Taming Aspect Composition: A Functional Approach

Roberto E. Lopez-Herrejon and Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{rlopez, batory}@cs.utexas.edu

Abstract

Aspect Oriented Programing is a promising paradigm that challenges traditional notions of program modularity. Despite its increasing acceptance, aspects have been documented to suffer limited reuse, unpredictable behavior, and difficult modular reasoning. We develop an algebraic model that treats aspects as program transformations and uncovers aspect composition as the source of the problems mentioned. We propose an alternative model of composition that eliminates these problems, preserves the power of aspects, and lays out an algebraic foundation on which to build and understand AOP tools.

1 Introduction

Aspect Oriented Programming (AOP) is a promising paradigm that challenges and enhances traditional notions of program modularity [11]. It has been widely applied to different languages but the most influential implementation is AspectJ [5][11][18]. AspectJ has sophisticated and powerful modularization mechanisms that bring clear benefits over traditional modules but also has equally significant drawbacks. Aspects have been documented to suffer limited reuse [12], unpredictable behavior [22], and difficult modular reasoning [9][1]. All these factors hinder useful software engineering practices such as step-wise development [10] and its natural materialization in *Component-Based Software Engineering (CBSE)* [27], where programs are developed incrementally by composing components one at a time.

Aspect semantics are defined by an event-based model [28]. We take an unconventional perspective in this paper by viewing aspects as program transformations [23]. A *program transformation* is a function that maps programs. Transformations provide an alternative but equally valid semantics for aspects as it shifts the focus from a programmers perspective to that of the compiler, where all crosscuts (both static and dynamic) are seen as transformations. Adopting this perspective raises aspects from code artifacts to mathematical entities (functions from programs to programs) and enables the development of mathematically-based models of aspects and their composition. Doing so allows us to show that the primary source of complexity in AspectJ is how aspects are composed.

We take this approach further by modeling crosscuts, aspect composition, and weaving as an algebra. We propose an alternative model of aspect composition that eliminates the above-mentioned problems *while preserving the power of AspectJ*. We believe this model lays an algebraic foundation on which to build and understand emerging modularization technologies and tools.

2 AspectJ Overview

AspectJ[1] is an extension of Java. Its goal is to modularize *aspects*, concerns that crosscut traditional module boundaries such as classes and interfaces, that would otherwise be scattered and tangled with the implementation of other concerns [5]. AspectJ has two types of crosscuts, static and dynamic, that we illustrate and interpret as transformations.

2.1 Static Crosscuts

Static crosscuts affect the static behavior of programs. AspectJ supports several types of static crosscuts [5][18]. In this paper we focus on *introductions*, also known as *inter-type declarations*, that add fields, methods, and constructors to existing classes and interfaces.

In AspectJ, standard Java classes and interfaces are referred to as *base code*. Consider class `Point` defined below:

```
class Point {
  int x;
  void setX(int v) { x = v; }
}                                                    (1)
```

The following aspect `TwoD` adds (introduces) a second coordinate value to class `Point`. It adds field `y` and method `setY` to `Point`:

```
aspect TwoD {
  int Point.y;
  void Point.setY(int v) { y = v; }
}
```

When these two files are composed or *woven* by the AspectJ compiler `ajc` using the command:

```
ajc Point.java TwoD.java
```

1. We used version 1.2.1 for this paper.

The result is a new class `Point'` with the introduced members underlined below:

```
class Point' {
  int x;
  void setX(int v) { x = v; }
  int y;
  void setY(int v) { y = v; }
}                                                    (2)
```

AspectJ generally uses more sophisticated rewrites than those shown in this paper. The composed code snippets we present simplify illustration and are behaviorally equivalent to those produced by `ajc`.

Static crosscuts as transformations. From the program transformation perspective, base code such as `Point` in `(1)` represents a value to which a function (a program transformation) or aspect is applied. For instance, class `Point'` in `(2)` can be written as the following expression:

```
Point' = TwoD ( Point )
```

That is, `Point` is a base program (or value) and `TwoD` is a function that maps `Point` to `Point'`.

2.2 Dynamic Crosscuts

Dynamic crosscuts, in contrast, run additional code when certain events occur during program execution. The semantics of dynamic crosscuts are understood and defined in terms of an event-based model [28]. As a program executes, different events fire. These events are called *join points*. Examples of join points are: variable reference, variable assignment, execution of a method body, method call, etc. A *pointcut* is a predicate that selects a set of join points. *Advice* is code executed before, after, or around each join point matched by a pointcut.

The following aspect is the familiar logging example. Its interpretation is: run the advice code (underlined) `after` (advice type) the execution of methods in class `Point` whose name starts with '`set`' (*pointcut in italics*).

```
aspect Logging {
  after():  execution(* Point.set*(..))
  { println("Logged"); }
}                                                    (3)
```

From a compiler perspective, an equivalent interpretation is: *insert* the advice code after the body of any method in class `Point` whose name starts with '`set`'. For example, if aspect `Logging` is woven into class `Point'` in `(2)` the result is equivalent to:

```
class Point" {
  int x;
  void setX(int v) {x = v; println("Logged"); }
  int y;
  void setY(int v){ y = v; println("Logged"); }
}                                                    (4)
```

Dynamic crosscuts as transformations. Like static crosscuts, dynamic crosscuts are also transformations. For example, class `Point"` in `(4)` can be written as the expressions:

```
Point" = Logging(Point') = Logging(TwoD(Point))
```

That is, class `Point"` is the result of applying two transformations, or from an AOP perspective the result of weaving two aspects, into class `Point`.

AspectJ provides an array of sophisticated mechanisms to define powerful pointcuts and to perform complex rewrites when weaving aspects into programs. We argue that all dynamic crosscuts can be understood as transformations including pointcut designators such as `cflow`, `args`, `this`, and `target`, which expose context information of a join point [5].

Consider `cflow(Y)` where `Y` is a pointcut. Suppose `Y` captures a specific method execution or method call. `cflow(Y)` is the set of join points that occur during the execution of `Y`, from the time that the method is called to the time of the return [5]. An interesting question to ask is if a join point `X` occurs within the control flow of `Y`? The pointcut that expresses this is concisely written in AspectJ as:

```
cflow( Y ) && pointcut_for_X
```

From a compiler's perspective, control flow advice is a transformation that is composed from four simple transformations: (i) introduce a control flow stack `S`, (ii) before each `Y` join point, push a marker `M` on `S`, and (iii) after each `Y` join point, pop `M` off `S`. For the duration that `M` is on `S`, any join point that occurs does so within the control flow of `Y`. And finally, (iv) at each `X` join point, check to see if `M` is on `S`; if so, execute the advice code.

In general, aspect compilers, such as `ajc`, demonstrate that aspects are transformations: `ajc` takes a base program and aspects as input and produces a woven program as output. Even so, regarding dynamic crosscuts, especially `cflow`, as transformations remains controversial [15][13]. However, when given proper consideration, optimization and weaving techniques such as those presented in [4][14][21] are examples of program transformations, sophisticated indeed, but transformations nonetheless.

2.3 Advice Precedence

Recognizing aspects are transformations is a key first step in understanding AspectJ. The next step is to understand how aspects are composed.

Multiple pieces of advice can be applied to the same join point. *Advice precedence* determines the order in which advice is woven. AspectJ deals with precedence differently depending on where the pieces of advice are defined, either in the same aspect or in different aspects [5].

Advice in different aspects. AspectJ programmers can optionally specify a `declare precedence` statement in an aspect such as:

```
declare precedence: Aspect3, Aspect2, Aspect1;
```

Aspects are listed from higher to lower precedence (precedence order), and are woven in reverse order (weaving order). In the above example, the advice of $\texttt{Aspect}_1$ is woven first, then the advice of $\texttt{Aspect}_2$, and finally the advice of $\texttt{Aspect}_3$, the aspect with the *highest precedence* in this declaration.[2] If there is no `declare precedence` statement, precedence of aspects is undefined. In such cases, *the AspectJ compiler chooses the order in which to weave aspects*; *users do not know what order will be selected.*[3]

To understand the significance of this rule, consider an elementary question in mathematics: given a value (`2`) and functions that double its input `double(x)`, adds three to its input `add3(x)`, and subtracts two `sub2(x)`, what is the result of their composition? *This question makes no sense mathematically because the order in which functions are composed matters* (i.e., `double(sub2(add3(2))` ≠ `sub2(double(add3(2))`). This is the reason why *expressions*, not *sets of operations*, are evaluated. The counterpart to this question from a program transformation perspective is: given a value (`BaseProgram`) and functions (`aspect1(x)`, `aspect2(x)`, `aspect3(x)`), what is the result of their composition? This is the question that AspectJ poses to its users, and it makes no sense mathematically for the same reasons given earlier. The situation is actually worse because AspectJ decides the aspect composition order and hence determines the semantics of a program. *This means that the semantics of a program produced by AspectJ can be unpredictable.* This ordering rule is a problem in AspectJ's model of composition.

Advice within an aspect. The precedence of advice within an aspect is governed by the following rules, copied verbatim from [5]:

If two pieces of advice are defined in the same aspect, then there are two cases:

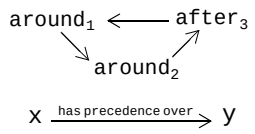
- *If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier.*
- *Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.*

2. Note that the mathematical concept of precedence has exactly the opposite meaning than precedence in AspectJ. AspectJ precedence means apply last, whereas mathematical precedence means apply first.

3. A special case is when an aspect extends another aspect, advice from a subaspect has higher precedence (i.e., is woven later) than that of its superaspect. This can be changed with `declare precedence` [5][18].

These rules present two problems: 1) they can lead to precedence circularity, i.e., the compiler does not know in which order to weave advice, and 2) they cannot express all composition (weaving) orders. The circularity problem is well-known [5], but the latter is not. A single aspect with three pieces of advice (identified by subscripts) illustrates both:

```
aspect Circular {
  void around1():execution(void test.main(..))
   { println("A1"); proceed(); println("A1"); }
  after3(): execution(void test.main(..))
   { println("A3"); }
  void around2():execution(void test.main(..))
   { println("A2"); proceed();println("A2"); }
}                                                    (5)
```



First, when the rules are applied, a circular precedence is created, as illustrated in the diagram to the right. To resolve the problem, programmers must manually reorder advice and ensure the resulting order eliminates circularity and leads to a semantically appropriate weaving for the task at hand, a non-trivial and lengthy process.

Second, some composition orderings are impossible to realize. Suppose we want the following output sequence (`A2`, `A1`, `<main>`, `A1`, `A3`, `A2`), which is achieved by weaving $\texttt{around}_1$ first, then $\texttt{after}_3$, and then $\texttt{around}_2$. In what order should advice $\texttt{around}_1$, $\texttt{after}_3$, and $\texttt{around}_2$ be listed in an aspect to achieve this weaving order?

The above rules dictate that $\texttt{around}_2$ must be listed before $\texttt{around}_1$ (because $\texttt{around}_1$ must be woven first). Advice $\texttt{after}_3$ must also appear before $\texttt{around}_2$ (to weave $\texttt{after}_3$ first). Thus, the ordering so far is: $\texttt{after}_3$ then $\texttt{around}_2$ then $\texttt{around}_1$. But $\texttt{after}_3$ must also appear after $\texttt{around}_1$ (for $\texttt{around}_1$ to be woven before $\texttt{after}_3$). It is impossible for $\texttt{after}_3$ to be *both* before $\texttt{around}_2$ and after $\texttt{around}_1$. Thus no linear ordering of the advice $\texttt{around}_1$, $\texttt{around}_2$, and $\texttt{after}_3$ can achieve the desired weaving order. It is challenging to apply the rules. We invite readers to try it.

The only way to realize such a weaving is to store each advice in a separate aspect and use `declare precedence`:

```
declare precedence: Around2, After3, Around1;
```

Pragmatically this means that it is in general impossible to compose individual aspect files to produce a compound aspect file.

In summary, the current rules for precedence make reasoning with multiple aspects unnecessarily difficult. But precedence is not the only problem with aspect composition. Fundamental software engineering practices such as stepwise development are not satisfactorily supported by AspectJ, as the following section shows.

3 An Incremental Development Example

Incremental or step-wise development is a fundamental and common programming practice [7][10][27]. It aims at building complex programs from simpler ones by progressively adding programmatic details. We use class `Point` of Section 2 but here we see it from an incremental development perspective. Our example consists of four steps. We use subscripts to denote the version of `Point` at a given step and underline the code that is added by each increment.

Base. Class $\texttt{Point}_0$ defines a 1-dimensional point with an `x` coordinate and corresponding `setX` method:

```
class Point0 {
   int x;
   void setX(int v) { x = v; }
}                                                    (6)
```

First increment. Adds coordinate `y` and its `setY` method to $\texttt{Point}_0$. The result is:

```
class Point1 {
   int x;
   void setX(int v) { x = v; }
   int y;
   void setY(int v) { y = v; }
}
```

Second increment. Counts how many times the set methods are executed. Adding both increments to base yields:

```
class Point2 {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int counter = 0;
}
```

Third increment. Adds a `color` field and its corresponding set method to $\texttt{Point}_2$:

```
class Point3 {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int counter = 0;
   int color;
   void setColor(int c) { color = c; }
}
```

Now let us consider how to implement this example using AspectJ.

Base. Identical to `(6)` because classes are the base code of AspectJ applications.

First increment. We define aspect `TwoD` with two introductions to class `Point`, one to add field `y` and the other to add method `setY`:

```
aspect TwoD {
   int Point.y;
   void Point.setY(int v) { y = v; }
}
```

The command that composes class $\texttt{Point}_0$ and aspect `TwoD` and achieves a program equivalent to $\texttt{Point}_1$ is:

```
ajc Point0.java TwoD.java
```

Second increment. Aspect `Counter` introduces field `counter` to class `Point` and advises the execution of all set methods to increment this counter[4]:

```
aspect Counter {
  int Point.counter = 0;
  after(Point p) : execution(* Point.set*(..))
                   && target(p)
  { p.counter++; }
}                                                    (7)
```

The composition that produces a program equivalent to $\texttt{Point}_2$ is:

```
ajc Point0.java TwoD.java Counter.java
```

Third increment. Aspect `Color` adds a `color` field and a `setColor` method:

```
aspect Color {
 int Point.color;
 void Point.setColor(int c) { color = c; }
}
```

The composition of the base with the three increments is achieved by:

```
ajc Point0.java TwoD.java Counter.java Color.java
```

However, this time the result is not $\texttt{Point}_3$, but instead:

```
class Point3' {
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int counter;
   int color;
   void setColor(int c){ color=c; counter++; }
}
```

That is, the `setColor` method of $\texttt{Point}_3'$ increments `counter` (underlined above) unlike $\texttt{Point}_3$. *Pragmatically this means that developers face the paradox that building a program incrementally using AspectJ and manually may yield different results.* To produce $\texttt{Point}_3$ using AspectJ, we should have used a more constrained version of `Counter` that captures execution join points only of `setX` and `setY` methods:

4. In AspectJ there are other ways to define `Counter`. Shortly, we will present the rationale behind this implementation decision.

```
aspect Counter {
  int Point.counter = 0;
  after(Point p) : (execution(* Point.setX(..))
                 || execution(* Point.setY(..)))
                 && target(p)
  { p.counter++; }
}                                                    (8)
```

An obvious question is: why was `Counter` not defined like `(8)` in the first place? Doing that certainly would solve *this* problem, but we must consider other properties of software modules that are also desirable for aspects. Among them is reusability, i.e., we want to treat aspects as components as in CBSE and reuse them *as is*. For example, suppose `Counter` is redefined as `(8)`, but now we want to build program $\texttt{Point}_3\texttt{'}$ instead. We would have to revise `Counter` back to `(7)` as the version in `(8)` cannot be used. The question is: why can we not reuse the same aspect for both cases? The problem is that aspect weaving does *not* distinguish among development stages of a program. We show how to solve this problem in the next section.

4 An Algebraic Model of Aspects

In this section, we develop an algebraic model that reveals the complexity of AspectJ composition. We then propose an alternative model of composition that is based on program transformations, retains the power of AspectJ, supports step-wise development, simplifies advice precedence, and facilitates reasoning using aspects. Our model has three operations that build upon the notions of introduction, advice, and weaving. We start with a model of introduction and its associated operation.

4.1 Introduction Addition

An *introduction* is a function that maps an input program to an augmented output program. Recall aspect `TwoD` and class $\texttt{Point}_0$ whose composition was modeled algebraically as:

$$\texttt{Point}_1 = \texttt{TwoD}\ (\ \texttt{Point}_0\ ) \qquad (9)$$

where $\texttt{Point}_0$ and $\texttt{Point}_1$ are values, `TwoD` is a function that maps class $\texttt{Point}_0$ to class $\texttt{Point}_1$. Appealing to intuition, we can rewrite `(9)` as a summation of the introductions of `TwoD` with $\texttt{Point}_0$:

$$\texttt{Point}_1 = \texttt{TwoD} + \texttt{Point}_0 \qquad (10)$$

Operation + is called *introduction addition*. It is a binary operation that performs disjoint union on *program fragments*, which are sets of variables and methods. The introductions of an aspect define a program fragment, as they too form a set of variables and methods. Variable and method names can be qualified to indicate their class or interface membership; this is especially useful when dealing with multiple classes and packages.

+ denotes disjoint set union. To simplify notation we omit set brackets `{}`. For example, aspect `TwoD` is the program fragment (set) containing `y` and `setY`, and class $\texttt{Point}_0$ is the fragment (set) containing `x` and `setX`. Their addition is:

$$\begin{aligned}\texttt{Point}_1 &= \texttt{TwoD} + \texttt{Point}_0\\ &= (\texttt{setY} + \texttt{y}) + (\texttt{setX} + \texttt{x})\\ &= \texttt{setY} + \texttt{y} + \texttt{setX} + \texttt{x}\end{aligned}$$

meaning $\texttt{Point}_1 = \{\texttt{setY, y, setX, x}\}$.

As + is disjoint set union, introduction addition has the following properties:

Identity. `0` is the *empty program* (i.e., a program fragment that contains no members). If `X` is a program fragment:

$$\texttt{X} = \texttt{X} + \texttt{0} = \texttt{0} + \texttt{X}$$

Commutativity. + is commutative because set union is commutative.

Associativity. + is associative because set union is associative. This is a useful property, as it allows us to substitute definitions. For example, `TwoD` is the sum:

$$\texttt{TwoD} = \texttt{setY} + \texttt{y}$$

which we could substitute into `(10)` to produce an equivalent definition of $\texttt{Point}_1$:

$$\texttt{Point}_1 = \texttt{setY} + \texttt{y} + \texttt{Point}_0$$

Operation + differs from AspectJ introduction in two regards: a) Introduction addition does not support member overriding. We believe overriding is rarely used and can be circumvented with a more structured design. b) Introduction addition supports the introduction of new classes. This distinction is important as adding functionality to programs often requires new classes [20].[5] For example, suppose program fragment `F` contains class `Z` with member `r` and class `W` with member `t`. This is written as:

$$\texttt{F} = \texttt{Z.r} + \texttt{W.t}$$

Fragment `G` contains class `Q` with member `u` and class `W` with member `v` is:

$$\texttt{G} = \texttt{Q.u} + \texttt{W.v}$$

The introduction summation of `F` and `G` is their union: class `Z` with member `r`, class `W` with members `t` and `v`, and class `Q` with member `u`:

$$\begin{aligned}\texttt{F} + \texttt{G} &= (\texttt{Z.r} + \texttt{W.t}) + (\texttt{Q.u} + \texttt{W.v})\\ &= \texttt{Z.r} + (\texttt{W.t} + \texttt{W.v}) + \texttt{Q.u}\end{aligned}$$

That is, the set of terms `{W.t, W.v}` with the same class prefix `W` are members of class `W`. In general, + allows us to

5. Aspects *can* encapsulate nested classes and nested interfaces, but *not* classes and interfaces that are not nested.

add any number of new classes, interfaces, and new members to existing program fragments.

4.2 Advice Addition

Recall the `Logging` aspect:

```
aspect Logging {
  after(): execution(* Point.set*(..))
  { println("Logged"); }
}
```

Such advice can be regarded as an implicit method introduction and an implicit call to such method. For example, when woven into class $\texttt{Point}_0$ in (6), aspect `Logging` can be regarded as the transformation that results in:

```
class Point  {
   int x;
   void setX(int v) { x = v; printLog(); }
   static void printLog(){ println("Logged");}
}
```

Method `printLog` is an explicit method that contains the advice body (the log message) and it is called at the end of the body of method `setX` (after the method execution).

Pure advice is a named advice that separates the two concerns: method introduction and its corresponding method call. With this separation, we can conceptually rewrite aspect `Logging` as:

```
aspect Logging {
  static void Point.printLog()
   { println("Logged"); }

  Log is after():execution(* Point.set*(..))
        --> Point.printLog();
}
```

`Log` is the name given to the pure advice, `printLog` is the method that contains the advice body, and the `-->` arrow indicates the method call. We made this separation syntactically to help visualize our analysis.

We model an aspect as a vector of two entries. The first entry, called the *advice part*, is the aspect's advice and the second entry, called the *introduction part*, is the aspect's introductions. The vector for `Logging` is:

```
Logging = [Log, printLog]
```

As another example, consider aspect `Counter` of (7). A pure advice version of it is:

```
aspect Counter {
   CounterP is after(Point p):
     execution(* Point.set*(..)) && target(p)
     --> Point.counterInc(p);

   static void Point.counterInc(Point p)
     { p.counter++; }
   int Point.counter = 0;
}
```

and its vector is:

```
Counter = [CounterP, counterInc + counter]
```

Note that the second entry of the vector adds two introductions `counterInc` (the method of the advice body) and `counter` (the variable).

Consider the `Circular` aspect of (5) whose pure version could be:

```
aspect Circular {
   pointcut pcd() : execution(void test.main(..));
   static void test.m1(){ ... }
   static void test.m3() { ... }
   static void test.m2(){ ... }
   a1 is void around1(): pcd() --> test.m1();
   a3 is void after3():       pcd() --> test.m3();
   a2 is void around2(): pcd() --> test.m2();
}
```

This aspect contains several pieces of advice. Its vector requires another operation that we call *advice addition*. Denoted by ⊕, advice addition models advice precedence. ⊕ has simple semantics: `a3`⊕`a1` means apply advice `a1` first and then `a3`. Suppose advice is woven in the textual order listed in an aspect. `Circular` would have the vector:

```
Circular = [ a2 ⊕ a3 ⊕ a1, m2 + m3 + m1 ]
```

Pure advice is a function and ⊕ denotes function composition. Advice addition has the following properties:

Identity. `1` is the *null pure advice* (i.e., pure advice that does not capture any join points). Null advice corresponds to the identity transformation; its application does not affect a program. If `a` is pure advice:

```
a = a ⊕ 1 = 1 ⊕ a
```

Not Commutative. The order in which advice is applied matters. ⊕ is not commutative, just as function composition is not commutative.[6]

Associativity. ⊕ is associative as function composition is associative. Let `a1`, `a2`, and `a3` be pieces of pure advice. The following equalities hold:

```
a3 ⊕ a2 ⊕ a1= (a3 ⊕ (a2 ⊕ a1))
            = ((a3 ⊕ a2) ⊕ a1)
```

Associativity allows us to substitute definitions. For example, if `a21` = `a2`⊕`a1` then:

```
a3 ⊕ a21 = a3 ⊕ ( a2 ⊕ a1 )
```

4.3 Advice Weaving

Advice weaving, denoted by *, is the operation of transforming an input program into a program with advice code

6. Two pieces of advice commute when they capture disjoint sets of join points.

inserted. Suppose a is pure advice and P is a program. The result of applying (weaving) a into P is program P′:

$$P' = a*P$$

Advice weaving has the following properties:

Identity. 1 is the null pure advice — again, pure advice that does not capture any join points. If P is a program fragment, P = 1*P. That is, P does not change when woven with 1.

Right Associative. * is right associative. Let a2 and a1 be pure advice and P be a program. a2*a1*P means apply a1 to P first, then apply a2. Algebraically:

$$a2 * a1 * P = (a2 * (a1 * P))$$

Distributes over introduction addition. Advice weaving distributes over introduction addition, i.e., * distributes over +. Let P be a program, a be pure advice, and P′=a*P. Now suppose P=X+Y+Z, where X, Y, and Z are arbitrary program fragments. We have:

$$\begin{aligned} P' &= a*P \\ &= a*(X + Y + Z) \\ &= a*X + a*Y + a*Z \end{aligned}$$

Advice applies to *all* join points in the program to which it is woven. That is, advice is woven into all program fragments (classes, interfaces, and introductions) that appear in the right-hand operand of operation *. Thus it is immaterial if the program fragment is viewed as a whole (P) or as the summation of its parts (X+Y+Z). This distributivity property is central to AOP.

Weaving Axiom. Let a1, a2, a3, and a4 be pure advice and P be a program. The weaving axiom establishes that advice is woven from right to left:

$$\begin{aligned} &(a4 \oplus a3 \oplus a2 \oplus a1) * P \\ &\quad = (a4 \oplus a3 \oplus a2)* a1 * P \\ &\quad = (a4 \oplus a3)* a2 * a1 * P \\ &\quad = a4 * a3 * a2 * a1 * P \end{aligned}$$

All aspect compilers implement the Weaving Axiom — they all weave advice in weaving order.

Given the operations +, *, and ⊕, we are now ready to model aspect composition.

4.4 Aspect Composition

Let aspects A1 and A2 be modeled by the vectors $A1=[a_1,i_1]$ and $A2=[a_2,i_2]$. We denote the AspectJ aspect composition operation by ◊. The AspectJ composition of A2 with A1 (with A1 being applied first) is:

$$\begin{aligned} A2 \lozenge A1 &= [a_2,i_2] \lozenge [a_1,i_1] \\ &= [\ a_2 \oplus a_1,\ i_2 + i_1\ ] \end{aligned}$$

◊ is similar to vector addition; the coordinates of vectors are summed: + sums program fragments, ⊕ sums advice in weaving order.

As another example, program P is modeled by the vector [1,p]. 1 is null pure advice and p is the introduction sum of the members of P. Weaving aspect A1 into P and then weaving aspect A2 is:

$$\begin{aligned} &A2 \lozenge A1 \lozenge P \\ &\quad = [a_2,i_2] \lozenge [a_1,i_1] \lozenge [1,p] \\ &\quad = [\ a_2 \oplus a_1 \oplus 1,\ i_2 + i_1 + p] \\ &\quad = [\ a_2 \oplus a_1,\ i_2 + i_1 + p] \end{aligned}$$

Continuing with the vector analogy, we denote the program that results from weaving pure advice into program fragments as the *length* of the vector. Let V be a vector, its length |V| is computed by:

$$|V| = |\ [a,i]\ | = a*i$$

That is, the length of a vector is its advice part woven with its introduction part. This follows from the fact that aspects can advise all join points of a program[7]. Thus the program that is produced by weaving A1 and then A2 into P is the expression:

$$|A2 \lozenge A1 \lozenge P| = a_2 * a_1 * (\ i_2 + i_1 + p\ ) \quad (11)$$

More generally, for aspects $A_1 \ldots A_n$ to be woven in this order into P, the result is:

$$\begin{aligned} &|A_n \lozenge A_{n-1} \lozenge \ldots \lozenge A_1 \lozenge P| \\ &\quad = (a_n*a_{n-1}*\ldots a_1)*(i_n+i_{n-1}+\ldots+i_1+p) \end{aligned} \quad (12)$$

That is, the result of weaving a sequence of aspects into a program equals the weaving of advice in weaving order into the program that is the introduction sum of the program's members and aspect introductions. (12) represents the "shape" or *architecture* of any program produced by AspectJ. We refer to this model of composition as the *Vector Model*.

We can use this result to identify the source of the problems noted earlier in Section 3 about incremental program development using AspectJ. It can be seen in the expansion of (11):

$$\begin{aligned} &|A2 \lozenge A1 \lozenge P| \\ &\quad = a_2 * a_1 * (i_2 + i_1 + p) \\ &\quad = a_2*\underline{a_1}*i_2 + a_2*a_1*i_1 + a_2*a_1*p \end{aligned}$$

The offending term is underlined. It means that to apply aspect A2, the programmer is required to know how an advice from a previous development step (a_1) affects an introduction added in the current step (i_2). More generally, the products that cause problems in incremental development are underlined below:

7. Readers familiar with advice that advises itself will recognize that the pure advice part advises its introduction part.

$\ldots * a_{k+2} * a_{k+1} * a_k * \underline{a_{k-1}} * \underline{\ldots} * \underline{a_2} * \underline{a_1} * i_k + \ldots$

In other words, a programmer needs to know how previously applied pieces of pure advice a_j affect later introductions i_k where $j < k$. These are the terms that make step-wise development difficult. This problem is aggravated when a large number of aspects are composed and the development involves multiple steps.

4.5 The Functional Model

An alternative way to compose aspects is to equate *aspect composition with function composition* (hence the name of the model). Let aspect `A=[a,i]`. We can model `A` as the function:

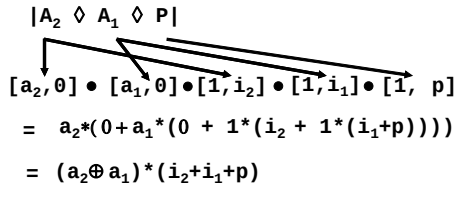
```
A(x) = a * ( i + x )
```

That is, `A` adds its introductions (`i`) to its program fragment input (`x`) before weaving its advice (`a`). So applying aspect `A1` to program `P` and then applying aspect `A2` is:

$$\begin{aligned} A2(A1(P)) &= a_2 * (i_2 + a_1 * (i_1 + p)) \\ &= a_2 * i_2 + a_2 * a_1 * i_1 + a_2 * a_1 * p \end{aligned} \quad (13)$$

Note that the offending pure advice a_1 disappears from the first summand ($a_2 * i_2$). This generalizes to the composition of any number of aspects: *the products that make step-wise development difficult are never generated.*

We now have two different models of aspect composition: the Functional Model (above) and the Vector Model of AspectJ. An obvious question arises: which model is more expressive?

> **Theorem**. The Functional Model can synthesize more programs than the Vector Model *provided that aspects are reused as is (i.e., pure advice and introductions are not modified).*
>
> **Proof**. An arbitrary Vector Model expression can be mechanically translated into an equivalent Functional Model expression. The figure below shows how to do this for the expression `|A2◊A1◊P|`.
>
> 
>
>
> `A2` is decomposed into its pure advice and introduction parts; the same for `A1`. Each part is a function, which when composed with `p`, sums introductions first and weaves advice last, yielding the program that would have been produced by AspectJ. This is a mechanical translation that can be applied to any Vector Model expression. However, translating an arbitrary Function Model expression into an Vector Model expression is impossible reusing aspects *as is*. Expression `(13)` is a program shape that cannot be produced by the Vector Model, because pure advice affects *all* introductions regardless of when and how they are added.♦

We can demonstrate the intuition behind the mathematics by recalling the `Point` example of Section 3. Let us add a third dimension to `Point`, which is defined by aspect `ThreeD` that introduces a `z` variable and `setZ` method. Assuming that the `Counter` aspect advises all set methods as in `(7)`, we can build at least 4 programs:

```
(a) Color( ThreeD( TwoD( Counter( Point0 ))))
(b) Color( ThreeD( Counter( TwoD( Point0 ))))
(c) Color( Counter( ThreeD( TwoD( Point0 ))))
(d) Counter( Color( ThreeD( TwoD( Point0 ))))
```

`(a)` is a program that counts the executions of `setX`. `(b)` counts the executions of `setX` and `setY`. `(c)` counts the executions of `setX`, `setY`, and `setZ`. `(d)` counts the execution of all set methods. Each of these programs is synthesized by reusing and composing `Counter` *as is*. Using AspectJ, *these four specifications would produce the same program — all would produce program (d). To build all four programs would require four different versions of* `Counter`. This example illustrates the Functional Model to be more expressive than the Vector Model for aspect reuse.

To summarize, problems in step-wise development arise using AspectJ when pointcuts are not bounded to a set of classes, methods, and variables at a specific stage of program development. Common examples are pointcuts that capture the set of all calls to one or more methods, and wildcard patterns. Subsequent introductions that are captured by these pointcuts give rise to the problems discussed here. These problems are avoided using the Functional Model.

5 Perspective

We are now investigating how to generalize the Functional Model to include other AspectJ capabilities such as `declare parents`, abstract aspects, abstract pointcuts, and aspect inheritance. Our goal is to build tools based on our model and to evaluate their potential in an experimental setting. To this end, we are collaborating with the *Aspect Bench Compiler (abc)* research group to create a joint project [6][3]. Even at this early stage, we can gauge the utility of our results and its relationship to other work.

5.1 Significance of Results

Our research can improve aspect composition in AspectJ in the following ways. First, the precedence rules for ordering pieces of advice within an aspect (Section 2.3) can be eliminated. We propose a simpler rule: apply advice in order in

which it is listed in an aspect file. We believe this rule will simplify the ordering algorithms currently utilized by aspect compilers and will help AspectJ programmers by reducing the effort to determine a composition order. As we have shown, no tool can compose aspects into a single compound aspect with the current set of precedence rules. With our changes, the composition of aspect files is possible.

Second, the rules that AspectJ uses to assign precedence to aspect files can also be eliminated. We propose that a precedence be declared for *all* aspect files to define their composition order. Alternatively, the compiler could raise an error when users fail to specify an order and where ordering matters. Again, we believe this change will simplify advice ordering algorithms for multiple aspect files and will also help AspectJ programmers because now compiler output will be predictable.

AOP researchers have raised the issue that it should be unnecessary to specify a composition ordering when aspects are provably commutative. We agree. In cases where pointcuts have disjoint sets of join points their corresponding advice is commutative. Existing tool support can help identify these situations [5]. However, it is still necessary to specify *when* these pieces of advice are to be applied. This may require an enhancement of existing tools.

5.2 Related Work

The relationship between program transformations and aspects is not new. Lämmel studied the implementation of aspects as programs transformations in the declarative paradigm [19]. Kniesel et. al developed `JMangler`, a backend tool to support AOP that relies in transformations at the byte-code level [17][11]. We extend these ideas by showing how a program transformation view can lead to an algebraic model of aspect composition.

McEachen and Alexander address the problems caused by weaving bytecode that already contains woven aspects [22]. A *foreign aspect* is an aspect that has been woven to a class (or classes) and whose resulting bytecode is later imported by a third party that has no access to the aspect's source code. Foreign aspects are problematic as they can: a) not capture all intended join points, b) capture unintended join points, c) inadvertently interact with other aspects. The authors advocate developer guidelines to design carefully the scope of pointcuts, use of abstract pointcuts to control the set of join points advisable by foreign aspects, and promote adequate pointcut documentation. Our work provides a foundation to understand the problems caused by foreign aspects and a solution to eliminate them. The Functional Model can be enforced by a compiler whereas enforcing guidelines are the responsibility of programmers.

Modular reasoning with AOP has been a controversial issue [9]. Kiczales and Mezini claim that in the presence of aspects "the complete interface of a module can only be determined once the complete configuration of modules in the system is known" [16]. In other words, aspects entail global reasoning that they define as "having to examine all the modules in the system or subsystems". The Vector Model of AspectJ mathematically corroborates their claim and shows the negative implications it has for incremental development. The Functional Model of composition eliminates the need of global reasoning without restricting the power of AspectJ.

Rinard et al. propose a framework to classify aspects based on their interactions with other aspects and base code [25]. They present program analysis tool that alerts users of cases where modular reasoning (a user-defined property) could be compromised so that they can take corrective action when deemed necessary. *Open modules* proposes a module system whereby an interface describes the pointcuts and join points that are advisable by the pieces of advice of other modules thus promoting modular reasoning about aspects [1].

Complementary to our approach, there is work that aims to improve aspect reuse by making significant language changes. Gybels and Brichau propose a logic-based crosscut language to better decouple aspects from programs [12]. Rho and Kniesel propose aspect *uniform genericity*, application of logic metavariables in language constructs, as a way to promote reuse and to significantly expand the generic capabilities of AspectJ [26]. Incidentally, they too take a transformation view of aspects.

Classpects are an attempt at unifying aspects and classes [24]. Classpects are classes enhanced with bindings. A *binding* associates an advice type (`before`, `after`, `around`) and a pointcut with a call to a list of methods. These methods replace the advice body, similar to what we did when we transformed advice into pure advice.

Feature Oriented Programming (FOP) is the study of feature modules for the synthesis of product-line programs [7]. A *feature* is an increment in program functionality. Features in FOP are composed by function composition, yielding programs whose semantics are predictable and unambiguous. A question often raised in the AOP community is: what is the difference between features and aspects? We believe there was no clear answer until this work. *Aspects and features are both increments in program functionality. What is different is their composition models: features use the Functional Model, aspects use the Vector Model.* Because their composition models differ, FOP and AOP are not directly comparable, that is FOP is not a subset of AOP, and vice versa. However, we believe both are instances of a

more general model, one that uses the full power of pointcuts and uses function composition to compose aspects.

The full implementation of the Functional Model will require advances in both aspect compilers and Java compilers. One of the issues is *separate compilation* [8][2], which is gaining attention in the programming languages community. Our need for separate compilation comes from the commutativity of operation +, which permits the introduction of a method before introducing other members on which the former may depend. The validation of the Functional Model does not require separate compilation of aspects as we can assume the existence of source code.

6 Conclusions

Aspect Oriented Programming should be in the repertoire of tools and techniques used by software developers. But the current model and flagship tool of AOP, AspectJ, has significant limitations: aspect reuse is hard, woven programs can have unpredictable behavior, modular reasoning using aspects is difficult, and step-wise development of programs is error-prone. We explored these limitations and found that the primary source of complexity in AspectJ is its model of aspect composition.

To address these problems, we took an unconventional approach that equates aspects with program transformations. Doing so allowed us to raise aspects from code artifacts to mathematical entities (functions from programs to programs) and enabled us to develop an algebra that modeled aspect composition. Our algebra not only exposed the source of the problems in aspect composition, it also revealed a solution. By equating aspect composition with function composition, the problems were eliminated and the power of AspectJ was preserved.

Our goal is to build tools to validate our algebra. While there is considerable work ahead, from our experience in program synthesis of product-lines [7], we are confident this goal can be accomplished. Further, we believe that our work lays an algebraic foundation on which to build and understand AOP tools.

Acknowledgements. We thank Oege de Moor, Chris Lengauer, Axel Rauschmayer, Jim Cordy, and Dewayne Perry for their comments on drafts of this paper.

This research is sponsored in part by NSF's Science of Design Project #CCF-0438786.

7 References

[1] J. Aldrich. Open Modules: Modular Reasoning about Advice. *ECOOP 2005*.

[2] D. Ancona, G. Lagorio, and E. Zucca, “True Separate Compilation of Java Classes”, *PPDP 2002*.

[3] P. Avgustinov, et al., “abc: An Extensible AspectJ Compiler”, *AOSD 2005*, Chicago, USA.

[4] P. Avgustinov, et. al. “Optimizing AspectJ”, *PLDI 200*5.

[5] AspectJ Manual, `http://www.eclipse.org/aspectj/doc/progguide/language.html`.

[6] Aspect Bench Compiler. `http://www.aspectbench.org`

[7] D. Batory, J.N. Sarvela, A. Rauschmayer, “Scaling Step-Wise Refinement”, *IEEE TSE*, June 2004.

[8] L. Cardelli, “Program Fragments, Linking, and Modularization”, *POPL 97*.

[9] C. Clifton, G.T. Leavens. “Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy”. SPLAT *2003*.

[10] E.W. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.

[11] R.E. Filman, T. Elrad, S. Clarke, M. Aksit. *Aspect-Oriented Software Development.* Addison-Wesley, 2004

[12] K. Gybels and J. Brichau, “Arranging Language Features for More Robust Pattern-based crosscuts”, *AOSD 2003*.

[13] K Gybels and K. Ostermann, discussions at *SPLAT 2005*.

[14] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. *AOSD 2004*.

[15] G. Kiczales, personal email communication, 2003.

[16] G. Kiczales, M. Mezini. “Aspect-Oriented Programming and Modular Reasoning”. *ICSE 2005*.

[17] G. Kniesel, P. Costanza, M. Austermann. “JMangler - A Framework for Load-Time Transformation of Java Class Files”. *SCAM 2001*.

[18] R. Laddad. *AspectJ in Action. Practical Aspect-Oriented Programming*. Manning, 2003.

[19] R. Laemmel, “Declarative Aspect-Oriented Programming”, *PEPM 1999*.

[20] R.E. Lopez-Herrejon, D. Batory, W. Cook. “Evaluating Support for Features in Advanced Modularization Techniques”. *ECOOP 2005*.

[21] H. Masuhara, G. Kiczales, “Modeling Crosscuting Aspect-Oriented Mechanisms”. *ECOOP 2003.*

[22] M. McEachen, R.T. Alexander. Distributing Classes wth Woven Concerns - An Exploration of Potential Fault Scenarios. *AOSD 2005.*

[23] Partsch, H., Steinbrüggen, R.: Program Transformation Systems. *ACM Computing Surveys*, September (1983).

[24] H. Rajan, K.J. Sullivan, “Classpects: Unifying Aspect- and Object-Oriented Programming”, *ICSE 2005.*

[25] M. Rinard, A. Salcianu, S. Bugrara. “A Classification System and Analysis for Aspect-Oriented Programs”, *FSE 2004*.

[26] T. Rho, G. Kniesel. LogicAJ - A Uniformly Generic Aspect Language. Submitted for publication.

[27] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 2002.

[28] M. Wand, G. Kiczales, C. Dutchyn, “A Semantics for Advice and Dynamic Join Points in Aspect Oriented Programming”, *TOPLAS 2004*.