Datalog as a Pointcut Language in Aspect-Oriented Programming

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

Programming Tools Group, Oxford University, United Kingdom
{elnar.hajiyev,neil.ongkingco,pavel.avgustinov}@comlab.ox.ac.uk
{oege,damien.sereni,julian.tibble,mathieu.verbaere}@comlab.ox.ac.uk

Abstract

AspectJ's pointcut language is complex, yet often not expressive enough to directly capture a desired property. Prolog has been suggested as an alternative, but Prolog queries may not terminate, and they tend to be verbose. We solve expressiveness, termination and verbosity by using *Datalog* plus *rewrite rules*.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Experimentation, Languages, Performance

Keywords Semantic pointcuts, pointcut sematics, Datalog, aspect-oriented programming

1. Introduction

Aspect-oriented programming allows one to intercept events at runtime by writing patterns. These patterns are called *pointcuts*, and in fact they are queries over the program structure. The most popular language that embodies this idea is AspectJ [5].

AspectJ pointcuts are quite complex, and indeed after several years of development, there is still discussion about the semantics of the way it matches type names. Despite this complexity, there are many desirable patterns that cannot be expressed.

Below we review that problem, and then proceed to present our solution. We follow the pioneering work of many previous authors (*e.g.* [3]) in using a logic programming language. Their suggestion was to use Prolog. However, Prolog queries often do not terminate and can be very inefficient, making it unsuitable for use in a production compiler for an aspect-oriented language. Instead, we contend that a very small subset of Prolog, named *Datalog* [2] is in the sweet spot between expressiveness and efficiency.

To support the claim of expressiveness, we demonstrate that the problematic example presented in earlier works on logic pointcuts and discussed below can be nicely expressed in Datalog. Furthermore, we show that the whole of the AspectJ pointcut language can be reduced to Datalog via simple rewrite rules. This also demonstrates how we can overcome the problem that logic languages tend to be verbose, offering an alternative user-friendly surface syntax based on AspectJ patterns.

Finally, we present some experiments to show that Datalog queries are efficient, by evaluating the same query with an AspectJ compiler, with Prolog and with our own system, named CodeQuest [4].

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

2. AspectJ pointcuts

A famed example of aspect-oriented programming is display updating in a figure editor. Whenever a figure element is moved, the display needs to be updated. In AspectJ, a solution reads as follows:

```
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();
  }
}
```

The pointcut move captures all method calls that may result in a move. Below the pointcut, there is a piece of *advice*: whenever a move occurs, call the static method needsRepaint() on the Display class. The use of **!cflowbelow**(..) expresses that we only wish to repaint the display after a top-level operation is completed, so that the use of (say) setX nested in moveBy does not trigger an unnecessary update.

The definition of the pointcut move is brittle, in the sense that we must remember to change it whenever we change the implementation of the figure editor. For instance, if we introduced a new class named *Polygon*, we would have to list all methods that may move it. While the aspect has achieved a textual separation of the display updating concern, we must keep the tangled mental picture in order to maintain the code.

3. Datalog

In words, we wish to identify methods M that satisfy the following property:

> M may write to a field F that may be read in the repainting routine Display.repaint().

Clearly, after the completion of every top-level call of such a method M, the display needs updating. A number of authors have suggested that a logic programming language such as Prolog is suitable for this task. Very briefly, a logic query consists of a predicate such as needsDisplayUpdate(M). An evaluator finds instantiations for the free variable M. The logic program itself consists of *clauses* of the form:

$$p(X_1, \ldots, X_n) \leftarrow q_1(Y_1, \ldots, Y_m), \ldots, q_k(Z_1, \ldots, Z_l).$$

This should be read as a backward implication: the left-hand side is true if all predicates on the right-hand side of $\leftarrow$ are true. In Prolog, the arguments of predicates may be composite data structures. Our proposal is not to allow that, instead using only *Datalog* subset of Prolog [2]. In fact, we further impose the technical restriction that all queries are 'stratified' and 'range-restricted': this guarantees that all queries (including recursive ones) terminate.

The definition of needsDisplayUpdate is a simple translation of its description:

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

That is, M needs a display update if there exists a field F which may be read by the repaint method of the Display class. and that same field F may be written by M.

To define mayRead and mayWrite we need to determine whether a relevant field set or get may occur in the dynamic scope of a method execution. We therefore need a predicate named mayCallStar(M,M') which tells us whether method M may call method M' — it is the reflexive transitive closure of the mayCall relation. Furthermore, isWithinShadowStar(S,S') is the reflexive transitive closure of the isWithinShadow relation. Using these, we define a relation between methods and shadows:

```
callContains (M,S') ← mayCallStar(M,M'),executionShadow(S,M'),
                      isWithinShadowStar(S',S).
```

In words, callContains (M,S') holds when M may execute S' (it is thus similar to the 'predicted cflow' pointcut proposed by Kiczales).

Now we are in a position to define mayRead and mayWrite:

```
mayRead(M,F) ← callContains (M,S), getShadow(S,F,_).
mayWrite(M,F) ← callContains (M,S), setShadow(S,F,_).
```

To complete the example, we have to work out definitions of isWithinShadowStar and mayCallStar. We have already given several examples of transitive closure, so details of that are omitted. The definition of the mayCall relation is not trivial, because it has to take account of possible overriding. Fortunately it is well-known how to do that in Prolog (see for instance JQuery [6]), and those definitions are all directly expressible in Datalog.

4. User-friendly syntax

Datalog is powerful, but for really simple pointcuts (like identifying calls to a method with a specific signature) it is verbose and awkward. By contrast, the AspectJ pointcut language shines in such examples, not least because any valid method signature is also a valid AspectJ method pattern. This is one of the reasons newcomers find AspectJ easy to pick up: if you know Java, you know how to express simple pointcuts. Can we give Datalog a similarly attractive syntax?

One might also be concerned about the formal expressive power of Datalog. When it comes to the finer points of AspectJ pointcuts, can they really be expressed as Datalog queries?

The answer to both of these questions is 'yes'. We have constructed a translation from AspectJ pointcuts to Datalog, which consists solely of simple rewrite rules. It is our intention to open up that implementation to advanced users, so they can define new query notations, along with rules for reducing them to Datalog.

Here is an example rule, used in the translation of **call** pointcuts:

aj2dl(call (*methconstrpat*), C,S) →
∃ X,R: (**methconstrpat2dl**(*methconstrpat*,C,R,X),callShadow(S,X,R))

The constructor **aj2dl**(PC,C,S) is used to drive the translation: it takes a pointcut PC, the current aspect class C, and a shadow S. This will be reduced to a Datalog query containing just C and S as free variables. Pattern PC is further rewritten using the **methconstrpat2dl** constructor. Interaction between term constructors that are rewritten and other Datalog predicates is based merely on the Datalog variables that both are bound with. Therefore it is not necessary for a user to understand the outcome of a rewriting at all.

Our implementation uses *Stratego*, which allows to record the rewrite rules in concrete syntax, almost exactly as shown above [7].

5. Experimental results

It remains to address efficiency concerns. Can this expressive pointcut language be used in a production compiler for a language like AspectJ? We have conducted experiments by taking several AspectJ pointcuts and translating them to Datalog with the above rewrite system. We executed these queries using our own implementation of Datalog on top of a relational database system [4], and also with XSB, an optimising implementation of tabled Prolog [1]. The execution times are shown in the table below, in seconds unless otherwise indicated. The *abc* column is the time taken by the pointcut matcher in the *abc* AspectJ compiler.

base code	KSLOC	abc	XSB	CodeQuest
JHotDraw	21	2.4	>12 hours	13.2
Jigsaw	101	4.0	1264.0	3.4
Jigsaw	101	4.0	14832.7	0.5
Jigsaw	101	4.0	4801.2	28.4

Bearing in mind that CodeQuest is a research prototype with many opportunities for further optimisation, this is strong evidence that Datalog provides the right combination of expressiveness and efficiency. The overhead of constructing Datalog facts from a Java program is also not prohibitive when using incremental database update as discussed in [4].

Acknowledgments This research has been supported, in part, by EPSRC in the United Kingdom.

References

[1] *XSB*. `http://xsb.sourceforge.net/`.

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

[3] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International AOSD Conference*, pages 60–69. ACM Press, 2003.

[4] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: scalable source code queries with Datalog. In Dave Thomas, editor, *Proceedings of ECOOP 2006*, Lecture Notes in Computer Science. Springer, 2006.

[5] 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 2001*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[6] Edward McCormick and Kris De Volder. JQuery: finding your way through tangled code. In *Companion to the 19th annual ACM SIGPLAN OOPSLA'04 conference*, pages 9–10, New York, NY, USA, 2004. ACM Press.

[7] Eelco Visser. Meta-programming with concrete object syntax. In *Generative programming and component engineering (GPCE)*, pages 299–315, 2002.