to be instantiated, boilerplate code for instantiation is created. Therefore abstract virtual traits can omit large parts of the generated code. The factory method for object creation has to be omitted. As we can not create a concrete instance of an abstract virtual trait it is neither possible nor intended to create instances of this virtual trait and therefore the factory method is not needed. Additionally it is not necessary to generate the VC_FIX class, as it is only used for object creation. Abstract virtual traits can still be mixed in as described.

2.5 Inheritance of virtual traits

This section shows how virtual trait inheritance is handled. The following example shows how inheritance inside of a family is transformed:

```
1  @family trait ExprModel {
2    @virtual abstract trait Expr
3    @virtual trait Constant extends Expr {
4      var value: Int = 0
5    }
6  }
```

We see that Constant extends Expr (Line 3). This inheritance is reflected in the generated traits and types:

```
1  abstract trait ExprModel extends AnyRef { outer =>
2    ...
3    type Constant >: Null <: AnyRef with Expr with
         VC_TRAIT$ExprModel$Constant
4    abstract trait VC_TRAIT$ExprModel$Constant extends
         VC_TRAIT$ExprModel$Expr { self: Constant =>
5      var value: Int = 0
6    }
7    def Constant(): Constant
8  }
9  class VC_FINAL$ExprModel extends ExprModel {
10   ...
11   type Constant = AnyRef with Expr with
         VC_TRAIT$ExprModel$Constant
12   class VC_FIX$ExprModel$Constant extends
         VC_TRAIT$ExprModel$Expr with
         VC_TRAIT$ExprModel$Constant
13   def Constant() = new VC_FIX$ExprModel$Constant()
14 }
```

Listing 2: Virtual trait inheritance (transformed)

The most important changes in Listing 2 compared to the example in Listing 1 are in Lines 3, 11 and 12. Here, we mix the inherited virtual trait Expr in. We first define the upper bound of the abstract type as all extended traits and after that mix in the own trait as before.

In Line 12 it is not possible to write the linearization exactly as in Line 3 and 11, because types cannot be extended or mixed in. So all traits from the complete hierarchy have to be mixed in explicitly. Therefore both VC_TRAITs are mixed in.

Because type Constant has Expr as upper bound and the self type is of type Constant, the trait can access all members of VC_TRAIT$ExprModel$Expr. So the desired behavior is achieved without losing the functionality of late binding.

2.6 Family inheritance

It should not only be possible that virtual traits can have an inheritance relation, but also trait families should have the ability to inherit from other families. We handle virtual traits specially in the linearization that are already defined in a parent virtual trait family and should be refined. We show a basic example of family inheritance with the following example:

```
@family trait ExprEval extends ExprModel {
  @virtual override abstract trait Expr {
    def eval: Int
  }
  @virtual override trait Add {
    def eval: Int = l.eval + r.eval
  }
}
```

When a trait family extends another one, it has to repeat all concrete type bindings from its super virtual trait families. The VC_FINAL class also has to repeat the VC_FIX class which mixes the traits together and refines the factory method.

```
1  trait ExprEval extends ExprModel { outer =>
2    type Expr >: Null <: AnyRef with
         VC_TRAIT$ExprModel$Expr with
         VC_TRAIT$ExprEval$Expr
3    trait VC_TRAIT$ExprEval$Expr extends
         VC_TRAIT$ExprModel$Expr { self: Expr =>
```

```
4       def eval: Int
5     }
6     ...
7     type Add >: Null <: AnyRef with BinExpr with
          VC_TRAIT$ExprModel$Add with
          VC_TRAIT$ExprEval$Add
8     trait VC_TRAIT$ExprEval$Add extends
          VC_TRAIT$ExprModel$Expr with
          VC_TRAIT$ExprEval$Expr with
          VC_TRAIT$ExprModel$BinExpr with
          VC_TRAIT$ExprModel$Add { self: Add =>
9       def eval: Int = l.eval + r.eval
10    }
11    ...
12  }
13  object ExprEval extends AnyRef {
14    class VC_FINAL$ExprEval extends ExprEval {
15      type Expr = AnyRef with VC_TRAIT$ExprModel$Expr with
          VC_TRAIT$ExprEval$Expr;
16      ...
17      def Add() = new VC_FIX$ExprEval$Add();
18      type VirtualB = AnyRef with BinExpr with
          VC_TRAIT$ExprModel$Add with
          VC_TRAIT$ExprEval$Add;
19      class VC_FIX$ExprEval$Add extends
          VC_TRAIT$ExprModel$Expr with
          VC_TRAIT$ExprEval$Expr with
          VC_TRAIT$ExprModel$BinExpr with
          VC_TRAIT$ExprModel$Add with
          VC_TRAIT$ExprEval$Add
20    }
21    def apply() = new VC_FINAL$ExprEval()
22  }
```

Listing 3: Family inheritance (transformed)

In Line 18 and 19 we can see that the linearization also includes traits from the parent families. These are mixed in first, as the traits from the current family have precedence.

2.7 Linearization

The desired trait linearization follows the rules of the Scala class linearization described in [6]. In addition to the base classes in the current family, the defined base classes and traits in the base families have to be included in the linearization. The base classes of the trait gain precedence over the inherited linearization of its families.

In the family ExprMultEval that was shown in the introduction the class Mult would have the following linearization:

```
Mult, Expr
```

The mixin traits for this linearization are:

```
VC_TRAIT$ExprModel$Expr,
VC_TRAIT$ExprEval$Expr,
VC_TRAIT$ExprModel$Mult,
VC_TRAIT$ExprMult$Mult,
VC_TRAIT$ExprEval$Mult,
VC_TRAIT$ExprMultEval$Mult
```

2.8 Constructor parameters

As traits do not support constructor parameters per default, constructor parameters of the virtual traits are added to the generated factory method. This is done by adding the constructor parameters with the same type signature to the factory method. So it is still possible to instantiate virtual traits with constructor parameters.

Rewriting the virtual trait Constant to use a constructor parameter results in:

```
@family trait ExprModel {
  @virtual trait Constant(val value: Int)
}
```

Transforming this example results in the code seen in Listing 4.

```
1   abstract trait ExprModel extends AnyRef { outer =>
2     abstract trait VC_TRAIT$ExprModel$Constant extends
          VC_TRAIT$ExprModel$Expr { self: Constant =>
3       val value: Int;
4     }
5     def Constant(value: Int): Constant
6   }
7   class VC_FINAL$ExprModel extends ExprModel {
8     def Constant(value: Int) = new
          VC_FIX$ExprModel$Constant(value)
9     class VC_FIX$ExprModel$Constant(val value: Int) extends
          VC_TRAIT$ExprModel$Expr with
          VC_TRAIT$ExprModel$Constant
10  }
```

Listing 4: Virtual trait with constructor parameter (transformed)

As seen in Lines 5 and 8, we introduce the constructor parameters in both the abstract and the concrete factory method. This ensures that instantiating the virtual trait needs constructor parameters. The constructor parameters are then passed along to the VC_FIX class (Line 9) which is a concrete class and therefore can have constructor parameters. The immutable variable value in Line 3 is defined by the constructor parameter of the VC_FIX class in Line 9 and thus giving value the correct value.

This is only a partial solution though, as it prohibits passing constructor parameters to virtual base traits.

Passing constructor parameters to virtual base traits.

When we pass constructor parameters to virtual base traits the transformation changes to accommodate the fact that the scope of the current virtual trait has to be visible. So we mix in another trait which binds the parameter of the current virtual trait to the parameter of the parent virtual trait.

The following example passes a constructor parameter to its base trait:

```
1   @family trait ExprModel {
2     @virtual trait Constant(val value: Int) extends Expr
3     @virtual trait Squared(val toBeSquared: Int) extends
          Constant(square(toBeSquared)) {
4       def square(x: Int) = x * x
5     }
6   }
```

In Line 3 we pass the constructor parameter toBeSquared to a method square which is defined in the body of the same virtual trait (Line 4). Therefore it is necessary to introduce another trait which we mix in, to make it possible that the constructor parameter can use functions declared in the virtual class (see Line 7 of Listing 5). This trait has the same self type as the VC_TRAIT. This allows to see all introduced members.

```
1   abstract trait ExprModel extends AnyRef { outer =>
2     ...
3     def Squared(toBeSquared: Int): Squared
4     type Squared >: Null <: AnyRef with Constant with
          VC_CONS$ExprModel$Squared with
          VC_TRAIT$ExprModel$Squared
```

```
5     trait VC_TRAIT$ExprModel$Squared extends
          VC_TRAIT$ExprModel$Expr with
          VC_TRAIT$ExprModel$Constant { self: Squared =>
6     }
7     trait VC_CONS$ExprModel$Squared { self: Squared =>
8           val toBeSquared: Int
9           val value: Int = square(toBeSquared)
10    }
11  }
12  class VC_FINAL$ExprModel extends ExprModel {
13    ...
14    def Squared(toBeSquared: Int) = new
          VC_FIX$ExprModel$Squared(toBeSquared)
15    type Squared = AnyRef with Constant with
          VC_TRAIT$ExprModel$Squared
16    class VC_FIX$ExprModel$Squared(_toBeSquared: Int)
          extends { val toBeSquared = _toBeSquared } with
          VC_CONS$ExprModel$Squared with
          VC_TRAIT$ExprModel$Expr with
          VC_TRAIT$ExprModel$Constant with
          VC_TRAIT$ExprModel$Squared
17  }
```

Listing 5: Virtual class with constructor parameter passed to base trait (transformed)

The factory method in Line 3 takes the new parameter as an argument. The virtual class transformation itself does not change, but another trait that starts with VC_CONS is added (Line 7). This trait manages the assignment of the passed constructor parameter to the base class. As it has the self type Squared, it can see all members that are declared in the virtual class. We then mix this trait into the VC_FIX class in Line 16. To prevent name clashes, we rename the constructor parameter of the VC_FIX class.

2.9 Summary

We showed that virtual traits can be transformed to a combination of nested traits, virtual types and factory methods. The transformation takes into account that virtual traits are late-bound and and offer family polymorphism.

The presented encoding can be automated, so we introduce an implementation using annotation macros that performs this transformation.

3. TECHNICAL REALIZATION

We implemented a prototype of virtual traits in Scala. This implementation includes the transformation of virtual trait families, virtual traits and constructor parameters. It does not support nested virtual traits and passing of constructor parameters to base classes.

The implementation uses annotation macros which are a part of macro paradise. Macro paradise includes macro features that are not (yet) included in the Scala compiler. Macro paradise is developed as a compiler plugin and can insert and modify compiler phases. Annotation macros are macros that are bound to static annotations. Listing 6 shows in Line 2 how a macro transformation can be invoked in an annotation. During the compile run macro paradise will invoke a separate compile run for each annotation macro and pass the AST of the annottee to the macro implementation. The annotation macro can then modify the abstract syntax tree of the annotated member and return this for further processing in the compiler. An important fact is that this expansion happens before the typer has run. This allows a very flexible and powerful manipulation of the AST. Annotation macros allow typing of members outside the annottee in the separate compile run, though.

```
1  class family extends StaticAnnotation {
2    def macroTransform(annottees: Any*) = macro family.impl
3  }
4
5  object family {
6   def impl(c: Context)(annottees: c.Expr[Any]*):
7         c.Expr[Any] = {
8    ...
9    val result: c.Tree = {
10     annottees.map(_.tree).toList match {
11      case (cd @ ClassDef(mods, name, tparams,
12              Template(parents, self, body))) :: rest =>
13      val classDef =
14      q"""abstract trait $name[..$tparams]
15          extends ..$parents { outer =>
16           ..${transformBody(body,name,parents)}
17         }"""
18     val newObjectDef =
19      q"""object ${name.toTermName} {
20          def apply() = new ${finalClassName(name)};
21           ${finalClass(name, body, parents)}
22         }"""
23
24     q"{ $classDef; $newObjectDef }"
25    }
26   }
27   c.Expr[Any](result)
28  }
29 }
```

Listing 6: Annotation macro implementation (excerpt)

The transformation is implemented in the @family annotation. We iterate over all members in the family body and transform all virtual traits we find. The final class is also added in this macro. Listing 6 shows that we pattern match over the annottees (Line 10). If we encounter a class definition (Line 11-12) the transformation is applied (Lines 13-24). The transformation is done mainly in two steps. First we transform the body of the family and all included virtual traits (Line 16). In the method transformBody virtual traits are expanded into abstract types and factory methods. The second step introduces the VC_FINAL class that establishes the final type bindings. The body of the VC_FINAL class is built in the method finalClass (Line 21). Finally we return both the modified trait (classDef) and the newly introduced companion object (newObjectDef) in Line 24. In contrast to the proposed transformation, we embed the VC_FINAL class inside of the companion object of the virtual trait family. This is needed as an annotated trait or class can only expand into a trait or class with an optional companion object.

The implementation of transformBody and finalClass need to have information about the linearization of its inner virtual traits. It therefore has to determine the linearization as described in section 2.7. For this it needs knowledge about the parent families and their virtual traits. Annotation macros only expose the AST of the annotated member, so the information about the parent families cannot be obtained using pattern matching on ASTs.

To determine the linearization of virtual traits we therefore use a combination of pattern matching on the own family AST and reflection on the already expanded parent fam-

ilies. The class linearization of the parent families can be determined by using reflection on all parents and combining these using the Scala class linearization rules. The linearization of a virtual trait can then be combined from the own defined virtual trait parents and those parents that are found using reflection in the linearization of the parent families. This is done by accessing the upper bound of the abstract types that are introduced for every virtual trait and therefore we can determine the linearization of the virtual trait in this family. The linearization algorithm then merges the obtained information into the complete linearization in the current family. The class mixins can then be determined by searching for instances of the VC_TRAIT in every family in the family linearization.

We tested the implementation using two small case studies, one being an extended version of the expression model, the other one is a program that emulates smart homes and implements the sensors as virtual traits. So the actual implementation can be mixed together from the different existing sensor families. Additionally, we ported test cases from CaesarJ to verify correct behavior. As CaesarJ features a slightly modified class linearization, the tests had to be modified to accommodate the linearization presented in this paper.

The implementation can be found at [7].

4. DISCUSSION

This paper shows the possibility to model virtual traits in Scala by implementing them as annotation macros. The theoretical transformations cover most cases to have a correct implementation of virtual traits with respect to the virtual class calculus described by Ernst, Ostermann, and Cook [4].

Scala contains virtual types which does not cover polymorphic instantiation and virtual inheritance. This is added by the transformation described in Section 2. Annotation macros provide the flexibility to add these features to Scala to provide virtual trait support.

However, the implementation and transformation have some shortcomings.

4.1 Restrictions in the transformation

The transformation does not cover any type safety checks with respect to virtual traits. Type safety is currently assumed to be given by Scala's typer. This can result in unclear error messages as well as undesired behavior. In future work basic type checking on an earlier point can be added to ensure type safety with respect to virtual traits.

This work does not cover extending virtual traits outside of families. Currently this is prohibited by the use of abstract types which traits cannot extend. To allow extending of virtual traits outside of families additional research needs to be done.

Currently the visibility modifiers are not honored in the presented transformation. Annotating a member with e.g. **private[this]** would currently result in being private with respect to the generated trait and not necessarily with respect to the virtual trait.

4.2 Restrictions in the implementation

The current implementation does not cover nested virtual traits (e.g. it is not possible to introduce **@virtual** traits inside another **@virtual** trait) so it is only possible to obtain one nesting layer. Further work has to be done to show that the transformation and implementation make it possible to recursively nest virtual traits and achieve full deep mixin-composition.

The proposed implementation uses the keyword **class** instead of **trait** as traits are always marked abstract by the parser and do not allow constructor parameters. As annotation macros run only after the parser phase there is no easy and elegant way to use the keyword trait without changing the compiler or removing constructor parameter support. Directly modifying the compiler enables changes to the parser to allow non-abstract traits and traits with constructor parameters.

Reflection and pattern matching over ASTs.

As annotation macros do not provide full AST visibility the use of reflection poses some issues with respect to determining the virtual trait linearization. Many operations have to be programmed in two different ways though they obtain the same information. One implementation retrieves the linearization in the own AST and another one uses the reflection API. This makes code reuse difficult. The use of reflection has advantages though. Typechecking the parent family will ensure that the macro already expanded and that the Scala type system accepts the expansion of the parent family. This does not provide type safety for virtual traits, but it rules out a variety of possible situations where unsafe code could be generated in the own family. If there are cyclic references, it is unfortunately possible to run into endless loops, though.

4.3 Constructor parameters

Also the current approach to constructor parameters for mixins is very limited. Scala traits do not support constructor parameters, because there is no guarantee that constructor parameters in mixins will be preserved in classes with which a mixin can be combined. Other languages like CaesarJ allow constructor parameters in mixins, but every constructor is rewritten to a method that cannot be hidden. Therefore subclasses can only override their implementation.

The problem with redefining constructor parameters consists of the fact that constructors can be used in other virtual traits of the base hierarchy. After changing the constructor signature, these virtual traits of the base class that have to use the new constructors due to late binding, do not know what to pass to the changed constructors. This can be seen in the following example:

```
1  @family trait ExprModel {
2    @virtual trait Constant(val test: Int)
3    @virtual trait Constants {
4      def zero: Constant = new Constant(0)
5      def one: Constant = new Constant(1)
6    }
7  }
8
9  trait ExprTest extends ExprModel {
10   @virtual override trait Constant(val testString: String)
11 }
```

Listing 7: Constructor refinement example

Listing 7 shows two fundamental issues. The first problem is that in **trait** Constants an instance of **trait** Constant is created (Line 4 and 5). As long as the constructor signature does not change, it is statically known that one parameter of type Int is expected. By overriding the constructor in

ExprTest (Line 10) this is not given anymore. As Constants is not changed, it does not know how to instantiate Constant anymore. The second problem is that with the name change of the parameter the member test is not available anymore so each call to it fails. So this cannot be allowed either. These issues could be circumvented by only allowing to add new constructor parameters in new families. Moreover all new constructor parameters would need a default value.

Constructor parameter support is therefore hard to achieve without breaking type safety. Scala takes the approach to disallow constructor parameters for mixins completely, which is the easiest approach, but limits the use of virtual traits. Constructors are different from methods, which are preserved in subclasses with their signature. This does not hold for constructors. Constructors in subclasses are not inherited and may therefore be defined completely different. This poses a problem for mixins. CaesarJ takes the approach of exposing constructors as methods which cannot be hidden in subclasses. It is only possible to override it. This has the disadvantage that constructors become methods that can even be called after the object has already been created. It remains as future work to find a solution to the constructor problem that allows mixins, but still maintain type safety.

4.4 Linearization

When rewriting virtual traits, our encoding adds a second axis of inheritance. Besides the "vertical" inheritance relationship that is explicit coded between traits in a virtual trait hierarchy, it adds an implicit "horizontal" inheritance relationship between an overriding virtual trait and the trait it overrides. These two axes are then collapsed into one using a specific order so that our macros can create valid Scala trait definitions. There are essentially two sensible orderings to choose from. Either an overriding trait first extends traits from the vertical axis and then from the horizontal or vice versa. For example, for a family A of virtual traits T and S, where S extends T and a subfamily B that overrides both T and S, we can either let B.S extend A.S with B.T or B.T with A.S.

We chose the second order for the following reason. Scala maintains the invariant that for any traits C and D, if D is a subtrait of C it comes before C in any class linearization in which C and D occur. By chosing the second order, we maintain a straightforward extension of this invariant: if D is a subtrait of C it comes before C *and any trait that overrides* C in any class linearization in which C and D occur.

4.5 Code size

For every virtual trait we generate an abstract type, a factory method and a class that is needed for instantiation. So we expect a linear overhead compared to regular traits. However, future work will focus on evaluating the impact of virtual traits on the size of the codebase.

5. RELATED WORK

Virtual traits presented in this paper are based on the virtual class calculus by Ernst, Ostermann, and Cook in [4]. First, virtual classes were introduced in BETA, but got documented only several years later in [8]. Later languages like gbeta and CaesarJ extended the model that was introduced by BETA.

5.1 BETA

BETA is a programming language that is purely object oriented. It introduces nested classes and also unified classes and procedures into patterns [9]. Patterns unify classes, procedures, functions, coroutines, processes and exceptions. Subpattern are like nested classes in other programming languages. BETA introduces patterns which abstract over classes, coroutines, concurrent processes and exceptions. Because patterns can be virtual, it is not only possible to have virtual classes but also all of the above mentioned features can all be virtual.

BETA itself is block structured so patterns can be textually nested [9]. However, it does not support deep mixin-composition.

5.2 gbeta

gbeta [10] is a statically typed programming language originating in BETA. In contrast to BETA it supports a fully general version of family polymorphism and can propagate combinations of classes and methods and thus supporting deep mixin composition.

Virtual classes in gbeta are realized by introducing open placeholders which are declared as a feature of its enclosing object "and it may be refined (to a subclass) in a subclass of the enclosing class. [...] gbeta supports a very general notion of virtual classes where a refinement declaration (traditionally known as a *further-binding* of the virtual class) may refine that class by means of a mixin-based combination process that recursively propagates into the block structure." [10].

5.3 CaesarJ

CaesarJ [11] takes virtual classes and brings them to a Java based programming language. It allows better modularity and reusability of components. CaesarJ itself does not only include virtual classes but also introduces aspect oriented features to enable further modularity by extracting features into aspects.

CaesarJ declares both virtual classes and class families with the keyword **cclass** in contrast to conventional Java classes which still carry the keyword **class**. Nested cclasses automatically are virtual classes. Neither can Java classes reside inside cclasses, nor can cclasses reside inside Java classes. CaesarJ classes can be nested an unlimited number of times. CaesarJ introduces the keyword **outer** to reference the enclosing class.

CaesarJ classes can be declared as a specialization of any number of CaesarJ classes. This is done by using the **extends** keyword and the different classes are separated by the mixin operator **&** [12]. Also if a class family defines a nested virtual class which already exists in the context of the collaboration, this virtual class is overridden. This is called a refinement or a further-binding. In contrast to the proposed solution for Scala it is not necessary to specify that the class is an overridden class using any keyword.

CaesarJ class families are not allowed to have constructor parameters and all constructors of parent families are called during the instantiation of a CaesarJ class family. CaesarJ nested classes are instantiated by calling the constructor using the **new** keyword on the class family.

5.4 Tribe

With Tribe, Clarke, Drossopoulou, Noble, and Wrigstad present a simplified version of the *vc* calculus [4] and allow more flexible path-based types and avoid to add additional conceptual overhead [13]. In contrast to *vc* surrounding instances can not only be accessed by calling **this**.out, but enables to use out also on instances of virtual classes. Also they propose a more flexible way of referring to types of virtual classes. In *vc* it is only allowed to refer to virtual classes by an instance of their enclosing class. Tribe also allows referencing by the type of the enclosing class. Referring to the virtual class using the instance of the enclosing class is more specialized. In an example where the family Graph contains a virtual class Node, the natural subtype relation g.Node $\leq$ Graph.Node is valid. g.Node denotes an instance of Node inside family g whereas Graph.Node denotes an instance of Node in *some* graph family. Scala supports projection types (Graph#Node) which can be used for the second case. Calling out (or **outer** in our transformation) on instances and not only on **this** could possibly be modeled using **val**s that refer to the enclosing instance in our virtual trait representation. Therefore some of the proposed features may be added to virtual traits in future work.

6. SUMMARY

In this paper we present virtual traits. A virtual trait can be overridden in a subtrait of its outer family trait and therefore refine its implementation. Virtual traits can be encoded on top of existing Scala features. We presented an encoding and showed that this encoding can be implemented on top of existing Scala features. The implementation uses annotation macros that provide a flexible way of realizing the proposed transformation. We will continue to work towards integrating virtual traits into Scala based on the formalization in [4].

References

[1] Martin Odersky and Matthias Zenger. "Scalable Component Abstractions". In: *OOPSLA*. ACM, 2005, pp. 41–57.

[2] Erik Ernst. "The expression problem, Scandinavian style". In: *ON MECHANISMS FOR SPECIALIZATION* (2004), p. 27.

[3] Vaidas Gasiunas and Ivica Aracic. "Dungeon: A Case Study of Feature-Oriented Programming with Virtual Classes". In: *Proceedings of the 2nd Workshop on Aspect-Oriented Product Line Engineering*. Oct. 2007.

[4] Erik Ernst, Klaus Ostermann, and William R. Cook. "A Virtual Class Calculus". In: *SIGPLAN Not.* 41.1 (2006).

[5] Eugene Burmako. "Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming". In: SCALA '13. ACM, 2013, 3:1–3:10.

[6] *The Scala Language Specification Version 2.9*. URL: `http://www.scala-lang.org/files/archive/nightly/pdfs/ScalaReference.pdf`.

[7] Manuel Weiel. *Virtual classes for Scala implemented as annotation macro*. URL: `https://github.com/xmanu/scala-virtual-classes-annotation-macros`.

[8] O. L. Madsen and B. Moller-Pedersen. "Virtual Classes: A Powerful Mechanism in Object-oriented Programming". In: *SIGPLAN Not.* 24.10 (1989).

[9] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. "The BETA programming language". In: *DAIMI Report Series* 16 (1987).

[10] Erik Ernst. "gbeta-a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance". In: *DAIMI Report Series* 29.549 (2000).

[11] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. "An Overview of CaesarJ". In: *Transactions on Aspect-Oriented Software Development I*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 135–173.

[12] *CaesarJ language specification*. URL: `http://www.caesarj.org/index.php/CJLS/Classes`.

[13] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. "Tribe: a simple virtual class calculus". In: *Proceedings of the 6th international conference on Aspect-oriented software development*. ACM. 2007, pp. 121–134.