

Datalog as a Pointcut Language in Aspect Oriented Programming

Programming Tools Group, University of Oxford, United Kingdom

AOP for:

- Dealing with cross-cutting concerns
- Runtime debugging and error detection
- Coding style enforcement

Pointcuts are the key to find locations of interest in the code

Wish list:

- Robust pointcuts
- Expressive semantic pointcuts
- Easy to learn, write and read

Problems

• Lack of expressiveness

All methods have to be explicitly listed
Just syntactic pattern matching
What if there are hundreds of methods?

• Not Robust

Need to update pointcuts consistently
What if library API methods change?

Motivating example in AspectJ

Safe Enumeration:

No calls to `nextElement()` of an Enumeration allowed if source object has been changed

- 1) First intercept creation of an *Enumeration* object for each *Vector* object

```
after(Vector ds) returning (Enumeration e):  
    call(Enumeration+.new(..)) && args(ds) { ... }
```

- 2) Then intercept an update of the same *Vector* object

```
after(Vector ds):  
    vector_update() && target(ds) { ... }
```

- 3) If there is a call to *nextElement* of an *Enumeration* and *Vector* object has changed raise an exception!

```
before(Enumeration e):  
    call(Object Enumeration.nextElement())  
    && target(e) { ... }
```

A pointcut to intercept changes of a *Vector* object requires listing of all methods that could modify it:

```
pointcut vector_update () :  
    call (* Vector.add* (..)) ||  
    call (* Vector.clear()) ||  
    call (* Vector.insertElementAt(..)) ||  
    call (* Vector.remove*(..)) ||  
    call (* Vector.retainAll(..)) ||  
    call (* Vector.set*(..));
```

Aspects Require Better Pointcuts!

Common answer: Prolog

```
vector_update_method(M) :-
// M is a method in
// a class V named Vector
  typeDecl(V, 'Vector', _, _),
  methodDecl(M, _, V, _, _),
// N is a method named nextElement
// in an implementation I of the
// Enumeration interface
  typeDecl(E, 'Enumeration', _, _),
  implements(I, E),
  methodDecl(N, 'nextElement', I, _),
// N may read field F
// (also via a3 chain of calls)
  mayRead(N, F),
// M may write field F
  mayWrite(M, F).
```

Some systems that use this architecture:
Alpha, Aspicere, Carma, Compose, LogicAJ, Sally, Soul, etc.*

Disadvantages of Prolog:

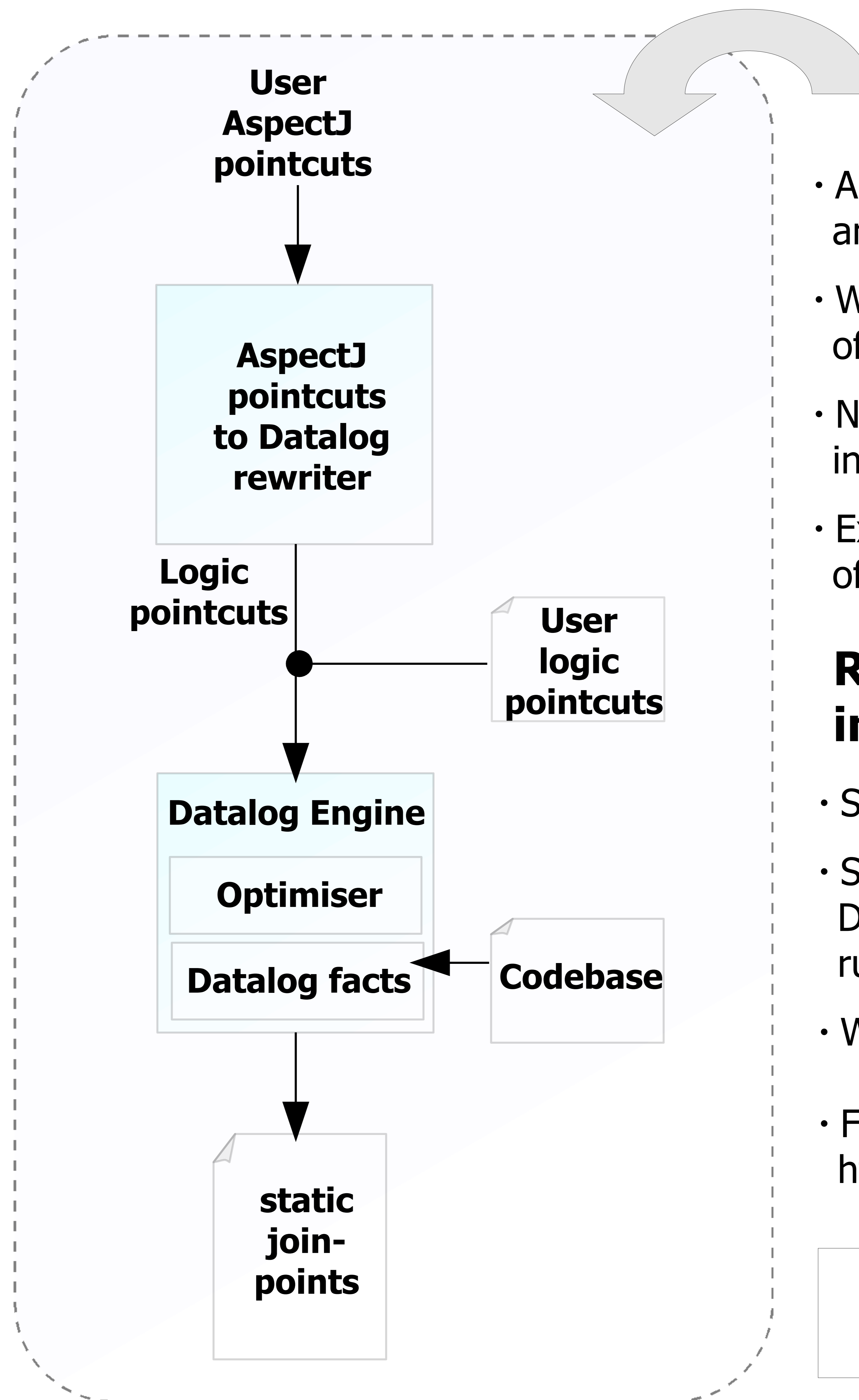
- Need mode annotations, cuts
- Queries may still not terminate
- Facts in memory do not scale

Datalog is the right solution!

- Use DataLog = Prolog - data structures
- Expressive enough, efficient and scalable
- Queries are always guaranteed to terminate

But...

- Name patterns expressed in a logic language are too verbose and unreadable
- Not every developer is immediately ready to learn Datalog



We propose

- A combination of purely syntactic AspectJ and more semantic Datalog pointcuts
- Where the former are a syntactic sugar of the latter
- Novice developers can define pointcuts in a more familiar style
- Experts can resort to the full power of logic

Rewriting AspectJ pointcuts into Datalog:

- Separate static and dynamic pointcuts
- Static pointcuts can be converted into Datalog by a set of simple transformation rules
- We use Stratego Transformation System
- Full set of rules is available online at: <http://progtools.comlab.ox.ac.uk/projects/>

Semantics of Pointcuts!

Rule, used to translate get (set) pointcuts:

```
aj2dl(get(fieldpat), C, S) =>
```

```
∃ F, R: (fieldpat2dl(fieldpat, C, R, F),
         getShadow(S, F, R))
```

And *fieldpat2dl* in its turn gets rewritten to:

```
fieldpat2dl(typepat memmpat, C, R, F) =>
```

```
∃ T: (typepat2dl(typepat, C, T),
       fieldmemmpat2dl(memmpat, C, R, F),
       field(F), hasType(F, T))
```

And so on, rules apply exhaustively until no more rules can be applied.

Example rewrite rules