

# Shadow Programming: Reasoning about Programs using Lexical Join Point Information

Pengcheng Wu

Karl Lieberherr

Northeastern University, Boston, USA  
{wupc,lieber}@ccs.neu.edu

**Abstract.** The expressiveness of AspectJ's dynamic join point model has been shown in many useful applications, while the static join point model (also called *lexical shadows*) has been studied less. We propose a notion of shadow programming that exposes a program's adapted lexical shadow information to compile time language constructs to enable customized static analysis and more expressive join point selection mechanisms. In particular, within the framework of the AspectJ language and compiler, we have designed and implemented two compile time language constructs, called *Statically Executable Advice* and *Pointcut Evaluator* respectively, to show how the lexical shadow information can be used.

## 1 Introduction

Aspect-oriented programs consist of base programs and a list of aspects. Aspects are composed with base programs using a so called *join point model* [10]. In AspectJ's terminology, *join points* are well-defined points during a program execution, around which extra aspectual code (i.e., *advice*) can be executed; *pointcut designators* pick out certain join points and expose values at those points. To compile aspect-oriented programs, aspect weavers (for example, the AspectJ compiler) are needed to parse the base programs and aspects, and to match lexical points in the program text against the pointcut designators. The matched lexical points are called *join point shadows* [16, 7], and advice code will be injected into those places. At run time, the advice will be executed on *dynamic join points* that are run time instances of the lexical shadows.

In Aspect-Oriented Programming (AOP) [10, 5] languages like AspectJ, *dynamic join point models* have been extensively studied and their expressiveness has been shown in many interesting applications, while much less has been done on the lexical shadow side. Shadows have only served as internal implementation constructs for AOP language compilers so far, however, it is also our view that a lot more can be exploited for other purposes as well. The examples include, but are not limited to, static program analysis and to support more expressive join point selection mechanisms.

Following this thought, this paper presents a notion of *shadow programming*, in which shadow information is exposed to aspect programmers who can take advantage of the information to reason about the program's static properties using

supported compile time facilities. In particular, to show the feasibility and the usefulness of this notion, two compile time facilities, called *Statically Executable Advice* and *Pointcut Evaluator* respectively, are presented for programmers to write compile time program analyzers and to define sophisticated join point selection algorithms. We have implemented both facilities as extensions to the AspectJ language and the compiler. We believe more compile time facilities can be developed along this direction.

**Outline.** The rest of the paper is organized as follows. Section 2 discusses the motivations of this work. Section 3 introduces our lexical join point model adapted from AspectJ's internal lexical shadow structures. Section 4 presents the two shadow programming facilities and their use cases. Section 5 briefly describes the implementations. Section 6 compares this work with other related work and Section 7 concludes the paper.

## 2 Motivations

### 2.1 Compile Time Program Analyzers

To make programs more understandable, efficient or evolvable, it is important for programmers to follow some programming conventions. For example, an object's clients should never directly access the object's fields, if there are corresponding getters and setters available.

Unfortunately, most programming conventions are not checked by modern programming language compilers that usually just do regular type checking and code generation. Our observation is that the lexical shadow notion of aspect-oriented programming language compilers contains rich static information about program structure, which can serve as excellent ground for nontrivial programming convention checking. As a matter of fact, the AspectJ language has already supported very simple program property compile time checking. For example, to check that in any getter method, there should not be any operation to change an object's states, one can put the following declare error statement in an aspect:

```
aspect Foo {
    declare error : set(* *.* ) && withincode(* *.get*(. .))
                  : "Side effect operations not allowed!";
}
```

When this aspect is compiled with a base program, the AspectJ compiler will try to match the program text against the pointcut designator expression `set(* *.* ) && withincode(* *.get*(. .))`, and if any match is detected, the specified error message will be printed out along with the corresponding program text's lexical address (file name and line number), so that we know a violation has occurred. However, this feature is still very primitive in two senses: (1) the conventions it can characterize are limited to those that can be directly expressed using AspectJ's pointcut designators; (2) programmers have no direct access to the lexical shadow information that is important to implement more complex compile time checkers, as we will see later.

In fact, the aforementioned `declare error` statement fails to report the complete violation set. If a getter method does not directly update a field, instead, it calls another getter method that updates a field, it can only report the violation of the latter method, while treating the former one as a nonviolation. Note that this misbehavior is not due to a programming bug, but due to the fact that AspectJ's `declare error` mechanism is limited in its expressiveness. Exposing lexical shadow information for programmers to access is essential for realizations of such kind of compile time checkers.

In [12], motivated by another programming convention checker, the Law of Demeter checker, we proposed an extension to AspectJ, called Statically Executable Advice, which allows programmers to define more complex computation using lexical shadow information. That proposal can be realized only if lexical shadow information is accessible to programmers at compile time. Observing that, now we have refined the proposal and implemented it in the AspectJ compiler (version 1.1). In this paper, we discuss its applications in a broader context of reasoning about program properties using exposed shadow information.

## 2.2 More Expressive Join Point Selection Mechanism

Many aspect-oriented programming languages follow AspectJ's join point selection mechanism, called pointcut designators [10]. In this mechanism, there are static primitive pointcut designators such as `call`, `execution`, `get`, and `set` to match method or constructor call sites, method or constructor bodies, field read accesses and updates respectively. Programmers can use a simple pattern language to specify the signatures of those lexical points to be selected. Logical connectors `||`, `&`, and `!` can be used to further refine selections. There are also *dynamic* pointcut designators including `cflow`, `this`, `target`, `args` and `if`. They refine join points selection at run time, and we will not cover them in this paper.

The expressiveness of the AspectJ's pointcut designator selection language is still very limited. One of the limitations is that it does not support join point selection based on particular properties of a lexical shadow other than those that can be directly expressed in the simple pattern language syntax. For example, one cannot select a call site `call(* *.store())` (call to a method named *store*) such that the target type has a field named `id`, which is useful in enterprise applications to map an instance of a class back to a persistent store. As a matter of fact, asking whether AspectJ can support selection of join points based on special properties of classes/methods has been one of the most common questions in the AspectJ user community. Just as two additional examples from the AspectJ users mailing list, people were asking how to select the executions of a method in a class but not in its subclasses [8] and how to select the executions of a method in nonanonymous classes [15]. They are either very difficult or impossible to be expressed in AspectJ's pointcut designator language in its current form.

New pointcut selection primitives have been proposed to address this issue. For example, in another AOP system, JBoss AOP [9], a new pointcut expression `hasField(..)` has been proposed and implemented to help specify the foremen-

tioned field scenario. But those extensions tend to be specific, and more general solutions are still needed.

Since the properties that a programmer wants to specify about program lexical points can be arbitrary, any general solution suggests programmer accessible lexical shadow information and a mechanism to reason about shadow information. Based on the programmer accessible shadow assumption, we propose a new pointcut designator expression called *Pointcut Evaluator* as a general approach to this problem and it has also been implemented in the framework of the AspectJ compiler (version 1.1).

### 3 Lexical Join Point Model

#### 3.1 Lexical Shadows in AspectJ Compiler

We briefly introduce what lexical shadows are in the AspectJ language and compiler. According to the AspectJ language model, join points are points in the execution of a program. Join points are also called *dynamic join points*, because they exist only at run time. Each dynamic join point has its corresponding static part in the program text, which is called *lexical shadow* [16, 7]. The AspectJ compiler operates on potential lexical shadows and matches those shadows against pointcut designators. For matched shadows, extra code will be injected to call advice at an appropriate time and to construct dynamic join point instances that are accessible to advice code at run time through a keyword variable `thisJoinPoint`.

Lexical shadows are abstractions of entities in a program, and they contain rich static information about those entities. As an example, a method call shadow contains the name of the method, the static types of the target object and the arguments, and in which method body (which is another shadow) the call is invoked. Currently, there are nine kinds of shadows in the AspectJ language and compiler [7]. The most common ones are method (or constructor) execution<sup>1</sup>, method (or constructor) call, field get, and field set.

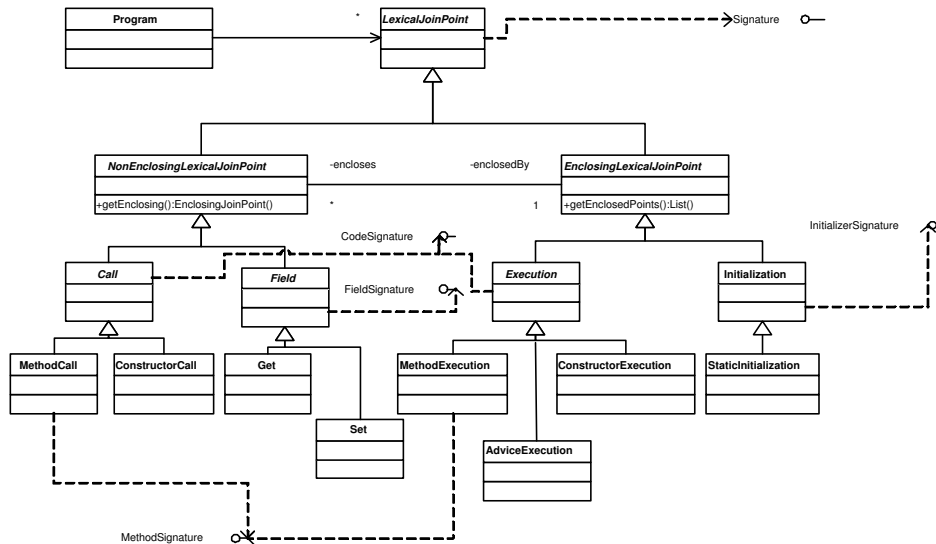
So far shadows have only served for the compiler's internal implementation's purpose, i.e., for matching programming entities against pointcut designators. Our goal is to make shadow information available to programmer accessible compile time facilities. The shadow class structure of the AspectJ compiler is for its internal use only and thus is unsuitable for programmers to access. We have designed and implemented a new structure based on the compiler's shadow structure information. We call the new structure *lexical join point structure*.

#### 3.2 Lexical Join Point Structure

Fig. 1 is the UML diagram of our abstraction of lexical join points.

---

<sup>1</sup> The term *execution* may be a little confusing to those who are not very familiar with AspectJ, since it sounds like a run time thing. You can just view it as the body of a method or a constructor.



**Fig. 1.** Lexical Join Points

A program consists of a set of lexical join points, each of which fits into one of the two categories: `NonEnclosingLexicalJoinPoint` and `EnclosingLexicalJoinPoint`. An `EnclosingLexicalJoinPoint` may contain any number of `NonEnclosingLexicalJoinPoint` objects.

Lexical join points are further classified according to their kinds. The nine concrete classes<sup>2</sup> correspond to the nine kinds of shadows defined in the AspectJ language respectively. Some of the lexical join point classes implement signature interfaces (indicated as circled lines in the figure) so that programmers can access needed information through well-defined APIs. Those signature interfaces are all defined in AspectJ's public reflective API package `org.aspectj.lang.reflect`, which should be familiar to AspectJ programmers. Letting lexical join point classes implement specialized signature interfaces deliver better API accessibility to programmers, while in the current AspectJ language, programmers only have direct access to the general `Signature` interface and in programs, they often have to cast it down to more specialized interfaces according to the kind of the current join point.

The signature interfaces lend the programmers the capability to access type hierarchy information as well, which could come from one of two sources. The first source is Java's reflection system where class `Class` provides entry points to type information about a loaded class. This is a feasible source since AspectJ is using a byte code weaving approach and it is fair to assume that all the base

<sup>2</sup> Abstract class names are in the *italic* font, the circled lines represent interface types and the dashed lines represent implementation relationship between classes and interfaces.

program classes have been in the byte code form before weaving time. The other source is AspectJ compiler's internal type information abstracted as class `TypeX`, which collects all the class information in the program, either from source text or byte code. Class `TypeX` has almost the same APIs as Java's reflective `Class` and thus it is easy to unify them into a general one. To simplify our implementation, we use the reflection system as our source for type hierarchy information. For example, for any lexical join point object, we can ask in which *static* type this lexical join point is defined by calling method

```
Class getDeclaringType()
```

which is declared in interface `Signature`. In the next section, we will see examples how to make use of type information to reason about programs at compile time.

## 4 Shadow Programming Facilities

We show how we can take advantage of the exposed lexical join point information in two shadow programming facilities we have designed and implemented in the AspectJ compiler.

### 4.1 Statically Executable Advice

AspectJ's `declare error` construct is a useful static checking feature, but it is not expressive enough to check complex program properties. By extending AspectJ's `declare` mechanism, we can support user-defined complex static checking logic using available lexical join point information. This new construct is called *Statically Executable Advice* that can be declared in an aspect definition using the following syntax.

```
declare advice : pointcut expression : Identifier ;
```

The *pointcut expression* can be any legal AspectJ pointcut expression as long as no `cflow`, `if`, `args`, `this` and `target` is used in it<sup>3</sup>. We add this restriction because we want the pointcut expression to be statically resolvable, so that we can apply statically executable advice at compile time. The *Identifier* has to be the name of a user-defined class implementing interface `IStaticallyExecutableAdvice` as shown in Listing 1.

The idea is that for each of the `declare advice` declarations, the AspectJ compiler will load the class indicated by *Identifier* and create an instance from it. During the pointcut matching phase, if a shadow matches the specified pointcut expression, the compiler will create a lexical join point object corresponding to the shadow and call the corresponding method on the *Identifier* instance with the new lexical join point object as the argument. For example, if the current lexical shadow is a method call and it matches the pointcut expression, then the `onCall` method will be invoked on the *Identifier* instance with a `MethodCall` lexical join point created from the shadow as the argument. In addition, the `start()`

<sup>3</sup> In AspectJ's terminology, a pointcut designator without `cflow`, `if`, `args`, `this` and `target` is called a *statically determinable pointcut*.

method will be invoked before the whole compilation process begins and the `finish()` method will be invoked after the whole compilation process has finished.

**Listing 1.** `IStaticallyExecutableAdvice.java`

```
interface IStaticallyExecutableAdvice {
    void beforeExecution(Execution e);
    void afterExecution(Execution e);
    void beforeInitialization(Initialization ini);
    void afterInitialization(Initialization ini);
    void onCall(Call c);
    void onField(Field f);
    void start();
    void finish();
}
```

Please note that for each of the two direct subclasses of class `EnclosingLexicalJoinPoint`, namely `Execution` and `Initialization`, there are both `before` and `after` methods declared for it. But for each of the two direct subclasses of class `NonEnclosingLexicalJoinPoint` there is only an `on` method. The semantics is that if an enclosing lexical join point, say  $e$ , matches the pointcut expression, then its `before` method will be executed prior to all of the `on` methods for the matched nonenclosing lexical join points enclosed in  $e$ , and its `after` method will be executed after all of those `on` methods. This temporal order is important to simulate the lexical scopes of shadows.

One of the major advantages of using the `declare advice` construct is that we can make use of AspectJ's pointcut designator selection mechanism to specify where in the program we want to apply the checking logic. This expressiveness is similar to the traversal strategies and Visitor methods in the Demeter system [13] in that pointcut expressions play the roles of traversal strategies and statically executable advice methods play the roles of Visitor methods.

The following use cases exposes the usage and features of the statically executable advice.

**Case 1: Side Effect Checking for Getters** As discussed in Section 2.1, AspectJ's `declare error` mechanism is not expressive enough to statically report whether a getter method may have side effects (change the states of some objects). We need to check the following properties:

1. If a getter method directly has field updating operations, then this method has side effect;
2. If a getter method calls some non-getter methods (whose names do not start with `get`), we assume non-getter methods always have side effects and so is this getter method;
3. If a getter method calls other getter methods, then whether that getter method has side effect depends on those methods, and we need to record this dependency information for future processing;

4. After having processed all of the getter methods, then from the methods that have been marked as having side effects, we compute the transitive closure following the reversed dependency relationships and mark those methods as having side effect as well;
5. All of the unmarked getter methods are side effect free methods.

It is easy to see that even this simple program property cannot be easily checked. Fortunately, with our new statically executable advice construct, one can implement this static checker elegantly.

First, we need to specify what shadows we want to check and what is our class that implements the static checker. It is specified using our `declare advice` construct in an aspect as in Listing 2.

**Listing 2.** CheckerAspect.java

```
public aspect CheckerAspect {
    declare advice : withincode(* *.get*(..)) : EffectFreeChecker;
}
```

Basically, we need to check every lexical shadow occurring in the body of a getter method, which is specified as pointcut designator expression `withincode(* *.get*(..))`. Class `EffectFreeChecker`, which implements interface `IStaticallyExecutableAdvice`, is the class whose instance does the actual checking. A supporting class `MethodNode` maintains a global map that maps a getter method's signature to its corresponding `MethodNode` instance. Each `MethodNode` instance maintains the side effect dependency relationships of other `MethodNode` instances on itself. The supporting class `MethodNode` also maintains a global list of `MethodNode` instances that have been identified as having side effect.

With this supporting class, we now only need to implement three statically executable advice methods in class `EffectFreeChecker` (Listing 3).

When the compiler matches a field operation within the body of a getter method, the above `onField` method is invoked. If the operation happens to be a `Set` operation (determined by line 3, Listing 3), then that enclosing getter method is directly marked as with side effect.

On the other hand, as shown in the `onCall` method, if a method call shadow is matched in a getter method, we first make sure this call is also to a getter method (if not, we immediately mark the enclosing getter method "having side effect"), then we establish the side effect dependency relationships for future processing (lines 11 - 12, Listing 3).

Once all of the shadows in the program have been processed, the `finish()` method is invoked. The sole purpose of the `finish` method is to call a utility method `reachTransitiveClosure()` that starts from the nodes in the side effect method node list and marks every `MethodNode` instance reachable via the transitive closure of dependency relationships as having side effect. After this process finishes, all the getter methods that have side effects will have been marked so.

When the base program and aspect `CheckerAspect` (Listing 2) are compiled using our extended AspectJ compiler, the getter method side effect checking will be executed automatically during the compilation process.

Listing 3. EffectFreeChecker.java

```
class EffectFreeChecker implements IStaticallyExecutableAdvice {
2  public void onField(Field f) {
    if(f instanceof Set)
4     MethodNode.addSideEffectNode(f.getEnclosing().getSignature());
    }
6  public void onCall(Call c) {
    if(!c.getName().startsWith("get")) {
8     MethodNode.addSideEffectNode(c.getEnclosing().getSignature());
    return;
10   }
    MethodNode theCall = MethodNode.getMethodNode(c.getSignature());
12   theCall.addDependencyRel(c.getEnclosing().getSignature());
    }
14  public void finish(){
    reachTransitiveClosure();
16  }
    //other empty methods are skipped
18 }
```

The implementation of this checker is very succinct due to two reasons: (1) the exposed lexical join point information adapted from shadows is available to the statically executable advice construct for free, while traditional approaches typically require a lot of parsing and abstract syntax tree traversals to get similar information; (2) as pointed out earlier, AspectJ's pointcut designator expression can declaratively instruct the compiler to only run the checker on relevant shadows, which otherwise would require a lot more code.

**Case 2: Law of Demeter Checker** The Law of Demeter [14] (LoD) is another example of a programming convention we would like to check.

The class form of the LoD is a variant of the LoD. A class is less coupled with other classes if it follows the class form of the LoD. The essence is to restrict the classes whose methods can be invoked in a class's method. It can be summarized as: in a method of a class, we can only call methods on the following set of classes

- the class itself;
- classes of the class's fields;
- classes of the method's parameters;
- classes whose constructors are invoked in the method body.

It is clear that checking the class form of the LoD only requires statically available information. But we could not implement a sound checker in AspectJ, because using its dynamic join point model we could only do checking on a per-execution basis. With our new statically executable advice construct, we have easily implemented a sound LoD static checker.

Listing 4. LoDCheckerAspect.java

```
public aspect LoDCheckerAspect{
    declare advice : (execution(* *.*(..)) || call(* *.*(..)) || call(*.new(..)))
                    && withincode(* *.*(..))
                    : LoDChecker;
}
```

Listing 5. LoDChecker.java

```
public class LoDChecker implements IStaticallyExecutableAdvice {
2   HashSet permissibleTypes = new HashSet();
   List potentialViolations=new ArrayList();
4   Class thisClass;
   public void onCall(Call c) {
6       if(c instanceof ConstructorCall) {
           permissibleTypes.add(c.getDeclaringType());
8           return;
       }
10      java.lang.reflect.Field[] fields = thisClass.getDeclaredFields();
       for(int i=0; i<fields.length; i++) {
12          if(c.getDeclaringType() == fields[i].getType())
              return;
14      }
       if(permissibleTypes.contains(c.getDeclaringType()))
16          return;
       potentialViolations.add(c);
18   }
   public void beforeExecution(Execution e) {
20       permissibleTypes.clear();
       thisClass = e.getDeclaringType();
22       permissibleTypes.add(thisClass);
       Class[] paraTypes = e.getParameterTypes();
24       for(int i=0; i<paraTypes.length; i++)
           permissibleTypes.add(paraTypes[i]);
26   }
   public void afterExecution(Execution e) {
28       Iterator it = potentiallyViolations.iterator();
       while(it.hasNext()) {
30           Call c =(Call)it.next();
           if(!permissibleTypes.contains(c.getDeclaringType()))
32               System.err.println("An LoD violation at: " + c.getSourceLocation());
       }
34       potentialViolations.clear();
   } }
}
```

Listing 4 uses declare advice to specify where in the program the checks take place and which class implements the checking logic. We need to capture all method call sites residing in a method body to check their target types. We also need to capture constructor calls since they provide permissible types and we want to collect them. The shadows corresponding to `execution(* *.*(..))`

are the method bodies we are checking. Listing 5 is the implementation of the `LoDChecker` class that performs all the necessary checking for the class form of the LoD.

The instance of class `LoDChecker` maintains the permissible types (in a `HashSet`, Listing 5) in the context of a method body. Within a method body, if the compiler finds any constructor call shadow (Listing 5, lines 6 - 9), its type is added to the set of permissible types for the method; if a method call shadow is found instead, we first check whether the target type is one of the field types or already known permissible types, if not, then this call *may be* a violation call (Listing 5, lines 10 - 17). We cannot determine whether a method call site really violates the LoD until we have processed every shadow in the method body, since there may be other constructor calls coming after that method call, which will bring about more permissible types. This explains why we only report violations in the `afterExecution` method, right before the process leaves a method body being checked.

**Summary** Our experiences with the two static checkers suggest that with the lexical join point information exposed at compile time, one can use statically executable advice to implement static checkers to check nontrivial program properties. Of course, the program properties that can be checked in this construct also depend on the expressiveness of AspectJ's pointcut designators. With more expressive designators added in, we can check even more interesting properties.

## 4.2 Pointcut Evaluator

As discussed earlier, in practice, there are many requirements for mechanisms to select join points based on properties of shadows, which usually cannot be expressed using the simple pattern matching syntax supported by AspectJ in its current form. The exposed lexical join points provide excellent grounds for more expressive join point selection mechanisms. One such mechanism is a compile time facility called *Pointcut Evaluator* which we propose and have implemented in the AspectJ compiler.

Again, using AspectJ's `declare mechanism`, in an aspect's definition, one can declare a pointcut evaluator variable which can be referred later in a pointcut designator definition. Here is the syntax.

```
declare evaluator : Identifier
                  : Identifier
                  [ : Instantiation Option ] ;
```

The first *Identifier* is the name of an evaluator variable that will refer to an instance, while the second one is the name of a class, from which the instance referred by the evaluator variable will be instantiated. The class has to implement a simple interface `IPointcutEvaluator` as shown in Listing 6. The optional *Instantiation Option* controls how the evaluator variable instance is instantiated, and currently, the option can be either `perCompile` (it is the default option, if no

option is specified), or `perPCD`. Option `perCompile` indicates that there will be just a singleton instance associated with the evaluator variable during the whole compilation process; while option `perPCD` indicates that for each pointcut designator definition, there will be a separate instance associated with an evaluator variable used in the designator.

**Listing 6.** `IPointcutEvaluator.java`

```
public interface IPointcutEvaluator {
    public boolean eval(LexicalJoinPoint ljp);
}
```

A declared pointcut evaluator variable can be referred in a pointcut designator to refine the join point selection. An evaluator variable can appear anywhere in a pointcut designator where an AspectJ's standard primitive designator is expected. When the AspectJ compiler is matching a potential shadow against a pointcut designator, and if an evaluator variable is used in the designator, then the `eval` method will be called on the instance referred by the evaluator variable, with a `LexicalJoinPoint` object constructed from the shadow as the argument. Then the boolean return value from the `eval` method will be used together with logical connectors and the standard primitive designators to determine the final matching of the shadow. We will see how it is used in the following use case.

**Use case: Contract checking for `equals/hashCode`** Modern software systems often rely on their components to obey some contracts [17] to ensure that systems behave correctly. Usually type systems cannot check contracts and thus they cannot be statically enforced. One such example is the contract between method `equals(Object)` and method `hashCode()` in Java's `Object` class. As documented in the Java API documents, under the entry of class `Object`, this contract is literally specified as the following statement:

- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects **must** produce the same integer result.

The importance of this contract is evident from the fact that it has caught significant attentions from both professional programmers [1] and academic researchers [4]. To address this contract, in his well known book *Effective Java* [1], Joshua Bloch makes it a rule that *Always override `hashCode` when you override `equals`*.

The rationale of that rule is that if you have overridden the `equals` method without overriding the `hashCode` in a class, it is almost certain that the class will fail to obey the aforementioned contract. Following that rule (which is statically checkable) itself, however, will not guarantee that contract will be obeyed. Instead, we cannot determine it until run time. So to determine whether a class obeys this contract, both compile-time and runtime checking is required.

It is impossible to implement this contract checking in the current AspectJ language, not just because it cannot provide the needed static checking support,

but also because its pointcut designator mechanism is not expressive enough for what is needed for the dynamic checking. It is not easy to implement this in other software engineering tools either. But our pointcut evaluator construct provides an elegant solution to this task.

Listing 7 and Listing 8 are the implementation of this contract checker using the pointcut evaluator facility.

**Listing 7.** ContractCheckingAspect.java

```
aspect ContractCheckingAspect {
    declare evaluator: withHashCode : HashCodeChecker;
    pointcut p(Object b): execution(* *.equals(Object)) && args(b) && withHashCode;
    after(Object b) returning(boolean r): p(b) {
        if(r) { //equals returns true, then test the contract.
            if(thisJoinPoint.getThis().hashCode() != b.hashCode())
                System.err.println("Contract violation!");
        }
    }
}
```

**Listing 8.** HashCodeChecker.java

```
public class HashCodeChecker implements IPointcutEvaluator {
    public boolean eval(LexicalJoinPoint ljp) {
        //only returns true if the class overrides hashCode method
        Class thisType = ljp.getDeclaringType();
        Method[] methods = thisType.getDeclaredMethods();
        for(int i=0; i<methods.length; i++)
            if(methods[i].getName().equals("hashCode"))
                return true;

        //if it overrides equals(), must also override hashCode()
        if((ljp instanceof MethodExecution) && ljp.getName().equals("equals"))
            System.err.println("Must also override hashCode()!");
        return false;
    }
}
```

In aspect `ContractCheckingAspect`(Listing 7), a pointcut evaluator variable `withHashCode` is declared to refine the join point selection of pointcut designator `p`, which captures executions of `equals` method only if the target class also overrides the `hashCode` method. The static determination whether a class overrides `hashCode` actually is implemented by class `HashCodeChecker`(Listing 8), from which the instance referred by variable `withHashCode` is created. Class `HashCodeChecker` also checks the rule (a class that overrides `equals` must also override `hashCode`) at compile time. The `after` advice in Listing 7 does the run time checking of the contract

## 5 Implementations

The implementations of the statically executable advice and pointcut evaluator facilities turned out to be seamless in the Eclipse AspectJ compiler (version 1.1), due to the compiler's extensibility.

There are two important concepts in the architecture of the AspectJ compiler, which are *Shadow* and *Shadow Munger* [7]. Shadow as an abstraction of the program text provides the grounding for our exposed lexical join point model, on which our two compile time facilities are based. Shadow munger is an abstraction of compile time actions when a shadow matches a pointcut designator expression. Two examples of shadow munger are *declare error processor* and *advice weaver*, which respectively prints out a specified error message and weaves the advice into the appropriate places when a shadow in the program is matched. The statically executable advice construct is just implemented as another shadow munger, called `AdviceMethodLauncher`. It constructs a lexical join point object from the matched shadow and uses it as the argument to call an `IStaticallyExecutableAdvice` method corresponding to the shadow on the instance created in the declare advice statement.

On the other hand, in a pointcut designator, a pointcut evaluator variable is treated the same as other standard primitive pointcuts, except that its semantics is to invoke the `eval` method on the instance referred by the evaluator variable with the lexical join point object corresponding to the current shadow as the argument. Its returning value is used to determine the matching of the shadow against the pointcut designator.

## 6 Related work

Josh by Chiba and Nakagawa [2] is a new AOP language based on a compile time reflection library called Javassist. In Josh, programmers can have pointcut designators that are implemented as static Java methods using the Javassist API to access static information of the base program, just as we can refine the join point selections in pointcut evaluators using exposed lexical join point information. Our approach has a better integration with AspectJ's well accepted join point model, while Josh users have to program in two models, namely AspectJ-like join point model pointcuts and Javassist's compile time reflection model. Our exposed lexical join point information is obtained almost for free from the AspectJ compiler's internal shadow information, while Josh has to get similar information from a special compile time reflection library.

Eichberg, Mezini and Ostermann [3] use the XQuery language as a join point selection mechanism while the underlying shadow model is an XML representation of class information (translated from class files using a special tool). However, programmers should find our model more accessible, since in our model, there is no need for special translation from Java programs to their XML representations.

There are also expressive pointcut languages [6, 19, 11, 18] based on logic programming and its unification mechanism. These languages support join point

selection on various data models, including the abstract syntax tree, the static type system, execution trace and heap objects. Arbitrary pointcut predicates can be written with regard to the data models to select join points. Aspects written in these pointcut languages tend to be less coupled to syntactic properties of base programs, and thus better aspect reusability can be achieved, just as the case in our shadow programming model. Due to logic programming's declarative nature and its built in unification mechanism, pointcut expressions in those languages are very concise.

SCoPE [20] is an extended AspectJ compiler that optimizes conditional pointcuts (if pointcuts) so that when if pointcuts only refer to statically available information, the SCoPE compiler can evaluate them at compile time and thus there is no runtime overhead associated with them, just like our Pointcut Evaluator facility. The expressiveness of these two features are comparable.

## 7 Conclusion

We observe that shadow information in AOP languages, particularly in AspectJ, can be also exploited for tasks other than compiler implementations, such as customized compile-time analysis and more expressive join point selection. The notion of shadow programming is proposed to expose the shadow information for usage by compile-time facilities. Concretely in the AspectJ language and compiler, we have designed and implemented the exposed shadow information and API as our lexical join point structure. Two shadow programming facilities, statically executable advice and pointcut evaluator, have been designed and implemented. Use cases for them have been presented exposing their usefulness and feasibility. We believe that more shadow programming facilities can be developed along this line and they will broaden the application domain of AOP languages.

## Acknowledgements

We would like to thank Jeffrey Palm and Therapon Skotiniotis for their valuable comments on earlier drafts of this paper. We are also grateful to the anonymous reviewers of GPCE'2005 for their helpful comments.

## References

1. Joshua Bloch. *Effective Java Programming Language Guide*. Sun Microsystems Inc., 2001. Pages 25-35.
2. 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.
3. Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*. Springer, LNCS, 2004.

4. Matthias Felleisen. Functional objects. In *Invited Talk at ECOOP*, 2004. Also available at <http://www.ccs.neu.edu/home/matthias/Presentations/ecoop2004.pdf>.
5. R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis*. <http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf>, October 2000.
6. 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.
7. Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.
8. Charles N. Harvey III, Jun 2004. <http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg02460.html>.
9. JBoss Inc. JBoss AOP. <http://www.jboss.org>.
10. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mike Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, pages 327–353, Budapest, 2001. Springer Verlag.
11. Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable Pattern Implementations Need Generic Aspects. In *Proceedings of Workshop on Reflection, AOP and Meta-Data for Software Evolution*, June 2004.
12. Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A Case for Statically Executable Advice: checking the Law of Demeter with AspectJ. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 40–49. ACM Press, 2003.
13. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
14. Karl J. Lieberherr and Ian Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 24(3):67–78, March 1989.
15. Marius Marin, Sep 2004. <http://dev.eclipse.org/mhonarc/lists/aspectj-users/-msg02944.html>.
16. Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003)*, pages 46–60. LNCS, 2003.
17. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997. Second Edition.
18. Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the European Conference on Object-Oriented Programming*, 2005.
19. Tobias Rho and Günter Kniesel. LogicAJ - Uniform Genericity for Aspect Languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn, December 2004.
20. Tomoyhuki Aotani and Hidehiko Masuhara. Compiling Conditional Pointcuts for User-Level Semantic Pointcuts. In *Software engineering Properties of Languages and Aspect Technologies Workshop*, Chicago, IL, USA, Mar 2005.