JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©**JOT, 2008**

Vol. 0, No. 0, Z

Method Proxy-Based AOP in Scala

Daniel Spiewak and **Tian Zhao**
University of Wisconsin – Milwaukee, {dspiewak,tzhao}@uwm.edu

This paper describes a fully-functional Aspect-Oriented Programming framework in Scala – a statically typed programming language with object-oriented and functional features. This framework is implemented as internal domain-specific languages with syntax that is both intuitive and expressive. The implementation also enforces some static type safety in aspect definitions.

1 INTRODUCTION

Aspect-Oriented Programming (AOP) implementations such as AspectJ provide language extensions of pointcuts and advices to insert code of crosscutting concerns into the base program through bytecode transformation. In this paper, we describe a framework to implement an AOP extension to the Scala language [13] using higher-order functions as AOP proxies [1]. This framework allows programmers to specify pointcuts and aspects using a Domain Specific Language (DSL) [5] embedded within Scala. Our technique uses Scala's higher-order functions to intercept method calls with minimal syntactic overhead imposed on the base program. This framework allows developers to define pointcuts by specifying class types and method signatures. The framework also allows access to context variables, while aspects can insert advice code before or after the advised body.

The rest of the paper is organized as follows: In Section 2, we present the main idea using an example. Section 3 explains the syntax of pointcuts and advices. Section 4 explains the implementation details. We discuss the benefits and tradeoffs of our framework in Section 5. and related works are in Section 6.

2 EXAMPLE

AOP enables developers to write code that is modularized to a point beyond the capabilities of vanilla object-oriented design. This kind of modularization is expressed declaratively in the form of cross-cutting concerns. A simple example of this can be seen in the Java implementation of a basic `Circle` class:

```
public class Circle {
  private int x, y;

  public void setX(int x) {
```

[1]The source code is available at: `http://www.codecommit.com/scala-aop.zip`

```
      this.x = x;
      repaint();
  }
  public void setY(int y) {
      this.y = y;
      repaint();
  }
  private void repaint() { ...  }
}
```

The problem with this class is a tight coupling between the `Circle` class and the repeated invocation of the `repaint()` method. Every point in the class where values may have changed, code must be duplicated to make a call and ensure the screen is updated. Aspect-oriented programming provides a solution to this problem by treating the `Circle` repaint as a cross-cutting concern. AOP design says that the repaint logic and the logic dictating *when* to repaint should both be factored out into a separate module, called an *aspect*. An aspect contains *pointcuts* to specify sets of the joinpoints while *advices* specify the code to insert at various joinpoints using the pointcuts. An aspect is basically a `class` with additional constructs allowing it to define logic which is to be "weaved" into relevant classes. Using AspectJ:[2], we can define an aspect `CirclePainter` to insert calls to repaint method after each call to the setter methods of Circle class.

```
public aspect CirclePainter {
  pointcut move(): call(void Circle.setX(int)) ||
                   call(void Circle.setY(int));

  after(Circle c): move() && target(c) {
      c.repaint();
  }
}
```

Our goal is to create a framework which can enable aspect-oriented patterns within the confines of an object-oriented programming language. No bytecode manipulation should be utilized, nor should any compiler modifications be required. Existing language constructs should be used to provide a quantification syntax that is both powerful and flexible. Any further aims (such as completely transparent join point interception) are secondary and should not come before the goal of avoiding semantic alteration. Scala is a language with flexible syntax and incorporates features from both object-oriented and functional languages [13], so it seems to be an excellent target for our experiment.

The first concern for an AOP implementation is to determine how the method interception is to take place. As far as we know, there is no way to do this transparently. Taking the Scala implementation of a `Circle` class as an example:

```
class Circle {
  private var ix, iy

  def x_=(x:Int) = { ix = x }
  def y_=(y:Int) = { iy = y }
```

[2]http://www.eclipse.org/aspectj

```
}
```

Notice the differences between Scala and Java implementation of this example include the use of keyword "var" to declare to private variables x and y, and the use of keyword "def" to declare the methods to set values. The functions "x_=" and "y_=" are setter functions similar to "setX" and "setY" in the Java implementation of this example.

Scala does not provide a mechanism with which an external framework can intercept calls to arbitrary methods within the `Circle` class without modifying `Circle` itself. This can be considered a key requirement for transparent join-point weaving. Thus, our Scala AOP implementation must use a non-transparent mechanism for method interception. There are alternative ways such as using Mixins to override the method in question within the confines of the language. These techniques are discussed later in the paper.

The most obvious way to accomplish this would be to utilize a delegate. Every method in the `Circle` class would make a call to some singleton in the AOP framework which would actually perform the work of the method in question, triggering any advice given matching pointcuts. The delegate function will need to accept a functional which specifies the intended body of the original method. This functional will ensure that the delegate itself is not coupled to the methods weaved. By making use of Scala's powerful mixin mechanism combined with its alternate method call syntax, it is syntactically valid to construct a syntax closer to the following:

```
class Circle(...) extends AOP {
  ...
  /* weavable */
  def x_=(x:Int) = defun { ix = x  }

  /* not weavable */
  def y_=(y:Int) = { iy = y  }
}
```

where `defun` is a function within `AOP` which is a `trait`. Note that the code block $\{\ ix = x\ \}$ is treated as an anonymous function; while `defun` takes a *call-by-name* parameter, which is not evaluated until it is used. With this syntax, the difference between "weavable" methods and those which are not is the addition of the `defun` "tag" between the = and the opening curly-brace.

Now we are ready to define aspects in Scala. Below is the Scala implementation of the "CirclePainter" aspect with a pointcut "move" and an advice to insert the call to "repaint" method after each join-points specified in the pointcut. Our syntax for pointcuts and advices resembles that of AspectJ but still has significant differences.

```
class CirclePainter extends Aspect {
  val move = pointcut (classOf[Circle]::('x_=, 'y_=)) -> *

  after(move) { c: Circle =>
    c.repaint();
  }
}
```

Prior to the start of an application, we must instantiate aspect classes and stored them in the singleton object “AOP”.

```
object Main extends Application {
  AOP.addAspect(new CirclePainter())

  val c = new Circle()
  c.x = 10
  c.y = 20
}
```

The “CirclePainter” aspect is able to insert calls to “repaint” method after each call to setter methods. Note that the assignment “c.x = 10” has the effect of the method call “c.x_=(10)”.

3 SYNTAX

Pointcuts The most interesting part of the Scala AOP framework is the syntax used to specify pointcuts. One of the goals of our framework is to create a fully-modular and transparent AOP implementation. With the exception of the `defun` delegate tag, the weaved code should have no knowledge of the aspects. Likewise, the dispatch code (that which makes calls to the weaved classes) should have absolutely no knowledge of the AOP process. The only code to be tightly coupled should be the aspects themselves. This requires a robust syntax for specifying pointcuts, otherwise transparent and specific join points cannot be achieved.

Before looking at the syntax itself, it is worth considering the precise requirements for a “robust” pointcut quantification mechanism. The goal is to specify unambiguously which methods are to be matched. Thus, several points of the method signature must be handled to consider the syntax sufficiently robust: Return type, Containing class, Method name. For a truly unambiguous specification, the syntax would have to consider method parameter types as well.

Taking inspiration from AspectJ, and keeping the restrictions of Scala’s syntax always in mind, we can derive the EBNF grammar for our pointcut specifications shown in Figure 1.

$$
\begin{array}{rcl}
\textit{pc} & ::= & \text{pointcut‘(’ } \textit{type-sig} :: \textit{meth-sig} \text{ ‘)’} -\!> \textit{type-sig} \\
\textit{type-sig} & ::= & \text{‘(’ } \textit{type} \ \{, \textit{type}\} \text{ ‘)’} \\
 & | & \star \\
\textit{type} & ::= & \text{classOf‘[’ } \textit{type-name} \text{‘]’} \\
\textit{meth-sig} & ::= & \text{‘(’ } {}'\textit{meth-name} \ \{, {}'\textit{meth-name}\} \text{ ‘)’} \\
 & | & \star
\end{array}
$$

Figure 1: Syntax of Pointcuts

Note that parenthesis around a type signature or a method signature are omitted if there is only one type name or method name. Also note that in Scala, any term of the form ′mysym is an instance of Symbol class, which is essentially a String but is unique for equal strings so that they can be compared using reference equality.

```
pointcut (classOf[Circle]::'x_=) -> *
pointcut ((classOf[Circle], classOf[Square])::'y_=) -> *
pointcut (classOf[Circle]::*) -> classOf[Int]
pointcut (*::('x, 'y_=)) -> (classOf[String],classOf[Int])
```

Figure 2: Examples of pointcuts

A few pointcuts of this grammar are shown in Figure 2. The meaning of these examples is as follows. The first one defines a pointcut constrained on methods with any return type in class `Circle` and with name "x_=". Likewise, the third declaration returns a pointcut constrained on methods returning type `Int` in class `Circle` with any name. Note that the $\star$ character is used as a wildcard here rather than the Scala conventional underscore (_). This is because the underscore has special meaning in the Scala syntax and is unusable in a DSL like the above.

Advices Pointcuts are not the only construct which require a DSL in our pure-Scala implementation of AOP; advice also has a certain DSL-like syntax. This syntax defines both `before` and `after` advice. Both advice types can optionally require the target instance be passed as a parameter. After-advice also may optionally receive the return value of the method wrapped within an instance of Scala's `Option` monad. All advice forms (including those accepting parameters) are fully type-checked at compile-time. Thus, advice may not expect an instance of `String` as the target instance if the pointcut only allows capturing within instances of `Circle`.

The grammar for our advice syntax is shown in Figure 3. Note that *pc-name* refers to pointcut variables and *var-name* refers to context variables that point to either the target objects or the return values.

advice	::=	before '(' *pc-name* ')' '{'
		[*target* =>] *body*
		'}'
	|	after '(' *pc-name* ')' '{'
		[*after-params* =>] *body*
		'}'
target	::=	*var-name* : *type-name*
ret-val	::=	*var-name* : Option '['*type-name*']'
after-params	::=	*target*
	|	'('*ret-val*, *target*')'

Figure 3: Syntax of Advices

As with the pointcut syntax, the main idea behind the grammar is to provide an intuitive and minimal syntax for defining advice. Another primary concern of this grammar is to provide a mechanism for accessing context data from the join point. For this particular syntax, the two kinds of contextual data which are made available are the target instance and the return value (for `after` advice only). It is important to note that most AOP implementations also allow access to parameter values as part of the context data. Our current syntax does not support this feature due to the way in which method calls are "intercepted". In fact, obtaining the target instance itself must be handled through a special work-around (using a mixin for `defun` rather than a singleton namespace).

The contextual data is passed to the parameters of the functional representing the advice (if parameters are specified). It is an important requirement that all of the context parameters are type-checked against whatever possible types may be represented by the context data itself. To allow these parameters of such specific types and no others, method overloading is used within another DSL builder object, which is an object which exists solely to satisfy the next syntactical step in the DSL. This is an important implementation note as it depends upon a feature, method overloading for first-class functions, which exists solely as a product of Scala's blending of the object-oriented and functional paradigms.

Recall the example of "CircleAspect" in our syntax of pointcuts and advices:

```
class CircleAspect extends Aspect {
  val move = pointcut (classOf[Circle]::('x_=, 'y_=)) -> *

  after(move) { c: Circle =>
      c.repaint()
  }
}
```

where we define a pointcut to capture all calls to the `x_=` and `y_=` methods within the `Circle` class. We then use the pointcut to specify `after` advice to repaint the `Circle` instance.

One sticky point in this syntax is that we would like the parameter in the advice functional to be type-checked. The parameter represents the target instance upon which the call was "intercepted", therefore it should be checked against any containing type the pointcut may match. In our example it is just the single class, `Circle`, but in a more realistic example there may be several possible containing classes. The DSL implementation will have to gather these different types and infer a least common supertype against which the functional parameter may be checked. This all must happen at compile time. Thus, like many other aspects of the syntax, carefully chosen compiler "tricks" must be utilized in order to achieve the desired result.

4 IMPLEMENTATION

The implementation of internal domain-specific languages is a topic which has received some attention in recent years [15, 5]. The rise of frameworks such as Ruby on Rails (and subsequent clones) has led to increased interests in the art of API construction. Many articles [4] have been written describing in great detail the steps one must take to construct an internal DSL in dynamic languages such as Ruby, Groovy, or Python. While little material [5] has been devoted to the construction of DSLs in *static* languages (such as Java and Scala), many of the same techniques apply. The general steps in creating an internal DSL are as follows:

1. Specify a desired syntax for the DSL
2. Adjust the syntax to comply with language syntax constraints
3. Identify the elements of the syntax as recognized by the language parser
4. Create an API which satisfies the desired structure and semantics

Steps one and two have already been completed (see examples above). Step three usually melds into step two, as the process of adjusting the syntax is highly dependent on being able to parse the constructs according to the language specification. Usually, this step involves instantiating the grammar for several different cases and marking down what each element represents within the confines of the parent language.

Method Call Interception in Scala

To satisfy the syntax of method call interception, we declare a `trait AOP` containing a method `defun` which accepts a single no-parameter functional as a parameter. This function should make a call elsewhere in the framework to handle the *before* advice, call to the functional, and then make a third call to the framework to handle *after* advice. The implementation is fairly simple and its simplied form is reproduced below:

```scala
trait AOP {
  def defun[A](fun: =>A): A = {
    handleBefore()

    val back = try {
      fun
    } finally {
      handleAfter()
    }
    back
  }
}
```

The type parameter `A` represents the return type for the specified functional. By making this the return type for `defun`, type inference will ensure that the weaved method will maintain the appropriate signature, thus preserving type checking. It is interesting to note that this implementation of `defun` merges the *after* advice for both the case that an exception was thrown, and the case that the method returned normally. AspectJ allows these as separate join points. While it would certainly be possible to implement this functionality in our framework, it has been omitted for the sake of simplicity.

DSL API

The implementation of our DSL API revolves around a single core class: `Aspect`. This class contains methods like `pointcut`, `before` and `after`, as well as functions as the primary containers for extra syntactic sugar such as the `*` symbol in its various permutations. At a minimum, this class has to contain the following:

```
abstract class Aspect {
  protected val * = AnyClass()

  def pointcut[A](sig: SigClause[A]) = new PCBuilder(sig)

  def before[R, T](pc: Pointcut[R, T]) = { ... }
  def after[R, T](pc: Pointcut[R, T]) = { ... }
}
```

Note that A, R, and T are type parameters. There are a few classes in this snippet which are purely internal to the DSL implementation; for example: `PCBuilder`. This class is necessary because the construction of a pointcut is a multi-step process in our syntax. The class and method signatures are combined into a single value, `SigClause`. This is passed to the `pointcut` method, which returns `PCBuilder`.

The `before` and `after` methods both return an instance of `AdviceApplicator`, which defines an `apply()` method which takes a function value as a parameter. This function value is what actually defines the body of the advice. By wrapping this function within an instance of `AdviceApplicator`, we are able to merge the separate cases: with and without context parameters.

`AdviceApplicator` in turn adds the advice body in question to a `HashMap` within `Aspect` which maintains a list of all advices defined within the current aspect and their corresponding pointcuts. These maps (one for before, one for after) are searched after a `defun` invocation in order to find the advices for a given method. The process of actually matching against a specific method is only run once per method, at which point the result is cached within another pair of `HashMap`(s) for improved performance on subsequent invocations.

Implementation of Pointcuts

Figure 4 illustrates the process of defining DSL for pointcuts. Starting from left to right, the first thing we see is a method called `pointcut`. This method is implemented in the superclass `Aspect`. Reading further we see that it must accept a single parameter and return an object which defines the -> method. This single parameter will be of some intermediate type which contains information about both the containing class(es) and the symbols representing the methods to be matched. This instance will be constructed using the right-associative `::` method called on an instance of type `(Symbol, Symbol)` and taking as a parameter an instance of type `Class`.

This is where we run into a bit of a problem. Unlike Ruby, Scala does not support

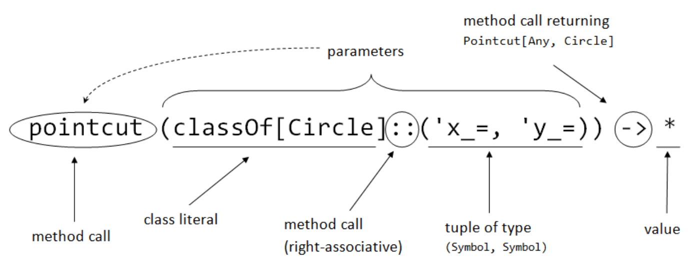

Figure 4: Pointcut as DSL

the notion of "open classes." This means that we cannot simply add a method to the `Tuple2[Symbol, Symbol]` class (the underlying class for the 2-tuple type). To implement this feature, we will have to make use of Scala's implicit type conversions [13]. We define a conversion method in the `Aspect` superclass which takes a `Symbol` 2-tuple as a parameter and returns a new instance of some class which defines the `::` method. In the framework, this class is called `PCSymbol`. If there is a call to `::` on a 2-tuple, Scala compiler translates it to a call to the conversion method to first turn the tuple into a `PCSymbol` object and then call `::` on this object.

Of course we want to generalize this solution as much as possible. Thus, we must define conversions not just for `Symbol` 2-tuples, but *every* tuple plus the individual `Symbol` type. Scala only defines tuples up to 23 values, so this is a much less daunting task than it could be. All of these conversions return an instance of `PCSymbol` as a result, allowing the implementation to be abstract from the specifics of the pointcut syntax.

Also, we must define a set of conversions from n-tuples of `Class` to another abstraction class, `PCClass`. As with symbols, these conversions are necessary to allow both individual class literals as well as n-tuples of literals as both containers and return types. This technique enables implementation of `*` as the wildcard simply represents a special implementation of `PCClass`.

Returning to our syntax analysis, we see that the `pointcut` method does not actually create the pointcut itself, but instead generates an additional intermediary instance which defines the `->` method. This method (like most in Scala) is left-associative and takes an instance of `PCClass` as a parameter. Here again we see the value of converting all class literals and class literal n-tuples to a single abstractive type.

This is a common pattern in DSL implementations: methods which build an instance which contains *some* of the data required to obtain the final result and which also define the methods necessary for the next step in the DSL syntax. Most internal DSLs of moderate complexity are built in this fashion. More complex frameworks such as Ambition [1] and scala-rel [14] must make use of a tree-based approach in which they actually build a parse tree from method and field invocations called when the DSL executes. This technique is obviously far more flexible, but also more complicated than our AOP DSL calls for.

The final product of our pointcut DSL will be the the creation of an instance of type `Pointcut`. The purpose of this class is mainly to hold the `PCClass` and `PCSymbol` data pending use in advice. The class is also polymorphic on the containing and return types defined as part of the pointcut declaration. This will allow advice context to be type-checked based on what types are possible with a given pointcut. In cases where multiple containing classes are specified, a least common superclass is inferred by the compiler and this becomes the type against which `Pointcut` is parameterized. This inference is actually handled as part of the implicit conversion between `Class` n-tuple and `PCClass`.

Here is the pointcut example reproduced with the steps taken by the runtime (in order):

```
val move = pointcut (classOf[Circle]::('x_=, 'y_=)) -> *
```

1. Call to method `symbol2PCSymbol` passing instance of `(Symbol, Symbol)` as a parameter. Method returns instance of `PCSymbol`
2. Call to method `class2PCClass` passing instance of `Class` as a parameter. Method returns instance of `PCClass`
3. Call to method `::` in class `PCSymbol` passing instance of `PCClass` as a parameter. method returns instance of `SigClause`
4. Call to method `pointcut` passing instance of `SigClause` as a parameter. Method returns instance of `PCBuilder`
5. Resolve value `*` to instance of `PCClass`
6. Call to method `->` in class `PCBuilder` passing instance of `PCClass` as a parameter. Method returns an instance of `Pointcut`
7. Assign instance of `Pointcut` to value `move`

Syntax edge cases such as specifying * for the method symbols or the containing type can be handled by adding specific methods and values (similar to how * is handled). With these final pieces, the DSL implementation is complete and capable of generating any pointcut supported by the framework.

Implementation of Advices

Before looking into the precise details of the implementation, perhaps it would be useful to re-examine a concrete example of the advice syntax:

```
class CircleAspect extends Aspect {
  ...
  after(move) { c:Circle =>
    c.repaint()
  }
}
```

Once again, we must break down the syntax so as to identify the critical elements which must be implemented in the API. Moving from left-to-right (following the order of evaluation), the first step is a call to the `after()` method of class `Aspect`. We're passing the `move` pointcut as part of the invocation. Looking ahead somewhat, we see that whatever the return value of the `after` function may be, it must respond to calls to the `apply()` method [3], accepting a function parameter. Here, as in standard method calls, Scala allows the omission of parentheses if the single parameter is delimited by spaces. Thus, the second half of the syntax is invoking the parentheses "operator" on the return value of `after()` by omitting the parentheses.

It is in the implementation of this `apply()` method that method overloading becomes critical. This function must accept function parameters of three varieties: without parameters, with a single parameter of the same type as the pointcut target, and (in the case of `after` advice) a function accepting an `Option` monad of the pointcut return type as well as an instance of the pointcut target type. Thus, assuming that class `Pointcut` is polymorphic on two type values `R` and `T`, we can define the signatures for the `apply()` methods within the context of a containing class `AdviceApplicator` as follows:

```
class AdviceApplicator[R,T](pc:Pointcut[R,T]) {
  def apply(fun:(=>Unit)) = ...
  def apply(fun:(T)=>Unit) = ...
  def apply(fun:(Option[R],T)=>Unit) = ...
}
```

It is interesting to note that the first signature of `apply()` will not accept a functional at all. Rather, the special type `(=>A)` for some type `A` actually represents a call-by-name parameter of type `A`. Thus, when advice is declared with "no parameters", it is actually compiled as an inner scope evaluated by name within the `apply()` method. Under the surface, the Scala compiler treats call-by-name and function parameters in a very similar fashion, but the distinction is still interesting to note.

For the sake of the example, all three `apply()` signatures were shown within the same class. However, this is not the case in the actual framework implementation. The distinction is necessary because `after` advice has an additional parameter (return value) which may be handled. This context parameter is not available to `before` advice and should be type-checked as unspecified for any `before` functional. Within the framework, there are actually *two* `*Applicator` classes, one of which extends the other to provide the `after`-specific overload.

Putting it Together

In summary, the pointcuts are created using a multi-step builder process with multiple classes and symbolic method names. These pointcuts are polymorphic on the containing and return types. A single pointcut instance is passed to either the `before()` or

[3]`apply()` is Scala's equivalent to C++'s `operator()()` function, handling calls to the parentheses "operator"

`after()` methods along with a functional which represents the expressions to execute at a given join point. These functionals can take parameters which allow them to access context data such as target instance. These context parameters are type-checked against the pointcut contravariantly so that the advice may accept the context parameters up-cast to any super-type of the least upper bound common to all context types defined by the pointcut. If multiple advices are found for a given pointcut, the advices will be called in order of declaration.

At the point of method call, execution is immediately transfered to a mixin method, `defun`. This method takes a functional which is to be the body of the actual method. The `defun` method first calls to the singleton object `AOP` passing `this` from the mixin. `AOP` then executes all `before` advice and returns to `defun`. `defun` executes the implementation functional, ignoring any exceptions and saving the return value. Within a `finally` clause, a second call is made to the `AOP` singleton which executes all `after` advice, passing both the target instance and an `Option` monad containing the return value or `None`. Finally, `defun` yields the return value it previously saved and flow is returned to the original method. This immediately returns the value passed back from `defun`.

The entire framework revolves around a main singleton: `AOP`. This `object` contains the list of aspects which in turn contain a map of pointcuts to advice. This top-level controlling class must be a singleton so as to allow join points to have global scope. An interesting addition to the framework could be to implement a form of "aspect scoping," where certain classes or even certain instances can only be affected by a specific set of aspects. For simplicity's sake, this concept was not pursued in the prototype version of the library. Such scoping would also be inconsistent with other AOP implementations such as AspectJ which are limited to the concept of globally applied cross-cutting.

5 DISCUSSION

In examining our AOP framework, several factors emerge which distinguish it from other implementations of the aspect-oriented concept. A few of these differences represent limitations in the implementation, but many of them may be seen as advantages over other implementations such as AspectJ. We examine these differences in some detail, considering both theoretical and practical implications.

More Robust Pointcuts

So far, we have focused on minimizing the impact of our framework on base code by using a simple `defun` call in each method that needs to be advised. While this approach allows us to emulate the way that pointcuts are defined in other AOP implementations, it does not help aspect writer to define more robust pointcuts. Pointcuts defined using method and class names are fragile and may become invalid if these names are modified during refactoring or maintenance.

There are many proposals to make pointcuts more robust. Some of these proposals use naming conventions, annotations, or dedicated interfaces in base code as markers for pointcut definitions [9, 10]. The purpose of this is to map join points to view-based abstractions and then define pointcuts using these views [7, 11]. These approaches make the base code less oblivious to the aspects. In essence, the base code must include some hints – directly or indirectly – for the aspect writers to create pointcuts. The degree of entanglement between the base program and aspects is similar no matter which approach is taken.

In this framework, the base code and the aspects are directly connected through `defun` calls. However, we are not restricted to just one such function. We can define many different `defun` functions to represent various kinds of join points. For example, one can define the following pointcut in AspectJ to capture calls to all mutator methods in the base code.

```
pointcut mutators() call( * set*(..) );
```

In this example, the pointcut assumes that base code will always define mutators using the naming pattern `set*`. This expectation is implicit and any violation can easily go undetected, as the AspectJ compiler will not raise warnings for failing to capture a join point which had been captured in a previous iteration. There are techniques to prevent this [16], but they require additional tool support. A view-based approach [7] solves the problem by creating constraints on the view mappings from base programs and by ensuring that base-code revisions do not violate the constraints.

For example, we can define a function called `mutator` in `AOP`, which is called by each mutator method.

```
class Circle(...) extends AOP {
  ...
  def x_=(x: Int) = mutator { ix = x }
  def y_=(y: Int) = mutator { iy = y }
}
```

The `mutator` function would behave in the same way as `defun` except that it would carry additional information used to indicate that the intercepted methods are mutators.

```
trait AOP {
  def mutator[A](fun: =>A): A = { defun(defun, 'mutator) }

  def defun[A](fun: =>A, pcType: Symbol): A = { ...  }
}
```

Aspects could use the `mutator` function to specify pointcuts so that there is no need to resort to stack inspection to discover the caller's signature:

```
val move = pointcut 'mutator
```

The syntactic overhead of this approach is similar to source code annotations. The differentiating factor is that it allows base code to pass context variables directly to aspects. For example, the code below uses an `mutator` function with a parameter so that aspects which wish to advise these join points may utilize the context variable through the parameter.

Of course, mutator must be changed to accommodate the additional parameter and its implementation must pass the parameter along to the advice. This can be accomplished within the framework by adding an overload to AdviceApplicator which allows a optional added parameters of type Any* (var-args of Any). These additional parameters could handle any context variables passed to the defun implementations. The only problem with this is that all type-checking on those parameters is lost when they are up-cast to Any. This problem could be overcome in the same way as target and return value context parameters are handled, but only in the case of a fully defined pointcut, rather than a defun symbol.

The syntax for passing method parameters to the defun function could be fairly simple:

```
class Circle(...) extends AOP {
  ...
  def x_=(x:Int) = mutator(x) { ix = x }
  def y_=(y:Int) = mutator(y) { iy = y }
}
```

While this approach makes base code less oblivious to aspects, it also makes the aspects much more robust by avoiding the selection of context variables from method parameters through dynamic join points, something which can easily break down if the base code changes method signatures, rearranges parameter lists, or renames field variables. The functions like mutator partially serve the purpose of view abstractions for join points while remaining within the base language, allowing the compiler to ensure correct application.

Static Checking

By creating an implementation of aspect-oriented programming as an internal domain-specific language, we have made it possible for the Scala compiler itself to perform some amount of static checking on aspects. The theoretical capabilities of such static checking are not fully explored by our framework, but there is great potential for future enhancement in this area. At the moment, the primary example of such static checking in the framework is demonstrated in the access of context variables by advice (both before and after). Consider the following example of a simple pointcut which matches any methods in the hypothetical class Person which either return String or StringBuilder:

```
val strings = pointcut (classOf[Person]::*) ->
                 (classOf[String], classOf[StringBuilder])

before(strings) { p: Person =>
  ...
}
after(strings) { (ret: Option[CharSequence], p: Person) =>
  ...
}
```

In both the *before* and *after* advice, we are accepting parameters which represent context variables on the join point. In both cases, these context variables are statically checked, proving correctness at compile-time. Based on the pointcut alone, the framework is able to infer that the most-specific type of the invocation target will be `Person`, while the most-specific return type will be the unification of `String` and `StringBuilder`, which happens to be `CharSequence`. This type unification is computed statically by Scala's compiler, based on the type parameter constraints defined on the implementation class, `PCClass`.

Of course, this sort of checking is only possible in situations where the return type and/or containing class are used as criteria in defining the pointcut. However, this checking does illustrate what is possible, hinting at further potential for such compile-time checking in future implementations. For example, returning to the concept of named interceptors, it is possible to define a named interceptor such that it only functions on a method with a specific return type, or perhaps one which requires certain context parameters to be present. This sort of checking is possible to implement within the confines of Scala's type system, whereas other runtime AOP implementations (such as Spring AOP) must perform all checking dynamically.

Performance Considerations

One of the advantages of an AOP implementation based on artifact instrumentation is that of performance. The framework or runtime may incur some overhead, but the method interception and identification process will not. Unfortunately, this is not the case with a pure-language implementation. Our framework must go through several steps in the process of passing flow control from the base code to the relevant advice. First, the `defun` call is handled. This requires obtaining stack information to determine the base code method signature. This is then used in a linear search through every aspect in the system to locate a match for the given method. Once a match is located, the associated advice is invoked, passing the appropriate parameters.

This process imposes significant overhead on any advised base code. Our tests involved two identical methods, one advised, the other unadvised. These methods were then invoked separately over a large number of iterations. It was discovered that the overhead imposed by a successful match and advising was roughly 38 microseconds, averaged over 1,000,000 iterations. This translates to slightly more than a hundred times slower than Spring's AOP implementation. More than 90% of this overhead is caused by the method used to interrogate the call stack. We performed some experimental benchmarking with a hard-coded method signature, avoiding the need to check call stacks. The overhead in this experiment is 1.58 microseconds for before advice and 1.98 for after advice.

We believe that the performance can be further improved in two key areas. One optimization would be to avoid using call stacks to look up calling method signature. For example, earlier we showed that it is possible to use function such as `mutator` to capture join points, and use its symbolic name to define pointcuts. Additionally, our framework implementation makes heavy use of closures and function parameters. Scala's implementation for these constructs may be inefficient on the JVM due to a lack of native support for first class functions. This may be improved by research currently in progress on projects such as Sun's *Da Vinci Machine* [4].

6 RELATED WORK

In comparison to AOP implementations that use bytecode transformation, our framework has its limitations in capturing join points. For example, our pointcuts are similar to the pointcut designators `execution` and `within` in AspectJ [8]. However, we do not have pointcuts similar to `call` or `cflow`. Aspects in this framework can obtain context variables such as receivers and return values. We can allow aspects to access any context variables, including method parameters, by adding new `defun` functions with parameters.

Many AOP implementations try to minimize the impact of language extensions by using source code annotations or external configuration files. For example, AspectJ allows aspects be written in Java syntax with annotations to designate pointcuts. Spring and JBoss AOP use XML. These changes make aspects look more like regular Java programs so that aspect writers need not learn a new language. Unfortunately, it still requires the same amount of effort to learn the predefined annotations and XML tags. Annotations and XML effectively take the place of a formal language, defining domain-specific semantics and syntax. Often, it is no easier to work with these internal constructs than it is to learn external constructs defined by a separate language. Also, such constructs require an external tool for any sort of static checking, whereas our framework allows compile-time verification of soundness under certain situations with nothing more than the Scala compiler itself.

Proxy-based AOP

Proxy-based AOP implementations such as Spring AOP do not transform bytecode and rely on object proxies instead to apply advice code by intercepting method calls at runtime While the compiled approaches can result in more efficient program, proxy-based AOP implementation requires no special compilation process.

However, AOP frameworks based on object proxies have the drawback that when a method invokes another method on the `this` pointer, advice on the latter method is not executed[5]. The reason is that the object identity referenced by `this` pointer is not the same as the object identity of the proxy object. Therefore, calls through `this` pointer

[4]http://openjdk.java.net/projects/mlvm

[5]http://blog.xebia.com/2006/08/18/the-problem-with-proxy-based-aop-frameworks/

are intercepted by proxies. Though one can fix this problem by obtaining a reference to the proxy context, the resulting program becomes explicitly dependent on the implementation details of the framework. Proxy-based AOP frameworks in Java have this inherent problem. For example, Spring AOP uses JDK interface proxies as well as CGLIB – both of which are proxies for Java objects. Though Java does not provide the ability to add proxies to methods, more advanced languages such as Scala do offer this through higher-order functions. As indicated earlier, we differ from Spring AOP in that our proxy is method-based instead of object-based so that we can intercept method calls invoked on *this* pointer. In addition, we do not rely on external configuration files or Java annotations to denote pointcuts and advice applications. All our AOP constructs are based on Scala.

Another similar approach is Composition Filters [2] where filters are aspects that act as proxies to messages to impose additional functionalities. These functionalities can be activated based on conditions specified in the filters. The filters have distinct syntax and complex semantics, which may be difficult to be implemented as an internal DSL. There is also a radical proposal of machine model for AOP [6] where aspects are inserted at runtime as proxies to actual objects. Method calls are intercepted by the proxies to decide the actual code to run in a way similar to virtual method dispatch. Though this approach can be quite flexible and more efficient than other proxy-based AOP frameworks, it is not applicable to applications run on Java virtual machines.

Other Scala Implementations

A few attempts have been made to implement AOP in pure Scala. It's interesting that this language more than others seems to see such attention. These implementations have varied properties and are often useful as design patterns for solving many common scenarios. Foremost among these implementations are mixins [3, 13].

Scala mixins Scala mixins [13] are similar to those available in other languages such as Ruby in that they are almost literally inserting code into a particular class. It is not inheritance so much as method rewriting. This property can be exploited to implement a form of base-code oblivious method interception (unlike our `defun` implementation which is base-code aware). To the best of our knowledge, the technique was first demonstrated in Scala by Martin Odersky in his discussion of the Scala compiler architecture [12]. This idea was further expanded upon by Jonas Bonér[6] to provide fully composable interception.

Figure 5 is a simplified implementation of AOP using this technique, which shows the `Circle` class completely oblivious to any advising. The AOP itself is driven entirely by the driver code during instantiation. Creating a new `Circle` instance using `with CirclePainter` actually creates a new subclass of `Circle` which uses the `CirclePainter` mixin. `CirclePainter` brings into the class the overridden `x_=(Int)` and `y_=(Int)` methods which wrap added functionality around the super-

[6]http://jonasboner.com/2008/02/06/aop-style-mixin-composition-stacks-in-scala

```
class Circle {
  private var ix, iy;

  def x_=(x:Int) = { ix = x }
  ...
}
trait CirclePainter {
  override def x_=(x:Int) = {
    val back = super.x_=(x)

    this.asInstanceOf[Circle].repaint()

    back
  }
  ...
}
val c = new Circle with CirclePainter
c.x = 12
```

Figure 5: Base and driver code

class (`Circle`) implementation. This can effectively be thought of as the advice. When the `x_=(Int)` method is invoked on this special instance, it is calling the override within `CirclePainter`.

This implementation has the very interesting property of being totally transparent to the base-code. Advising is effectively enabled or disabled at instantiation time by the driver-code. This pattern can be useful in solving problems traditionally approached using AOP or dependency injection. This pattern also has the advantage of having full access to the parameters of the method being advised. This is something our implementation *could* support, but only by modifying or overloading `defun`.

This approach has two primary shortcomings. It relies on the driver to setup the mixin in the instantiation syntax. This breaks the AOP principle of driver-code obliviousness. It is possible for code to use the `Circle` class without properly repainting it on all mutator calls merely by neglecting to include the mixin syntax. The driver code is tightly coupled to the design of `CirclePainter` as a mixin. But more importantly, there is no way to specify gradient pointcut quantifiers using a simple override.

With most AOP implementations, it is possible to define a pointcut which will match any method with a given return type, or perhaps any method with a given return type *and* a single parameter of another given type. This control over the pointcut match quantifiers can be extremely useful when examining more general use-cases than a simple repaint. Likewise, it allows the implementation of the circle repaint advise in a single point, rather than in every override. While technically the mixin allows the logic to be factored out into a delegate method using standard procedural techniques, it still requires a certain amount of redundancy. More than that, any time a new mutator is added to the `Circle` class, a new override must be added to the `CirclePainter` trait. Thus, the advice is tightly coupled to the structure of the base code.

Our AOP implementation allows powerful pointcut quantification. For example, we could define the `move` pointcut from previous examples to match any method which takes a single `Int` parameter and returns `Unit`. The pointcut would then match both `x_=(Int)` and `y_=(Int)` in a single quantifier. Further, if the base-code designer chose to add a mutable property `radius`, it too would be automatically matched by the pointcut without any changes to the aspect. Thus, while the advice is coupled to the base-code structure, the coupling is far looser than in the case of mixins. This loose coupling increases flexibility and eases concurrent development of the separate layers.

Scala views It has been shown that Scala's implicit type conversions (also called views) can be used to implement AOP-style security checks[7]. The idea is to perform additional operations (such as security checks) when an object is implicitly converted to another type. The views from one class to another can be introduced by normal methods with the `implicit` modifier. This approach is limited because views alone cannot insert `after` advice. Also, it does not support quantification over join points since each view can only insert advice into the methods defined in the view's return type.

7 CONCLUSION

Scala's powerful constructs and flexible syntax allow for the implementation of varied techniques within the confines of the language itself. By making full use of the available techniques, implementing AOP does not strictly require the modification of language semantics. Our framework fully implements the core concepts of AOP and represents the potential for new innovation in the field without the unnecessary hardship of modifying a compiler. The framework can easily be extended to encompass other concepts in AOP, as well as opening the door to new design patterns within other hybrid functional / object-oriented languages. As future work, we would like to improve the performance of the implementation in the areas mentioned earlier. Also, it would be interesting to test the effectiveness of the framework with some larger applications.

REFERENCES

[1] Ambition. http://ambition.rubyforge.org.

[2] L. Bergmans and M. Aksit. Principles and design rationale of composition filters. *Aspect-Oriented Software Development. Addison-Wesley*, pages 0–32, 2004.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, 1990.

[7]http://scala.sygneca.com/code/aop-style-security-check

[4] Martin Fowler. Implementing an internal DSL, September 2007. http://martinfowler.com/dslwip/InternalOverview.html.

[5] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in Java. In *Companion to the 21th Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 855–865, 2006.

[6] M. Haupt and H. Schippers. A machine model for Aspect-Oriented Programming. *Lecture Notes in Computer Science*, 4609:501, 2007.

[7] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. A Model-driven Pointcut Language for More Robust Pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 501–525, 2006.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference onbergmans2004pad Object-Oriented Programming (ECOOP)*. Springer-Verlag, June 2001.

[9] Gregor Kiczales and Mira Mezini. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of International Conference on Software Engineering*, 2005.

[10] Gregor Kiczales and Mira Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming*, 2005.

[11] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views - a case study. *Computer Languages, Systems and Structures*, 32(2-3):140–156, 2006.

[12] Martin Odersky. The scala experiment: can we provide better language support for component systems? In *Proceedings of the 33rd ACM Symposium on Principles of programming languages*, pages 166–167, 2006.

[13] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language (2nd edition). Technical report, EPFL, 2006.

[14] Scala-Rel. http://code.google.com/p/scala-rel.

[15] Tim Sheard, Zine el-abidine Benaissa, and Emir Pasalic. DSL implementation using staging and monads. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 81–94, 1999.

[16] Maximilian Stoerzer and Juergen Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, 2005.