The abc Group

Datalog Semantics of Static Pointcuts in AspectJ 1.2.1

abc Technical Report No. abc-2006-2

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

July 2006

aspectbench.org

Contents

1 Introduction

AspectJ [3] is currently the most popular aspect-oriented programming language. It features an extremely rich *pointcut language* that allows programmers to intercept runtime events (which are known as *joinpoints* in the literature), and run additional code before, after or instead of intercepted joinpoints.

Pointcuts are thus just patterns that range over joinpoints, and the way pointcut matching is performed is an integral part of the semantics of any aspect-oriented programming language. Curiously, even though a large amount of research effort has gone into investigating AOP semantics (*e.g.* [1, 2, 5–7, 10–14, 16–19]), such studies have focussed on the operational semantics of advice, taking into account only a very simple pointcut language. For instance, in [16], the base language identifies program points for instrumentation by explicitly including labels, and pointcuts are simply sets of such labels.

However, in order to understand a language like AspectJ fully, it is vital to have a well-defined semantics for pointcut matching. The fact that no such document exists has been a continuing source of serious bugs in AspectJ compilers and misconceptions among proponents of the technology. In spite of this, the user community continues to make requests for a yet more expressive pattern language, only exacerbating the problem.

The aim of this document is to present a complete semantics of static pointcut matching in AspectJ. Our approach is to provide a semantics that reduces the complex pointcuts found in the language to simple sets of program points (or source locations — *shadows* in the literature), leaving the dynamic part of the story to an operational semantics in the style of [16], which is already well-understood.

We achieve this goal by applying a simple syntactic term rewriting system to AspectJ pointcuts, translating them into Datalog queries over relations defined over the object program. Datalog is a logic query language that originated in the theoretical database community [8]. We restrict ourselves to *safe Datalog*, a fragment of Prolog that has a straightforward least-fixpoint semantics; furthermore, all safe Datalog queries are guaranteed to terminate. Evaluating the produced queries over a database of facts derived from the object program in CodeQuest [9] — our own implementation of Datalog based on RDBMS — results in a set of shadows that match the pointcut. Our semantics is thus executable.

Our rule set is built on top of the Stratego term rewriting system [15]. This technical report concentrates on exhibiting the entire set of around 90 rules; since the focus is not the exact syntax of Stratego, we present a slight typographical modification of the actual source code, in order to improve legibility.

We present the semantics of AspectJ 1.2.1 here, but we believe this style of term rewriting-based semantics is an excellent platform for discussing general pointcut languages. Some more information regarding the general ideas inherent in this work can be found in a companion paper [4].

2 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 [8] 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.

2.1 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:

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

where regexpmatch(X, P) is a 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, given 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.

2.2 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 **not**(E) is an expression whenever E is.
- We allow the use of disjunction, represented by a semicolon.

3 Rewriting basics

3.1 Primitive predicates

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. Also applies to packages (e.g. package `x.y` is a child of package `x`).
hasSubtype(T1, T2)	T2 is a direct subtype of T1.

Figure 1: Primitive Predicates: Program Structure

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 1. 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:

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 2: Primitive Predicates: Shadows

methodParamTypes(Method, Type, Pos, NextPos)

which 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.

While java packages are not hierarchical, we have found that defining a hierarchy on packages using the hasChild primitive was needed to properly rewrite type patterns that contain the `..` wildcard. Thus hasChild(A,B) holds if B is a direct subpackage of A (e.g. package `x.y` is a child of package `x`).

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. Figure 2 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.

3.2 Pre-defined derived predicates

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. Figure 3 lists all the derived predicates of this type.

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) ).
```

Predicate	**Description**
package(P)	P is a package
class(C)	C is a class.
interface(I)	I is an interface.
primitive(T)	T is a primitive type.
reftype(C)	C is a reference type (i.e. a class or an interface).
reforprimtype(T)	T is a reference or primitive type.
array(T)	T is an array type.
type(T)	T is a type (reference, primitive or array).
packageOrType(T)	T is a package or a type.
constructor(M)	The method M is a constructor.
method(M)	M is a method.
callable(M)	M is a method or a constructor.
field(F)	F is a field.
shadow(S)	S is a shadow.
modifier(M)	M is a modifier.
name(N)	N is a name.

Figure 3: Pre-defined Derived Predicates: Types

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.

Figure 4 lists all the pre-defined predicates that are not used as types.

3.3 Master rewriting strategies

Our rules consist of a set of *modules*, each dealing with a specialised subset of the AspectJ pointcut language. In the code listings, we adopt the convention that term constructors are shown in **bold font**, and metavariables which capture subexpressions of the current left-hand side are typeset in *italics*. Both left- and right-hand side are enclosed in brackets ([[..]]) to make reading easier.

A module declaration consists of a series of import statements, followed by a list of rules. Each rule specifies a pattern, and, following an arrow (→), what any expression matching the pattern should be rewritten to. Optionally, there can be a **where** clause that gives side conditions — most of the time, these consist of the generation of fresh logic variable names using the *newname* construct, but in a few cases they are actual conditions and will be discussed in detail.

The top-level module ties everything together: It imports all other modules, as well as some libraries, and defines the overall *rewriting strategy*.

```
strategies
  main = io−wrap(makeQuery ; innermost(tx))
```

We see that the *main* strategy consists of the sequential composition of two auxiliary strategies: *makeQuery* and *innermost(tx)*. It is applied to the AST obtained from parsing an AspectJ pointcut. First, the makeQuery strategy creates the overall structure of the generated Datalog query: A pointcut is rewritten to a query with two free variables, a context C and a shadow S. The intended interpretation is that the pointcut, when evaluated from the given context, selects precisely those shadows S which make the query true.

```
rules
  makeQuery: pointcut         → [[  ?predname(C : type, S : shadow) ←aj2dl(pc, C, S).  ]]
```

Predicate	**Description**
isInPackage(X,P)	The syntactic element X is contained (perhaps indirectly) by package P.
returns(M,T)	The return type of method M is T.
hasSubtypePlus(T1,T2)	The transitive closure of the hasSubtype primitive.
hasSubtypeStar(T1,T2)	The reflexive transitive closure of the hasSubtype primitive.
hasModifier(X,M)	The syntactic element X has a modifier M. X may be a method or a field. M is an identifier for the modifier (not the string form).
hasStrModifier(X,Mod)	Mod is the string form of the modifier of the syntactic element X.
hasType(F,T)	The field F has type T.
hasName(X,N)	The syntactic element X has the name N.
hasChildPlus(A,B)	Transitive closure of the hasChild primitive.
hasChildStar(A,B)	Reflexive transitive closure of the hasChild primitive.
declaresConstructor(C,M)	The constructor M is declared in class C.
declaresMethod(C,M)	The method M is declared in the class C.
declaresField(C,F)	The field F is declared in class C.
afterParam(X,P1,P2)	In the method X, there exists a parameter with position P2 that comes after a parameter of position P1.
paramPos(P)	P is a position in the method parameters.
hasNoArgs(M)	The method M has no paramters.
defaultPackage(P)	P is the default package of the compilation.
topLevelPackage(P)	The package P is not contained by any package (other than the default package).
simpleTypeLookup(C, N, T)	Given the context C and a name N, use Java name lookup to find type T.
overrides(Cur, Super)	Holds if the method Super is overridden by Cur in a subclass, or if Super and Cur are the same method.

Figure 4: Pre-defined Derived Predicates

```
                                    where <newname>("Shadow") ⇒S;
                                          <newname>("Context") ⇒C;
                                          <to−pc−expr−and−name> pointcut ⇒(pc, predname)
```

The name of the generated query and the actual pointcut expression are extracted from the AST using the auxiliary strategy *to-pc-expr-and-name*. Finally, the pointcut expression *pc* is wrapped in the **aj2dl** term constructor.

All the rules that are actually part of our semantics are labelled with **tx:**. They eliminate the term constructor **aj2dl** introduced by *makeQuery* by rewriting it to Datalog and further term constructors. Recall that the *main* strategy calls *innermost(tx)* — this ensures our rewrite rules are applied exhaustively.

Sometimes, it is convenient to rewrite a term constructor to a *true(X)* predicate, when it imposes no further constraint on the selected shadows. Our rules do this in various places. We shall do this whenever the semantics justify it.

3.4 Named pointcuts

AspectJ allows the user to define *named pointcuts* — named pointcut expressions that can be used when defining other pointcuts. In general, a named pointcut can take a number of formal parameters, which can be bound with the usual AspectJ binding constructs (like args, this *etc.*).

Each named pointcut is transformed to a Datalog query with the same name. To avoid name clashes, and to avoid involving the orthogonal issue of poincut-name resolution in our semantics, we assume that all named pointcuts have been renamed to have distinct names. Recall that each pointcut must be evaluated in the context of the type it is lexically defined in. Named pointcuts may be defined in a different context to pointcuts that refer to them. Therefore, an additional primitive Datalog relation is required: *pointcutContext(N,C)* holds if the pointcut with name N should be evaluated in context C.

Thus, transforming a reference to a named pointcut into Datalog is easy: We simply refer to a query of the same name.

```
⟦ aj2dl(pcname(pcargspat), C, S) ⟧ → ⟦ NewC : type ˆ (pointcutContext(pcname, NewC),
                                                      pcname(NewC, S)) ⟧
                                         where <newname>("Context") ⇒NewC
```

3.5 Boolean expressions over pointcuts

Pointcuts can be combined with the usual Boolean connectives — logical conjunction, disjunction and negation. Since all of these are present in Datalog, we can simply rewrite the conjunction of two pointcuts to the conjunction of the two Datalog expressions that they are rewritten to. A similar approach deals with the remaining operations.

```
⟦ 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)) ⟧
```

4 Patterns

4.1 Member patterns

A *member pattern* in AspectJ is a pattern that picks out particular class members (fields, methods or constructors). In general, such a pattern can either be a *simple name pattern* (a string of identifier characters, possibly containing the wildcard ∗), in which case all members from all classes whose name textually matches the pattern are selected, or it can be of the form namepat.snamepat or namepat..snamepat, where *namepat* is a *name pattern* ranging over types or both packages and types, and *snamepat* is a simple name pattern. The former case selects all members whose name textually matches *snamepat* which are contained in a type that matches *namepat*, and the latter all members matching *snamepat* that are in a type contained in a type or package matching *namepat*.

For details on the matching of name patterns to type, see Section 4.7.

Because of a subtlety in dealing with static members, we must differentiate between member patterns for methods, for fields and for constructors.

4.1.1 Method Member Patterns

The **methmembpat2dl** term constructor is applied to four entities: the member pattern to match, the matching context C, the receiver type of the method call R and the method X matching the member pattern.

In the base case, the pattern is just a simple name pattern *snamepat*. We rewrite this to Datalog that asserts X matches *snamepat*, is a method, and is declared in some supertype of the actual receiver type R. The subtype relationship is expressed by hasSubtypeStar(Z, R), and the fact that Z declares the method X by hasChild(Z, X).

```
[ methmembpat2dl(snamepat, C, R, X) ]         → [ snamepat2dl(snamepat, X), method(X),
                                              Z : type ^ (hasSubtypeStar(Z,R),
                                              hasChild(Z,X)) ]
                           where <newname> "Z" ⇒Z
```

If the method member pattern is compound and can be decomposed as namepat..snamepat, then we want to assert the following:

- There is a package or type Y that matches *namepat*.
- There is a class Z transitively contained in Y such that Z is a supertype of the receiver type R.
- There is a method X that matches *snamepat*, defined in some type P. Moreover,
    - If X is static, then P is actually equal to Z.
    - If X is not static, then P is some supertype of Z.

All this translates into Datalog in the following fashion:

```
[ methmembpat2dl(namepat..snamepat, C, R, X) ]→[ Y : packageOrType ^ Z : type ^ P : type ^
                       (wcnamepat2dl(namepat, Y), hasChildPlus(Y, Z), hasSubtypeStar(Z,R),
                        method(X), snamepat2dl(snamepat, X), hasChild(P, X),
                        ((hasStrModifier(X, static ), equals(P, Z));
                              (not(hasStrModifier(X, static)), hasSubtypeStar(P, Z)))) ]
                  where <newname> "Y" ⇒Y;
                        <newname> "Z" ⇒Z;
                        <newname> "P" ⇒P
```

Note that because .. is considered a wildcard, we used **wcnamepat2dl**(*namepat*, Y) to rewrite *namepat*. For a discussion of why this is important, see Section 4.7.

Suppose now that the member pattern decomposes as typepat.snamepat. In this case, we want to assert the following:

- There is some type Z matching *typepat* that is a supertype of the receiver type R.
- There is a method X that matches *snamepat*, defined in some type P. Moreover,
    - If X is static, the P is actually equal to Z.
    - If X is not static, P is some supertype of Z.

```
[[ methmembpat2dl(typepat.snamepat, C, R, X) ]]→[[Z : type ^ P : type ^
                        (typepat2dl(typepat, C, Z), hasSubtypeStar(Z, R),
                        snamepat2dl(snamepat, X), method(X), hasChild(P, X),
                        ((hasStrModifier(X, static ), equals(P, Z));
                              (not(hasStrModifier(X, static)), hasSubtypeStar(P,Z)))) ]]
                 where <newname> "Z" ⇒Z;
                       <newname> "P" ⇒P
```

4.1.2 Field Member Patterns

Field member patterns are rewritten using the **fieldmembpat2dl** term constructor, which is applied to four entities: the pattern, the matching context C, the receiver type of the field access R, and the actual field X that matches the pattern.

Again, in the base case (when the pattern is just a simple name pattern), it is easy to see how the rewriting should proceed: we assert that there is a field X matching *snamepat*, and which is defined in some supertype Z of R.

```
[[ fieldmembpat2dl(snamepat, C, R, X) ]]          → [[ snamepat2dl(snamepat, X), field(X),
                                                    Z : type ^ (hasSubtypeStar(Z,R),
                                                    hasChild(Z,X)) ]]
                 where <newname> "Z" ⇒Z
```

Suppose now that the pattern decomposes into namepat..snamepat. It is important to contrast this with the corresponding case for **methmembpat2dl**, which had to consider whether the method was static or not. No such distinction is necessary with fields, and indeed this is the reason why we can't treat all member patterns uniformly.

With fields, we simply require the existence of a type Y that matches *namepat* (note that, again, we use **wcnamepat2dl** to rewrite a pattern that contains the wildcard ..), which transitively contains some type Z, which is a supertype of R. Given that, a field matching the pattern must be defined in some supertype P of Z.

```
[[ fieldmembpat2dl(namepat..snamepat, C, R, X) ]] → [[ Y : packageOrType ^ Z : type ^ P : type ^
                        (wcnamepat2dl(namepat, Y), hasChildPlus(Y, Z), field(X),
                        snamepat2dl(snamepat, X), hasSubtypeStar(Z,R),
                        hasChild(P, X), hasSubtypeStar(P, Z)) ]]
                 where <newname> "Y" ⇒Y;
                       <newname> "Z" ⇒Z;
                       <newname> "P" ⇒P
```

Finally, let us consider the case when the member pattern is of the form typepat.snamepat. Again, due to the fact that it isn't necessary to distinguish between static and non-static fields, this rule is somewhat simpler than the corresponding rule for **methmembpat2dl**. We simply require a type Z matching *typepat*, which should be a supertype of R, and such that some supertype, P, defines a field X matching the pattern.

```
[[ fieldmembpat2dl(typepat.snamepat, C, R, X) ]]→[[Z : type ^ P : type ^
                        (typepat2dl(typepat, C, Z),
                        snamepat2dl(snamepat, X), field(X),
                        hasSubtypeStar(Z, R), hasChild(P, X),
                        hasSubtypeStar(P,Z)) ]]
                 where <newname> "Z" ⇒Z;
                       <newname> "P" ⇒P
```

4.1.3 Constructor Member Patterns

The final possibility is that a member pattern refers to a constructor. If this is the case, then the final simple name pattern *must* be the string " new" — otherwise constructor member patterns look just like method member patterns.

Consequently, **constrmembpat2dl** is applied to four entities: a pattern, a matching context, a receiver type and a variable that binds to the constructor. Since constructors are not inherited in Java, these rules are significantly simpler than those for methods and fields.

In the basic case, the pattern is simply the string new, and to match it we simply require that the receiver type R declares a constructor X.

```
[[ constrmembpat2dl(new, C, R, X) ]]        → [[ constructor(X), hasChild(R,X) ]]
```

Otherwise, the pattern could be of the form namepat..new. We simply assert that there exists a type or package Z which matches *namepat* (note again the use of **wcnamepat2dl** because of the wildcard), which transitively contains the receiver type R, which defines some constructor X.

```
[[ constrmembpat2dl(namepat..new, C, R, X) ]]→[[Z : packageOrType ˆ
                              (wcnamepat2dl(namepat, Z), constructor(X),
                              hasChildStar(Z,R), hasChild(R,X)) ]]
                       where <newname> "Z" ⇒Z
```

Finally, the pattern could be of the form typepat.new. In that case, we require that the receiver type matches *typepat* and declares the constructor X.

```
[[ constrmembpat2dl(typepat.new, C, R, X) ]]→[[typepat2dl(typepat, C, R),
                              constructor(X), hasChild(R,X) ]]
```

4.2 Field patterns

Field patterns in AspectJ pick out specific fields, and occur only in get() and set() pointcuts. Unlike field member patterns (discussed in Section 4.1.2), which talk exclusively about textually matching the name of the field to a pattern, field patterns can also specify the modifier and a type for the field that is to be picked out.

Thus, a field pattern in general consists of a *field modifier pattern* (the discussion of which we defer to Section 4.5), a *type pattern* (discussed in Section 4.7) and a field member pattern. Each of these is further rewritten with the appropriate term constructor, and the results are conjoined with the assertion that the field has the appropriate type.

```
[[ fieldpat2dl(fmodpat typepat membpat, C, R, F) ]]→ [[  T : type ˆ (fmodpat2dl([fmodpat], F),
                                                      typepat2dl(typepat, C, T),
                                                      fieldmembpat2dl(membpat, C, R, F),
                                                      field (F), hasType(F, T))
                                                    ]]
                              where <newname> "Type" ⇒T
```

4.3 Method or constructor patterns

Method patterns in AspectJ occur whenever a particular set of methods needs to be picked out — namely, in call () and execution() pointcuts. They consist of a *method modifier pattern* (discussed in Section 4.5), a type pattern for the return type (Section 4.7), a method member pattern that determines the method name and containing type (Section 4.1.1), a *formals pattern* that selects methods based on their parameter types (Section 4.4), and an optional *throws pattern* that imposes constraints on the throws clause of the method.

At first, we choose to ignore the throws pattern. A method pattern with no throws clause is rewritten in the usual way, wrapping each constituent part with the appropriate term constructor and conjoining the results with the requirement that the return type T of the method should match *typepat*.

```
[[ 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)) ]]
                       where <newname> "Type" ⇒T
```

Now consider the case when the method pattern does specify a throws pattern. We simply delegate to the above rewrite rule and rewrite the throws pattern using the **throws2dl** term constructor, which ensures the declared exceptions of the method X conform to *throwspat*.

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

Constructor patterns look just like method patterns, with the exception that they do not specify a return type, and always end in .new. Apart from these minor points, however, their handling is exactly identical to that of method patterns, discussed above.

```
[ methconstrpat2dl(cmodpat constrmembpat(formalspat), C, R, X) ]→[cmodpat2dl([cmodpat], X),
                        constrmembpat2dl(constrmembpat, C, R, X),
                        formals2dl([formalspat], C, X, 1) ]

[ methconstrpat2dl(cmodpat constrmembpat(formalspat) throws throwspat, C, R, X) ]
                    → [ methconstrpat2dl(cmodpat constrmembpat(formalspat), C, R, X),
                          throws2dl([throwspat], C, X) ]
```

4.4 Formal patterns

A *formals pattern* occurs in method patterns; conceptually, it is a list of type patterns which correspond to the parameter types of the matching method. In addition to this, the wildcard .. has a special meaning here: it is taken to denote zero or more parameters of arbitrary type. Thus, [] is a formals pattern that specifies methods with no parameters, [..] matches all methods, regardless of parameters, and [A, .. , B] matches methods with at least two parameters whose first parameter has type A, and last parameter has type B.

Because we restrict the rewrite rules to *safe Datalog*, which doesn't allow the use of lists or other similar data structures, we need to provide some suitable encoding of each method's parameter types in the extensional predicates. Our approach is to represent them with the methodParamTypes(M, T, X, Y) relation. The intended interpretation is that M is a method, which has a parameter of type T at parameter position X, and Y is the position of the next parameter (so X+1 — arithmetic is not available in safe Datalog, either), or 0 if there is no next parameter.

Now we can present our rewrite rules for **formals2dl**. This term constructor is applied to four entities: the formals pattern, enclosed in square brackets, the matching context C, the method M that we are matching against the pattern, and the current parameter position X. Recall from Section 4.1.1 that **formals2dl** is originally invoked with a value of 1 in the last position, indicating that we start matching the formals pattern at the first parameter position.

If the pattern is empty, then we assert that M has no arguments.

```
[ formals2dl([], C, M, X) ] → [ hasNoArgs(M) ]
```

If the pattern at any stage consists of simply the formal parameter wildcard .., then we need not assert anything else — we rewrite this to a special true(X) predicate that does not impose further constraints on the result.

```
[ formals2dl([..], C, M, X) ]        → [ true(X) ]
```

If the current formals pattern is a single type pattern, then we assert that the parameter at position X of the method M has a type T that matches *typepat*, and that the next parameter position is 0 (i.e. this is the last parameter). Note the side condition on Line 4, where we prohibit the *typepat* to be matched against the formal wildcard .., which is handled in the above case.

```
1  [ formals2dl([typepat], C, M, X) ]  → [ T : type ˆ (typepat2dl(typepat,C,T),
2                                              methodParamTypes(M, T, X, 0)) ]
3                    where <newname> "ParamType" ⇒T;
4                          <not(?FormalWildcard)> typepat
```

If the current element of the formals pattern is the formal parameter wildcard .., then we assert that there is some parameter position (i.e. integer) that is after the current position X, and that we can match the remaining *formalspat* starting at that position. Note that this rule uses a non-trivial side condition: Line 5 ensures that the remainder is not the empty list (if it is, then the second of these rules should be applied).

```
1  [[ formals2dl([.., formalspat],  C, M, X) ]]      →
2                          [[ N : paramPos ^ (afterParam(M, X, N),
3                                 formals2dl([formalspat], C, M, N)) ]]
4                          where <newname> "Next" ⇒N;
5                                 <not(eq)> (formalspat ,[])
```

The final possibility is that the current element of the formals pattern is some type pattern, and this is not the end of the pattern. In that case, we assert that the parameter of M at position X has a type that matches *typepat*, that the next parameter position is N, and that we can match the remaining *formalspat* starting at N. Note again the use of non-trivial side conditions: In addition to requiring the remainder of the pattern to be non-empty, Line 7 prohibits *typepat* from binding to the formal wildcard .., which is handled in the above case.

```
1  [[ formals2dl([typepat, formalspat],  C, M, X) ]]         →
2                          [[ T : type ^ N : paramPos ^ (typepat2dl(typepat, C, T),
3                                 methodParamTypes(M, T, X, N),
4                                 formals2dl([formalspat], C, M, N)) ]]
5                          where <newname> "ParamType" ⇒T;
6                                 <newname> "Next" ⇒N;
7                                 <not(?FormalWildcard)> typepat;
8                                 <not(eq)> (formalspat, [])
```

4.5 Modifier patterns

Java allows different sets of modifiers for fields, methods and constructors. Each of these can be matched against by modifier patterns. For this reason, we distinguish between the three different kind of modifier patterns — the rewriting implementation is exactly identical in all three cases.

Each of **mmodpat2dl**, **cmodpat2dl** and **fmodpat2dl** is applied to two entities: A list of modifier patterns and the method, constructor or field that is being matched. If the list of modifiers is empty, we rewrite to true(M), since there are no more conditions to add.

If the first element of the list is a positive modifier, we assert that M has that modifier, and recursively rewrite the tail of the list. Note the use of the <*toString*> strategy, which converts from the AST node representing each modifier to the string representation of that modifier. The reason for this is that we chose to represent modifiers as strings in the extensional predicates.

Finally, if the first element of the list is a negated modifier, we assert that M does *not* have that modifier, and again recursively rewrite the tail.

```
[[ mmodpat2dl([], M) ]]                    → [[ true(M) ]]
[[ mmodpat2dl([methmod methmods], M) ]]→[[Mod : modifier ^ (hasModifier(M,Mod),
          hasName(Mod,str), mmodpat2dl([methmods], M)) ]]
     where <toString> methmod ⇒str; <newname> "Mod" ⇒Mod
[[ mmodpat2dl([!methmod methmods], M) ]]→[[Mod : modifier ^ (not(hasModifier(M,Mod)),
          hasName(Mod,str), mmodpat2dl([methmods], M)) ]]
     where <toString> methmod ⇒str; <newname> "Mod" ⇒Mod

[[ cmodpat2dl([], M) ]]                        → [[ true(M) ]]
[[ cmodpat2dl([constrmod constrmods], M) ]]→[[ Mod : modifier ^ (hasModifier(M,Mod),
          hasName(Mod,str), cmodpat2dl([constrmods], M)) ]]
     where <toString> constrmod ⇒str; <newname> "Mod" ⇒Mod
[[ cmodpat2dl([!constrmod constrmods], M) ]]→[[ Mod : modifier ^ (not(hasModifier(M,Mod)),
          hasName(Mod,str), cmodpat2dl([constrmods], M)) ]]
```

```
        where <toString> constrmod ⇒str; <newname> "Mod" ⇒Mod

  ⟦ fmodpat2dl([], M) ⟧                      → ⟦ true(M) ⟧
  ⟦ fmodpat2dl([fieldmod fieldmods], M) ⟧ → ⟦ Mod : modifier ˆ (hasModifier(M,Mod),
            hasName(Mod,str), fmodpat2dl([fieldmods], M)) ⟧
        where <toString> fieldmod ⇒str; <newname> "Mod" ⇒Mod
  ⟦ fmodpat2dl([!fieldmod fieldmods], M) ⟧ → ⟦ Mod : modifier ˆ (not(hasModifier(M,Mod)),
            hasName(Mod,str), fmodpat2dl([fieldmods], M)) ⟧
        where <toString> fieldmod ⇒str; <newname> "Mod" ⇒Mod
```

4.6 Throws patterns

Method patterns in AspectJ can specify a list of exception patterns in a throws clause. The interpretation of each exception pattern, however, depends on the *lexically first* character: According to the AspectJ language documentation, if the first character is !, then the exception pattern matches if *none* of the method's declared exceptions match the class name pattern that is obtained by deleting the !. In all other cases, the exception pattern matches if *some* declared exception matches the entire class name pattern.

The authors consider this to be unpleasant language design, since it gives semantic meaning to parentheses (throws !Ex and throws (!Ex) mean different things). It has certainly proven to be a point of confusion for users of the language. Nevertheless, expressing the desired semantics is not a problem for our rewriting approach.

An empty throws pattern is rewritten to the dummy true(M) predicate.

```
  ⟦ throws2dl([], C, M) ⟧        → ⟦ true(M) ⟧
```

If the pattern is not empty, and the first exception pattern lexically starts with !, then we assert that there exists no type E that is thrown by the method and that matches *classnamepat*, after which we recursively rewrite the remainder of *throwspat*.

```
  ⟦ throws2dl([!classnamepat, throwspat], C, M) ⟧ → ⟦ not(E : type ˆ(classnamepat2dl(classnamepat, C, E),
                                   throwsException(M, E))), throws2dl([throwspat], C, M) ⟧
                        where <newname> "Exception" ⇒E
```

If the first exception pattern does not lexically start with !, then we assert that there exists some type E that is thrown by the method and that matches *classnamepat*, after which we recursively rewrite *throwspat*.

```
  ⟦ throws2dl([classnamepat, throwspat], C, M) ⟧→ ⟦ E : type ˆ(classnamepat2dl(classnamepat, C, E),
                                   throwsException(M, E)), throws2dl([throwspat], C, M) ⟧
                        where <newname> "Exception" ⇒E
```

Note that in order to write the rewrite rules in this simple fashion, the underlying grammar for AspectJ pointcuts needs to be aware of this issue and parse exception patterns correctly. This turned out to be quite tricky, emphasising the fact that a somewhat questionable design was being implemented.

4.7 Type patterns

4.7.1 Name patterns

AspectJ has two kinds of patterns for ranging over types:class name patterns and type patterns (discussed in the next sections). They both use *name patterns* as building blocks.

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.

Name patterns may include the `*` and `..` wildcards and are matched differently, depending on whether or not they contain wildcards. Patterns that do are matched against fully-qualified type names. In contrast, patterns that do not contain wildcards are matched using Java name lookup — so that, for example, it is possible to refer to `java.lang.String` as simply `String`.

Some special encoding in the extensional predicates was required to simplify matching of name patterns to fully qualified names. In Java, package pkg1.pkg2 can be defined even if there exists no definition of the package pkg1. In other words, definition of package pkg1.pkg2 does not imply a containment relationship between pkg2 and pkg1. However, in order to simplify matching of name patterns to fully qualified names, a containment relationship between all elements divided by a dot in all package declarations has been encoded in the hasChild relation. Moreover, a default package, whose name is denoted by an empty string ", is stored in the packageDecl relation and every type that is not defined in any package is set to be a child of the default package. Also the leading element of every package declaration is also a child of the default package.

The **snamepat2dl** constructor is used to rewrite simple name patterns. In this term constructor we assert that simple name pattern S matches the given pattern. A `*` name pattern is rewritten to the true(S) predicate, leaving S unrestricted.

```
[[ snamepat2dl(*, S) ]]              → [[ true(S) ]]
```

If, however, the pattern isn't just a single `*`, we assert that the name of S can be matched against the pattern. Note the side condition in Line 3 below: it ensures that this case and the one above do not overlap. This is important, as a singleton `*` as a name pattern is a special case, matching anonymous and local classes (which wouldn't figure in the hasChild relation).

Again, we use the <*toString*> strategy to convert the name pattern to a Datalog string representation.

```
1   [[ snamepat2dl(snamepat, S) ]]     → [[ N : name ^(hasName(S, N), re_match(str, N)) ]]
2                          where <newname> "Name" ⇒N;
3                                <not(eq)> (snamepat, "*");
4                                <toString>(snamepat) ⇒str
```

As discussed above, the AspectJ semantics of matching a name pattern depend on the presence or absence of a wildcard in the pattern. The **wcnamepat2dl** term constructor is used in the presence of wildcards: each name pattern is then matched to the fully qualified name of a type.

If the name pattern is a simple name pattern then we assert that there exists a child of the default package P whose name matches the pattern.

```
    [[ wcnamepat2dl(snamepat, T) ]]     → [[ P : package ^
(defaultPackage(P),
                              hasChild(P, T), snamepat2dl(snamepat, T)) ]]
                         where <newname> "P" ⇒P
```

If the name pattern is of the form namepat.snamepat, we assert that there exists a package or type T1 that matches the pattern *namepat* and has a child T that matches the pattern *snamepat*.

```
    [[ wcnamepat2dl(namepat.snamepat, T) ]]→[[T1 : packageOrType ^(wcnamepat2dl(namepat, T1),
                              hasChild(T1, T), snamepat2dl(snamepat, T)) ]]
                         where <newname> "Enclosing" ⇒T1
```

Similarly, if the name pattern is of the form namepat..snamepat, we assert that there exists a package or type T1 that matches the pattern *namepat* and transitively has a child T that matches the pattern *snamepat*.

```
    [[ wcnamepat2dl(namepat..snamepat, T) ]]→[[T1 : packageOrType ^(wcnamepat2dl(namepat, T1),
                              hasChildPlus(T1, T), snamepat2dl(snamepat, T)) ]]
                         where <newname> "Enclosing" ⇒T1
```

Let us now consider name patterns which do not contain wildcards (*exact* name patterns). Once again, the intended semantics in this case is to match the type that is obtained by performing Java name lookup from the position of the pointcut declaration, if any. Here it becomes obvious why we have carried around the context parameter C.

The rules follow the Java Language Specification very closely. A simple type pattern matches a type T if we can perform simple type lookup on the pattern and obtain T. A pattern of the form namepat.snamepat is matched by performing *package or type lookup* on *namepat*, and finding a child of the result that has a name matching *snamepat*.

```
[ exactnamepat2dl(snamepat, C, T) ]      → [ simpleTypeLookup(C, str, T) ]
                         where <toString>(snamepat) ⇒str
[ exactnamepat2dl(namepat.snamepat, C, T) ]→[Pot : packageOrType ˆ
                              (potnamepat2dl(namepat, C, Pot),type(T),
                               hasChild(Pot, T), hasName(T, str)) ]
                          where <newname> ”PoT” ⇒Pot;
                              <toString>(snamepat) ⇒str
```

Note that a pattern of the form namepat..snamepat should never be considered here, as `..` is a wildcard, and as such is handled by **wcnamepat2dl**.

Let us consider package or type lookup. Again, the structure of the Datalog query follows the JLS: if we are considering just a simple name pattern, then either we can resolve it to a type using simple type lookup (in which case the result of the lookup is that type), or we cannot – in which case it must denote a package, and we rewrite it using the **pnamepat2dl** term constructor.

```
[ potnamepat2dl(snamepat, C, T) ]       → [ (simpleTypeLookup(C, str, T));
                                (not(T1 : type ˆ(simpleTypeLookup(C, str, T1))),
                                 pnamepat2dl(snamepat, T)) ]
                          where <newname> ”T1” ⇒T1;
                              <toString>(snamepat) ⇒str
```

When dealing with the case namepat.snamepat, we first look up *namepat* as a package or type; if this succeeds, we look for a child type of the result whose name is *snamepat*. If, on the other hand, no such child type exists, then the pattern must denote a package, and correspondingly we rewrite it with **pnamepat2dl**.

```
[ potnamepat2dl(namepat.snamepat, C, T) ]→[Pot : packageOrType ˆ
                                ((potnamepat2dl(namepat, C, Pot),type(T),
                                  hasChild(Pot, T), hasName(T, str)) ;
                                 (not(T1 : typeˆ(potnamepat2dl(namepat, C, Pot),
                                  hasChild(Pot, T1), hasName(T1, str))),
                                  pnamepat2dl(namepat.snamepat, T))) ]
                          where <newname> ”PoT” ⇒Pot;
                              <newname> ”T1” ⇒T1;
                              <toString>(snamepat) ⇒str
```

Finally, to rewrite package name patterns, we again follow the JLS: for a simple name pattern, we look for a package that has that name. For a pattern of the form namepat.snamepat, we look for a package matching *namepat* that has a child matching *snamepat*.

```
[ pnamepat2dl(snamepat, P) ]→[package(P), hasName(P, str), topLevelPackage(P) ]
                          where <toString>(snamepat) ⇒str
[ pnamepat2dl(namepat.snamepat, P) ]→[P1 : package ˆ
                                (pnamepat2dl(namepat, P1), hasChild(P1,P),
                                package(P), hasName(P, str)) ]
                          where <newname> ”P1” ⇒P1;
                              <toString>(snamepat) ⇒str
```

Now, to decide which set of rewrite rules we should use, we need to check the presence or absence of wildcards in a name pattern at the topmost level. This is done in the **namepat2dl** term constructor: A pattern consisting of a single `*` by definition matches all types. For more complex patterns, we determine whether they contain a wildcard using the <*contains−wildcard*> strategy, and pick either **wcnamepat2dl** or **exactnamepat2dl** to further rewrite them.

```
[ namepat2dl(∗, C, T) ]              → [ true(T) ]
[ namepat2dl(namepat, C, T) ]        → [ wcnamepat2dl(namepat, T) ]
                          where <contains−wildcard>(namepat) ; <not(eq)> (<fqname>(namepat), ”∗”)
[ namepat2dl(namepat, C, T) ]        → [ exactnamepat2dl(namepat, C, T) ]
                          where <not(contains−wildcard)>(namepat) ; <not(eq)> (<fqname>(namepat), ”∗”)
```

4.7.2 Class patterns

Class patterns range over reference types and consist of name patterns connected with boolean operators and the subtype operator '+'. The **classnamepat2dl** constructor is used to rewrite class patterns and is applied to three entities: the class name pattern to match, the matching context C, and the reference type Cls matching the class name pattern.

Boolean connectives (conjunction, disjunction and negation) of class name patterns are translated into equivalent Datalog connectives similarly to the rules for rewriting a combination of pointcuts, shown above.

```
[ classnamepat2dl(classnamepat1 && classnamepat2, C, Cls)
            → (classnamepat2dl(classnamepat1, C, Cls), classnamepat2dl(classnamepat2, C, Cls)) ]
[ classnamepat2dl(classnamepat1 || classnamepat2, C, Cls)
            →  (classnamepat2dl(classnamepat1, C, Cls)); (classnamepat2dl(classnamepat2, C, Cls)) ]
[ classnamepat2dl(!classnamepat, C, Cls) ]
            → [ not(classnamepat2dl(classnamepat, C, Cls)) ]
```

If the class pattern is just a name pattern *namepat*, we assert there exists a reference type Cls that matches *namepat*.

```
[ classnamepat2dl(namepat, C, Cls) ]
            → [ namepat2dl(namepat, C, Cls), reftype(Cls) ]
```

If the name pattern is used together with a subtype operator '+', we assert that there exists a type X that matches *namepat* and the reference type Cls is its subtype.

```
[ classnamepat2dl(namepat+, C, Cls) ]      → [ X : type ^(namepat2dl(namepat, C, X),
                                    hasSubtypeStar(X,Cls), reftype(Cls)) ]
                        where <newname> "X" ⇒X
```

4.7.3 Type patterns

Type patterns are different to class patterns in two ways: they range over primitive types as well as reference types, and there is an additional array operator `[]` that allows patterns to range over array types. The **typepat2dl** constructor is used to rewrite type patterns and is applied to three entities: the type pattern to match, the matching context C, and the type T matching the type pattern.

Boolean connectives are rewritten as above:

```
[ typepat2dl(typepat1 && typepat2, C, T) ]→[ (typepat2dl(typepat1, C, T), typepat2dl(typepat2, C, T)) ]
[ typepat2dl(typepat1 || typepat2, C, T) ] → [ (typepat2dl(typepat1, C, T)); (typepat2dl(typepat2, C, T)) ]
[ typepat2dl(!typepat, C, T) ]             → [ not(typepat2dl(typepat, C, T)) ]
```

When matching primitive types we simply need to make sure that the type T has a name of the corresponding primitive type.

```
[ typepat2dl(void, C, T) ]          → [ hasName(T, void) ]
[ typepat2dl(boolean, C, T) ]       → [ hasName(T, boolean) ]
[ typepat2dl(float, C, T) ]         → [ hasName(T, float) ]
[ typepat2dl(double, C, T) ]        → [ hasName(T, double) ]
[ typepat2dl(short, C, T) ]         → [ hasName(T, short) ]
[ typepat2dl(int, C, T) ]           → [ hasName(T, int) ]
[ typepat2dl(long, C, T) ]          → [ hasName(T, long) ]
[ typepat2dl(char, C, T) ]          → [ hasName(T, char) ]
[ typepat2dl(byte, C, T) ]          → [ hasName(T, byte) ]
```

If the type pattern is just a name pattern *namepat*, we assert that there exists a type T that matches *namepat*.

```
[ typepat2dl(namepat, C, T) ]       → [ namepat2dl(namepat, C, T), type(T) ]
```

If the name pattern is used together with a subtype operator '+', as in **classnamepat2dl**, we assert that there exists a type T1 that matches *namepat* and the type T is its subtype.

```
[ typepat2dl(namepat+, C, T) ]      → [ T1 : type ^(namepat2dl(namepat, C, T1),
                                                    hasSubtypeStar(T1, T)) ]
                      where <newname> "Enclosing" ⇒T1
```

Finally, **typepat2dl** also matches array patterns.

Whenever a type pattern is matched with the [] operator, we assert that there exists a type T1 that matches *typepat* (which is rewritten recursively), and T is an array type over T1. Here we use a special extensional predicate, arrayDecl(T, ET, N), that helps to encode array types of any dimension. The interpretation of this relation is that an array type T has an element type ET and name N.

```
[ typepat2dl(typepat[], C, T) ]      → [ T1 : type ^(typepat2dl(typepat, C, T1), arrayDecl(T, T1, _)) ]
                       where <newname> "ElemType" ⇒T1
```

5 Primitive pointcuts

5.1 `adviceexecution`

An `adviceexecution` shadow is the whole body of an advice method. Since the corresponding pointcut takes no arguments, it matches *any* such shadow. The definition is therefore straightforward.

```
[ aj2dl(adviceexecution(), C, S) ] → [ adviceexecutionShadow(S) ]
```

5.2 `call`

A `call` shadow can be either

- a call to a Java method (not including any code to evaluate method arguments); or
- a call to an object constructor (not including code to evaluate arguments, but including the object allocation).

Each such shadow has an associated method or constructor, and a type. The Datalog literal *callShadow(S,X,R)* holds if S is a call-shadow where X is a method or constructor declaration, and R is the static type of the receiver of this call. If R does not have a declaration for the callee (in the case that it is an inherited method), then X refers to the declaration of the closest inherited method.

A `call` pointcut takes one argument, which is a pattern ranging over method or constructor signatures. In order to rewrite a `call` pointcut to Datalog, in terms of a context C and a shadow S, we must express two constraints: firstly, S must be a `call` shadow; and secondly, in the context C, a signature of the call-shadow must match the pattern from the pointcut. The possible signatures of Y are encoded as pairs (X, R), where X is a more general definition of Y (with matching signature), and R is the declaring type of Y.

If Y is a constructor, it is not inherited, and so only method equality is considered — it has no more general definition. In contrast, if Y is a method, then a more general definition X is any method declaration (abstract or not) such that Y overrides or is equal to X.

```
[ aj2dl(call(methconstrpat), C, S) ] → [ X : callable  ^ Y : callable  ^ R : type ^
                                    (methconstrpat2dl(methconstrpat, C, R, X),
                                     overrides (Y, X), callShadow(S, Y, R)) ]
                      where <newname> "SuperMethod" ⇒X;
                            <newname> "ConcreteMethod" ⇒Y;
                            <newname> "Recv" ⇒R
```

5.3 `execution`

An execution shadow is either

- the whole body of a Java method; or
- the body of a constructor *after* its call to another constructor or super constructor.

Similar to `call`, each such shadow has an associated method or constructor, and a type. The Datalog literal *executionShadow(S,Y)* holds if S is the `execution` shadow of a method or constructor Y.

An `execution` pointcut takes one argument, which is a pattern ranging over methods and constructors. There are two constraints to express when rewriting this pointcut in Datalog: the potential matching shadow S must be an `execution` shadow, and the signature of the corresponding method or constructor Y must match the pattern from the pointcut. The possible signatures of Y are encoded as pairs (X, R), where X is a more general definition of Y and R is the declaring type of Y.

If Y is a constructor, its only more general definition is itself. In contrast, if Y is a method, then a more general definition X is any method declaration (abstract or not) such that Y overrides or is equal to X.

```
⟦ 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)) ⟧
                              where <newname> "SuperMethod" ⇒X;
                                    <newname> "ConcreteMethod" ⇒Y;
                                    <newname> "Recv" ⇒R
```

5.4 `get` and `set`

A `get` shadow is the code that retrieves the contents of a field on some object. The Datalog literal *getShadow(S,F,R)* holds if S is a `get` shadow for a field F, where the static type of the object (that the field is on) is R.

The corresponding get pointcut takes one argument, which is a pattern ranging over field signatures. Again, the Datalog translation involves two parts: checking the kind of shadow, and ensuring that the pattern from the pointcut matches the field and static type associated with the shadow.

```
⟦ aj2dl(get(fieldpat ), C, S) ⟧ → ⟦ F : field  ˆ R : type ˆ
                                   (fieldpat2dl(fieldpat , C, R, F), getShadow(S, F, R)) ⟧
                              where <newname> "Field" ⇒F;
                                    <newname> "Receiver" ⇒R
```

The only differences between `set` and `get` pointcuts and shadows is that `set` shadows consist of the code that *writes* to a field on some object.

```
⟦ aj2dl(set(fieldpat ), C, S) ⟧ → ⟦ F : field  ˆ R : type ˆ
                                   (fieldpat2dl(fieldpat , C, R, F), setShadow(S, F, R)) ⟧
                              where <newname> "Field" ⇒F;
                                    <newname> "Receiver" ⇒R
```

5.5 `handler`

A `handler` shadow is the block of code inside an exception handler — that is, the catch part of a `try`... `catch` construct. The Datalog literal *handlerShadow(S,X)* holds if S is a `handler` shadow and the static type of the exception caught by the block is X.

The `handler` pointcut takes one argument, which is a pattern ranging over classes. A Datalog translation of this pointcut, in terms of a shadow S, must constrain S to be a `handler` shadow, where the static exception type of that shadow matches the pattern in the pointcut.

```
[[ aj2dl(handler(classnamepat), C, S) ]] → [[ X : type ^
                              (classnamepat2dl(classnamepat, C, X), handlerShadow(S,X)) ]]
                    where <newname> ”Ex” ⇒X
```

5.6 `initialization`

An `initialization` shadow is the block of code, in a constructor, that initialises an object after the `super()` call. The Datalog literal *initializationShadow(S,X)* holds if S is an `initialization` shadow and X is the signature of the constructor lexically containing S.

The `initialization` pointcut takes one argument, which is a pattern ranging over constructors. A Datalog translation of this pointcut, in terms of a shadow S, must constraint S to be an `initialization` shadow, lexically contained in a constructor that matches the pattern from the pointcut.

```
[[ aj2dl(initialization (constrpat), C, S) ]] → [[ X : constructor ^
           R : type ^ (methconstrpat2dl(constrpat, C, R, X),
           initializationShadow (S,X)) ]]
        where <newname> ”Constr” ⇒X;
              <newname> ”Recv” ⇒R
```

5.7 `pre-initialization`

A `pre-initialization` shadow is the block of code at the very beginning of a class constructor which evaluates the arguments (if any) to the `this()` or `super()` call. The Datalog literal *preinitializationShadow(S,X)* holds if S is a `pre-initialization` shadow and X is the signature of the constructor lexically containing S.

The `pre-initialization` pointcut takes one argument, which is a pattern ranging over constructors. A Datalog translation of this pointcut, in terms of a shadow S, must constraint S to be a `pre-initialization` shadow, lexically contained in a constructor that matches the pattern from the pointcut.

```
[[ aj2dl(preinitialization (constrpat), C, S) ]] → [[ X : constructor ^ R : type ^
                              (methconstrpat2dl(constrpat, C, R, X), preinitializationShadow(S,X)) ]]
                    where <newname> ”Constr” ⇒X;
                          <newname> ”Recv” ⇒R
```

5.8 `staticinitialization`

A `staticinitialization` shadow is the block of code that performs the static initialization for a class. This includes assigning default values to any static fields, and also the contents of the `static` {...} block (if present). The Datalog literal *staticinitializationShadow(S,X)* holds if S is a `staticinitialization` shadow for a class X.

The `staticinitialization` pointcut takes one argument, which is a pattern ranging over classes. A Datalog translation of this pointcut, in terms of a shadow S, must constrain S to be a `staticinitialization` shadow, where the class containing the shadow matches the pattern in the pointcut.

```
[[ aj2dl(staticinitialization (classnamepat), C, S) ]] → [[ X : type ^
                              (classnamepat2dl(classnamepat, C, X), staticinitializationShadow(S,X)) ]]
                    where <newname> ”Class” ⇒X
```

5.9 `within`

The `within` pointcut does not correspond to a single kind of pointcut. It takes a single argument, which is a pattern ranging over classes, and only matches shadows that are lexically within classes matching the pattern.

```
[[ aj2dl(within(classnamepat), C, S) ]]  → [[  X : type ˆ Y : type ˆ
                                              (classnamepat2dl(classnamepat, C, X),
                                               hasChildStar(X,Y), isWithinClass(S,Y)) ]]
                          where <newname> ”Class” ⇒X;
                                <newname> ”Class” ⇒Y
```

5.10 `withincode`

The `withincode` pointcut is similar to `within`. It takes a single argument, which is a pattern ranging over methods and constructors, and only matches shadows that are lexically within methods or constructors matching the pattern.

```
[[ aj2dl(withincode(methconstrpat), C, S) ]]
           → [[  X : callable  ˆ R : type ˆ S1 :  shadow ˆ
                         (methconstrpat2dl(methconstrpat, C, R, X),
                          executionShadow(S1,X), isWithinShadow(S,S1)) ]]
                   where <newname> ”Method” ⇒X;
                         <newname> ”Recv” ⇒R;
                         <newname> ”Shadow” ⇒S1
```

Acknowledgements We would like to thank all members of the *abc* team, in particular Eric Bodden, Laurie Hendren, Ondřej Lhótak 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.

References

[1] 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.

[2] 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.

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

[4] Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of Static Pointcuts in AspectJ. Technical Report abc-2006-3, AspectBench Compiler Project, 2006. `http://aspectbench.org/techreports#abc-2006-3`.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.