The abc Group

Semantics of Static Pointcuts in AspectJ

abc Technical Report No. abc-2006-3

Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco,
Oege de Moor, Damien Sereni, Julian Tibble, Mathieu Verbaere

Programming Tools Group, Oxford University Computing Laboratory, Wolfson Building,
Parks Road, Oxford OX1 3QD, UK

{pavel.avgustinov,elnar.hajiyev,neil.ongkingco,
oege.de.moor,damien.sereni,julian.tibble,mathieu.verbaere}@comlab.ox.ac.uk

November 17, 2006

a s p e c t b e n c h . o r g

Contents

List of Figures

List of Tables

Abstract

In aspect-oriented programming, one can intercept events by writing patterns called *pointcuts*. The pointcut language of the most popular aspect-oriented programming language, AspectJ, allows the expression of highly complex properties of the static program structure.

We present the first rigorous semantics of the AspectJ pointcut language, by translating static patterns into safe (*i.e.* range-restricted and stratified) Datalog queries. Safe Datalog is a logic language like Prolog, but it does not have data structures; consequently it has a straightforward least fixpoint semantics and all queries terminate.

The translation from pointcuts to safe Datalog consists of a set of simple conditional rewrite rules, implemented using the Stratego system. The resulting queries are themselves executable with the CodeQuest system. We present experiments indicating that direct execution of our semantics is not prohibitively expensive.

1 Introduction

Aspect-oriented programming enables one to intercept events at runtime by writing patterns called *pointcuts*. The intercepted events are named *joinpoints*. Whenever a pointcut matches a joinpoint, extra code (called *advice*) is run. The most popular language that embodies these ideas is AspectJ, an extension of Java [28]; there is also a modern aspect-oriented version of C#, named Eos-U [36]. Typical applications include runtime verification of system-wide invariants, the implementation of authentication and authorisation mechanisms, as well as various caching and pooling strategies [30].

With the growing popularity of aspect-orientation, numerous researchers have started to investigate the semantics of aspect-oriented languages (*e.g.* [2, 3, 10, 13, 14, 24, 26, 27, 32, 40, 43, 45–47]). All such studies have focused on the operational semantics of advice, taking a very simple pointcut language. For instance, in [43], the core language identifies program points for instrumentation through explicit labels, and pointcuts are sets of such labels.

Yet in AspectJ itself, the pointcut language is very complex, allowing the programmer to capture intricate properties related to the static structure of the program. This complexity is a continuing source of serious bugs in AspectJ compilers (cf. the discussion in Section 6). Nevertheless the AspectJ user community continues to make requests for a yet more expressive pattern language, only exacerbating the problem.

In this paper, we bridge the gap between existing operational semantics of advice and current practice, by giving a complete semantics of the static matching of AspectJ 1.2.1 pointcuts. In particular, our semantics can be used to reduce complex pointcuts to sets of labels that refer to source locations, and then the dynamic part of the story is told by an operational semantics in the style of [43].

Our semantics consists of a translation from AspectJ pointcuts into Datalog queries over relations defined in the object program. Datalog is a logic query language that originated in the theoretical database community [17]. We restrict ourselves to the *safe* fragment that has a straightforward least-fixpoint semantics; furthermore all safe Datalog queries terminate. The translation from pointcuts to Datalog takes the form of about 90 conditional rewrite rules. The full definition is thus quite short and elegant. It is available for download as an accompanying technical report [6].

This semantics is put to work in three ways. First, it serves as a crisp definition to discuss tricky points in the language design, and has enabled us to lay bare several long-standing bugs in AspectJ implementations. Second, the semantics is executable, and we present comparative experiments with an industrial-strength compiler to show the costs of directly executing the semantics are not prohibitive. Finally, our semantics provides a framework for the design and discussion of further language extensions that the AspectJ user community is clamouring for [7, 8, 11, 23].

It is not possible to prove a correspondence result with previous semantics, as the only existing definition of AspectJ is an informal description on the web [4]. However, our testing with respect to the standard implementation, and subsequent discussion of discrepancies with the AspectJ designers, provide ample confidence that our formal semantics captures the intended meaning.

Many previous works have suggested the use of logic programming for writing pointcuts in aspect-oriented programming, but invariably they use Prolog [16, 20, 29]. In the present setting, that would be inappropriate because the semantics of Prolog itself is quite complex, even with tabled resolution to give better termination behaviour. Furthermore, we tried to run our experiments with (a tabled variant of) Prolog, but found that execution times prohibit its application in practice.

In summary, this paper makes the following contributions:

- The identification of safe Datalog as a suitable intermediate form for pointcuts in aspect-oriented programming.
- The use of term rewriting to reduce complex pattern-based pointcuts to Datalog queries.
- The first rigorous semantics of the AspectJ 1.2.1 pointcut language.
- Experimental evidence that it is feasible to directly execute our semantics, on AspectJ programs of up to 100KSLOC.

The structure of the paper is as follows. In Section 2, we provide a brief introduction to AspectJ, focussing on the pointcut language. We then proceed to discuss existing semantics for the aspect-oriented paradigm in Section 3. In particular, we enunciate the difference between *static* and *dynamic* pointcuts. Dynamic pointcuts refer to runtime properties such as the call stack — they are matched at runtime, and their semantics is by now well understood (*e.g.* [43, 45]). By contrast, static pointcuts are matched against the static structure of the program, and they are the focus of the present paper. Next, we provide a brief introduction to safe Datalog in Section 4. We then explain informally how static pointcuts can be mapped to Datalog queries in Section 5. The heart of the paper is Section 6, which shows how a simple set of rewrite rules suffices to translate AspectJ's static pointcuts into Datalog. Rather than presenting a shallow overview of the complete semantics, we detail the most complex issues in AspectJ's design where a rigorous approach is indispensable. In Section 7 it is demonstrated that this semantics directly leads to a viable implementation strategy. We then briefly speculate on the use of Datalog to directly express new forms of pointcut in Section 8. A brief roadmap of related work is provided in Section 9 before concluding in Section 10.

2 AspectJ pointcuts

AspectJ is a variant of Java, extended with aspect-oriented features [28]. These features allow a programmer to write a single piece of code that consistently affects the behaviour of multiple modules in a program.

The novel contribution of aspect-oriented languages, which was not present in previous work on class composition (*e.g.* [39]), is known as "pointcut and advice". A *pointcut* is a predicate over events that occur during the execution of a program. These events (called *joinpoints*) are composite — they have duration and may be nested. A piece of advice is a block of code that is executed when a pointcut matches a joinpoint. Advice can be run before the matched joinpoint, after it, or instead of it.

We will introduce the pointcut notation of AspectJ with the aid of an example, taken from the textbook by Laddad [30]. The task in hand is to automatically enforce the following requirement of the Swing GUI library:

> "Once a component is visible, the event-dispatching thread (sometimes called the AWT thread) is the only thread that can safely access or update the state of the realized component. The rule exempts certain methods, allowing them to be safely called from any thread." [30]

Laddad's solution works by intercepting calls to methods that would update the state of a component from the wrong thread, and then queueing them for execution in the event-dispatching thread. The pointcut routedMethods, shown in Figure 1, matches calls to just those methods that would violate the invariant. It relies on five other programmer-defined named pointcuts — each one is defined in terms of pointcut primitives, using the Boolean connectives *and* (&&), *or* (||), and *not* (!). Three built-in primitives are also used:

```
1  pointcut viewMethodCalls() :
2      call(* javax..JComponent+.*(..));
3
4  pointcut modelMethodCalls() :
5      call(* javax..*Model+.*(..))
6   || call(* javax.swing.text.Document+.*(..));
7
8  pointcut uiMethodCalls() :
9      viewMethodCalls() || modelMethodCalls();
10
11 pointcut threadSafeCalls() :
12     call(void JComponent.revalidate())
13  || call(void JComponent.repaint(..))
14  || call(void add*Listener(EventListener))
15  || call(void remove*Listener(EventListener));
16
17 pointcut excludedJoinpoints() :
18     threadSafeCalls()
19  || within(SwingThreadSafetyAspect)
20  || if(EventQueue.isDispatchThread());
21
22 pointcut routedMethods() :
23     uiMethodCalls() && !excludedJoinpoints();
```

Figure 1: Pointcuts for Swing thread-safety enforcement

if takes a boolean-valued Java expression as an argument; it matches a joinpoint if the expression evaluates to true before (or after, depending on the kind of advice) the joinpoint occurs.

within takes a pattern ranging over types as an argument; it matches any joinpoint that was caused by executing code lexically within a type that matches the pattern.

call also takes a pattern as an argument, which ranges over method or constructor signatures; it matches any call-joinpoint to a method or constructor that has a signature matching the pattern.

There are several wildcards used in AspectJ patterns. The first is $*$: it matches any series of characters that can appear in a Java identifier (not '.'). So, for example, Line 14 of Figure 1 matches call-joinpoints to any method with a name that starts with "add", ends with "Listener", and takes a single argument of type EventListener.

A + wildcard can only appear in a pattern that ranges over types. It means 'match any subtype'. It appears in Line 2 of Figure 1, which matches any call to a method on a type matching the pattern javax..JComponent or any subtype of such a type.

The wildcard .. matches any sequence of full-stops and Java identifiers that begins and ends with a full-stop. For example, javax..*Model matches "javax.swing.AbstractListModel", but it would not if the pattern were javax..Model.

Note that the .. wildcard has a special meaning when used in the formal parameter list of a method pattern, as seen at the end of Line 2 — in that case it matches an arbitrary number of parameters of arbitrary type.

3 Existing AOP semantics

There is a large amount of work on the semantics of aspect-oriented programming, *e.g.* [2, 3, 10, 13, 14, 24, 26, 27, 32, 40, 43, 45–47]. None of these works involve a pointcut language that approaches the complexity of the pointcut language of AspectJ. We contend, however, that it is possible to understand the semantics of AspectJ by following the framework introduced in [43].

```
1  public class X {
2     void f1(int n) { }
3     void f2(int n) { f1(n); }
4     void f3(int n) { f2(n); }
5
6     public static void main(String[] args) {
7       X x = new X();
8       x.f3(0);
9       x.f2(1);
10    }
11 }
12
13 aspect A {
14    pointcut a (int n) :
15      call(* f*(..))
16     && args(n)
17     && cflowbelow(execution(* f3(..)));
18
19    before(int n) : a(n) {
20     System.out.println(thisJoinPoint + ":␣n="+n);
21    }
22 }
```

Figure 2: Example AspectJ program.

A key idea of [43] is to make the semantics a two-step process, involving a surface language and a core language. The surface language in our case is AspectJ. It has the rich notation for pointcuts that allows programmers to alter the behaviour of the mainline program without modifying the program text directly. That property is sometimes called *obliviousness* in the literature on aspect-oriented programming. The core language, by contrast, augments the mainline program with explicit labelled instrumentation points. Pointcuts refer directly to sets of such instrumentation points. The operational semantics of the core language observes the execution of labelled instrumentation points and executes advice where labels in pointcuts match labels at runtime.

In most of the papers that have built on [43], the translation from surface to core is quite simple [13, 14]. Wang *et al.* consider a slightly more complex translation, which involves doing more of the instrumentation at compile-time [46, 47]. The current paper continues that trend by offering a translation from the AspectJ surface language to a suitable core language, replacing the rich pointcuts by labelled instrumentation points. We do not offer an operational semantics with it, as that would require augmenting a complete operational semantics of Java.

In the literature on compiling aspect-oriented programs [22, 33], instrumentation points in the static program are called *shadows*, whereas their runtime counterparts are named *joinpoints*. We shall follow that terminology below.

To illustrate, consider the AspectJ program in Figure 2. It contains a pointcut definition (Lines 14–17), and a piece of advice (extra code) that is triggered by that pointcut (Lines 19–21). The pointcut a(n) makes use of the **cflowbelow**(p) primitive: conceptually this matches if a joinpoint is properly nested inside any joinpoint that matches p. Pointcut primitives such as **cflowbelow** are inherently dynamic, and in general they cannot be resolved by static matching of shadows (in the example, the call to f1 from f2 is advised or not depending on whether f2 was called from f3). The same holds true for **args**, which exposes the actual value of an argument at runtime. Apart from these two primitives, the pointcuts are entirely static and could thus be replaced by sets of shadows in the program. This is illustrated in the translated program in Figure 3. Here the mainline program has been augmented with explicit labels for shadows (there are actually more shadows than shown here, for instance for class initialisation). Accordingly, the static pointcuts have been replaced by sets of labels.

The purpose of the remainder of the paper is to pin down the process by which AspectJ pointcuts are

```
public class X {
    void f1(int n) { L1 : {} }
    void f2(int n) { L2 : { L3 : {f1(n);} } }
    void f3(int n) { L4 : { L5 : {f2(n);} } }

    public static void main(String[] args) {
      L6 : { X x = (L7 : {new X()});
             L8 : {x.f3(0)};
             L9 : {x.f2(1)}; }
    }
}

aspect A {
    pointcut a (int n) :
       label(L3,L5,L8,L9)
    && args(n)
    && cflowbelow(label(L4));

    before(int n) : a(n) {
     System.out.println(thisJoinPoint + ":␣n="+n);
    }
}
```

Figure 3: Translated program.

reduced to this form. We shall show how each static pointcut is defined in terms of a relational query; by running those queries on a mainline program, one obtains the sets of labels as in Figure 3.

4 Datalog

We shall use the Datalog query language to express the semantics of AspectJ pointcuts. Datalog is similar to Prolog, and syntactically is a subset of Prolog, but excludes the ability to construct new data type values such as lists. While we give a brief introduction to Datalog, we refer to the reader to [17] for more details. A Datalog program is a set of clauses (backward implications) of the form:

$$p(X_1, \ldots, X_n) \leftarrow q_1(Y_1, \ldots, Y_{m_1}), \ldots, q_k(Y_1, \ldots, Y_{m_k}).$$

where each X_i is a variable, and each Y_j is either a variable or a constant. Each q_j is a positive or negated occurrence of either a predicate or a *test* such as $X < Y$. A variable occurs *positively* in a clause if it occurs in a positive predicate on the right-hand side of the clause, but not if it only occurs in a test. Intuitively, a test such as $X < Y$ cannot be used to generate values of X and Y making the test true, unlike a predicate $p(X, Y)$.

The semantics of Datalog programs, at least in the absence of negation, are straightforward. Each predicate $p(X_1, \ldots, X_n)$ defines an n-ary relation, and clauses are interpreted as inclusions between relations. The meaning of the program is then the least solution of this set of inclusions. For instance, the Datalog program $p(X) \leftarrow p(X)$, while non-terminating as a Prolog program, is a *bona fide* definition of the empty relation in Datalog.

4.1 Safe Datalog

The use of negation in Datalog programs is more problematic, as negation is not a monotonic operator, and so the fixpoint need not exist. Concretely, a program such as $p(X) \leftarrow \neg p(X)$ does not define a relation p, and indeed $p(X)$ is neither true nor false for any X. *Safe* Datalog is a subset of Datalog that provides a sufficient (but not necessary) condition that guarantees that every program can be evaluated to a set of relations. Safe Datalog imposes two conditions: range restriction and stratification.

Range Restriction In a *range-restricted* Datalog program, each variable in the head (*i.e.* left-hand side) of a clause must appear positively on the right-hand side. Furthermore, each variable on the right-hand side must appear positively at least once. This restriction rules out programs such as $p(X,Y) \leftarrow q(X)$, as Y is left unconstrained. Programs such as:

$$r(X) \leftarrow \neg q(X), \text{regexpmatch}(\text{“a.*”}, X).$$

where $\text{regexpmatch}(P,X)$ is a regular expression pattern matching test, are likewise disallowed. Both the above queries are undesirable as the relations defined cannot directly be computed: the $p(X,Y)$ relation may be infinite (any value of Y can be used), while evaluating the $r(X)$ relation may require evaluating infinitely many regular expression matches.

Stratification Furthermore, in a *stratified* Datalog program, negation may not be used in recursive cycles. A program is stratified if there is some strict partial order $<$ on predicates such that whenever p depends negatively on q, then $p > q$. That is, a predicate may never depend negatively on itself. This prohibits such programs as $p(X) \leftarrow q(X), \neg p(X)$.

Any safe Datalog program defines a set of relations as the least fixpoint of the recursive inclusions in the program. Furthermore, this solution may be effectively computed, and efficient algorithms are known for evaluating safe Datalog programs. Finally, all relations evaluated are finite, provided the *primitive* predicates (undefined predicates providing access to the database) denote finite relations.

These properties of safe Datalog are highly desirable in our setting. First, Datalog has a clear and straightforward semantics, unlike Prolog, in which the operational and declarative semantics do not coincide. This guarantees that defining the semantics of AspectJ pointcuts by translation to Datalog is valid. Beyond pure semantics, the efficiency of Datalog allows our translated AspectJ pointcuts to be evaluated — leading to a directly implementable semantics.

4.2 Extensions

For convenience, we shall make use of a number of extensions to pure Datalog. These are just syntactic sugar, and may be eliminated in a translation back to pure Datalog (which we omit for space reasons).

- We use a variant of Datalog in which each variable is annotated with a *type*. In any clause, the type of the variables defined in the head are given explicitly, as follows:

  $$p(X_1 : p_1, \ldots, X_n : p_n) \leftarrow E.$$

  where the p_i are predicates and E is any Datalog expression. This is equivalent to the untyped clause:

  $$p(X_1, \ldots, X_n) \leftarrow p_1(X_1), \ldots, p_n(X_n), E.$$

  Furthermore, we insist that any free variable appearing on the right-hand side be introduced by an existential quantifier, again giving its type. We use the syntax `X : p ^ E` to represent the existential quantification $\exists X (p(X) \wedge E)$. A typed Datalog program is necessarily range-restricted.

- Datalog expressions can use negation arbitrarily, so that $\textbf{not}(E)$ is an expression whenever E is.

- We allow the use of disjunction, represented by a semicolon.

5 Pointcuts are queries

We now aim to show how pointcuts in AspectJ can be regarded as Datalog queries over a relational representation of the program. The correspondence presented here is informal, and it is only intended to help the reader build an intuition before diving into the formal details in the next section.

Consider the example translation from Figure 2 to Figure 3. The program of Figure 2 is stored as a set of primitive relations. That set includes, for example, a relation for recording method declarations:

methodDecl(MethodId,Name,Sig,DeclaringType,ReturnType).

The first field is the *identifier*: this can be thought of as the identity of the corresponding node in the abstract syntax tree. The primitive relations also record *shadows* in the program: these are the labels shown in Figure 3. For instance, we have a relation

callShadow(ShadowId,Method,RecvType)

This relates the identity of a shadow (labelled instrumentation points like L1, L2 in Figure 3) to a method called at that shadow, and the static type of the receiver. Furthermore, there is an extensional relation that records method bodies, as these are also joinpoints in AspectJ:

executionShadow(ShadowId,Method)

We are now ready to express the static pointcuts of Figure 2 as Datalog predicates. We first consider

call(∗ f∗(..))

This pointcut corresponds to the following Datalog predicate:

```
pc1(S : shadow) :− M : method ˆ N : name ˆ
                   callShadow(S,M,_),
                   methodDecl(M,N,_,_,_),
                   regexpmatch('f.∗',N) .
```

Evaluating pc1 will yield the following set of solutions for S: { L3,L5,L8,L9 } – precisely the translation in Figure 3.

The other static pointcut in Figure 2 is

execution(∗ f3(..))

It is easy to give a naïve translation into Datalog, namely

```
pc2(S : shadow) :− M : method ˆ N : name ˆ
                   executionShadow(S,M),
                   methodDecl(M,N,_,_,_),
                   regexpmatch('f3',N) .
```

This time evaluation yields only one solution, namely S: { L4 }.

In order to extend this intuitive correspondence to a full formal semantics, we need to decide exactly on the set of primitive relations. Furthermore, the above translation is naïve, because in fact we need to take into account where a pointcut is declared, as the context determines how names in the pointcut are resolved. For that reason, in the formal semantics, pc1 and pc2 would need to take an additional parameter.

6 Semantics of static pointcuts

6.1 Overall structure

As described above, our goal is to determine, for each static pointcut, which set of labelled instrumentation points it denotes, as this will pin down its semantics. We achieve this by giving a set of rewrite rules that translate the static pointcuts in AspectJ to Datalog predicates. The resulting predicate corresponding to a given pointcut has two free variables — the first denotes the Java type in which the pointcut is being evaluated (this parameter is used to handle name lookup), and the second ranges over shadow labels; the values of that variable making the predicate true are precisely those labels which the pointcut denotes.

The rewriting rules are split up into contexts such as pointcuts, method patterns and name patterns. For each such context we introduce a different term constructor (**aj2dl** for pointcuts, **methconstrpat2dl** for method patterns, *etc.*), and the purpose of the rewriting process is to eliminate these constructors. When they have all been eliminated, the translation process is complete.

In our rules we adopt the conventions that both left- and right-hand side of the rewrite rule are enclosed in brackets ([..]) to make reading easier. Identifiers shown in **bold font** are term constructors, identifiers in *italics* are metavariables that capture subexpressions of the current term.

[**aj2dl**(*pc1* && *pc2*, C, S)] → [(**aj2dl**(*pc1*, C, S), **aj2dl**(*pc2*, C, S))]
[**aj2dl**(*pc1* || *pc2*, C, S)] → [(**aj2dl**(*pc1*, C, S)); (**aj2dl**(*pc2*, C, S))]
[**aj2dl**(!*pc*, C, S)] → [**not**(**aj2dl**(*pc*, C, S))]

Figure 4: Rewrite rules for Boolean combinations of pointcuts

The **aj2dl** constructor is used to rewrite an AspectJ pointcut to Datalog. More precisely, **aj2dl**(*pc*, C, S) should be interpreted as a Datalog expression with free variables C and S, such that the expression is true iff S is a shadow (instrumentation label) in the denotation of *pc*, and C is the class in which the pointcut *pc* is located. The context information provided by the class parameter C is necessary, as the class in which a pointcut is located affects its semantics (through the use of Java name lookup in pointcuts).

An expression of the form **aj2dl**(*pc*, C, S) is rewritten to a pure Datalog expression, in a syntax-directed fashion. The rules in Figure 4 show how logical operators may be eliminated from pointcuts, and converted into the equivalent Datalog logical operators.

6.2 Primitive predicates

In order to express pointcuts in Datalog, a set of primitive predicates (also referred to as *extensional predicates* in the deductive databases literature) must be supplied to query the structure of the program. The set of primitive predicates must at least encode as much of that structure as is required to evaluate AspectJ pointcuts. An extreme viewpoint would be to just store the abstract syntax tree of the mainline program, and write queries over that structure. However, we shall need quite complex derived notions, such as the type hierarchy (represented by a relation hasSubtype). While this information could be defined purely in terms of the syntax of the program, it would clutter our semantics of pointcuts to do so. We therefore abstract away from this irrelevant detail, and use the set of primitive predicates in Figure 5. This set captures just enough information about the structure of the program to evaluate AspectJ pointcuts.

While the use of Datalog usually allows a simple and direct expression of queries, our treatment of method parameters shows that an encoding may sometimes be necessary. The methodParamTypes predicate is used to obtain, for each method, the list of types of formal parameters. As Datalog does not allow the use of data structures such as lists, or indeed arithmetic, this cannot be expressed directly. Instead, we define a relation:

methodParamTypes(Method, Type, Pos, NextPos)

that holds if the formal parameter of Method at position Pos has type Type. The NextPos field records the position of the *next* parameter of M (*i.e.* Pos + 1), or 0 if there is no next parameter. This field is needed because arbitrary arithmetic is not available in Datalog, and is used to iterate over parameter types.

In addition to the primitive database predicates describing the structure of the program, we include predicates listing the *shadows* in the program. Shadows represent the static instrumentation points recognised by the AspectJ language; as such, pointcuts denote sets of shadows. Again, because our focus is on the matching behaviour of pointcuts, we have chosen to represent shadows directly as primitive predicates. In the terms of Section 3, this amounts to abstracting from the details of inserting labels at every instrumentation point of the mainline program. Figure 6 lists the relevant primitive predicates. Each of these corresponds to a kind of shadow defined by the AspectJ language — for instance, the callShadow predicate describes method or constructor call shadows. The type stored for each call shadow should be interpreted as the receiver type for virtual method calls, while for static method calls and constructor calls this is just the declaring type of the callee.

It is worth noting that no part of the matching semantics is preempted by these predicates. Picking out all call shadows, for example, is a simple mechanical task that can be achieved by case analysis on the program AST: we just have to collect all method calls. We are concerned with the matching of pointcuts, *i.e.* the process by which the set of all method calls is constrained to just those which are matched by a

Predicate	Description
packageDecl(P, N)	P denotes a package with name N.
typeDecl(T, N, IsInt, P)	T denotes a type with name N, declared in package P. IsInt is true if T is an interface.
primitiveDecl(T, N)	T denotes a primitive type with name N.
arrayDecl(T, ET, N)	T denotes an array type with element type ET and name N.
methodDecl(M, N, S, DT, RT)	M denotes a method with name N, signature S, return type RT and declared in type DT.
constructorDecl(C, S, Cls)	C denotes a constructor with signature S for class Cls.
fieldDecl(F, DT, T, N)	F denotes a field with name N, of type T, declared in type DT.
compilationUnit(CU, P)	CU denotes a compilation unit in package P.
singleImportDecl(I, N)	I denotes an import declaration, importing the type with name N.
onDemandImportDecl(I, N)	I is an on-demand import declaration, for all types in the type or package with name N.
methodModifiers(M, Mod)	Method M has the modifier Mod.
fieldModifiers(F, Mod)	Field F has the modifier Mod.
modifiers(Mod, N)	Modifier Mod has string representation N.
methodThrows(M, T)	Method M declares throwing exception T
methodParamTypes(M, T, Pos, Next)	Method M has a parameter of type T at position Pos. Next is the next position after Pos.
hasChild(A, B)	Syntactic element B is a directly lexically enclosed by A.
hasSubtype(T1, T2)	T2 is a direct subtype of T1.

Figure 5: Primitive Predicates: Program Structure

given pointcut. We can think of the predicates in Figure 6 simply as a way of bounding the domain of the Datalog variable S.

Predicate	Description
callShadow(S, M, Recv)	Call to a method or constructor M with receiver type Recv.
executionShadow(S, M)	Execution of a method M.
initializationShadow(S, C)	Initialisation of an object (body of C after parent constructor calls).
preinitializationShadow(S, C)	Pre-initialisation of an object (body of C before parent constructor calls).
staticinitializationShadow(S, T)	Initialisation of the static members of a class T.
getShadow(S, F, Recv)	Read access to a field F, on an object of static type Recv.
setShadow(S, F, Recv)	Write access to a field F, on an object of static type Recv.
handlerShadow(S, Exn)	Execution of a handler for exception Exn.
adviceexecutionShadow(S)	Execution of advice.

isWithinClass(S, Cls)	Shadow S is contained in class Cls.
isWithinShadow(S1, S2)	Shadow S1 is contained within shadow S2.

Figure 6: Primitive Predicates: Shadows

6.3 Pre-defined derived predicates

Below we shall make use of some pre-defined derived predicates (also called *intensional predicates* in the deductive databases literature), as a convenient shorthand in defining the semantics of pointcuts.

The simplest examples are those predicates that are used as types, such as constructor, method, field, type. Most of these are self-explanatory, but there are some exceptions: callable (M) holds when M is a method or a constructor; similarly packageOrType(T) is the union of the package and type predicates.

Other pre-defined predicates include hasName(X,N), which is true when X is an entity (method, type, package, ...) that has name N. All of these are obtained via simple projections of the primitive relations.

A more complex class of pre-defined predicates are those used for traversing hierarchical data. A typical example is the reflexive transitive closure of the immediate hasSubtype relation:

```
hasSubtypeStar(T : type,T : type).
hasSubtypeStar(T : type,S : type) ←
  U : type ˆ ( hasSubtype(T,U), hasSubtypeStar(U,S) ).
```

The final category of pre-defined predicates concerns the lookup of type names in Java. The most important of these is predicate simpleTypeLookup(C,N,T). It relates a type C, a name N and a type T precisely when inside C, looking up a type by name N would result in T according to the Java Language Specification. Furthermore N is assumed to be a simple name, not containing dots.

For space reasons, we do not include a list of the pre-defined derived predicates in this paper. The companion technical report contains the details of these predicates.

6.4 Rewrite rules

As explained above, we aim to give a semantics to the AspectJ pointcut language by rewriting it to Datalog in a term-based, purely syntactic fashion. The complete set of rewrite rules for doing that consists of about 90 rules. While it is pleasing that so few rules suffice to pin down the whole pointcut language, space forbids a thorough description of all rules in this paper (full details can be found in the companion report [6]). Rather than give a cursory overview of all rules, we present an in-depth discussion of two particular features of the AspectJ pointcut language. The specific choice of constructs we describe is significant, as we focus on language features that have been a source both of confusion among AspectJ users and implementation bugs. This illustrates the necessity for a precise semantics to clarify the pointcut language.

In developing our semantics, we felt it was important that the rewrite rules are directly executable, so they can be easily tested on tricky examples. At first we developed the rules by directly rewriting abstract syntax trees that represent pointcuts, but in this form the rules quickly became unreadable. We therefore implemented them using the Stratego system of Visser *et al.*; its main attraction is that rewrite rules can be specified in a concrete syntax [41] — in our case, concrete AspectJ pattern syntax on the left-hand side of rules, and concrete Datalog syntax on the right-hand side. Furthermore, Bravenboer *et al.* have developed a grammar for AspectJ that we adopted for this project [9]. The rules shown in this paper are slight typographical modifications of their implementation in Stratego.

6.4.1 Call and execution pointcuts

AspectJ offers two ways to intercept method invocations: one for intercepting at the call site, and another for intercepting the execution of a method body in the defining class. For the first alternative, one uses the **call** pointcut, and for the second **execution**. The seemingly different matching behaviours of **call** and **execution** have led to considerable confusion, and a comprehensive discussion can be found in [7] (although it is now somewhat dated, since it deals with AspectJ 1.1.1 and the language semantics has evolved). To illustrate the difference between **call** and **execution**, consider the following type hierarchy:

```
class A { void m() {} }
class B extends A {}
class C extends B { void m() {} }
```

along with the pointcuts

```
pointcut c() : call(∗ B.m(..));
pointcut e() : execution(∗ B.m(..));
```

and the sequence of calls

(**new** A()).m(); (**new** B()).m(); (**new** C()).m();

The call pointcut c() will match the latter two calls, whereas the execution pointcut e() only matches the definition of m() in C. The situation is further complicated in the presence of static methods: if both definitions of m() are declared static in the above example, c() only matches the second call, and e() does not match at all.

This potentially surprising matching behaviour is caused by the difference between call and execution shadows. Any call statement gives rise to a call shadow, irrespective of where the called method is actually declared. An execution shadow, on the other hand, spans an entire method body, and as such by definition can only be present in those classes that contain a (re-)definition of the method. In the example above, the different behaviour is caused by B having no definition of m(), not any difference in the interpretation of the pattern.

The precise semantics is easily brought out by writing both pointcuts c() and e() as Datalog queries. We shall first do that by hand, to give the reader a sense of what needs to be generated; once the goal is clear, we shall present the rewrite rules to achieve it. Consider first the call pointcut c():

```
c(Context : type, Shadow : shadow) ←
  M : method ^ Recv : type ^ M_Real : method
   ( matchpat(Context,M,Recv) ,
     overrides (M_Real,M) ,
     callShadow(Shadow,M_Real,Recv) ).

matchpat(Context : type, M : method, Recv : type) ←
  T : type ^ MD : type ^
   ( simpleTypeLookup(Context,'B',T),
     hasSubtypeStar(T,Recv),
     methodDecl(M,'m',_,MD,_),
     ( ( hasStrModifier(M,static),
         equals(MD,T) )
       ;
       ( not(hasStrModifier(M,static)),
         hasSubtypeStar(MD,T) ) ) ) .
```

In words, this query consists of two parts: one that matches the method pattern to find a candidate method M with a matching signature, and another that picks out an appropriate call shadow Shadow.

The subsidiary predicate matchpat first finds the definition T of a type named 'B', as well as a method M named 'm' that is declared in some type MD. In order to correctly combine these two results, matchpat must ensure that T has an implementation of M. According to the AspectJ rules, "has an implementation of" means "declares, overrides, or inherits" for virtual methods; for static methods it just means "declares". Therefore, if the method is static, MD and T must be equal; if the method is not static, T can also be a subtype of MD. Note that this picks out all more general methods m of B.m() and their declaring classes MD — they are our representation of the set of signatures that the method pattern matches.

The parameter Recv of matchpat is intended to bind the static receiver type of the shadow — that is, the static type of the receiver for virtual calls, and the declaring type for static calls. At this stage, the only requirement is that Recv is T or a subtype of T, the type mentioned in the pointcut.

The second part of the query is simple in comparison: We find all methods M_Real overriding the more general definition M that we retrieved in matchpat, and pick out any call shadows that refer to M_Real and have a suitable static receiver type Recv. It is worth stressing that the overrides () predicate we use is reflexive, as a method could be its own most general definition.

Now contrast the above query c with the Datalog definition of the **execution** pointcut:

```
e(Context : type, Shadow : shadow) ←
  M : method ^ Recv : type ^ M_Real : method ^
   ( matchpat(Context,M,Recv),
     overrides (M_Real,M),
     hasChild(Recv,M_Real),        // crucial  difference
```

executionShadow(Shadow,M_Real)) .

It is remarkably similar to our earlier pointcut for **execution**: Again, we find overriding methods M_Real for the result of matchpat, but now we also assert that the receiver type of the shadow (which, for execution pointcuts, is just the type containing the method body) contains a definition of M_Real; if it didn't, there would be no execution shadow. Finally, we pick out all shadows for M_Real as matched by the pointcut.

[**aj2dl**(call(*methconstrpat*), C, S)] → [X : callable ˆ Y : callable ˆ R : type ˆ
(**methconstrpat2dl**(*methconstrpat*, C, R, X),
overrides (Y, X), callShadow(S,Y,R))]

[**aj2dl**(execution(*methconstrpat*), C, S)] → [X : callable ˆ Y : callable ˆ R : type ˆ
(**methconstrpat2dl**(*methconstrpat*, C, R, X),
overrides (Y, X), hasChild(R, Y),
executionShadow(S,Y))
]

Figure 7: Top-level rewrite rules for call and execution pointcuts

We can now conclude that the seemingly different matching behaviour is based purely on the difference between call and execution shadows; the method pattern in either case is matched in exactly the same way, but execution shadows only arise if a class declares a method, whereas for a call shadow it is sufficient to simply inherit it.

We are ready, therefore, to present the rewrite rules that give the semantics of **call** and **execution** in full generality. The rules (Figure 7) follow the argument above: one rule picks out a callShadow, and the other an executionShadow (with the side condition that there needs to be an actual declaration of the method in the receiver), but the method or constructor pattern is treated uniformly.

That pattern is further rewritten using the **methconstrpat2dl** term constructor. Rewrite rules are given in Figure 8 (those shown are for method patterns — constructor patterns are very similar). We see that the method pattern is further broken down into its constituent parts: a modifier pattern, a type pattern for the return type, a pattern matching class members, and a formals-pattern (*i.e.* a pattern over the method's formal parameters), as well as an optional pattern on the **throws** clause of the method. Each of these is rewritten to Datalog using the appropriate term constructor, and the resulting Datalog expressions are conjoined in the natural way.

[**methconstrpat2dl**(*mmodpat typepat membpat*(*formalspat*), C, R, X)]
→ [T : type ˆ(**mmodpat2dl**([*mmodpat*], X), **typepat2dl**(*typepat*, C, T),
methmembpat2dl(*membpat*, C, R, X), **formals2dl**([*formalspat*], C, X, 1), returns(X, T))]

[**methconstrpat2dl**(*mmodpat typepat membpat*(*formalspat*) throws *throwspat*, C, R, X)]
→ [**methconstrpat2dl**(*mmodpat typepat membpat*(*formalspat*), C, R, X), **throws2dl**(*throwspat*, C, X)]

Figure 8: Rewriting for method patterns — constructor patterns are similar

We refrain from listing all the auxiliary term constructor rewrite rules here, the interested reader may find our full set of rules in our technical report [6]. For now, we just note that out of the additional constructors, **mmodpat2dl** and **formals2dl** are implemented in the natural fashion, *i.e.* they define a set of predicates asserting that the method has a certain modifier (or a certain formal parameter), recursively working through the list of modifier (or formal parameter) patterns. **typepat2dl** and **methmembpat2dl** ultimately rely on name patterns, and the implementation of those proved unexpectedly challenging. We discuss it in Section 6.4.2.

Thus, the last remaining term constructor is **throws2dl**, and there is a subtlety that can easily be missed when perusing the AspectJ documentation, but which appears clearly in our semantics. The rewrite rules are given in Figure 9. They are an illustration of the kind of recursive rewriting that is also employed with **mmodpat2dl** and **formals2dl**. An empty exception pattern is rewritten to a dummy predicate that is later eliminated. In the remaining two clauses, when rewriting exception patterns we differentiate between the case when the first character of the pattern is a '!', and when it is not. According to the AspectJ documentation, in the former case the pattern matches if none of the declared exceptions of a method match the rest of the pattern, and in the latter case we match if there is *some* exception in the throws clause that matches the pattern. So ! Ex is an exception pattern that starts with a '!', whereas '!' in the (! Ex) pattern is a part of the class name pattern and not of the exception pattern and therefore the third clause of the **throws2dl** rule will be applied.

While the authors are of the opinion that it is bad language design to put substantial meaning into parentheses (the throws pattern ! Ex has a different meaning from (! Ex), as shown above), it poses no problems for our method of pinning down the semantics. We hope that by giving a crisp semantics, issues such as these will become easier to discuss and rectify.

$$[\![\ \textbf{throws2dl}([], \mathrm{C}, \mathrm{M})\ ]\!] \rightarrow [\![\ \mathrm{true}(\mathrm{M})\ ]\!]$$

$$[\![\ \textbf{throws2dl}([!\textit{classnamepat}, \textit{throwspat}], \mathrm{C}, \mathrm{M})\ ]\!] \rightarrow [\![\ \textbf{not}(\mathrm{E} : \mathrm{type}\ \hat{}(\textbf{classnamepat2dl}(\textit{classnamepat}, \mathrm{C}, \mathrm{E}), \mathrm{throwsException}(\mathrm{M}, \mathrm{E}))), \textbf{throws2dl}(\textit{throwspat}, \mathrm{C}, \mathrm{M})\ ]\!]$$

$$[\![\ \textbf{throws2dl}([\textit{classnamepat}, \textit{throwspat}], \mathrm{C}, \mathrm{M})\ ]\!] \rightarrow [\![\ \mathrm{E} : \mathrm{type}\ \hat{}(\textbf{classnamepat2dl}(\textit{classnamepat}, \mathrm{C}, \mathrm{E}), \mathrm{throwsException}(\mathrm{M}, \mathrm{E})), \textbf{throws2dl}(\textit{throwspat}, \mathrm{C}, \mathrm{M})\ ]\!]$$

Figure 9: Rewrite rules for **throws2dl**

6.4.2 Type name patterns

Type name patterns in AspectJ pointcuts allow sets of types to be concisely denoted, and are invaluable in expressing many commonly-used pointcuts. However, the semantics of type name patterns is surprisingly subtle, and as such it is useful to examine it more closely.

Type patterns allow the use of wildcards to denote sets of types. Examples of type patterns include: Foo, a*..B (read: any name starting with a, followed by a dot-separated list of identifiers, followed by B) and *. While it is quite clear what set of *strings* is represented by a name pattern, the subtlety lies in the interpretation of these as names. This has been a source of confusion in the AspectJ community, and indeed the *ajc* [4] and *abc* [1] compilers for AspectJ do not implement the same rules. Furthermore, this is the cause of a substantial bug in the reference AspectJ implementation [25].

To see how this issue arises, consider a name pattern, for definiteness say a*..B. There are two possible interpretations of this pattern. The first possibility is that this should range over *fully qualified* names. In this case, classes a.B and aa.c.B would be matched, but not d.a.B (we use the convention that lowercase identifiers denote packages).

Alternatively, the name pattern might denote any class that can be referred to by a name of the form a*..B *from the aspect*. That is, in this view the normal Java name lookup rules are used to interpret names. This matches any type matched in the previous way (any type can be referred to by its fully qualified name), but may match *more* types. For instance, the simple pattern B would match any class B in the same package as the aspect.

In fact, the semantics of AspectJ use either form of matching, depending on context. The key is the presence or absence of the *wildcards* * and .. in the pattern. If a wildcard appears anywhere in the pattern, then the pattern should be interpreted as ranging over fully qualified names, and no name lookup is performed. However, if there are no wildcards in the pattern, then it is interpreted as a Java name, and denotes whichever types (if any) can be referred to by the given name from the context of the aspect. Finally, the name pattern * (a single asterisk) is treated as a special case. This matches *any* type, including anonymous types such as anonymous inner classes, which otherwise would be matched by no name pattern.

We shall now give the semantics of type name patterns in more detail.

Type patterns in AspectJ are built up from the following. A *simple name pattern*, representing an identifier, is just a literal possibly containing the wildcard $*$. A general name pattern is a sequence of simple name patterns, separated by either . or .., where the latter is a wildcard denoting any sequence of Java identifiers and full-stops beginning and ending with a full-stop. Given any name pattern pat, both pat and pat+ (representing any subtype of pat) are type name patterns, and type name patterns are closed under logical connectives. Finally, given a type name pattern tpat, tpat[] is a type name pattern representing an array with element type matching tpat.

Type name patterns are translated using the **typepat2dl** term constructor. The top-level rules (excluding logical connectives) are shown in Figure 10. The name pattern is translated *via* the **namepat2dl**, and the appropriate type (exact match, subtype or array type) is selected.

[**typepat2dl**(*namepat*, C, T)]→[**namepat2dl**(*namepat*, C, T), type(T)]
[**typepat2dl**(*namepat*+, C, T)]→[T1 : type ˆ(**namepat2dl**(*namepat*, C, T1), hasSubtypeStar(T1, T))]
[**typepat2dl**(*typepat*[], C, T)] → [T1 : type ˆ(**typepat2dl**(*typepat*, C, T1), arrayDecl(T, T1, _))]

Figure 10: Type Patterns: Subtypes and Array Types

The **namepat2dl** rewrite context defines the semantics of name patterns. As discussed above, the AspectJ semantics separates out the cases of name patterns containing wildcards from exact name patterns. This is achieved in **namepat2dl** (Figure 11) using an auxiliary test contains−wildcard that checks the presence of wildcards in a pattern. Note that the **namepat2dl** predicates match by name only, and in particular can match either packages or types. The end result is constrained to a type in **typepat2dl**.

[**namepat2dl**($*$, C, T)]→[true(T)]
[**namepat2dl**(*namepat*, C, T)]→[**wcnamepat2dl**(*namepat*, T)]
  where contains−wildcard(*namepat*) ; fqname(*namepat*) $\neq$"$*$"
[**namepat2dl**(*namepat*, C, T)]→[**exactnamepat2dl**(*namepat*, C, T)]
  where not(contains−wildcard(*namepat*)); fqname(*namepat*) $\neq$"$*$"

Figure 11: Name Patterns: Testing for Wildcards

The semantics of simple name patterns and wildcard name patterns are defined in Figure 12. Such a name pattern is matched (including wildcards) to the fully qualified name of an element.

Finally, the rules for wildcard-free name patterns are shown in Figure 13. These are rather more involved, reflecting the complexity of name lookup in Java. Two predicates are defined: **exactnamepat2dl** matches a name pattern against type or package names, while **pnamepat2dl** matches package names only. In accordance with the Java specification, when a name is looked up in a context where it is unknown whether a package or type is denoted, the name should be interpreted as a type whenever possible, and only interpreted as a package if no type of that name exists. Finally, name lookup for simple names is performed by the simpleTypeLookup predicate discussed earlier.

7 Experimental results

The Datalog queries we produce in the semantics are executable, and so it is natural to ask whether that semantics can be used directly as the basis of an AspectJ implementation. In this section, we seek to determine whether that is at all feasible in practice, by running queries that correspond to some realistic pointcuts on sizable mainline programs.

There are a number of options for executing Datalog queries. Perhaps the most obvious one is to capitalise on the fact that Datalog is a subset of Prolog, so we could just use a Prolog interpreter. However, to stay true to the declarative semantics of Datalog, such an interpreter must use *tabled resolution*, because otherwise non-termination might occur. That consideration suggests the use of XSB, the leading optimising,

[**snamepat2dl**(∗, S)]→[true(S)]
[**snamepat2dl**(*snamepat*, S)]→[N : name ˆ(hasName(S, N), regexpmatch(*snamepat*, N))]
  where *snamepat* ≠"∗"

[**wcnamepat2dl**(*snamepat*, T)]→[P : package ˆ (defaultPackage(P), hasChild(P, T),
    snamepat2dl(*snamepat*, T))]
[**wcnamepat2dl**(namepat.snamepat, T)]→[E : packageOrType ˆ(**wcnamepat2dl**(*namepat*, E),
    hasChild(E, T), **snamepat2dl**(*snamepat*,
    T))]
[**wcnamepat2dl**(namepat..snamepat, T)]→[E : packageOrType ˆ(**wcnamepat2dl**(*namepat*, E),
    hasChildPlus(E, T),
    snamepat2dl(*snamepat*, T))]

Figure 12: Name Patterns: Simple Name Patterns and Wildcards

[**exactnamepat2dl**(*snamepat*, C, T)]→[
    (simpleTypeLookup(C, *snamepat*, T));
    (**not**(T1 : type ˆ(simpleTypeLookup(C, *str*, T1))), **pnamepat2dl**(*snamepat*, T))]
[**exactnamepat2dl**(namepat.snamepat, C, T)]→[
    Pot : packageOrType ˆ((**exactnamepat2dl**(*namepat*, C, Pot), type(T), hasChild(Pot, T),
        hasName(T, *snamepat*)) ;
    (**not**(T1 : typeˆ(**exactnamepat2dl**(*namepat*, C, Pot), hasChild(Pot, T1), hasName(T1,
        snamepat))),
        pnamepat2dl(namepat.snamepat,
        T)))]

[**pnamepat2dl**(*snamepat*, P)]→[package(P), hasName(P, *snamepat*), topLevelPackage(P)]
[**pnamepat2dl**(namepat.snamepat, P)]→[P1 : package ˆ (**pnamepat2dl**(*namepat*, P1), hasChild(P1,P),
    package(P), hasName(P, *snamepat*))]

Figure 13: Name Patterns: Exact Name Patterns

tabled implementation of Prolog [38]. However, in a few preliminary experiments we found that the queries corresponding to pointcuts can take several hours to complete. Moreover, it took a considerable amount of optimisation done by hand to reduce XSB run times to such a level. Clearly the point of directly executing the semantics is lost if hand optimisation is required.

We therefore decided to use CodeQuest [21], our own implementation of Datalog based on a relational database system. CodeQuest first optimises Datalog rules to reduce the number of unnecessary computations (applying a specialised version of the well-known *magic sets* transformation). It then translates the optimised Datalog queries into SQL to take the advantage of several decades of research on RDBMS optimisations. For the measurements reported below, we used Microsoft SQL Server Express 2005 as the RDBMS backend of the CodeQuest system.

For each of the experiments, a customised version of the *abc* [5] compiler collects the facts implied by the program's structure, which are used to populate CodeQuest's primitive relations. The database is then indexed in a straightforward manner, namely by creating a separate index on each field of every table. The *abc* compiler also collects information on the shadows in the program that are matched by pointcuts, which is used to verify the correctness of the result of CodeQuest's evaluation of the Datalog pointcuts. The overall architecture is shown in Figure 14.

The experiments themselves are run on four AspectJ software projects of various sizes. Weka [49] is a data mining tool written in Java, consisting about 10KSLOC. It is instrumented with an aspect that checks for changes to an object's hash code while it is in a hash set. JHotDraw [18] is a popular open source Java drawing program, with approximately 21KSLOC. It has been instrumented with an aspect that checks for the safe use of Java enumerations, similar to concurrent modification exception checks in Java collections. ReWeave is a set of diagnostic aspects applied to the *abc* compiler itself, which has 51KSLOC. Finally, Jigsaw [42] is the w3c's leading-edge webserver, with approximately 101KSLOC, and has been instrumented

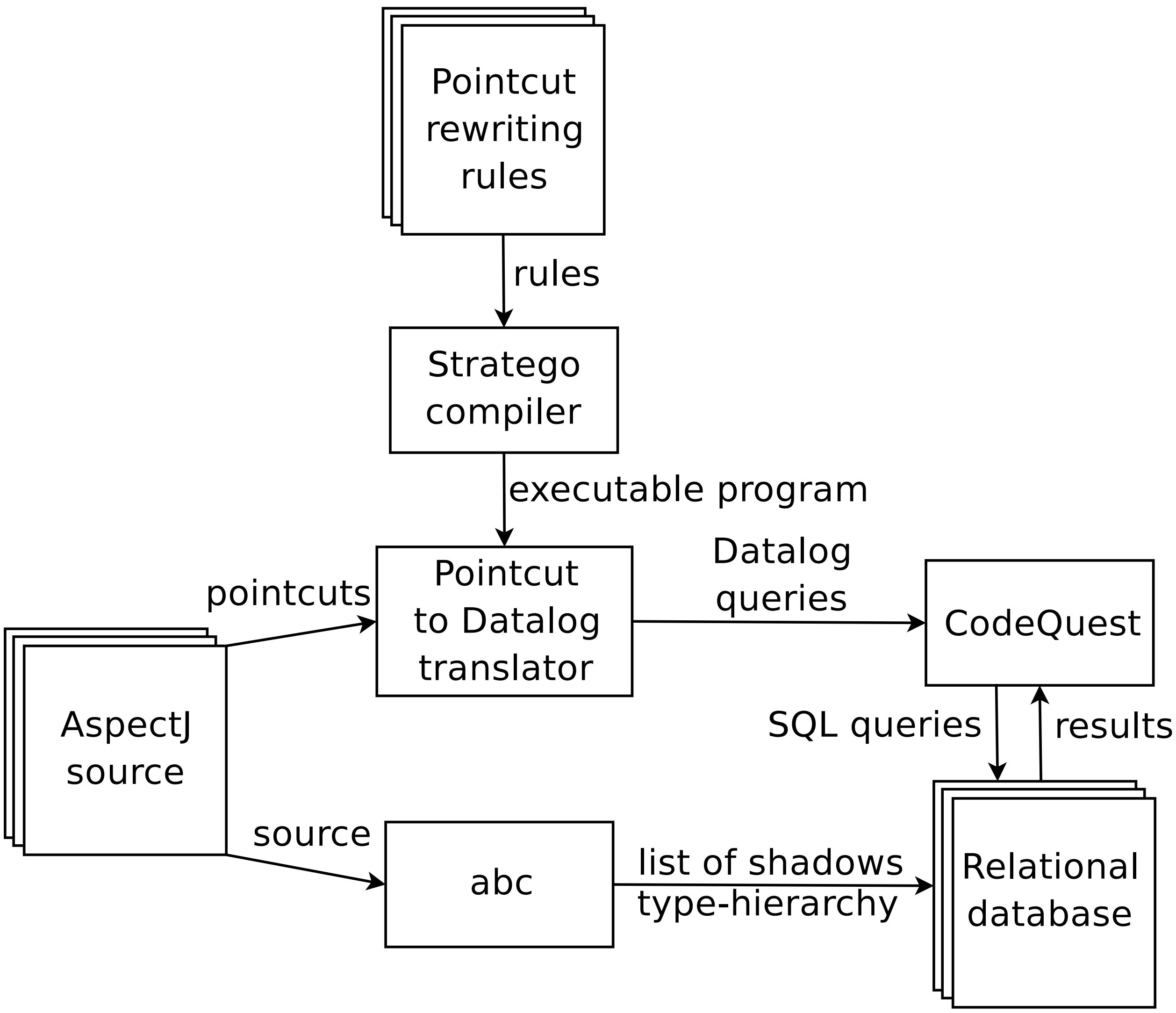

Figure 14: Experimental Setup

by an aspect that checks for the proper sequence of resource locks. Figure 15 details the pointcuts in each of the projects above that were used in the experiments. These are representative of typical AspectJ usage.

Figure 16 summarises the results of the experiments, with the matching time in CodeQuest for each of the pointcuts. It is interesting to observe that the performance scales well with the project size: from one project to the next, the size roughly doubles, and that is reflected in query execution times. We should stress at this point that CodeQuest is a research prototype, and it is easy to envisage many optimisations that will improve performance.

Now that we have established that the pointcut matching times are acceptable, it is natural to ask what penalty must be paid for storing the program structure and the shadows in a relational database, and for building indices on those primitive relations. We compare the performance of matching in CodeQuest with that of the industrial AspectJ compiler *ajc*. CodeQuest matching times are composed of two parts: the time it takes to populate and index the facts into the database (PI), and the time to execute the aggregate query for the project (AQ). The aggregate query for a each project is the disjoint union of the Datalog equivalents of all of its pointcuts. This is then compared to the total compile time for the project in *ajc*. Figure 17 shows the results of this comparison.

The evaluation times of the aggregate queries are much lower than the simple total of the individual

Name	Pointcut
Weka1	! **within**(RealHashtableNativeAspect) && **call**(∗ Map.put(..))
Weka2	! **within**(RealHashtableNativeAspect) && (**call**(∗ Map.get(..)) || **call**(∗ Map.remove(..)) || **call**(∗ Map.containsKey(..)))
Weka3	! **within**(RealHashtableNativeAspect) && **call**(∗ Map.remove(Object))
JHotDraw1	**call**(∗ Vector.add∗(..)) || **call**(∗ Vector.clear ()) || **call**(∗ Vector.insertElementAt(..)) || **call**(∗ Vector.remove∗(..)) || **call**(∗ Vector.retainAll (..)) || **call**(∗ Vector.set∗(..))
JHotDraw2	**call**(Object Enumeration.nextElement())
JHotDraw3	**call**(Enumeration+.new(..))
ReWeave1	**set**(∗ ∗) && !(**within**(Aspect) || **within**(IdentityPair) || **within**(org.aspectj..∗) || **within**(org.aspectbench..∗))
ReWeave2	**get**(∗ ∗) && !(**within**(Aspect) || **within**(IdentityPair) || **within**(org.aspectj..∗) || **within**(org.aspectbench..∗))
ReWeave3	**execution**(∗ abc.weaving.weaver.Weaver.resetForReweaving())
ReWeave4	**execution**(∗ abc.weaving.weaver.Weaver.weaveAdvice()) && !(**within**(Aspect) || **within**(IdentityPair) || **within**(org.aspectj..∗) || **within**(org.aspectbench..∗))
ReWeave5	**adviceexecution**() && **within**(Aspect) || **within**(IdentityPair) || **within**(org.aspectj..∗) || **within**(org.aspectbench..∗)
Jigsaw1	**execution**(∗ ∗(..)) && !**within**(CflowDepth || LockUpdater || LockChecker)
Jigsaw2	**call**(∗ org. w3c.tools. resources. ResourceReference+.lock())
Jigsaw3	**call**(∗ org. w3c.tools. resources. ResourceReference+.unlock())

Figure 15: Pointcuts used in experiments

Pointcut	CodeQuest
Weka1	4.68 s
Weka2	5.91 s
Weka3	5.24 s
JHotDraw1	11.36 s
JHotDraw2	7.83 s
JHotDraw3	7.53 s
ReWeave1	9.61 s
ReWeave2	9.62 s
ReWeave3	13.29 s
ReWeave4	17.45 s
ReWeave5	7.50 s
Jigsaw1	36.93 s
Jigsaw2	31.34 s
Jigsaw3	31.03 s

Figure 16: Pointcut matching times with CodeQuest

pointcuts that comprise it due to the large number of common terms in the Datalog equivalents of those pointcuts. CodeQuest takes advantage of this during query evaluation by computing the result of common terms only once.

The table also shows a ratio comparing *ajc*'s total compile time with that of a hypothetical *ajc* compiler that uses Datalog for pointcut matching. The ratio itself is given by

$$\frac{\text{AJC} + \text{PI} + \text{AQ}}{\text{AJC}}$$

We compare the query evaluation time to the total compilation time in *ajc* as it is difficult to separate pointcut matching times in an AspectJ compiler (as an example, name patterns can be evaluated prior

Project	AJC	populate + index (PI)	aggregate query (AQ)	Ratio
WEKA	5.49 s	7.79 s	9.15 s	4.09
JHOTDRAW	5.05 s	7.73 s	12.15 s	4.94
REWEAVE	19.89 s	9.94 s	26.58 s	2.84
JIGSAW	32.25 s	14.93 s	43.50 s	2.81

Figure 17: Comparison with *ajc*

to matching pointcuts to shadows). The intent of this table is to compare Datalog matching times with an actual industrial-strength AspectJ compiler, and to show that the performance degradation will not be prohibitive if pointcut matching in *ajc* were to be replaced with Datalog in CodeQuest.

The results show that the matching times for Datalog queries in CodeQuest compare favourably with the total compile time of the *ajc* compiler: they do not exceed the *ajc* compiler's compilation times by more than a factor of five. For a comparison, *abc*, which is a research-oriented AspectJ compiler designed more for extensibility than speed, can be up to 30 times slower than *ajc*. Furthermore, the gap does seem to narrow as the size of the project increases.

The times for loading the database are admittedly quite high. An important property, however, is that this only needs to be done for each compilation unit separately, and thus when recompilation occurs, we can incrementally update the relevant database facts. In particular, library code would only be loaded once in the database. Details of this process are described in [21].

In summary, the above experiments show that directly using our semantics as a basis for an AspectJ compiler is *feasible*. Whether it is also possible to make it competitive with existing implementations remains to be seen. We believe that some penalty in compile time may be acceptable: firstly, a direct implementation is easily kept in one-to-one correspondence with the semantics; but also, Datalog is more expressive than the AspectJ pointcut language — facilitating useful pointcuts that cannot be expressed in AspectJ. We expound the latter point in the next section.

8 Directly expressing pointcuts in Datalog

Datalog, together with the primitive predicates described above, is a richer language than the AspectJ pointcut language. It is possible to write recursive pointcuts; bind types, members, and shadows; and directly query the type and lexical hierarchies of a program.

As a result, it can be beneficial to write pointcuts directly in Datalog. The richness of the language is useful because it allows a programmer to express pointcuts in terms of semantic, rather than syntactic, criteria — AspectJ pointcuts tend to fall into the second category. The problems with syntactic criteria, and the uses of logic languages in alleviating them, have been discussed in the literature before [16, 20, 29]. However, they suggest using a computationally complete logic language.

Related work has established the benefits of logic languages for expressing pointcuts (cf. Section 9). In this section, we aim to demonstrate that despite the restrictions of safe Datalog to enable strong termination guarantees, many interesting examples can still be expressed.

Figure 18 shows a well-known aspect for updating the view of an application structured using the Model-View-Controller modularisation. The move pointcut defined on Lines 2–7 is syntactic in nature: it enumerates all the methods that update the state of an element in the application-model. A more direct way of expressing the *intended* set of methods is:

> All methods that may write to a field that could then be read in the repainting routine Display.repaint().

Translating this straight into Datalog yields:

```
aspect DisplayUpdating {
  pointcut move():
    call(void FigureElement.moveBy(int,int)) ||
    call(void Point.setX(int))               ||
    call(void Point.setY(int))               ||
    call(void Line.setP1(Point))             ||
    call(void Line.setP2(Point));

  after() returning: move() && !cflowbelow(move()) {
    Display.needsRepaint();
  }
}
```

Figure 18: A famous aspect for decoupling model and view

```
needsDisplayUpdate(M) ←
    typeDecl(DC,'Display',_,_),
    methodDecl(Repaint,'repaint',DC,_,_),
    mayRead(Repaint,F),
    mayWrite(M,F).
```

Of course, we must now define exactly what we mean by mayRead and mayWrite. It is not enough to just consider what fields are read (or written) in one particular method — we must also consider any method that could be called transitively from it. When computing this, we shall also keep in mind the fact that methods can be overridden. The predicate mayCall, which contains caller-callee pairs and takes into account overriding, is defined as follows.

```
mayCall(M,M2) ←
    containsShadow(M,Call),
    callShadow(Call,M3),
    overrides(M2,M3).
```

```
containsShadow(M,S) ←
    executionShadow(S2,M),
    isWithinShadow(S,S2).
```

We are now able to define mayRead and mayWrite in terms of mayCallStar.

```
mayRead(M,F) ←
    mayCallStar(M,M2),
    containsShadow(M2,S),
    getShadow(S,F,_).

mayWrite(M,F) ←
    mayCallStar(M,M2),
    containsShadow(M2,S),
    setShadow(S,F,_).
```

Just as in examples in previous sections, we use the convention that, for any predicate pred, we write predStar to denote its reflexive transitive closure.

Finally, it only remains to specify the overrides predicate. It is well-known how to do that in a logic language, as demonstrated for instance in JQuery [35]. These definitions carry over to Datalog almost unchanged.

A common complaint about aspects is the lack of clear interfaces between modules. To a large extent this is a language-independent issue, and more a matter of software design [19]. It appears, however, that using semantic pointcuts such as those suggested above helps to decouple aspects from changeable implementation details.

9 Related Work

We have already made comprehensive references to previous work, so this section provides just a roadmap of the main highlights. The related work for this paper falls into four main categories: AOP semantics, logic pointcuts, logic languages for static analysis, and code queries.

AOP semantics In this paper, we have taken our cue from the approach of Walker *et al* [43]. The distinguishing characteristic of that semantics is the use of a core language with labelled instrumentation points. Arguably the semantics of Wand *et al.* [45] is more in line with the spirit of aspect-orientation, as all matching of events happens at runtime, thus foregoing that intermediate level. The connection between the two styles of semantics is given by partial evaluation: partially evaluating Wand's definitional interpreter yields a compiler in the style of Walker [34]. It would have been possible for us to make [45] our starting point, but as our focus are queries over the static program structure, it would have been a detour.

Logic pointcuts There is a large amount of previous works, including implemented systems, that propose a logic pointcut language in lieu of the patterns in AspectJ, notably [20]. In our opinion, the use of a Turing complete pointcut language has all the pitfalls associated with undisciplined meta-programming, for instance the potential of non-termination at compile time. Furthermore *just* using a logic language is syntactically very cumbersome for simple pointcuts that merely refer to method signatures. Finally, our experiments show that the use of Prolog is simply too inefficient in practice.

Static analysis Researchers in static analysis have long acknowledged that logic programming in general, and Datalog in particular, is a suitable notation for expressing static analyses; the work by Reps [37] is an early example. More recently Dawson *et al* revived that line of work, in [15]. Monica Lam and her students have continued that tradition, adding the new twist of an implementation via binary decision diagrams [31, 48]. We feel much is to be gained from combining those insights (mostly directed at traditional applications of static analysis), and re-using them in the context of aspect-oriented programming.

Code queries There is a rich and vast literature on 'code queries' in the software maintenance community, which makes connections to logic programming. It is in that context that the idea of storing a program in relational form originated. It would take us too far astray to review this field here; the interested reader is referred to a companion paper [21] for a comprehensive discussion of how that work influenced the design and implementation of our CodeQuest system.

10 Conclusions

We have presented the first rigorous semantics of a practical pointcut language, building on the pioneering work of Walker *et al.* [43]. In doing so, we have sorted out numerous subtle issues in the semantics of AspectJ, in particular regarding the matching of name patterns.

Of independent interest is the framework that we chose for this semantics, namely a translation to safe Datalog. Safe Datalog is a pure logic programming language, and it is not Turing complete as all queries are guaranteed to terminate. At the outset of this project it was not clear to us that the whole AspectJ pointcut language can be expressed in safe Datalog. The fact that this is indeed possible provides strong evidence that safe Datalog is a suitable intermediate form for pointcuts in aspect-oriented programming.

The translation took a pleasingly simple form, namely that of conditional rewrite rules. We believe that this makes our semantics accessible to a wide audience, including working AspectJ developers, as it requires no exotic mathematical machinery.

The semantics are executable, and our experiments provide strong evidence that a direct implementation is feasible. At present that still entails a considerable increase in compile time over an industrial-strength AspectJ compiler, but bearing in mind that we used a straightforward research prototype to evaluate the Datalog queries, we believe substantial further improvements are within reach. The additional expressive power that Datalog affords, plus the fact that the semantics *is* the implementation, may well be worth paying a penalty in compile time.

Finally, we have shown through an example that it is very useful for developers to have the option of writing pointcuts directly in Datalog. Clearly one would not wish to write *all* pointcuts in Datalog, as often AspectJ patterns are elegant and concise. Our semantics as a simple set of rewrite rules offers the possibility of a hybrid approach, where the two notations can be freely mixed as desired. Indeed, it is easy to envisage a system where advanced users can define new pointcuts of their own, via a set of rewrite rules that reduce them to existing primitives, much in the style of other extensible query languages [12]. Dean Wampler and Ron Bodkin have argued for a 'long form' of pointcuts in AspectJ, and our semantics provides precisely that [8, 44].

Acknowledgements We would like to thank all members of the *abc* team, in particular Eric Bodden, Laurie Hendren, Ondřej Lhoták and Ganesh Sittampalam for many interesting discussions on this paper, and also for our very enjoyable joint work on the *abc* compiler that made the present paper possible. Martin Bravenboer helped a lot with the use of Stratego. He also patiently incorporated numerous bug fixes and change requests into the AspectJ grammar in Stratego. Gregor Kiczales, Luke Ong and Sam Sanjabi provided helpful feedback on a draft of this paper.

References

[1] abc. The AspectBench Compiler. Home page with downloads, FAQ, documentation, support mailing lists, and bug database. `http://aspectbench.org`.

[2] Jonathan Aldrich. Open Modules: modular reasoning about advice. In Andrew P. Black, editor, *Proceedings of ECOOP 2005*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[3] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In Akinori Yonezawa and Satoshi Matsuoka, editors, *REFLECTION*, volume 2192 of *Lecture Notes in Computer Science*, pages 187–209. Springer, 2001.

[4] AspectJ Eclipse home page. http://eclipse.org/aspectj/, 2003.

[5] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *Proceedings of AOSD*, pages 87–98. ACM Press, 2005.

[6] Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Datalog Semantics of Static Pointcuts in AspectJ 1.2.1. Technical Report abc-2006-2, AspectBench Compiler Project, 2006. `http://aspectbench.org/techreports#abc-2006-2`.

[7] Ohad Barzilay, Yishai A. Feldman, Shmuel Tyszberowicz, and Amiram Yehudai. Call and execution semantics in AspectJ. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL*, pages 19–24, 2004. Technical report TR #04-04, Department of Computer Science, Iowa State University.

[8] Ron Bodkin. Pointcuts need a long form. `http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg05971.html`, 2006.

[9] Martin Bravenboer, Eric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for AspectJ — a case for scannerless generalized-LR parsing. In William Cook, editor, *Proceedings of OOPSLA*, page to appear. ACM Press, 2006.

[10] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: a minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *Proceedings of CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2004.

[11] Bill Burke. *has* and *hasfield* pointcut expressions. `http://aosd.net/pipermail/discuss_aosd.net/2004-May/000958.html`, 2004.

[12] Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible grammars for language specialization. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Database Programming Languages*, pages 11–31. Springer, 1993.

[13] Daniel S. Dantas and David Walker. Harmless advice. In *Conference record of POPL*, pages 383–396. ACM Press, 2006.

[14] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. PolyAML: a polymorphic aspect-oriented functional programming language. In Benjamin Pierce, editor, *Proceedings of ICFP*, pages 306–319. ACM Press, 2005.

[15] Stephen Dawson, C. R. Ramakrishnan, and David Scott Warren. Practical program analysis using general purpose logic programming systems. In Kathryn S. McKinley, editor, *Proceedings of PLDI*, pages 117–126. ACM Press, 1996.

[16] Kris de Volder. Aspect-oriented logic meta-programming. In Pierre Cointe, editor, *REFLECTION*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 1999.

[17] Hervé Gallaire and Jack Minker. *Logic and Databases*. Plenum Press, New York, 1978.

[18] Erich Gamma. JHotDraw. Available from `http://sourceforge.net/projects/jhotdraw`, 2004.

[19] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[20] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of AOSD*, pages 60–69. ACM Press, 2003.

[21] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: scalable source code queries with Datalog. In Dave Thomas, editor, *Proceedings of ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006.

[22] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Karl Lieberherr, editor, *Proceedings of AOSD*, pages 26–35. ACM Press, 2004.

[23] Jim Hugunin. Support for modifiers in typepatterns. `http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg01578.html`, 2003.

[24] Peter Hui and James Riely. Temporal aspects as security automata. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL workshop at AOSD*, Technical Report #06-01, pages 19–28. Iowa State University, 2006.

[25] Wes Isberg. Type patterns shouldn't match arbitrary suffixes. Bug report: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=141133`, 2006.

[26] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In Luca Cardelli, editor, *Proceedings of ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 2003.

[27] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, To appear, 2006.

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings of ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[29] Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. In *ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 116–126. 2004.

[30] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[31] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of PODS*, pages 1–12. ACM Press, 2005.

[32] Ralf Lämmel. A semantical approach to method-call interception. In *Proceedings of AOSD*, pages 41–55. ACM Press, 2002.

[33] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In Atsushi Ohori, editor, *Proceedings of APLAS*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[34] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2003.

[35] Edward McCormick and Kris De Volder. JQuery: finding your way through tangled code. In *Companion to OOPSLA*, pages 9–10. ACM Press, 2004.

[36] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *Proceedings of ICSE*, pages 59–68. ACM Press, 2005.

[37] Thomas W. Reps. Demand interprocedural program analysis using logic databases. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, volume 296 of *International Series in Engineering and Computer Science*, pages 163–196. Kluwer, 1995.

[38] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In Richard Thomas Snodgrass and Marianne Winslett, editors, *SIGMOD*, pages 442–453. ACM Press, 1994.

[39] Peri L. Tarr, Harold Ossher, and Stanley M. Sutton. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of ICSE*, pages 689–690. ACM Press, 2002.

[40] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In Mehmet Akşit, editor, *Proceedings of AOSD*, pages 158–167. ACM Press, 2003.

[41] Eelco Visser. Meta-programming with concrete object syntax. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Proceedings of GPCE*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2002.

[42] w3c. Jigsaw. `http://www.w3.org/Jigsaw/`, 2006.

[43] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In Colin Runciman and Olin Shivers, editors, *Proceedings of ICFP*, pages 127–139. ACM Press, 2003.

[44] Dean Wampler. Humane pointcut languages. `http://blog.aspectprogramming.com/articles/2006/04/21/`, 2006.

[45] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[46] Meng Wang, Kung Chen, and Siau-Cheng Khoo. On the pursuit of static and coherent weaving. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL*, number TR #06-01 in Technical Report, pages 43–52, 2006.

[47] Meng Wang, Kung Chen, and Siau-Cheng Khoo. Type-directed weaving of aspects for higher-order functional languages. In John Hatcliff and Frank Tip, editors, *Proceedings of PEPM*, pages 78–87. ACM Press, 2006.

[48] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog and binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[49] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java implementations*. Morgan Kaufmann Publishers, 2000.