SCoPE: an AspectJ Compiler for Supporting User-Defined Analysis-Based Pointcuts

Tomoyuki Aotani
Graduate School of Arts and Sciences,
University of Tokyo
aotani@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences,
University of Tokyo
masuhara@acm.org

Abstract

This paper proposes an approach called SCoPE, which supports user-defined analysis-based pointcuts in aspect-oriented programming (AOP) languages. The advantage of our approach is better integration with existing AOP languages than previous approaches. Instead of extending the language, SCoPE allows the programmer to write a pointcut that analyzes a program by using a conditional (`if`) pointcut with introspective reflection libraries. A compilation scheme automatically eliminates runtime tests for such a pointcut. The approach also makes effects of aspects visible to the analysis, which is essential for determining proper aspect interactions. We implemented a SCoPE compiler for the AspectJ language on top of the AspectBench compiler using a backpatching technique. The implementation efficiently finds analysis-based pointcuts, and generates woven code without runtime tests for those pointcuts. Our benchmark tests with JHotDraw and other programs showed that SCoPE compiles programs with less than 1% compile-time overhead, and generates a program that is as efficient as an equivalent program that uses merely static pointcuts.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Aspect-Oriented Programming Languages, Pointcuts, Compiler Design

Keywords AOP, Analysis-Based Pointcuts, Compiler Design

1. Introduction

Aspect-oriented programming (AOP) helps modularize crosscutting concerns [12, 18, 19], which have scattered and tangled implementations when traditional modularization mechanisms such as procedures and classes are used. Typical crosscutting concerns include logging, synchronization, persistence, performance optimization, parallelization and profiling.

One of the important mechanisms of AOP languages is the *pointcut and advice* mechanism, which is supported by most of current mainstream AOP languages including AspectJ [18]. It can be explained in terms of the three elements: *join points, pointcuts* and *advice*. A join point is a point in the execution of a program whose behavior can be affected by advice. A pointcut is a description that matches join points. An advice declaration, which consists of a pointcut and body statements, specifies to run its body statements in addition to or in place of the join points matching the pointcut.

Pointcuts are the key element to make aspects more declarative—more robust against software evolution, more reusable and easier to understand [17]—as they are the means of specifying how the effects of advice should crosscut the program. For the pointcuts that specify join points by using immediate properties of a program (e.g., the method signatures or the declaring types), advice declarations describing a complicated crosscutting behavior often have to enumerate signatures of methods to be advised. Such an aspect tends to be *fragile*, as even a small modification to the program (e.g., changing a method name or adding a new field to a class) could require changes in pointcuts, which prone to be omitted [14, 16].

This paper focuses on *analysis-based pointcuts* as a means of improving aspect robustness. Analysis-based pointcuts match join points based on properties obtained by static program analyses, such as control-flow reachability and side-effect freedom. Since such properties are more stable than the immediate properties, they contribute to the robustness of aspects against program evolution. Note that we focus on *static* program properties; pointcuts for examining dynamic properties, such as dynamic calling contexts (e.g., `cflow`), execution histories [1, 10, 33] and dynamic dataflows [24], are out of the scope of this paper.

This paper proposes a novel approach to supporting user-defined analysis-based pointcuts. While there have been several attempts [8, 11, 13, 25, 26] with similar goals, our approach has several advantages:

- Better integration with existing AOP languages: Our approach merely relies on existing AOP constructs (i.e., conditional pointcuts) and introspective reflection libraries. In addition, compilers can be implemented by slightly modifying an existing compiler thanks to our backpatching-based technique.
- Efficient compilation: Our implementation has a sufficiently small overhead (excluding the elapsed time for user-defined analysis) due to our backpatching technique.
- Clear semantics: Our approach does not change the semantics of existing AOP languages, but merely eliminates runtime tests for analysis-based pointcuts while preserving the behavior. This helps the programmer understand program behavior when aspect interacts with each other [23].

The rest of the paper is organized as follows. Section 2 shows examples of analysis-based pointcuts. Section 3 discusses challenges for supporting analysis-based pointcuts. Section 4 presents our SCoPE approach and illustrates how the pointcuts shown in Section 2 can be defined with SCoPE. Section 5 describes the implementation of our SCoPE compiler and its library support. Section 6 discusses advanced features. Section 7 measures compilation overheads and runtime performance. Section 8 discusses related work. Section 9 concludes the paper.

2. Analysis-based Pointcuts

In order to clarify our goal, we first present examples of analysis-based pointcuts, and how they contribute to more robust and expressive aspect definitions when compared to pointcuts based on method and field signatures. This section describes how analysis-based pointcuts identify join points by using hypothetical pointcut primitives. Our approach to defining those pointcuts will be explained in later sections.

2.1 Regular Expression Matching

Pointcuts with regular expression matching identify join points using regular expressions when matching class/method/field names in the join points[1]. This can be considered as an extension to wildcard-based type patterns in AspectJ, and could be useful for specifying names in more rigorous ways. For example, if AspectJ's type patterns and signature patterns were extended with regular expressions, the following pointcut would match any method call whose name consists of only lowercase characters.

```
pointcut executeLowercaseMethod():
  execution(.* .*\.[a-z]+(.*));
```

Even though regular expression matching might be too trivial to be called analysis-based pointcuts, we present this as the simplest example to explain our approach. Several AOP languages and frameworks, including PROSE [28], JAC [27] and Spring AOP[2], support regular expression matching.

2.2 Class Structure Analysis

Pointcuts that analyzes class structures match join points based on structural properties of the types. For example, `hasmethod` and `hasfield`, which are available in JBoss AOP and Aspectwerkz[3], analyze whether the class has a specified method or field. Those pointcuts are useful, for example, to implement the Inversion of Control principle in a transparent manner[4]. For example, the following aspect transparently installs a transaction monitor at object initialization. The pointcut `hasfield(TM tm)` matches classes that have a field `tm` of type `TM`.

```
aspect SetTransactionMonitor {
  pointcut initObjectWithTmField():
    execution(*.new(..)) && hasfield(TM tm)
  after() returning(): initObjectWithTmField() {
    // install a transaction monitor
  }
}
```

The use of `hasfield` pointcut in the above example increases the declarativeness of the aspect compared to the one defined using a list of individual class and package names.

[1] Such extensions sometimes have been requested in the AspectJ's mailing list: `http://dev.eclipse.org/mhonarc/lists/aspectj-dev/msg01653.html`

[2] `http://www.springframework.org/`

[3] Also available in a recent version of the ajc AspectJ compiler as an undocumented feature.

[4] `http://docs.jboss.org/aop/1.3/aspect-framework/examples/ioc_with_has/has.html`

2.3 Predicted Control Flow

Pointcuts that predict control flow identify join points based on the potential (or predicted) behavior at the current join point. We here show a slightly modified version of the `pcflow` pointcut, which is originally proposed by Kiczales [17].

The following aspect, when used with a figure editor application [18], redraws the screen whenever a figure object changes its visual properties. The `set` pointcut identifies any assignments to any fields in subclasses of class `Fig`. The `pcflowGet` pointcut, which is a hypothetical pointcut, limits the assignments to those whose target fields are referenced during the execution of `draw` method. As a result, the advice runs only at the assignments to the fields that represent visual properties such as positions and colors of figure elements.

```
aspect DisplayUpdating {
  pointcut figureMove(): set(* Fig+.*) &&
    pcflowGet(Fig+.draw());
  after() returning(): figureMove() {
    Canvas.update();
  }
}
```

Compared to an aspect that enumerates all relevant field names, the use of `pcflowGet` clarifies the intention of the programmer. In addition, the aspect is robust against additions of new figure element classes and additions of new fields to existing figure element classes.

2.4 Side-Effect Analysis

A pointcut that analyzes side-effects identifies join points whose potential execution performs only operations without side-effects, such as I/O operations and accesses to global variables. Such a pointcut would be useful for optimizing aspects since there are many optimization opportunities for side-effect free methods.

The following aspect, for example, adds memoization (caching) to methods by recording parameters and return values for each method, and by reusing the recorded return value when the method is called again with the same parameter[5].

```
aspect Memoization{
  Map<Integer,Integer> m;
  pointcut pureIntegerFunction(int key):
    execution(int *(int)) && args(key)
    && isSideEffectFree();

  int around(int key): pureIntegerFunction(key) {
    if (!m.containsKey(key))
      m.put(key,proceed(key));
    return m.get(key);
  }
}
```

In the example, the hypothetical pointcut `isSideEffectFree` matches join points whose potential execution performs no side-effecting operations. In combination with the `execution` pointcut that matches any integer method executions, the aspect properly adds memoization to purely functional methods.

2.5 Coding Style Checking

Pointcuts that check statement-level coding styles enhance policy enforcement mechanisms implemented using AOP languages.

[5] This is largely simplified from practical ones, which would usually limit itself to methods in a certain class, and would be generalized to support types other than integer.

The following aspect alerts programmers when a method in `MyClass` calls a method whose receiver type does not appear in any field of `MyClass`. The rule is derived from the style rules of the law of Demeter [22], but largely simplified[6]. The hypothetical pointcut `callToFieldType` matches a join point when the caller type has a field that has the same type as the static type of the receiver. By using the `declare warning` mechanism of AspectJ, the compiler reports method calls violating the rule.

```
aspect StyleChecking {
  pointcut violation(): call(* *(..)) &&
    !callToFieldType() && within(MyClass);
  declare warning(): violation()
    "receiver should be stored in a field."
}
```

Policy enforcement is a well-known applications of AOP languages [21]. However, pointcuts in current AOP languages are limited to express rules beyond static scoping and naming conventions. Analysis-based pointcuts would be a good approach to describe more complicated rules.

3. Challenges for Supporting User-Defined Analysis-Based Pointcuts

Even though analysis-based pointcuts are useful, they are not always available in practical AOP languages. Moreover, there is a wide-range of analysis-based pointcuts, and many of them are specific to particular domains or applications. It is therefore difficult to extend an AOP language to support all of them, or to provide generic pointcut primitives for constructing complicated analysis-based pointcuts.

Our approach is to support *user-defined* analysis-based pointcuts. Of course, this does not necessarily mean that every programmer has to write program analysis code since our approach allows to provide such pointcuts as a library.

Before presenting our approach, we discuss the challenges that should be tackled when proving user-defined analysis-based pointcuts. These challenges are common to our approach as well as several existing AOP languages and frameworks [8, 11, 13, 26, 35] that allow the programmer to define analysis-based pointcuts.

Runtime efficiency. Analysis-based pointcuts should have no runtime overheads. In other words, they should be compiled as if they are static pointcuts.

In AOP languages, a compiler generates executable code by inserting, for each advice declaration, instructions that execute an advice body into locations matching the pointcut of the advice. This particular process is called *weaving*. A location in which advice-executing instructions can be inserted is called a *join point shadow*. If compiler finds that a pointcut of an advice declaration matches all join points created at a shadow, it simply inserts the appropriate advice instructions at the shadow. We say such a pointcut is *static*. On the other hand, if compiler finds that a pointcut matches join points whose shadows are the same only when some runtime condition holds, it inserts advice executing instructions with a conditional branch to test the condition. Such a pointcut is *dynamic*. For example, `call` and `execution` are static, and `args` and `cflow` are dynamic in AspectJ.

Compilation speed. From a practical viewpoint, compilation speed is an important factor since industrial software development often requires compiling very large programs.

[6] The law of Demeter has more rules in addition to the one presented here, and requires a method call to satisfy any of them.

Clear semantics. Since analysis-based pointcuts in a program observe the behavior of the program itself, their semantics should be carefully designed so as not to introduce contradicting behavior. We will discuss this point in Section 4.2.

Access to program information. In order to define complicated program analyses, it should be possible to access the required program information. In particular, some program analyses such as predicted control flow analysis and side-effect analysis are inter-procedural. Therefore, full access to entire class hierarchy and bytecode (or expression) level information should be provided.

Tool support. Programs that uses analysis-based pointcuts should be well-supported by programming tools, such as integrated development environments, refactoring tools and visualizers. We will see that our approach is compatible with tools for an existing AOP language as it does not extend syntax and semantics of the language.

4. SCoPE Approach to Supporting Analysis-Based Pointcuts

We propose an approach called SCoPE (Static Conditional Pointcut Evaluation) for enabling user-defined analysis-based pointcuts. The approach can be summarized as follows:

- It lets programmers describe their own analyses in a conditional (`if`) pointcuts.
- It provides introspective reflection libraries so that the analysis can access to program information.
- It offers a compiler that eliminates runtime tests for the conditional pointcuts that implement analysis-based pointcuts.

As an example, consider the following pointcut which matches any method call whose name consists of only lowercase characters, as introduced in Section 2.1:

```
pointcut executeLowercaseMethod(): execution(* *(..))
  && if(thisJoinPoint.getSignature()
        .getName().matches("^[a-z]+$"));
```

The first part of the pointcut matches any method execution. The second part uses a conditional (`if`) pointcut to implement the analysis. In AspectJ, the programmer can write any boolean expression in an `if` pointcut in order to add arbitrary conditions to a pointcut.

The example uses Java and AspectJ reflection APIs in order to analyze the program. The special variable `thisJoinPoint` is an object that contains information about the join point, and the methods `getSignature()` and `getName()` retrieve signature and name of the method being executed.

Currently all AspectJ implementations compiles a conditional pointcut into runtime tests, which usually results in overheads. Our compiler eliminates those runtime tests so that the pointcut works as if it was static.

Note that the above pointcut definition uses only features already existing in AspectJ and similar AOP languages. In other words, our approach does not require any syntactic extensions to existing AOP languages. However, the above pointcut definition is not practical with current AspectJ compiler implementations as they evaluate the condition at runtime[7].

[7] When an expression in the conditional pointcut is very simple, an existing compiler might evaluate it at compile-time. However, as far as we confirmed, all existing implementations evaluate conditional expressions at runtime when they include `thisJoinPoint`, which is essential for analysis-based pointcuts.

```
1  static boolean hasfield(JoinPoint tjp, String fname){
2    try {
3      tjp.getSignature().getDeclaringType()
4                        .getField(fname);
5      return true; // when exists
6    } catch (Exception e) { return false; }
7  }
8
9  pointcut initObjectWithTmField():
10   execution(*.new(..))
11   && if(hasfield(thisJoinPoint,"tm"));
```

Figure 1. `hasfield` pointcut with SCoPE.

```
1  static boolean pcflowGet(JoinPoint jp, String entry){
2    for(Method m: implsOf(entry))
3      for(Instruction i: m.instructions)
4        if (i is getfield &&
5            i.hasSameSignature(jp))
6          return true;
7        else if (i invokes method m1 &&
8                 pcflowGet(jp, m1))
9          return true;
10   return false;
11 }
12
13 pointcut figureMove(): set(* Fig+.*) &&
14     if(pcflowGet(thisJoinPoint,"Fig.draw()"));
```

Figure 2. Predicted control flow pointcut with SCoPE.

```
1  static boolean isSideEffectFree(JoinPoint jp) {
2    for (Instruction i: getProceedBody(jp))
3      if (i is {get,set}{field,static})
4        return false;
5    return true;
6  }
7
8  pointcut pureIntegerFunction(int key):
9    execution(int *(int)) && args(key) &&
10   if(isSideEffectFree(thisJoinPoint));
```

Figure 3. Side-effect analyzing pointcut with SCoPE.

Our approach therefore includes a novel compiler, which recognize pointcuts that perform static analyses as static ones so that they introduce no runtime tests.

4.1 Definitions of Analysis-Based Pointcuts

Below are definitions of the analysis-based pointcuts presented in Section 2 rewritten using the SCoPE approach. Note that those definitions may not be ideal in terms of analysis precision and efficiency as we simplified them for explanation purposes.

4.1.1 Class Structure Analysis

Figure 1 shows a definition of `hasfield` pointcut. The `hasfield` is a method that returns `true` when there is a field with given name in the declaring type[8] of the join point. The conditional pointcut merely calls the method in l.11. Note that the definition does not check the type of the field for simplicity; it can be easily extended to do so.

4.1.2 Predicted Control Flow

Figure 2 is a sketch of the predicted control flow pointcut presented in Section 2.3. The method `pcflowGet(jp,entry)` recursively visits the instructions reachable from `entry`, and returns true when it finds an instruction that references a field that has the same signature as the one in `jp`. The method call `implsOf` in l.2 is provided in our library (discussed later), and returns a set of methods with the given signature in `entry`. The `Method` and `Instruction` objects are the metaobjects defined in a reflection library, and provide instruction-level information such as the body instructions in a method, kind of an instruction (e.g., call and execution), and field and method names accessed by the instruction.

Note that the definition is simplified from the actual one[9], which uses a loop rather than recursive calls to avoid stack overflow, and keeps a set of visited methods to avoid looping infinitely over recursive methods.

4.1.3 Side-Effect Analysis

Figure 3 is a sketch of side-effect analysis pointcut shown in Section 2.4. The pointcut uses `isSideEffectFree(jp)`, which returns `false` when there is an instruction that accesses an instance or class variable among the instructions reachable from the current join point.

The method `getProceedBody` in l.2 is provided by our library, and returns a set of instructions reachable from the `proceed` form in the advice that uses the pointcut. Since it collects instructions over method calls, the body of the method is a simple loop over those instructions. This simple analysis considers that any field access instructions are side-effecting as shown in l.3.

Assume we apply the memoization aspect in Section 2.4 to a method `int f(int)`. When the argument to `f` is not found in the table, the aspect evaluates a `proceed` form in order to run the body of `f`. The above library method `getProceedBody` therefore returns the instructions in `f` (and the instructions in the other methods called from `f`). Therefore, even though the memoization aspects adds side-effecting operations to `f`, the analysis properly excludes those operations by using `getProceedBody`. It might look complex, but is crucial when there are interactions between aspects as we will discuss in Section 4.2.

4.1.4 Coding Style Checking

Figure 4 is a definition of the coding style checking pointcut discussed in Section 2.5[10]. The `fieldTypes` method gathers a list of types of fields in a caller class. The `callToFieldType` method checks whether the callee type appears in the list. Similar to the class structure analysis, the analysis uses the Java reflection API to obtain field types.

4.2 Semantics of Analysis-Based Pointcuts

When there is more than one aspect, semantics of analysis-based pointcuts must be carefully designed so that the aspects properly interact with each other. This issue is discussed in detail in our previous work in detail [23]. We summarize here the required

[8] A declaring type of a join point refers the target type of the join point. For example, the declaring type of an execution join point is the class that defines the executing method. The declaring type of a call join point is the static type of the receiver object.

[9] It can be found at our project page, `http://www.graco.c.u-tokyo.ac.jp/ppp/projects/scope/`

[10] Because `if` pointcuts cannot be used within `declare` statements in AspectJ, the `declare` statement in Section 2.5 must be replaced with an advice that throws warning messages at runtime.

```
// returns a collection of field types declared in c and its ancestors
static Collection fieldTypes(Class c) {
  ArrayList fieldTypes = new ArrayList();
  for(;c != null; c = c.getSuperclass())
    for(Field f: c.getDeclaredFields())
      fieldTypes.add(f.getType());
  return fieldTypes;
}
// returns true if the callee type appears in any field declaration of
// the caller class
static boolean callToFieldType
    (JoinPoint.StaticPart ejp,
     JoinPoint.StaticPart sjp){
  Class caller=ejp.getSignature().getDeclaringType();
  Class callee=sjp.getSignature().getDeclaringType();
  // true if callee type exists
  return fieldTypes(caller).contains(callee);
}

pointcut violation(): call(* *(..)) &&
  within(MyClass) &&
  if(!callToFieldType(enclosingJoinPoint,
                      thisJoinPointStaticPart));
```

Figure 4. Coding style checking pointcut with SCoPE.

properties of analysis-based pointcuts to guarantee proper aspect interactions.

Required Property 1 (Aspect Visibility): *Effects of aspects should be visible from the analysis-based pointcuts.*

This property can be explained by using the `isSideEffectFree` pointcut in Figure 3. The pointcut should not match a purely functional method when another aspect (say tracing) adds side-effecting behavior to the method. This becomes possible if effects of the tracing aspect is visible from the `isSideEffectFree` pointcut.

SCoPE achieves the property by performing the analyses on the executable code in which aspects are woven. Section 5.3 explains how SCoPE realizes the property.

Note that our approach requires the analyses to be conservative when an analysis-based pointcut (say `if(`p_1`)`) in an advice declaration (say a_1) analyzes a code fragment that is advised by an advice declaration (say a_2) with another analysis-based pointcut (say `if(`p_2`)`). Since the code analyzed by p_1 is generated by an existing compiler, it contains a conditional branch that evaluates like `if(`p_2`){` the body of a_2 `}`.

We believe that our approach is a reasonable compromise to avoid complicating language semantics and compiler implementations. Another approach is to evaluate an analysis-based pointcut and weave the advice body one by one [13]. For example, the compiler first evaluates p_2 with respect to the unwoven code and weaves or does not weave the body of a_2. It then evaluates p_1 with respect to the code generated in the previous step, and weaves or does not weave the body of a_1. This makes analyses more precise because the code analyzed by p_1 does not contain the conditional branch `if(`p_2`)`. However, it does not work well when the two advice declarations are mutually affecting [20]; e.g., p_1 matches if and only if a_2 is *not* woven into the code and p_2 matches if and only if a_1 is woven into the code. It also complicates compiler implementations because they have to produce the woven code for each time an advice declaration is processed.

Required Property 2 (Awareness of Aspect Precedence): *The analysis-based pointcuts should be able to take aspect precedence into account when analyzing a program.*

This property can also be explained using `isSideEffectFree`. When a tracing aspect is applied to a purely functional method, the precedence between the tracing aspect and the memoization aspect should change the result of the `isSideEffectFree` pointcut. When the memoization aspect precedes the tracing aspect, the `isSideEffectFree` pointcut should not match the method because the operations to be memoized include the side-effecting operations of the tracing aspect. On the other hand, when the tracing precedes the memoization, the pointcut should match because the operations to be memoized still have no side-effects.

SCoPE realizes the property by offering library methods to access instructions only reachable from `proceed` forms in the advice, namely `getProceedBody` in Figure 5. Since `proceed` executes the advice bodies with lower precedence, the results from those library methods properly reflect aspect precedence.

5. Implementation Issues

We realized the SCoPE approach by resolving the following implementation level issues, which are discussed in subsequent subsections:

- In order to provide full access to program information, we provide flexible library support for describing a wide-range of user-defined analyses.
- In order to find analysis-based pointcuts described by using conditional pointcuts, we developed an efficient binding-time checking mechanism.
- We proposed a novel compilation scheme based on a back-patching technique. The scheme makes effects of aspects visible to analysis-based pointcuts and completely eliminates runtime overheads.

We implemented a SCoPE compiler for AspectJ on top of the AspectBench compiler [3]. The implementation is approximately 1900 lines of Java and Scala code including library adapters, and is publicly available[11].

5.1 Flexible library support

In order to define instruction-level program analyses, SCoPE uses bytecode translation libraries as an extension to the introspective reflection APIs in Java and AspectJ. Since those reflection APIs do not provide information beyond class structures, bytecode translation libraries are crucial in defining analyses that require bytecode-level information.

SCoPE does not rely on a particular library, but allows the programmer to choose a suitable one for their purposes. Current implementation supports ASM[12] and Soot [32][13], but supporting similar ones such as BCEL[14] and Javassist [7][15] is not difficult.

We provide an adapter for each bytecode library so that they can be used as if they are extensions to the Java and AspectJ reflection API. Since those bytecode libraries are primarily designed for program transformation, they require to manually load classes. Our adapter (1) provides methods that bridges between Java/AspectJ metaobjects (e.g., the ones of type `java.lang.Class` and `org.aspectj.lang.reflect.MethodSignature`) and metaobjects defined in the library, and (2) initializes the library by automatically loading classes.

[11] `http://www.graco.c.u-tokyo.ac.jp/ppp/projects/scope/`

[12] `http://asm.objectweb.org/`

[13] `http://www.sable.mcgill.ca/soot/`

[14] `http://jakarta.apache.org/bcel/`

[15] `http://www.csg.is.titech.ac.jp/~chiba/javassist/`

```
interface XAdapter {
  // finds a class object by name.
  C classNodeOf(String cname);

  // finds a field object by signature.
  F fieldNodeOf(FieldSignature fsig);

  // finds a method object by signature.
  M methodNodeOf(CodeSignature msig);

  // returns a set of methods (including overriding
  // ones) by signature.
  Collection<M> implesOf(CodeSignature msig);

  // returns a set of subclasses by class object.
  Collection<C> subclassesOf(C cnode);

  // returns a set of instructions potentially
  // reachable from the proceed form in advice
  List<I> getProceedBody(JoinPoint jp);
  List<I> getProceedBody(JoinPoint.StaticPart jp);
}
```

Figure 5. Adapter methods for a bytecode library X, where C, M, F and I stand for types of class, method, field and instruction.

X	ASM	Soot(Jimple)
C	`ClassNode`	`SootClass`
M	`MethodNode`	`SootMethod`
F	`FieldNode`	`SootField`
I	`AbstractInsnNode`	`Stmt`

Table 1. Meta-classes in bytecode libraries.

Figure 5 shows the interface of an adapter defined for each library. Since each library provides different meta-classes for representing classes, methods, fields and bytecode instructions, they are described as C, M, F and I, respectively. They correspond to classes as shown in Table 1.

The methods shown in Figure 5 can be classified into the next three groups:

- Methods that convert from a string or Java/AspectJ metaobject to a bytecode metaobject (i.e., `classNodeOf`, `fieldNodeOf`, and `methodNodeOf`).
- Methods that provide frequently used functions common to Java bytecode-level analysis such as collecting all subclasses of a class and all method implementations of a signature (i.e., `implesOf` and `subclassesOf`).
- The methods that provide functions specific to analysis-based pointcuts, such as collecting reachable instructions from `proceed` (i.e., `getProceedBody`).

Another role of the adapters is to initialize the bytecode libraries so that all classes in the program are loaded when SCoPE evaluates analysis-based pointcuts. Therefore, the analysis code can easily use methods in those libraries as if they were extensions to Java's and AspectJ's reflection APIs. Since those libraries usually require the programmer to manually load class files, the analysis code would otherwise become very awkward.

5.2 Finding Static Conditional Pointcuts

Our SCoPE implementation automatically detects analysis-based pointcuts (what we call the *binding-time checking*) and evaluates them (what we call the *pointcut evaluation*) at compile-time. This approach provides us with a better integration with existing development tools as it does not extend the language's syntax. We believe this property is important from a practical viewpoint, as modern software development involves a number of tools, such as integrated development environments, debuggers, and test generators.

type	method
org.aspectj.lang.JoinPoint	getArgs(), getTarget(), getThis()
java.lang.Class	newInstance()
java.lang.reflect.Field	get(Object), getBoolean(Object), ..., set(Object,Object), setBoolean(Object,boolean),...
java.lang.reflect.Method	invoke(Object,Object[ ])

Table 2. Predetermined dynamic methods in Java and AspectJ APIs.

The above approach, however, needs a mechanism to automatically find conditional pointcuts that can be evaluated at compile-time. Since there is no distinction between analysis-based pointcuts and other conditional pointcuts with runtime conditions, we devised a technique akin to the binding-time analysis in partial evaluation [15].

Below, we explain how SCoPE efficiently finds conditional pointcuts that can be evaluated at compile-time. We first present the rules whether a conditional pointcut can be evaluated at compile-time, and then explain an efficient yet effective algorithm to find such pointcuts.

5.2.1 Definition of Static Conditional Pointcuts

Intuitively, a conditional pointcut is static if its expression always returns the same value with respect to the same join point shadow.

We defined rules to determine whether a conditional pointcut is static, which are simple enough to be efficiently checked at compile-time, yet general enough to describe many useful program analyses. The rules are defined with several auxiliary definitions.

First we define *immutable class variables*. A class variable is immutable if it is final and of primitive type, or final and of immutable class such as `String` and `JoinPoint.StaticPart`. Correspondingly, a mutable class variable is a class variable that is not immutable.

Next we define a set of *dynamic methods*. A set of dynamic methods is a minimal set of the methods whose member is not predetermined as static, and either (1) is a native method, (2) is predetermined as dynamic (cf. Table 2), or (3) has any of the following subexpressions in its body:

- an assignment or reference to a mutable class variable, or
- a dynamic method invocation.

Several library methods are predetermined as static. They include native methods that are known to have no side-effects (e.g., `StrictMath.sin`), and methods in the bytecode library.

A dynamic method invocation is an expression that may dispatches to a dynamic method. When the static signature of the invoked method is an expression to invoke a method such as C`.`m`(...)`, and there exists a dynamic method with the same signature in C's *effective subclasses* (explained later), the invocation is dynamic.

Finally, a conditional pointcut `if(`e`)` is static if e has none of the following subexpressions:

- a variable bound by other pointcuts (e.g., `this`, `target` and `args`),
- an assignment or reference to a mutable class variable, or

```
aspect Tracing {
  static boolean flag = true;
  final static Map traced = new HashMap();

  pointcut isActive(): if(flag);
  pointcut nullArgument(Object x):
    args(x) && if(x == null);
  pointcut nullAccess():
    if(thisJoinPoint.getTarget() == null);
  pointcut tracedMethods():
    if(traced.get(thisJoinPoint.getSignature()
                    .getName()).boolValue())
}
```

Figure 6. Examples of dynamic conditional pointcuts.

- an invocation of a dynamic method.

The above rules are general enough to define useful analysis-based pointcuts. In fact, the rules do not prohibit use of local variables, object creations, assignments to instance fields, method calls, and so forth. As a result, the pointcut definitions presented in Section 4.1 can be made static.

At the same time, the rules determine the conditional pointcuts in Figure 6 as dynamic because:

(`isActive`) `flag` is a non-final field,

(`nullArgument`) `x` is a variable bound by the `args` pointcut,

(`nullAccess`) `getTarget` is predetermined as dynamic, and

(`tracedMethods`) `traced` is of `Map` type, which is not immutable.

5.2.2 Efficient and Precise Binding-Time Checking

It is not difficult to implement a naive binding-time checker based on the aforementioned rules. AspectJ compilers generate a boolean method (which we call *if-residue*) for each conditional pointcut. With the help of a bytecode library, it is simple to judge whether a method has any instruction that violates one of the rules in an inter-procedural manner.

However, precision and efficiency of binding-time checkers depend on how it computes *effective subclasses* of C when judging whether a method invocation is dynamic. We first explain why a naive implementation, which uses all subclasses of C in the entire program (including libraries), has problems, and then explain our algorithm based on the Rapid-Type Analysis [4].

Problems of A Naive Implementation. Assume a program has the `violation` pointcut in Figure 4 and also has a definition of `DbTable` class that implements `Collection` interface. The methods in `DbTable` have I/O side-effects since the class keeps values in an external database.

According to the above rules, `violation` is static if the method `callToFieldType` is static. Among other conditions, it requires the invocation of `Collection.contains` in l.17 to be static. This is confirmed by checking all definitions of `contains` in effective subclasses of `Collection`.

Naively, we might want to use all subclasses of `Collection` in the program as effective subclasses. This causes the next two problems:

- This slows down the analysis because it needs to check a large number of method definitions when the receiver type has many subclasses. This can be easily caused by the use of a collection library, and by the use of the `Object.equals` method, which frequently happens in Java programming.
- This can also degrade precision of the analysis as it can incorporate side-effecting methods which are never executed. In the above example, since `DbTable` is a subclass of `Collection`, the side-effecting operations in the `DbTable.contains` could make the `callToFieldType` dynamic, even though we know that `ArrayList` is the only runtime receiver class (see l.17).

Computing Effective Classes by Using Rapid Type Analysis. The above problem can be summarized as how to limit the set of classes to be examined when the algorithm encounters a method invocation. A similar problem can be found when performing a method inlining optimization in object-oriented languages, which determines a set of method declarations that actually executed from a method invocation. There are several analyses proposed to the method inlining problem, namely the Class Hierarchy Analysis (CHA) [9], the Rapid Type Analysis (RTA) [4], and Points-to Analysis (PA) [2,30]. The naive implementation discussed above can be classified as CHA.

We employ RTA because it is simple yet precise enough for our purpose. Intuitively, the analysis traverses a program from an entry point to collect a set of instantiated classes, and uses the set of classes to limit the set of method declarations that are actually executed from a method invocation. For the example discussed above (Figure 4), the analysis traverses the program from the expression in the conditional pointcut and finds that only `ArrayList` is instantiated[16] in the traversed code fragments. Then, when it encounters the call to `contains` in l.17, it only needs to examine the `contains` method declared in `ArrayList`.

Our algorithm is different from standard RTA in the following two respects. (1) While RTA usually assumes a single entry point for each program, ours focuses on more than one entry points in a program. In addition, code reachable from each entry point tends to be smaller than the entire program size. (2) Since our algorithm traverses from the expression in a conditional pointcut, the code reachable from the expression can reference an object that is instantiated in the code unreachable from the expression. However, we can safely ignore such a situation because the reference to such an object can only be obtained through a global variable or a free variable binded by other pointcuts such as `args` and `this` pointcuts, which makes the conditional pointcut dynamic.

5.3 Compilation Scheme

As mentioned, effects of aspects are visible to analysis-based pointcuts in SCoPE. In other words, analysis-based pointcuts observes woven programs through introspective reflection libraries. At the same time, SCoPE compiles the analysis-based pointcuts as static pointcuts; i.e., instructions for advice executions are inserted without runtime tests.

We realize those two contradicting properties by devising what we call the *backpatching* technique. The technique is advantageous to compiler developers because the technique works as a post-processor to a standard AspectJ compiler, and does not require major modifications to the existing compiler implementations.

We first explain our compilation scheme at conceptual level, and then present our implementation with the backpatching technique.

5.3.1 Double Compilation: A Conceptual Compilation Scheme

Conceptually, our compilation scheme is simple. It compiles the whole program, including aspects, twice:

- The first step is to merely produce woven code by using a standard compiler.

[16] We assume library methods called from `callToFieldType` and `fieldTypes` do not instantiate objects.

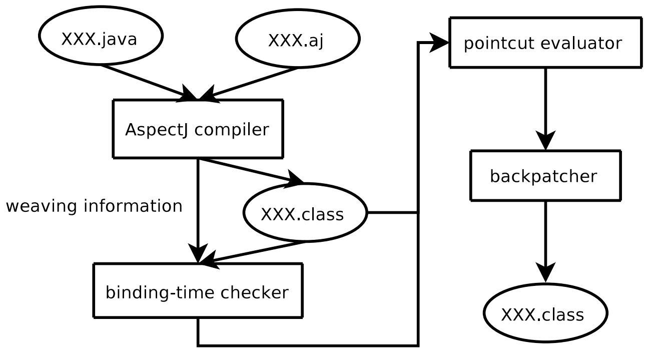

Figure 7. Compilation scheme with backpatching.

- The second step is to compile the program again, but whenever the compiler matches a conditional pointcut with respect to a join point shadow that it judges to be static, it evaluates the expression in the pointcut. When the evaluation uses a reflection library, the woven code generated at the first step is used as a code base. If the result is true, it inserts instructions to execute the advice body at the shadow. Otherwise, it does nothing.

Note that this scheme fully eliminates runtime overheads of the static conditional pointcuts, as the second step treats them in a similar manner to static pointcuts. At the same time, it achieves aspect visibility by using the woven code as the code base.

5.3.2 Backpatching: More Efficient Compilation Scheme

Our actual implementation uses the backpatching technique rather than double compilation, for more efficient compilation. As we see, double compilation takes twice as much time as compilation by standard compilers since it has to compile the whole program twice.

Figure 7 illustrates our compilation scheme. We explain each step by assuming it compiles the following program:

```
1  class Main {
2    public static void main(String[] args) { ... }
3    public String toString() { ... }
4  }
5
6  aspect TraceLower {
7    pointcut executeLowercaseMethod():
8      execution(* *(..)) &&
9      if(thisJoinPointStaticPart.getSignature()
10         .getName().matches("^[a-z]+$"));
11   before(): executeLowercaseMethod() { ... }
12 }
```

AspectJ Compiler. Using an AspectJ compiler, the woven class files are generated with *weaving information*. The woven classes of the above example look like this[17]:

```
1  class Main {
2    final static JoinPoint.StaticPart shadow$1 = ...;
3    final static JoinPoint.StaticPart shadow$2 = ...;
4    public static void main(String[] args) {
5      if(TraceLower.if$1(shadow$1)) // runtime test
6        TraceLower.aspectOf().before$1();
7       ...method body
8    }
9    public String toString()  {
10     if(TraceLower.if$1(shadow$2)) // runtime test
11       TraceLower.aspectOf().before$1();
12      ...method body
13   }
14 }
15
16 class TraceLower {
17   // if-residue
18   static boolean if$1(JoinPoint.StaticPart jp) {
19     return jp.getSignature()
20            .getName().matches("^[a-z]+$"));
21   }
22   void before$1() { ... } // advice body
23 }
```

[17] We present the woven classes at the source language level for readability, though actual code has a bytecode representation.

In the woven code, conditional pointcuts are compiled into boolean methods (e.g., ll.18–21) we call those methods *if-residues*. Each join point shadow that matches `execution(* *(..))` has a code fragment that calls the corresponding advice body when the if-residue returns true (e.g., ll.5, 10).

The AspectJ compiler has been modified so that it will produce weaving information, which is a list of the location of the join point shadow that calls an if-residue, a signature of the if-residue, and parameters of the if-residue. In the above example, the following table represents the weaving information:

shadow location	if-residue name	parameters
l.5	`if$1`	`shadow$1`
l.10	`if$1`	`shadow$2`

Binding-Time Checker. For each if-residue in the woven code, the binding-time checker decides whether it is static by using the algorithm outlined in Section 5.2. In the above example, `if$1` is judged as static.

Pointcut Evaluator. The pointcut evaluator runs static if-residues with respect to each join point shadow. Before evaluation, it creates a separate class loader, and resets the state of bytecode libraries used in the if-residues, and let them load woven classes. A separate class loader is essential as SCoPE itself is written in Java and uses a bytecode library.

It then invokes an if-residue for each join point shadow listed in the weaving information with a static if-residue. If the if-residue takes a `JoinPoint` object as its argument, it creates a `JoinPoint` object based on the weaving information.

An interesting optimization here is parallel evaluation. Since we analyzed that no if-residue has no side-effects, we can safely evaluate them at the same time using threads.

For the shadow at l.5 in the above example, it evaluates `TraceLower.if$1(shadow$1)`, which returns true.

Backpatcher. For each evaluated shadow, the backpatcher replaces the method invocation instruction of the if-residue with a constant instruction that produces a corresponding truth value obtained at the previous step. As a result, the woven code has conditional branches with constant expression like this:

```
1  class Main {
2    ...
3    public static void main(String[] args) {
4      if(true) // backpatched
5        TraceLower.aspectOf().before$1();
6      ...
7    }
8    public String toString()  {
9      if(false) // backpatched
10       TraceLower.aspectOf().before$1();
11     ...
```

```
  }
}
```

Even though the resulting code still has conditional branches, the code runs as efficient as the code that does not use conditional pointcuts thanks to runtime compilers in Java virtual machines. If we have to run the code on an interpretive virtual machine, they can be eliminated by simple post-processing.

6. Extended Features

This section discusses several extended features to SCoPE that need syntax changes to the language. Those features are not implemented yet as we are focusing on the features that do not require syntactic extensions.

6.1 Explicit Declaration vs. Automatic Detection

While current SCoPE automatically finds analysis-based pointcuts, it becomes possible to declare analysis-based pointcuts explicitly by extending the syntax of the underlying language.

One of the most modest extensions is to add annotations to such a pointcut. For example, the following pointcut explicitly declares that the conditional pointcut should be evaluated at compile-time by using an annotation:

```
pointcut initObjectWithTmField():
  execution(*.new(..))
  && @btcStatic if(hasfield(thisJoinPoint,"tm"));
```

Even with this approach, our proposed algorithm to find static conditional pointcuts is useful to confirm that the expression in the annotated conditional pointcut does not access runtime information.

Another use of annotations is to allow the user-defined program analysis to access global variables. When a class field is annotated with `@btcStatic`, we could treat the field as if it was immutable, which otherwise would be judged as mutable in our current algorithm. This feature would be useful for optimizing some analyses by caching intermediate results into a global table, so that the results can be shared when analyzing different join point shadows.

Currently, we plan to implement these extensions when the underlying compiler supports annotations.

6.2 Higher-Order Pointcuts

A more challenging extension is to make analysis-based pointcuts higher-order; i.e., to allow analysis-based pointcuts to take sub-pointcuts as its parameters like `cflow`.

The definitions of analysis-based pointcuts in Section 4.1 explains the motivation. For example, the predicted control flow uses a string value to specify a starting point of the analysis as shown in Figure 2. If we could specify a pointcut instead of a string, the pointcut definition becomes more reusable through the use of the abstract pointcut mechanism in AspectJ.

In order to realize this idea, we need to devise (1) a mechanism to pass a pointcut description to a Java method, and (2) a mechanism to retrieve information from the pointcut parameter from the Java method.

Our current plan is to introduce a new pointcut primitive that *reifies* a pointcut description into a variable, and to offer methods to access matching shadows of the reified pointcut. The syntax of the new primitive is:

$$\texttt{reify(}v\texttt{, }p\texttt{)}$$

where v and p ranges over variables and pointcuts, respectively. We believe this syntax allows the compiler to easily distinguish pointcut descriptions from Java expressions.

The variable v has type `Pointcut`, which has the following method:

```
// returns a list of join point shadows matching the pointcut
Collection<JoinPoint.StaticPart>
Pointcut.getShadows();
```

```
1 static boolean pcflowGet(JoinPoint jp, Pointcut entry){
2   for(JoinPoint.StaticPart s: entry.getShadows())
3     for(Method m: methodNodeOf(s.getSig()))
4     ... the rests are the same
5 }
6
7 pointcut figureMove(Pointcut entry): set(* Fig+.*) &&
8     reify(entry, execution(void Fig+.draw())) &&
9     if(pcflowGet(thisJoinPoint, entry));
```

Figure 8. Predicted control flow pointcuts with higher-order pointcut mechanism.

		regex	hasfield	LoD	JHotDraw
source (LoC)		23	40	122	20974
compiled	SCoPE	200	351	977	134918
(# of	abc (enum)	63	90	82	70674
instructions)	abc	193	342	969	133043
	abc(memo)	N/A	N/A	N/A	133718

Table 3. Source and compiled code sizes of benchmark programs.

By using those mechanisms, the predicted control flow pointcut can be redefined as shown in Figure 8. In the definition, the `execution` pointcut at l.8 is reified into variable `entry`, and passed to `pcflowGet` method as its second parameter. The `getShadows` method in l.2 retrieves all join point shadows matching the `execution` pointcut, which are used as starting points of the analysis.

Since the second parameter to `reify` in l.8 can be any pointcut description, it is possible to describe more complicated pointcuts, or to parameterize them by using the abstract pointcut mechanism.

There are a few studies on supporting user-defined higher-order pointcuts [31,34]. However, none of them considers syntactic compatibility with AspectJ-like languages and compile-time pointcut evaluations.

7. Performance Measurements

We evaluated compilation and runtime efficiency of our SCoPE implementation in order to demonstrate SCoPE's practical feasibility.

For micro benchmarks, we created small programs with aspects that use analysis-based pointcuts. The programs have simple iterations with empty method invocations. The pointcuts are the regular expression matching (**regex**), the class structure analysis (**hasfield**), and coding style checking (**LoD**) as presented in Section 4.

For larger-scale benchmarks, we also used JHotDraw[18], a graphical figure editor application, whose display updating mechanism is separated into an aspect by using the `pcflowGet` pointcut in Section 4.

For comparison purposes, we also created the equivalent programs that do not use analysis-based pointcuts, but use primitive pointcuts such as `call`, `execution` and `set` by enumerating all applicable signatures instead. Those programs exhibit best efficiency as they should have no runtime overheads at all, while having the same behavior as the ones with analysis-based pointcuts. We refer those programs as **abc(enum)**.

[18] `http://www.jhotdraw.org/`

	SCoPE				abc	ov.
	A	B	C	D		(%)
regex	4.23	0.25	0.07	0.02	4.97	–9.46
hasfield	5.03	0.25	0.09	0.02	5.46	–2.93
LoD	7.53	0.27	7.61	0.02	7.58	3.17
JHotDraw	95.20	0.51	4332.95	0.23	95.32	0.65

Table 4. Compilation times of programs with analysis-based pointcuts (sec.). The columns titled A–D in the SCoPE section give breakdowns of the internal steps, namely (A) AspectJ compilation, (B) binding-time check, (C) pointcut evaluation and (D) backpatching. The overheads exclude the times for pointcut evaluation.

Table 3 shows the sizes of the source and compiled code of those four programs, measured by lines of code and the sum of the number of bytecode instructions, respectively.

All benchmark programs were executed on top of Sun HotSpot Server Java VM 1.5.0_08 running on dual Xeon 3.06 GHz Linux machine with 6 GB memory. Compilation and execution times are measured by `System.currentTimeMillis()`. We executed/compiled each program for 100 times, and calculated the median values.

7.1 Compilation Speed

We compared the compilation times of the benchmark programs using our SCoPE compiler and the AspectBench compiler version 1.2.1 (abc).

Table 4 shows the compilation times for SCoPE and abc for the four benchmarking programs. The columns titled A–D are the times elapsed for the four internal compilation steps in SCoPE shown in Figure 7. The rightmost column shows the overheads introduced by SCoPE. We excluded the times for the pointcut evaluation because the step mainly runs user-defined analysis methods.

As we see at the JHotDraw row, compilation overheads are small enough even with medium-sized programs. On the other hand, the overall compilation times are not very small. However, this is due to the abc's compilation process, and could be improved when we switched the underlying compiler to a faster compiler, such as the one developed by the Eclipse project (ajc).

7.2 Runtime Overheads

In order to verify our claim that SCoPE fully eliminates runtime overheads of the conditional pointcut, we executed the four benchmarking programs in the three configurations, namely SCoPE, abc(enum) and abc.

By following the DaCapo benchmarking methodology [6], we split a single run into the initialization phase (**init**) and three consecutive iterations (**1st** to **3rd**), and separately measured time spent for each phase.

Table 5 summarizes the results. Each column corresponds to different configuration. The rightmost column shows the overheads of SCoPE executions relative to the ideal, i.e., abc(enum), executions.

7.2.1 Method Invocation with Advice

The first three benchmarking programs compares efficiency of an empty method invocation when empty advice with analysis-based pointcut is applied to the method. Each iteration of those programs invokes the method for 10000000 times. They perform no operations at the initialization phases.

As we see in the regex, hasfield and LoD rows in the table, SCoPE successfully eliminates the runtime tests at advice execution as the SCoPE compiled programs runs as fast as the ideal ones at the second and third iterations. There are overheads at the first iterations, though. We presume that this is due to a conditional branch with a constant value at advice invocation. Since a JVM first executes a program by using a bytecode interpreter, such a branch poses overheads until a dynamic compiler in JVM optimizes it.

			abc	abc		ov.
		SCoPE	(enum)	(memo)	abc	(%)
regex	1st	0.03	0.02	N/A	12.64	50
	2nd	0.02	0.02	N/A	12.43	0
	3rd	0.02	0.02	N/A	12.41	0
has-field	1st	0.04	0.02	N/A	4.63	100
	2nd	0.03	0.03	N/A	4.24	0
	3rd	0.02	0.02	N/A	4.24	0
LoD	1st	0.06	0.02	N/A	124.46	200
	2nd	0.03	0.03	N/A	124.10	0
	3rd	0.02	0.02	N/A	124.00	0
JHot-Draw	init	1.06	0.84	591.80	N/A	26.19
	1st	2.18	2.18	5.55	N/A	0
	2nd	1.88	1.87	3.33	N/A	0.53
	3rd	1.87	1.86	3.32	N/A	0.54

Table 5. Execution times (sec.)

The figures in the **abc** column are more than two orders of magnitude larger than the ideal cases. This means that an existing compiler can not eliminate overheads of analysis-based pointcuts even for very simple analysis methods such as regex and hasfield.

7.2.2 Overall Runtime Performance with JHotDraw

In order to measure performance with a practical application program, we measured execution times of JHotDraw. The initialization phase in JHotDraw sets up a canvas in a window with menus and icons, and draws four different kinds of figure elements on the canvas. One iteration in JHotDraw moves the four figure elements on a canvas 1000 times. In order to exclude effects of interaction with a window system, we executed the programs with a virtual screen (Xvfb).

As the JHotDraw rows in Table 5 show, the SCoPE compiled program, when compared against the ideal one, has a 26% overhead at the initialization phase, but no overheads during the iterations. We presume that the overheads at the initialization phase are due to the conditional branches with constant values, and also due to bloated code size, which in turn impacts dynamic compilation speed. As we see in Table 3, the compiled code generated by SCoPE has almost as twice number of instructions as the ideal one, even though the former includes the analysis methods, which are never executed at runtime.

For the abc column, we used an analysis-based pointcut with memoization. The analysis method is modified to record the analysis result into a global table in order to avoid running the analysis more than once for the same join point shadow. The execution time of the program without memoization was more than two hours. With memoization, overheads during iterations are reasonable. However, the initialization phase becomes tremendously slower as it has to analyze the program for a number of join point shadows at runtime.

8. Related Work

There are several studies on supporting user-defined analysis-based pointcuts. Josh [8] and LogicAJ [13] are new AOP languages similar to AspectJ. Those languages have their own weavers, and allow the programmer to define new pointcut primitives that uses program analysis by extending the weaver. As a result, tools for those languages such as integrated development environments and debuggers need to support newly defined pointcut primitives. Eichberg et al. presented a language to define user-defined analysis-based pointcuts by using the XQuery language [11]. The work however does not cover implementation of the overall compilation framework.

Alpha is a new AOP language that provides rich program information to user-defined pointcuts [26]. Although it is possible to define many useful analysis-based pointcuts in Alpha, its dynamic execution model would require a sophisticated compilation framework in order to achieve as efficient performance as static pointcuts.

As far as the authors know, no publications for those languages explicitly mention about the aspect visibility and aspect precedence awareness properties discussed in Section 4.2. Some languages, LogicAJ [13] for example, seem to address this problem by automatically determining evaluation order of pointcut primitives and iteratively weaving advice and inter-type declarations. We believe that our backpatching approach provides reasonable semantics and better compilation speed.

XAspects is a system that integrates domain-specific aspect languages and component languages [29]. Although XAspects is not a general purpose language, it employs a similar compilation scheme in which an AOP feature examines the properties of the code generated by the result of first compilation.

Model-based pointcuts are an alternative solution to the fragile pointcut problem [16]. Since this approach requires the programmer to maintain a conceptual model in response to the program changes, we believe that analysis-based pointcuts will complement the model-based pointcuts when the programmer can clearly describe analyses.

Shadow Programming is an approach that uses user-defined analysis-based pointcuts for static crosscutting (e.g., inter-type declarations and warnings) [35]. As far as the authors know, those user-defined pointcuts can not be used in advice declarations. In addition, user-defined pointcuts in Shadow Programming can only access to information at the join point shadow. As a result, it would be almost impossible to realize inter-procedural analyses with their approach.

Partial evaluation is an optimization technique to generate more efficient programs by using some of the parameters of a program [15]. Evaluation of conditional pointcuts in SCoPE can be seen as partial evaluation, because it evaluates the expression in a conditional pointcut by using information of join point shadows, which is already known at compile-time. However, when compared with a fully-fledged partial evaluation system, SCoPE is much simpler since it evaluates the expression when it does not use any runtime parameters at all. As a result, when a conditional pointcut contains both static and dynamic expressions as below, the pointcut is regarded as dynamic in SCoPE:

```
if(flag && thisJoinPoint.getSignature()
           .getName().matches("^[a-z]+$"))
```

Partial evaluation techniques, which regenerate an expression by replacing static subexpressions with evaluated values, could optimize more kinds of conditional pointcuts.

9. Conclusion

This paper presented an approach called Static Conditional Pointcut Evaluation (SCoPE), which supports user-defined analysis-based pointcuts. The contributions of the paper are as follows. (1) We pointed out that a number of useful analysis-based pointcuts can be written by using conditional pointcuts in existing AOP languages. (2) We showed a compilation scheme that fully eliminates the runtime overheads of user-defined analysis-based pointcuts. (3) We demonstrated that our compiler implementation for AspectJ, which is built on top of the AspectBench compiler, has very small amount of compile-time and runtime overheads when executed on a Java virtual machine.

Compared to other AOP languages that support user-defined pointcut primitives, SCoPE achieves better integration with an existing AOP language, namely AspectJ. Syntactically and semantically, SCoPE is fully compatible with AspectJ. In other words, SCoPE can be seen as an aggressively optimizing compiler for AspectJ. At the implementation level, our SCoPE compiler acts as a post-processor for an existing AspectJ compiler. It only requires the AspectJ compiler to produce information about conditional pointcuts. Therefore, it is easy to follow the evolution of the underlying language.

Our future work includes separate compilation and more precise binding-time checking. Current SCoPE implementation assumes whole-program compilation so that one must have a complete set of classes in hands at compile-time. This is because our binding-time checking algorithm requires a complete class hierarchy information in a program. Moreover, analysis-based pointcuts introduce additional dependency from an analyzing module to the analyzed modules, which would complicate the rules for separate compilation.

As for the binding-time checking, our current algorithm is not precise enough when a user-defined analysis and an advice declaration interfere. Assume an advice declaration calls `thisJoinPoint.getArgs()` in its body. When the advice is applied to a method that implements defining an analysis-based pointcut, the binding-time checking algorithm will find the call and concludes that the pointcut is dynamic. However, the call to `getArgs` can be evaluated at compile-time because the join point object will also be instantiated during the execution of the analysis, rather than supplied outside from the pointcut. We plan to improve the algorithm by employing an efficient points-to analysis algorithm such as the ones proposed in [5, 30] instead of the rapid type analysis mentioned in Section 5.2.2.

Acknowledgments

We would like to thank all anonymous reviewers of this paper and the members of the PPP group and PoPL meeting at University of Tokyo, especially Jan Hanneman and Kohei Sakurai for their comments. We also thank Karl J. Lieberherr, Kenichi Asai, Kris De Volder, Klaus Ostermann, Shigeru Chiba and his research group for their comments to the early stage of this work.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 345–364. ACM Press, 2005.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.

[4] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341. ACM Press, 1996.

[5] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings*

of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pages 103–114. ACM Press, 2003.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, 2006.

[7] Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 313–336. Springer, 2000.

[8] Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open AspectJ-like language. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111. ACM Press, 2004.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[10] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150. ACM Press, 2004.

[11] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems*, pages 366–381. Springer, 2004.

[12] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[13] Stefan Hanenberg Günter Kniesel, Tobias Rho. Evolvable pattern implementations need generic aspects. In *Proceedings of Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2004.

[14] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[15] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.

[16] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 501–525, 2006.

[17] Greger Kiczales. The fun has just begun, 2003. Keynote talk at AOSD'03.

[18] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer, 2001.

[19] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

[20] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Proceedings of Foundations of Aspect-Oriented Languages workshop*, 2005.

[21] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[22] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, 1989.

[23] Hidehiko Masuhara and Tomoyuki Aotani. Issues on observing aspect effects from expressive pointcuts. In *Proceedings of Workshop on Aspects, Dependencies and Interactions*, pages 53–61, 2006.

[24] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the 1st Asian Symposium on Programming Languages and Systems*, pages 105–121. Springer, 2003.

[25] Sean McDirmid and Wilson C. Hsieh. Splice: Aspects that analyze programs. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, pages 19–38. Springer, 2004.

[26] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 214–240. Springer, 2005.

[27] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24. Springer, 2001.

[28] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002.

[29] Macneil Shonle, Karl Lieberherr, and Ankit Shah. XAspects: an extensible system for domain-specific aspect languages. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37. ACM Press, 2003.

[30] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM Press, 1996.

[31] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 158–167. ACM Press, 2003.

[32] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pages 125–135. IBM Press, 1999.

[33] Robert J. Walker and Gail C. Murphy. Implicit context: easing software evolution and reuse. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 69–78. ACM Press, 2000.

[34] Geoffrey Washburn and Stephanie Weirich. Good advice for type-directed programming aspect-oriented programming and extensible generic functions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 33–44. ACM Press, 2006.

[35] Pengcheng Wu and Karl Lieberherr. Shadow programming: Reasoning about programs using lexical join point information. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, pages 141–156. Springer, 2005.