Feature-Oriented Programming with Object Algebras

Bruno C.d.S. Oliveira[1], Tijs van der Storm[2], Alex Loh[3], William R. Cook[3]

[1]National University of Singapore (oliveira@comp.nus.edu.sg)
[2]Centrum Wiskunde & Informatica (CWI) (storm@cwi.nl)
[3] University of Texas, Austin ({wcook,alexloh}@cs.utexas.edu)

Abstract. Object algebras are a new programming technique that enables a simple solution to basic extensibility and modularity issues in programming languages. While object algebras excel at defining modular features, the *composition* mechanisms for object algebras (and features) are still cumbersome and limited in expressiveness. In this paper we leverage two well-studied type system features, *intersection types* and *type-constructor polymorphism*, to provide object algebras with expressive and practical composition mechanisms. Intersection types are used for defining expressive *run-time* composition operators (combinators) that produce objects with multiple (feature) interfaces. Type-constructor polymorphism enables generic interfaces for the various object algebra combinators. Such generic interfaces can be used as a type-safe front end for a generic implementation of the combinators based on reflection. Additionally, we also provide a modular mechanism to allow different forms of *self*-references in the presence of delegation-based combinators. The result is an *expressive*, *type-safe*, *dynamic*, *delegation*-based composition technique for object algebras, implemented in Scala, which effectively enables a form of *Feature-Oriented Programming* using object algebras.

1 Introduction

Feature-oriented programming (FOP) is a vision of programming in which individual *features* can be defined separately and then composed to build a wide variety of particular products [5,20,41]. In an object-oriented setting, FOP breaks classes and interfaces down into smaller units that relate to specific features. For example, the `IExp` interface below is a complete object interface, while `IEval` and `IPrint` represent interfaces for the specific features of evaluation and printing.

```
trait IExp {
  def eval() : Int
  def print() : String
}
```

```
trait IEval { def eval() : Int }

trait IPrint { def print() : String }
```

Existing object-oriented programming (OOP) languages make it difficult to support FOP. Traditionally OOP encourages the definition of complete interfaces such as `IExp`. Such interfaces are implemented by several classes. However adding a new feature usually involves coordinated changes in multiple classes. In other

words, features often cut across traditional object-oriented modularity boundaries, which is centered on the behavior of individual objects. Such cross-cutting is a symptom of the *tyranny of the dominant decomposition* [46]: programming languages typically support development across one dominant dimension well, but all other dimensions are badly supported [22, 26, 46].

The main difficulty in supporting FOP in existing OOP languages stems from the intrinsic flexibility of FOP, which is challenging for programmers and language designers, especially when combined with a requirement for *modular type-checking* and *separate compilation*. Although research has produced many solutions to extensibility and modularity issues, most of these require advanced language features and/or careful advanced planning [11, 17, 18, 29, 31, 48, 50–52].

Object algebras [36] are a new approach to extensibility and modularity in OOP languages, which is based on a generalization of factories that creates families of related objects. The basic model of object algebras requires only simple generics, as in Java, without advanced typing features. For example, the following interface is an object algebra signature of simple expressions:

```
trait ExpAlg[E] {
  def lit(x : Int) : E
  def add(e1 : E, e2 : E) : E
}
```

Object algebras allow new features to be defined by implementing `ExpAlg`. For instance, classes implementing `ExpAlg[IPrint]` and `ExpAlg[IEval]` are algebras implementing printing and evaluation features respectively. Object algebras also allow extending the interface `ExpAlg` with new constructors [36]. As such object algebras provide a solution to the *expression problem* [14, 42, 49].

While object algebras excel at defining modular features, the *composition* mechanisms for object algebras (and features) are still cumbersome and limited in expressiveness. Combining algebras implementing `ExpAlg[IPrint]` and `ExpAlg[IEval]` to form `ExpAlg[IExp]` is possible, but tedious and cumbersome in Java. Moreover composition mechanisms must be defined separately for each object algebra interface, even though the composition follows a standard pattern. Finally, the basic model of object algebras does not support self-references, so overriding is not supported. The lack of good compositions mechanisms hinders the ability to express *feature interactions*, which is essential for FOP.

This paper provides object algebras with expressive and practical composition mechanisms using two well-studied type system features: *intersection types* [15] and *type-constructor polymorphism* [30, 43]. Both features (as well as their interaction) have been well-studied in programming language theory. For example Compagnoni and Pierce's $F^{\omega}_{\wedge}$ calculus [12], used to study language support for multiple inheritance, supports both features. Moreover, both features are available in the Scala programming language [32], which we use for presentation.

An intersection type, `A with B`, combines the interfaces `A` and `B` to form a new interface. Because the new interface is not required to have an explicit name, programmers can define generic interface composition operators, with types of the form `A => B => A with B`. These interface combinators allow object algebras to

be composed flexibly. While the interfaces are composed and checked statically, the composition of the algebras is done at runtime.

Type-constructor polymorphism refers to the ability for a generic definition to take a type constructor, or type function, as an argument. Since definitions like `ExpAlg` are type constructors, type constructor polymorphism is useful to abstract over such definitions. With type constructor polymorphism it is possible to define generic interfaces for object algebra combinators which are parametrized over the particular type of object algebras. In combination with meta-programming techniques this allows automating implementations of the combinators. As a result a single line is sufficient to implement the combinators for an extension or new object algebra interface. For example, the object `ExpComb`

```
object ExpComb extends Algebra[ExpAlg]
```

creates an object with combinators for the object algebra interface `ExpAlg`.

We also provide a modular mechanism to allow different forms of *self*-references in the presence of delegation-based combinators. As Ostermann [40] observes there are two important concerns related to self-references in delegation-based families of objects: 1) *virtual constructors*; 2) individual *object* self-references. The two issues are addressed using two types of self-references, which provide, respectively a notion of *family* and *object* self-references.

Ultimately, the object algebra composition mechanisms presented in this paper are *expressive*, *type-safe*,*dynamic* (composition happens at run-time), *delegation*-based and convenient to use. With these composition mechanisms a powerful and expressive form of FOP with object algebras is possible.

In summary, our contributions are:

- *FOP using object algebras:* We show that, provided with suitable composition mechanisms, object algebras enable a convenient and expressive form of FOP, which supports *separate compilation* and *modular type-checking*.
- *Generic object algebra combinators:* Using intersection types and type-constructor polymorphism, we show how to model general, expressive and type-safe composition mechanisms for object algebras.
- *Modular self-references:* We show a modular mechanism for dealing with self-references in the presence of delegation-based object algebra combinators.
- *Case studies:* We present two case studies illustrating the applicability of our techniques. The first is a typical test problem in FOP, the second involves composition and instrumentation of various operations on grammars. The code for the case studies and smaller examples is published online at:
  `https://github.com/tvdstorm/oalgcomp`

2 Object Algebras and Current Limitations

Object Algebras are classes that implement algebraic signatures encoded as parameterized interfaces, where the type parameter represents the carrier set of the algebra [36]. In `ExpAlg` the methods `Lit` and `Add` represent the constructors of the abstract algebra, which create values of the algebra in the carrier type `E`.

```
trait ExpEval extends ExpAlg[IEval] {
 def Lit(x : Int) : IEval = new IEval {
  def eval() : Int = x
 }

 def Add(e1 : IEval, e2 : IEval) : IEval = new IEval {
  def eval() : Int = e1.eval() + e2.eval()
 }
}

object ExpEval extends ExpEval
```

Fig. 1. An object algebra for evaluation of integer expressions.

```
trait ExpPrint extends ExpAlg[IPrint] {
 def Lit(x : Int) : IPrint = new IPrint {
  def print() : String = x.toString()
 }

 def Add(e1 : IPrint, e2 : IPrint) : IPrint = new IPrint {
  def print() : String = e1.print() + " + " + e2.print()
 }
}

object ExpPrint extends ExpPrint
```

Fig. 2. An object algebra for printing of integer expressions.

A class that implements such an interface is an *algebra* [21], in that it defines a concrete representation for the carrier set and concrete implementations of the methods. While it is possible to define an object algebra where the carrier set is instantiated to a primitive type, e.g. `int` for evaluation or `String` for printing, in this paper the carrier is always instantiated to an object interface that implements the desired behavior. For example, Fig. 1 and 2 define algebras for evaluating and printing expressions.

Provided with these definitions, clients can create values using the appropriate algebra to perform desired operations. For example:

```
def exp[E](f : ExpAlg[E]) : E =
   f.Add(f.Lit(5), f.Add(f.Lit(6),f.Lit(6)))

val o1 : IPrint = exp(ExpPrint)
val o2 : IEval = exp(ExpEval)
println("Expression: " + o1.print() + "\nEvaluates to: " + o2.eval())
```

defines a method `exp`, which uses the object algebra (factory) `f` to create values of an abstract type `E`. The example then creates objects `o1` and `o2` for printing and evaluation. The `ExpAlg` interface can be extended to define new constructors for additional kinds of expressions. A new class that implements `ExpAlg` provides a new operation on expressions.

One serious problem with the example given above is that two versions of the object must be created: `o1` is used for printing, while `o2` is used for evaluation. A true feature-oriented approach would allow a single object to be created that supported both the printing and evaluation features. A more serious problem arises when one feature (or algebra) depends upon another algebra. Such operations cannot be implemented with the basic strategy described above. In general, feature *interactions* are not expressible.

The original object algebra proposal [36] addressed this problem by proposing *object algebra combinators*. Object algebra combinators allow the composition of algebras to form a new object algebra with the combined behavior and also new behavior related to the interaction of features. Unfortunately, the object algebras combinators written in Java lack expressiveness and are not very practical or convenient to use, for three different reasons:

- *Composed interfaces are awkward:* The Java combinators are based on creating *pairs* to represent the values created by combining two algebras. From the client's viewpoint, the result had the following form (using Scala, which has support for pairs):

  ```
  val o : (IEval,IPrint) = exp(combineExp(ExpEval,ExpPrint))
  println("Eval: " + o._1.eval() + "\nPrint: " + o._2.print())
  ```

  The value `o` does combine printing and evaluation, but such pairs are cumbersome to work with, requiring extraction functions to access the methods. Combinations of more than two features require nested pairs with nested projections, adding to the usability problems.
- *Combinators must be defined for each object algebra interface:* There is a lot of boilerplate code involved because combinators must be implemented or adapted for each new object algebra interface or extension. Clearly, this is quite inconvenient. It would be much more practical if the combinators were automatically defined for each new object algebra interface or extension.
- *The model of dynamic composition lacks support for self-references:* Finally, combinators are defined using dynamic invocation, rather than inheritance. The Java form of object algebras does not support self-reference or *delegation*. Since self-reference is important to achieve extensibility, the existing object algebra approach lacks expressiveness.

As a result, while object algebras provide a simple solution to basic modularity and extensibility issues, existing composition mechanisms impose high overhead and have limited expressiveness for FOP. The remainder of the paper shows solutions to the 3 problems.

3 Combining Object Algebras with Intersection Types

Intersection types provide a solution to the problem of combining object algebras conveniently. Combining object algebras allows two different behaviors or operations, implemented by two specific algebras, to be combined so that they are both available at once. Intersection types avoid the heavy encoding using pairs and allow methods to be called in the normal way.

3.1 Scala's Intersection Types

In Scala, an intersection type[1] `A with B` expresses a type that has both the methods of type `A` and the methods of type `B`. This is similar to

```
interface AwithB extends A, B {}
```

in a language like Java or C#. The main difference is that an intersection type does not require a new nominal type `AwithB`. Furthermore, Scala's intersection types can be used even when `A` and `B` are type parameters instead of concrete types. For example,

```
trait Lifter[A,B] {
   def lift(x : A, y : B ) : A with B
}
```

is a **trait** that contains a method `lift` which takes two objects as parameters and returns an object whose type is the intersection of the two argument types. Note that such an interface cannot be expressed in a language like Java because it is not possible to create a new nominal type that expresses the combination of `A` and `B`. To create a new nominal type, Java requires concrete types, but here, `A` and `B` are type parameters.

3.2 Merging Algebras with Intersection Types

Intersection types allow easy merging of the behaviors created by object algebras. The `lift` operation defined in the previous section for combining objects is used in the definition of a `merge` operator for algebras. Conceptually, a `merge` function for an algebra signature F combines two F-algebras to create a combined algebra:

`merge`F`: (A => B => A with B) =>` F`[A] =>` F`[B] =>` F`[A with B]`

Unlike the solution with pairs described in Section 2, intersection types do not require additional projections. The additional function argument represents the `lift` function, of type `A => B => A with B`, that specifies how to compose two objects of type `A` and `B` into an object of type `A with B`. This lift function resolves conflicts between the behaviors in `A` and `B` by appropriately invoking (delegating) behaviors in `A with B` to either `A` or `B`. The lift function can also resolve *interactions* between features. In other words, the function argument plays a role similar to *lifters* in Prehofer's FOP approach [41].

From a conceptual point of view, the key difference between `combine` on pairs and `merge` is that the former uses a `zip`-like operation with pairs, and the later uses a `zipWith`-like operation with intersection types.

Figure 3 defines the `merge` combinator for expressions in Scala as the trait `ExpMerge`. The value of type `Lifter[A,B]` plays the role of the combination function in `merge`, while the two values `alg1` and `alg2` are the two object algebra arguments. The definition of `Lit` and `Add` uses the combination method `lifter` to combine the two corresponding objects, which are delegated by invoking the corresponding method on the arguments. Intersection types automatically allow the following subtyping relationships:

[1] In Scala these are often called *compound types*.

```
trait ExpMerge[A,B] extends ExpAlg[A with B] {
  val lifter : Lifter[A,B]
  val alg1 : ExpAlg[A]
  val alg2 : ExpAlg[B]

  def Lit(x : Int) : A with B =
    lifter.lift(alg1.Lit(x),alg2.Lit(x))

  def Add(e1 : A with B, e2 : A with B) : A with B =
    lifter.lift(alg1.Add(e1, e2),alg2.Add(e1, e2))
}
```

Fig. 3. Expression merging combinator with intersection types.

```
object LiftEP extends Lifter[IEval,IPrint] {
  def lift(x : IEval, y : IPrint) = new IEval with IPrint {
    def print() = y.print()
    def eval() = x.eval()
  }
}

object ExpPrintEval extends ExpMerge[IEval,IPrint] {
  val alg1 = ExpEval
  val alg2 = ExpPrint
  val lifter = LiftEP
}

def test2() = {
  val o = exp(ExpPrintEval)
  println("Eval: " + o.eval() + "\nPrint: " + o.print())
}
```

Fig. 4. Merging the printing and evaluation algebras.

`A with B <: A` *and* `A with B <: B`

These relationships ensure that no conversion/extraction is needed when delegating arguments, for example, `e1` and `e2` in `Add`. This is an advantage over using pairs, because extraction of the arguments from the pairs is not needed.

Figure 4 illustrates how to merge the printing and evaluation algebras to create an `ExpPrintEval` algebra. Clients can use this factory to create objects of type `IEval with IPrint`, which include `print` and `eval` in a single interface. The result is a seamless combination of the printing and evaluation features.

Unfortunately, it is still necessary to define some boilerplate. Each merge requires a lifter object (`LiftEP`), which in this case combines `IEval` with `IPrint`. Often, as in this case, such lifting operations simply create the new object and delegate the methods to the corresponding methods in either `x` or `y`. As we shall see in Section 5.2, in such cases, it is possible to define a generic lifting behavior.

The merge function is a very useful generic combinator, but not all uses will need all of its capabilities. For example, it is possible that the `lifting` function

```
class LiftDecorate[A](action : A => A) extends Lifter[A,Any] {
  def lift(x : A, y : Any) = action(x)
}

trait ExpDecorate[A] extends ExpMerge[A,Any] {
  val alg2 = ExpEmpty
  val lifter = new LiftDecorate(action)
  def action(x : A) : A
}
```

Fig. 5. Decoration combinator derived from merge and empty.

may be the primary technique needed, while the ability to combine algebras is not important. In other cases, the combination is important but the `lifting` function is standard.

4 Applying Uniform Transformations to Object Algebras

In addition to combining operations implemented by different object algebras, it is also often useful to augment or modify the behavior created by an algebra. One example is adding a generic tracing behavior at each step of evaluation. In this case the behavior being added is not specific to each constructor in the algebra, but is instead a uniform behavior to each evaluation step. We formalize this kind of transformation as a DECORATOR [19] that wraps each value constructed by the algebra with additional functionality.

The `decorate` combinator takes a function `A => A` and an object algebra `ExpAlg[A]` and produces a wrapped object algebra of type `ExpAlg[A]`:

```
decorate : (A => A) => ExpAlg[A] => ExpAlg[A]
```

Although `decorate` can be implemented directly, we choose to implement it in terms of the more generic `merge` combinator. In this use of `merge` it is the `lifting` function that matters, while the possibility to combine two algebras is not needed. As a result, we supply an `empty` algebra as the second algebra. Conceptually, the `decorate` combinator defines a `lifting` that applies the transformation `action` to its first argument, and ignores the second (empty) argument.

```
decorate action alg = merge(x => y => action(x), alg, empty)
```

Figure 5 gives the Scala definition of the `decorate` combinator for the expressions algebra. The `ExpDecorate` trait extends `ExpMerge` and sets the second algebra to an empty object algebra. An abstract method `action` specifies a decoration function, which is applied to objects of type `A`.

An empty algebra, defined in Fig. 6, is an algebra, of type `ExpAlg[Any]`, that does not define any operations. It instantiates the carrier type to `Any`, a Scala type that plays the role of the top of the subtyping hierarchy. Every method in the object algebra has the same definition: `new Object()`.

```
trait ExpEmpty extends ExpAlg[Any] {
  def Lit(x : Int) : Any = new Object()
  def Add(e1 : Any, e2 : Any) : Any = new Object ()
}
object ExpEmpty extends ExpEmpty
```

Fig. 6. The `Empty` expression algebra.

```
object TraceEval extends ExpDecorate[IEval] {
  val alg1 = ExpEval
  def action(o : IEval) = new IEval() {
    def eval() = {
      println("Entering eval()!")
      o.eval()
    }
  }
}
```

Fig. 7. A decorator for tracing.

4.1 Tracing by Decoration

Figure 7 defines a tracing mechanism using the decorator combinator. `TraceEval` defines an action that wraps an existing evaluator with a tracing evaluator, whose `eval` method first prints a message and then delegates to the base evaluator `o`. By extending `ExpDecorate[IEval]`, this action is applied to every evaluator created by the underlying evaluator `ExpEval`. When `exp` is invoked with `TraceEval`:

```
val o : IEval = exp(TraceEval)
println("Eval: " + o.eval())
```

the string `Entering eval()!` is printed 5 times in the console.

The combinators presented thus far contained copious amounts of boilerplating for both `merge` and `lift`. The `merge` combinator must be re-defined for each algebra even though they differ only by method names. Likewise, `lift` in most cases acts merely as a selector delegating to the appropriate algebra, yet has to be re-written for each algebra combination.

5 Generic Object Algebra Combinators

To avoid manually writing boilerplate code for combinators, we develop object algebra combinators interfaces and corresponding implementations generically.

A generic `merge` combinator defined on an algebraic signature F containing methods $C_1(args_1), ..., C_n(args_1)$ might look as follows:

```
trait FMerge[A,B] extends F[A with B] {
  val lifter : Lifter[A, B]
  val a1 : F[A]
  val a2 : F[B]
  def Ci(argsi) : A with B = lifter.lift(a1.Ci(argsi), a2.Ci(argsi))
}
```

```
trait Algebra[F[_]] {
  // Basic combinators
  def merge[A,B](mix : Lifter[A,B], a1 : F[A], a2 : F[B]) : F[A with B]
  def empty() : F[Any]

  // Derived combinator(s)
  def decorate[A](parent : F[A], action : A => A) : F[A] =
    merge[A,Any](new LiftDecorate(action),parent,empty)
}
```

Fig. 8. The generic interface for object algebra combinators.

The generic `merge` combinator extends an interface which is polymorphic in the types of the algebras it combines. Its methods are thin facades to invoke the underlying lifter on the two object algebras. There are two challenges in defining a generic `merge`: the first is defining its generic interface and the second is implementing the constructors. The former is solved using *type-constructor polymorphism* [30, 43], and the latter with reflection. The same idea is applied to other combinators, including `empty`.

5.1 A Generic Interface for Object Algebra Combinators

Scala supports type-constructor polymorphism by allowing a trait to be parameterized by a generic type, also known as a type constructor. With type-constructor polymorphism it is possible to provide a generic interface for object algebra combinators, as shown in Fig. 8. The `Algebra` trait is parameterized by a type constructor `F[_]`, which abstracts over object algebra interfaces like `ExpAlg`. Note that the annotation `[_]` expresses that `F` takes one type argument. The trait contains three methods. These methods provide a generalized interface for the object algebra combinators introduced in Sections 3 and 4, using the type constructor `F` instead of a concrete object algebra interface.

The `Algebra` interface is inspired by *applicative functors* [28]: an abstract interface, widely used in the Haskell language, to model a general form of effects. In Haskell[2], the interface for applicative functors is defined as:

```
class Applicative f where
  merge :: (a → b → c) → f a → f b → f c
  empty :: f ()
```

Like object algebra combinators, applicative functors are also closely related to `zip`-like operations. Our combinators can be viewed as an adaptation of the applicative functors interface. However, an important difference is that applicative functors require co-variant type-constructors (the parameter type occurs in positive positions only), whereas object algebra interfaces do not have such restriction. In fact most object algebras use invariant type-constructors (the parameter type can occur both in positive and negative positions). To compensate

[2] The actual interface in the Haskell libraries is different, but equivalent in expressiveness to the one described here as discussed by McBride and Paterson [28].

for the extra generality of the object algebra interface type-constructors, the `merge` operation restricts the type `c` to be the intersection type `A with B`.

An interesting observation is that `Algebra` can itself be viewed as an object algebra interface whose the argument type (constructor) abstracts over another object algebra interface. In other words, `Algebra` can be thought of as a factory of factories, or a *higher-order* factory: the `merge` and `empty` methods are factory methods that construct factory objects.

Combinators for specific object algebras Using `Algebra` we can create an object that contains object algebra combinators specialized for `ExpAlg` as follows:

```
object ExpComb extends Algebra[ExpAlg] {
  def merge[A,B](f : Lifter[A,B], a1 : ExpAlg[A], a2 : ExpAlg[B]) =
    new ExpMerge[A,B]() {
     val lifter : Mixer[A,B] = f
     val alg1 : ExpAlg[A] = a1
     val alg2 : ExpAlg[B] = a2
    }

   def empty() = ExpEmpty
}
```

and use these combinators in client code. For example:

```
import ExpComb._

val o = exp(merge(LiftEP,decorate(ExpEval,TraceEval.action),ExpPrint))
println("Eval: " + o.eval() + "\n PP: " + o.print())
```

creates an object `o`, which combines evaluation and printing and traces the evaluation method.

5.2 Default Combinator Implementations using Reflection

Reflection can be used to implement `merge`, `empty`, and a default `Lifter` for any given algebraic signature. Reflection does not guarantee static type safety, but if the default reflective implementation is trusted, then the strongly-typed generic interface guarantees the static type-safety of the client code.

Figure 9 gives generic implementations for `merge` and `empty` using reflection. These implementations can be used in the generic interface `Algebra` to provide a default implementation of the combinators. The idea is to use *dynamic proxies* [25] to dynamically define behavior. A dynamic proxy is an object that implements a list of interfaces specified at runtime. As such, that object can respond to all methods of the implemented interfaces by implementing the method `invoke` and the `InvocationHandler` interface. The dynamic proxy objects are created using the `createInstance` method:

```
def createInstance[A](ih : InvocationHandler)(implicit m : ClassTag[A]) = {
     newProxyInstance(m.runtimeClass.getClassLoader,
         Array(m.runtimeClass),ih).asInstanceOf[A]
}
```

```
def merge[A,B](lifter : Lifter[A,B], a1 : F[A], a2 : F[B])(implicit m :
    ClassTag[F[A with B]]) : F[A with B] =
  createInstance(new InvocationHandler() {
  def invoke(proxy : Object, method : Method, args : Array[Object]) = {
    val a = method.invoke(a1,args : _*)
    val b = method.invoke(a2,args : _*)
    lifter.lift(a.asInstanceOf[A],b.asInstanceOf[B]).asInstanceOf[Object]
  }})

def empty(implicit m : ClassTag[F[Any]]) : F[Any] =
  createInstance(new InvocationHandler() {
  def invoke(proxy : Object, method : Method, args : Array[Object]) =
      new Object()
  })
```

Fig. 9. Reflective, generic implementations of merge and empty.

This method relies on the JDK reflection API, which provides support for the creation of dynamic proxies, and Scala's *mirror-based* [8] reflection API to provide reflective information of type parameters. The use of Scala's mirror-based reflection requires an adjustment on the types of the combinators in `Algebra`. The combinators now need to take an additional *implicit parameter* [38] `m`, which contains a reflective description of type parameters. This additional parameter does not affect client code since it is implicitly inferred and passed by the Scala compiler.

Unfortunately there is a wrinkle in the our use of reflection: while supported by Scala, intersection types are not natively supported by the JVM. As a result the use of `createInstance` to dynamically generate an object with an intersection type is problematic. Fortunately there is a workaround which consists of creating a *nominal subtype* `S` of an intersection type `A with B`. This allows working around the JVM limitations, but adds extra parameterization (a type argument `S <: A with B`) to our combinators. In the paper, for clarity of presentation, we will ignore this issue and assume that `createInstance` works well with intersection types.

Generic Lifters The `delegate` function creates a generic lifter function.

```
def delegate[A,B](x : A, y : B)(implicit m : ClassTag[A with B]) =
  createInstance[A with B](new InvocationHandler() {
    def invoke(proxy : Object, method : Method, args : Array[Object]) = {
      try {
        method.invoke(x, args : _*)
      } catch {
        case e : IllegalArgumentException => method.invoke(y, args : _*)
      }}})
```

This function is quite useful to handle intersection types composed of interfaces whose methods are disjoint. An example is the intersection of `IEval` and `IPrint`. In the case that the sets of methods are not disjoint, methods in algebra `x` will have priority over those in algebra `y`.

With `delegate` and `merge` it is possible to define a combinator `combine`, which resembles the `zip`-like combinator with the same name proposed by Oliveira and Cook [36]. The difference is the result is an intersection type instead of a pair.

```
def combine[A,B](alg1 : F[A], alg2 : F[B])(implicit m1 : ClassTag[F[A
    with B]], m2 : ClassTag[A with B]) : F[A with B] =
  merge[A,B](new Lifter[A,B]() {
     def lift(x : A, y : B) = delegate[A,B](x,y)},
    alg1, alg2)
```

With `combine` we can combine features without having to explicitly define a lifter function. For example:

```
val o = exp(combine(decorate(ExpEval,TraceEval.action),ExpPrint))
println("Eval: " + o.eval() + "\n PP: " + o.print())
```

combines evaluation with printing and also enables tracing on `eval`.

In summary, generic object algebra combinators eliminate the need for manual definitions such as `ExpMerge` and `LiftEP`. Instead:

```
object ExpComb extends Algebra[ExpAlg]
```

creates a set of combinators, including `merge`, `empty`, `delegate`, `combine`, and `decorate` for `ExpAlg`, providing the necessary composition infrastructure.

6 Object Algebras, Self-References and Delegation

This section defines a generalization of object algebra interfaces which accounts for *self*-references in our delegation-based setting. Self-references are orthogonal and complementary to the generic and reflective combinators presented in Section 5. As such, for simplicity of presentation, we will first present the treatment of self-references on a specific object algebra interface and then discuss the adaptations needed to the generic object algebra interfaces.

6.1 Generalizing Object Algebras to Account for *Self*-References

Since the programming style in this paper is based on delegation an important question is how to account for self-references. The standard self-references provided by Scala's built-in class-based inheritance model do not provide an adequate semantics in the presence of delegation (or run-time inheritance). As Ostermann [40] points out, when dealing with with delegation-based object families there two important issues that need special care:

- *Object self-references:* When composing object algebras using combinators like `merge` or, more generally, delegating on another algebra, the self-reference to the constructed objects should refer to the *whole* composition rather than the individual object.
- *Virtual constructors:* Similarly to the semantics of *virtual classes* [18], the constructors of objects (that is the methods of the object algebras) should be late bound and refer to the composed objects rather than the objects in the local object algebras.

```
trait GExpAlg[In,Out] {
  def Lit(x : Int) : Out
  def Add(e1 : In, e2 : In) : Out
}

type ExpAlg[E] = GExpAlg[E,E]
```

Fig. 10. A generalized object algebra interface for expressions.

Both of these problems can be solved using two types of self-references: *object* self-references and *family* self references.

In order to account for these two types of self-references we first need a generalization of object algebra interfaces, as shown in Fig. 10. This generalization form has been studied before in the context of research on the relationship between the VISITOR pattern and Church encodings [39]. The idea is to distinguish between the uses of carrier types with respect to whether they are inputs (`In`) or outputs (`Out`). In type-theoretic terms, this means distinguishing between the positive and negative occurrences of the carrier type. It is easy to recover the conventional object algebra interface `ExpAlg[A]` simply by making the two type parameters in `GExpAlg` be the same.

6.2 Object Self-References

The generalized interface allows us to account for the type of object algebra interfaces, `OExpAlg`, with unbound (or open) object self-references:

```
type OExpAlg[S <: E, E] = GExpAlg[S, Open[S,E]]
type Open[S <: E, E] = (=> S) => E
```

The type `OExpAlg` is parameterized by two type parameters `E` and `S`. The type `E` is the usual carrier type for object algebras. The type `S` represents the type of the entire composition of objects, which must be a subtype of `E`. In `OExpAlg` the outputs are a function `(=>S)=> E`. The argument of this function (`=>S`) is the unbound self-reference, which is used by the function to produce an object of type `E`. To prevent early evaluation of the self argument, it is marked as a call-by-name parameter by placing `=>` before the argument type. Scala wraps call-by-name arguments in thunks to delay their evaluation until the function body needs their value.

Dependent features An example where using self references is important is when defining a feature which depends on the availability of another feature. Figure 11 illustrates one such case: a variant of the printing feature, which uses evaluation in its definition. The dependency on evaluation is expressed by bounding the type of the self-reference `S` to `IEval with IPrint`. This imposes a requirement that the self-reference and the input arguments of the different cases (such as `e1` and `e2`) implement both evaluation and printing. However, `ExpPrint2` does not have any hard dependency on a particular implementation of `IEval` and it

```
trait ExpPrint2[S <: IEval with IPrint] extends OExpAlg[S, IPrint] {
  def Lit(x : Int) = self => new IPrint() {
    def print() = x.toString()
  }

  def Add(e1 : S, e2 : S) = self => new IPrint() {
    def print() = e1.print() + " + " + e2.print() + " = " + self.eval()
  }
}
```

Fig. 11. An object algebra with a dependency.

```
trait CloseAlg[E] extends ExpAlg[E] {
  val alg : OExpAlg[E,E]

  def Lit(x : Int) : E = fix(alg.Lit(x))
  def Add(e1 : E, e2 : E) : E = fix(alg.Add(e1,e2))
}

def closeAlg[E](a : OExpAlg[E,E]) : ExpAlg[E] = new CloseAlg[E] {
  val alg = a
}
```

Fig. 12. Closing object self-references.

only defines the behaviour of the printing feature, though it may call evaluation in its implementation. The definition of the `print` method for `Add` uses the self reference (`self`). Note that Scala's built-in self-reference **`this`** is useless in this situation: **`this`** would have the type `IPrint`, but what is needed is a self-reference with type `IEval` **`with`** `IPrint` (the type of the composition).

Closing object self-references Before we can use object algebras with object self-references we must close (or bind) those references. This can be done using a closing object algebra, which is shown in Fig. 12. The closing algebra `CloseAlg[E]` extends the standard object algebra interface `ExpAlg` and delegates on an open object algebra `alg`, which is the algebra to be closed. In each case self-references are closed using *lazy fixpoints*. Lazy fixpoints are a standard way to express the semantics of *dynamic mixin inheritance* and bind self-references in denotational semantics [13] and lazy languages [34].

```
def fix[A](f : Open[A,A]) : A = {lazy val s : A = f(s); s}
```

To implement the lazy fixpoint we exploit Scala's support for lazy values. It is possible to achieve the same effect using mutable references, but Scala's lazy values provide a more elegant solution.

```
trait ExpPrint3[S <: IEval with IPrint] extends SelfExpAlg[S,IPrint]{
  def Lit(x : Int) = self => new IPrint() {def print() = x.toString()}

  def Add(e1 : S, e2 : S) = self => new IPrint() {
    def print() = {
      val plus54 = fself.Add(fself.Lit(5), fself.Lit(4));
      e1.print() + " + " + e2.print() + " = " + self.eval() +
        "  and " + "5 + 4 = " + plus54.eval();
    }
  }
}

def ExpPrint3[S <: IEval with IPrint] : OpenExpAlg[S,IPrint] =
  s => new ExpPrint3[S] {lazy val fself = s}
```

Fig. 13. A printing operation using family and object self-references.

6.3 Family Self-References

Another interesting type of self-references are *family* self references. The type `OpenExpAlg` is the type of (open) object algebra interfaces with both object and family self-references[3]:

```
type OpenExpAlg[S <: E, E] = (=> ExpAlg[S]) => GExpAlg[S, Open[S,E]]
```

Similarly to `OExpAlg`, `OpenExpAlg` is parameterized by two type parameters `E` and `S <: E`. `OpenExpAlg` is a function type in which the argument of that function (`=>ExpAlg[S]`) is the unbound family self reference. The output of the function is `GExpAlg[S, Open[S,E]]`. This type denotes that the input arguments in the algebra are values of the composition type `S`, whereas the output types are have unbound object self references (just as in Section 6.2).

It is possible to define a generic interface for object algebras interfaces with family self references:

```
trait SelfAlg[Self <: Exp, Exp] {
   val fself : ExpAlg[Self]
}
```

This interface can be combined with particular object algebra interfaces to add family self references. For example:

```
trait SelfExpAlg[Self <: Exp, Exp] extends
  GExpAlg[Self,Open[Self,Exp]] with SelfAlg[Self,Exp]
```

denotes integer expression object algebra interfaces with family and object self references. As shown in Fig. 13, this interface can be used to define object algebras with both types of self references. The `ExpPrint3` object algebra implements a modified version of `ExpPrint2` which adds some additional behavior to the printing operation in the `Add` case. The idea is to extend `ExpPrint3` so that the algebra

[3] Note that it is also possible to define a simpler type `(=>ExpAlg[S])=>ExpAlg[S]` which accounts only for object algebra interfaces with family self references.

```
trait Algebra[F[_,_]] {
  type FOpen[F[_,_],S <: T, T] = (=> F[S,S]) => F[S,Open[S,T]]

  // Basic combinators
  def merge[A,B,S <: A with B](mix : Lifter[A,B,S], a1 : FOpen[F,S,A], a2
      : FOpen[F,S,B]) : FOpen[F,S,A with B]
  def empty[S] : FOpen[F,S,Any]

  // Derived combinator(s)
  def decorate[A, S <: A](parent : FOpen[F,S,A], action : A => A) :
      FOpen[F,S,A] =
    merge[A,Any,S](new LiftDecorate(action),parent,empty[S])

  // closing combinators
  def fcloseAlg[S](a : F[S,Open[S,S]]) : F[S,S]
  def fclose[S](f : FOpen[F,S,S]) : F[S,S]
}
```

Fig. 14. Interface for generic combinators with self-references.

constructs a value denoting $5 + 4$ in one of the operations. The family self reference `fself` (which is available as a value from the extended interface `SelfAlg`) ensures that the constructors refer to the overall composed object algebra instead of the local `ExpPrint3` object algebra. It is the use of the family self-reference that enables a *virtual* constructor semantics. If `this.Add(this.Lit(5),this.Lit(4))` was used instead then the constructors would not have the expected semantics in the presence of compositions around `ExpPrint3`.

Closing references Both object self references and family self references can be closed with the following definition:

```
def close[S](f : OpenExpAlg[S,S]) : ExpAlg[S] = fix(compose(closeAlg,f))
```

Essentially, `close` first binds the object self references using `closeAlg` and then it binds the family self-references using a lazy fixpoint. Note that `compose` is the standard function composition operation.

6.4 Generic Combinators with Self-References

The generic combinators presented in Fig. 8 can be adapted to account for self-references, as shown in Fig. 14. The trait `Algebra` now has to abstract over a type constructor with 2 arguments, to account for the generalized form of object algebra interfaces. The type `FOpen` is a generalization of `OpenExpAlg`, for some algebra `F` instead of the specific `ExpAlg`. The combinators `merge` and `empty` must work on open object algebras instead of closed ones. Moreover, in the `merge` combinator the `Lifter` trait needs to be updated slightly to allow the use of object self-references by the lifting functions:

```
trait Lifter[A,B, S <: A with B] {
  def lift(x : A, y : B) : Open[S, A with B]
}
```

Finally, generic forms of closing operators are included in the `Algebra` interface.

As with the combinators in Fig. 8, reflection can also be used to provide generic implementations of the combinators. These implementations are straightforward adaptations of the ones presented in Section 5.2. Generic implementations for `fcloseAlg` can be defined using similar techniques.

Client code Finally, with all self-reference infrastructure and the suitably adapted generic combinators, client code can be developed almost as before. For example:

```
val o =
    exp(fclose(merge(LiftEP,decorate(ExpEval,TraceEval.action),ExpPrint3)))
println("Eval: " + o.eval() + "\nPrint: " + o.print())
```

composes the variant of printing using object and family self-references and decorates evaluation with tracing. Note that this code assumes adapted versions of `LiftEP` and `ExpEval`, which are straightforward to define. The main difference to previous code is that the `fclose` operation must be applied to the composition to bind the self references and be used by the builder method `exp`. Because of the use of family and object self references tracing is applied at each call of evaluation, which ensures the expected behavior for the example.

7 Case Studies

To exercise the expressivity of feature-oriented programing using object-algebra combinators we have performed two case studies. The first adapts an example by Prehofer [41], consisting of a stack feature, which optionally can be composed with counting, locking and bounds-checking features. The second involves various interpretations and analyses of context-free grammars.

Stacks In the first case study, we consider four features: Stack, Counter, Lock and Bound. The Stack feature captures basic stack functionality, such as `push`, `pop` etc. If the size of stack should be maintained, the Counter feature can be used. The Lock feature prevents modifications to an entity. Finally, the Bound feature checks that some numeric input value is within bounds.

Each feature is implemented as an object algebra for stacks which contains a single constructor `stack()`. If we consider the Stack feature to be the base feature, there are $2^3 = 8$ possible configurations. Each configuration requires a lifter to resolve feature interaction. For instance, lifting the counter feature to stack context involves modifying `push` and `pop` to increment and decrement the counter.

Many of these lifters require boilerplate for the methods without feature interaction. To avoid duplicating this code we have introduced default "delegating" traits. These traits declare a field for the delegatee object and forward each feature method to that object.

An example of such a delegator trait for the Counter feature is the following:

```
class StackWithCounter(self: => Stack with Counter, s: Stack, c: Counter)
  extends DStack with DCounter {
  val st = s; val ct = c
  override def empty() {self.reset; st.empty}
  override def push(s : Char) {self.inc; st.push(s)}
  override def pop() {self.dec; st.pop}
}

object LiftSCF extends Lifter[Stack,Counter, Stack with Counter] {
  def lift(st : Stack, ct : Counter) =
    self => new StackWithCounter(self, st, ct)
}
```

Fig. 15. Lifting the Counter feature to the Stack context

```
trait DCounter extends Counter {
  val ct: Counter
  def reset() { ct.reset }
  def inc() { ct.inc }
  def dec() { ct.dec }
  def size() = ct.size()
}
```

This trait is included, for instance, in the class that lifts the Counter feature to the Stack context, as shown in Fig. 15. The default behavior is to delegate the feature methods (e.g., `inc`, `push2`, etc.) to `Stack s` and `Counter c` respectively. To resolve feature interactions, however, the class `StackWithCounter` overrides the relevant methods, to customize the default behavior.

Whenever a feature method would normally call another method in the same feature on **`this`**, the explicit `self` reference should be used instead. An example where this is the case is the method `push2` in the Stack feature, which calls `push` twice:

```
def push2(a : Char) {self.push(a); self.push(a)}
```

If `push2` called `push` on **`this`**, any extension of `push` would be missed. This would result, for instance, in erroneous counting behavior when Stack is composed with Counter.

Grammars The second case study implements various interpretations of grammars. The signature of grammar algebras (shown in Fig. 16) contains constructors for alternative, sequential and optional composition, terminals, nonterminals and empty. We have implemented parsing, printing, and computing nullability and first-set of a grammar symbol as individual object algebras. Parsing requires special memoization which is applied using the `decorate` combinator. In a similar way, both nullability and first-set computation require decoration with an iterative fixpoint aspect. Furthermore, the first-set feature is always composed with the nullability feature, since the former is dependent on the latter.

```
trait GrammarAlg[In, Out] {
  def Alt(lhs: In, rhs: In): Out
  def Seq(lhs: In, rhs: In): Out
  def Opt(arg: In): Out
  def Terminal(word: String): Out
  def NonTerminal(name: String): Out
  def Empty(): Out
}
```

Fig. 16. The signature for grammar algebras.

Tracing and profiling parsers are obtained by dynamically composing with those aspects if so desired. The tracing feature is homogeneous in that it applies uniformly to all constructors of an algebra. Profiling depends on the parsing feature and therefore requires an explicit lifter to modify the parsing feature to update a counter. There is no feature interaction among the other features, so the default lifters produced by `combine` are sufficient.

The grammar case study illustrates the use of both object and family self-references. First, the optional constructor (`Opt`) in the signature of grammars can be desugared to an alternative composition with empty: `Opt(x)` = `fself.Alt(x, fself.Empty)`. Any grammar algebra that includes this desugaring does not have to explicitly deal with optional grammar symbols. The desugaring uses the family self-reference `fself` because the resulting term should be in the outermost, composed algebra, not the local one.

Object self-references play an important role in the profiling feature. The profiling feature accumulates a map recording the number of parse invocations on a certain grammar symbol. Using **`this`** to key into this map would create map entries on objects created by the inner, local object algebra. As a result, the objects stored in the map are without the features that may have been wrapped around these objects. For instance, in the composition `combine(combine(parse, profile), print)`, the keys in the profile map would not be printable.

Client code The following code creates a composite object algebra for grammars that includes parsing, nullability and first-set computation:

```
val f = fclose(
 combine[Parse, Nullable with First, Parse with Nullable with First](
  decorate(grammarParse, new Memo),
  combine[Nullable, First, Parse with Nullable with First](
   decorate(grammarNullable, new CircNullable),
   decorate(grammarFirst, new CircFirst))
  ))
```

The example shows how the three base algebras (`grammarNullable`, `grammarFirst`, and `grammarParse`) are first decorated with fixpointing and memoization behaviors. The resulting algebras are then composed using two invocations of `combine`.

The algebra `f` creates grammars with all three features built in:

```
// A ::= | "a" A
```

```
val g = Map("A" ->
  f.Alt(f.Empty, f.Seq(f.Terminal("a"), f.NonTerminal("A"))))
val s = g("A") // start symbol
s.parse(g, Seq("a", "a"), x => println("Yes"))
s.first(g) // -> Set("a")
s.nullable(g) // -> true
```

To look up non-terminals, all methods on grammars (`parse`, `first`, and `nullable`) receive the complete grammar as their first argument. The `parse` method furthermore gets an input sequence and a continuation that is called upon success.

8 Related Work

Generally speaking what distinguishes our work from previous work is the support for an expressive form of dynamic FOP that: 1) fully supports modular type-checking and separate compilation; 2) uses only well-studied, lightweight language features which are readily available in existing languages (Scala); 3) has very small composition overhead. In contrast most existing work either uses static forms of composition, which requires heavier language features and often has significant composition overhead; requires completely new languages/calculi; or does not support modular type-checking and separate compilation.

Feature-Oriented Programming To overcome the limitations of existing programming languages many of the existing FOP techniques [2–4, 6, 27, 41, 45] use their own language mechanisms and tools. *Mixin layers* [45] and extensions like *aspectual mixin layers* [4] are examples of language mechanisms typically used by such tools. Conceptually, mixin layers and aspectual mixin layers are quite close to our composition mechanisms, since our delegation-based model of composition has much of the same layering flavor. A difference is that mixin layers use static composition whereas we use run-time composition. As discussed throughout the paper, the `Lifter` interface used by the `merge` combinator plays a similar role to lifters in Prehofer's FOP approach [41]. Most of these language mechanisms and tools are implemented through code-generation techniques, which generally leads to an efficient and easy implementation strategy. However, this easy implementation strategy often comes at the cost of desirable properties such as separate compilation and modular type-checking. In contrast our object algebra based approach is fully integrated in an existing general purpose language (Scala); uses only already available and well-studied language features; and has full support for separate compilation and modular type-checking. The main drawback of our run-time composition mechanisms is probably performance, since delegation adds overhead which is difficult to eliminate.

More recently, researchers have also developed calculi for languages that support FOP and variants of it [1, 16, 44, 47]. These languages and calculi deal with import concerns such as type-checking or program analysis of all possible feature configurations. New languages developed for FOP typically provide novel language constructs that make features and their composition a part of the language. In contrast, our approach is to reuse existing programming language

technology and to model features and feature composition with existing OO concepts. An advantage of our approach is that by using standard programming language technology all the infrastructure (type-checking, program analysis, testing, tool support) that has been developed and studied throughout the years for that language is immediately available.

Family polymorphism Our work can be seen as an approach to *family polymorphism* [17], but it has significantly different characteristics from most existing approaches. Traditional notions of family polymorphism are based the idea of families grouping complete *class* hierarchies and using *static*, inheritance-based composition mechanisms. Most mechanisms used for family polymorphism, such as *virtual classes* [18], or *virtual types* [9] follow that traditional approach. In contrast our work interprets family polymorphism at the level of *objects* instead of *classes* and uses *run-time*, delegation-based composition mechanisms. One approach that also uses families of objects and run-time (delegation-based) composition is Ostermann's *delegation layers* [40]. As such delegation layers are conceptually closer to our approach. However, for modelling delegation layers support for virtual classes and delegation is assumed, which makes the approach quite heavyweight in terms of required language features.

Not many mechanisms that support family polymorphism are available in existing production languages. The CAKE pattern [33, 52], in the Scala programming language, is an exception. This pattern uses *virtual types*, *self types*, *path-dependent types* and (static) *mixin composition* to model family polymorphism. Even with so many sophisticated features, composition of families is quite heavyweight and manual. A particularly pressing problem, which has been acknowledged by the authors of Scala [52], is the lack of *deep* mixin composition. This means that all classes within a family have to be manually composed and then, the family itself has to be composed. In contrast, combinators like `merge` or `combine` take care of the composition of objects and the family.

Object algebras, visitors, embedded DSLs and Church encodings Section 2 already discusses how this work addresses the problem of limited expressiveness and large composition overhead required on previous work on object algebras. Object algebras are an evolution of a line of work which exploits Church encodings of datatypes [7] for various purposes: modular and generic visitors [35, 39], embedded DSLs [10,24], and generic programming libraries [23,37]. Most of that work considers only the creation of objects with single features, but not much attention is paid to creating objects composed of multiple features. Hofer et al. [24] use delegation in an optimization example. Their use of delegation is analogous to our use of dependent features in the printing operation presented in Section 6.2. However we express dependencies using a bound on the self-reference type, and use `merge` in client code to compose features. Their approach is more manual as they have to delegate behavior for each case. Furthermore they use pairs instead of intersection types and do not consider self-references. The generalization of object algebra interfaces in Fig. 10 was first used by Oliveira et al. to provide a unified interface for visitor interfaces. Oliveira [35] also explored a kind of

FOP using modular visitors. However there is a lot of overhead involved to create feature modules and compose features; and the approach requires advanced language features and does not deal with self-references.

9 Conclusion

Feature-oriented programming is an attractive programming paradigm. However it has been traditionally difficult to provide language mechanisms with the benefits of FOP, while at the same time having desirable properties like separate compilation and modular type-checking.

This work shows that it is possible to support FOP with such desirable properties using existing language mechanisms and OO abstractions. To accomplish this we build on previous work on object algebras and 2 well-studied studied programming language features: *intersection types* and *type-constructor polymorphism*. Object algebras provide the basic support for modularity and extensibility to coexist with separate compilation and modular type-checking. Intersection types and type-constructor polymorphism provide support for the development of safe and expressive composition mechanisms for object algebras. With those composition mechanisms, expressing *feature interactions* becomes possible, thus enabling support for FOP.

Although we have promoted the use of standard programming language technology for FOP, there is still a lot to be gained from investigating new programming language technology to improve our results. Clearly the investigation of better compilation techniques for object algebras and composition mechanisms is desirable for improving performance. Better language support for delegation would allow for more convenient mechanisms for object-level reuse, which are often needed with our techniques. Expressiveness could also be improved with new programming languages or extensions. For example when multi-sorted object algebras [36] are required the `Algebra` interfaces still have to be adapted. Although more powerful generic programming techniques can address this problem, this requires more sophisticated general purpose language features. With built-in support for object algebras as well as their composition mechanisms, there is no need for such advanced generic programming features and users may benefit from improved support for error messages.

References

1. Apel, S., Kästner, C., Grösslinger, A., Lengauer, C.: Type safety for feature-oriented product lines. Automated Software Engg. 17(3) (Sep 2010)
2. Apel, S., Kastner, C., Lengauer, C.: Featurehouse: Language-independent, automated software composition. In: ICSE '09 (2009)
3. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In: GPCE '05 (2005)
4. Apel, S., Leich, T., Saake, G.: Aspectual mixin layers: aspects and features in concert. In: ICSE '06 (2006)

5. Apel, S., Kästner, C.: An overview of feature-oriented software development. Journal of Object Technology 8(5), 49–84 (2009)
6. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. on Softw. Eng. 30(6), 355–371 (2004)
7. Böhm, C., Berarducci, A.: Automatic synthesis of typed lambda-programs on term algebras. Theor. Comput. Sci. pp. 135–154 (1985)
8. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: OOPSLA '04 (2004)
9. Bruce, K.B., Odersky, M., Wadler, P.: A statically safe alternative to virtual types. In: ECOOP'98 (1998)
10. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. 19, 509–543 (September 2009)
11. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for java. In: OOPSLA '00 (2000)
12. Compagnoni, A.B., Pierce, B.C.: Higher-order intersection types and multiple inheritance. Math. Structures Comput. Sci. 6(5), 469–501 (1996)
13. Cook, W.R., Palsberg, J.: A denotational semantics of inheritance and its correctness. In: OOPSLA '89 (1989)
14. Cook, W.R.: Object-oriented programming versus abstract data types. In: Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages. pp. 151–178. Springer-Verlag (1991)
15. Coppo, M., Dezani-Ciancaglini, M.: A new type-assignment for λ-terms. Archiv. Math. Logik 19, 139–156 (1978)
16. Damiani, F., Padovani, L., Schaefer, I.: A formal foundation for dynamic delta-oriented software product lines. In: GPCE '12 (2012)
17. Ernst, E.: Family polymorphism. In: ECOOP '01 (2001)
18. Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: POPL '06 (2006)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns : Elements of Reusable Object-Oriented Software. Addisson-Wesley professional computing series, Addisson-Wesley (1994)
20. van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA01). IEEE Computer Society (2001)
21. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Informatica (1978)
22. Harrison, W., Ossher, H.: Subject-oriented programming (A critique of pure objects). In: OOPSLA '93 (1993)
23. Hinze, R.: Generics for the masses. Journal of Functional Programming 16(4-5), 451–483 (2006)
24. Hofer, C., Ostermann, K., Rendel, T., Moors, A.: Polymorphic embedding of DSLs. In: GPCE '08 (2008)
25. Jones, P., Wollrath, A., Scheifler, R., et al.: Method and system for dynamic proxy classes (2005), uS Patent 6,877,163
26. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP (1997)
27. Lopez-herrejon, R.E., Batory, D., Cook, W.: Evaluating support for features in advanced modularization technologies. In: ECOOP '05 (2005)
28. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1), 1–13 (2008)

29. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: new-age components for old-fashioned java. In: OOPSLA '01 (2001)
30. Moors, A., Piessens, F., Odersky, M.: Generics of a higher kind. In: OOPSLA '08 (2008)
31. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA '06 (2006)
32. Odersky, M.: The Scala Language Specification, Version 2.9. EPFL (2011), http://www.scala-lang.org/docu/files/ScalaReference.pdf
33. Odersky, M., Zenger, M.: Scalable component abstractions. In: OOPSLA '05 (2005)
34. Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: Mri: Modular reasoning about interference in incremental programming. Journal of Functional Programming 22, 797–852 (2012)
35. Oliveira, B.C.d.S.: Modular visitor components. In: ECOOP'09 (2009)
36. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses: Practical extensibility with object algebras. In: ECOOP'12 (2012)
37. Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Trends in Functional Programming (2006)
38. Oliveira, B.C.d.S., Moors, A., Odersky, M.: Type classes as objects and implicits. In: OOPSLA '10 (2010)
39. Oliveira, B.C.d.S., Wang, M., Gibbons, J.: The visitor pattern as a reusable, generic, type-safe component. In: OOPSLA '08 (2008)
40. Ostermann, K.: Dynamically composable collaborations with delegation layers. In: ECOOP '02 (2002)
41. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: ECOOP '97 (1997)
42. Reynolds, J.C.: User-defined types and procedural data structures as complementary approaches to type abstraction. In: Schuman, S.A. (ed.) New Directions in Algorithmic Languages, pp. 157–168 (1975)
43. Reynolds, J.: Towards a theory of type structure. In: Programming Symposium, Lecture Notes in Computer Science, vol. 19, pp. 408–425 (1974)
44. Schaefer, I., Bettini, L., Damiani, F.: Compositional type-checking for delta-oriented programming. In: AOSD '11 (2011)
45. Smaragdakis, Y., Batory, D.: Implementing layered designs with mixin layers. In: ECOOP '98. pp. 550–570 (1998)
46. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.M.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99 (1999)
47. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. In: GPCE '07 (2007)
48. Torgersen, M.: The expression problem revisited – four new solutions using generics. In: ECOOP'04 (2004)
49. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list
50. Wehr, S., Thiemann, P.: JavaGI: The interaction of type classes with interfaces and inheritance. ACM Trans. Program. Lang. Syst. 33 (July 2011)
51. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: ICFP '01 (2001)
52. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL'05 (2005)