Partial Evaluation of Pointcuts

Karl Klose[1], Klaus Ostermann[1], and Michael Leuschel[2]
{klose,ostermann}@st.informatik.tu-darmstadt.de,
leuschel@cs.uni-duesseldorf.de

[1] Darmstadt Univ. of Technology, Germany
[2] Univ. of Duesseldorf, Germany

Abstract. This paper makes use of declarative languages, together with associated analysis and specialisation tools, to implement powerful and extensible query languages for aspect oriented programming. Pointcuts are predicates over events in the execution of a program, and are used to specify where an aspect should be applied. Expressive and extensible pointcut languages allow the programmer to abstract away from computational details and write pointcuts at a higher level and in a domain specific way.
Pointcuts are usually compiled by identifying a set of shadows — that is, places in the code whose execution is potentially relevant for a pointcut — and inserting dynamic checks at these places for those parts of the pointcut that cannot be evaluated statically. Today, the algorithms for identifying shadows and generating appropriate dynamic checks are specific for every pointcut designator. This makes it very tedious to extend the pointcut language.
We propose a logic based pointcut language and a framework for finding shadows and generating dynamic checks based on partial evaluation of the pointcut predicates. With this approach it is possible to produce both powerful and extensible pointcut languages together with efficient compilation and execution of the resulting program.

1 Introduction

Before proceeding, let us first recall the basic concepts of aspect-oriented programming. Since we will refer to AspectJ [14] several times in this work, we will discuss the terminology in the way it is used in this language.

Crosscutting concerns are concerns (i.e., coherent issues) in a software system which crosscut the traditional module structure, for example classes in object-oriented languages. Typical examples for this kind of concerns are logging of messages, transaction management and the enforcement of security policies. In object-oriented programming, such concerns usually become scattered over the system and entangled with other concerns.

Aspect-oriented programming (AOP) is a programming paradigm that allows to modularize crosscutting concerns in a single module called an *aspect*. In this way, aspects can easily be maintained, added and removed from software systems without changing other modules.

Pointcuts are used to describe at which point an aspect affects the execution of the basic program. The points that can be selected by a pointcut are called *joinpoints*, because they describe points where the execution of different aspects and the base program can join in the program flow. Pointcuts can be thought of as defining a set of joinpoints and a pointcut is said to be *triggered* at a joinpoint, if the joinpoint is in that set. The functionality that an aspect should introduce at a selected joinpoint is called *advice*. The advice is executed at every point in the execution which triggers the associated pointcut. In addition to the execution of advice, Pointcuts can be used for a wide range of purposes, such as reverse engineering [8], detection of application errors [19], or flexible instrumentation of applications [9].

To create a executable program, the different aspects must be combined with the base program to which they are applied. This process is called *weaving* and is in most implementations performed by a *weaver* component of the compiler. The weaver identifies the so called *shadows* the pointcuts in program, that is, places in the code, where the pointcut may be triggered. Although there are pointcuts that select only static places in the code, in most cases dynamic checks will be necessary to determine if a pointcut is triggered. The weaver generates and inserts these dynamic checks are at the appropriate places in the code.

The first pointcut languages were relatively static in that pointcuts could be mapped directly to static locations in the source code of the underlying program. Recently, there is a trend towards more dynamic pointcuts which quantify over dynamic information such as the call-stack [26, 23], dynamic argument values [10], the full execution trace of the application [1, 22], the structure of the dynamic heap [22], or even the future of the execution [15].

As said before, such complex dynamic pointcuts cannot easily be mapped to places in the source code. Rather, some kind of dynamic evaluation is needed. There are different options to implement the required dynamic evaluation:

- Using an interpreter for the language which also takes care of pointcut evaluation. Since this approach is obviously too slow for practical purposes, interpreters are mainly used in theoretical studies [16, 12] or as a basis of further optimization [21].
- Complete instrumentation of an application. With this option, dynamic events are generated and evaluated after every computation step in order to determine whether a pointcut applies. This strategy also obviously incurs a major performance penalty.
- Identification of the set of shadows of a pointcut (all places in the code whose execution will potentially trigger a pointcut) together with the insertion of dynamic checks at these shadows. This option can be combined with techniques like dynamic checks that are executed on the virtual machine level [3] or continuous weaving [11] to further improve the performance.

We will concentrate on the last option. This option is indeed used in many AOP language implementations such as AspectJ [14]. However, the algorithms for computing the set of shadows and computing the right dynamic checks are highly non-trivial. Worse yet, these algorithms are specific to the constructs of a

particular pointcut language. Hence, if the pointcut language is to be extended, the algorithm has to be revisited and extended as well.

This is not only very elaborate. It is also a major obstacle to keeping the pointcut language extensible. Extensible pointcut languages have been recognized as a way to make pointcuts more robust, precise, and high-level, to enable domain-specific libraries of pointcuts, and to put the pointcut language design into the hand of the programmers [10, 5, 22, 7].

We propose a *generic* approach to finding shadows and generating dynamic checks. In our approach, pointcuts are represented as queries in a logic programming language. Our main idea is to compute shadows and checks using techniques for partial evaluation of logic programs (frequently called *partial deduction*).

The contributions of this paper are as follows:

- This is the first work to embed the shadow search and dynamic check generation problem into the framework of partial deduction.
- Our solution can be used to generate major parts of an efficient compiler for pointcuts automatically.
- It becomes very easy to extend the pointcut language because our technique is generic in that it does not depend on the pointcut language.
- Our measurements show that our approach scales to reasonably large programs.
- We describe different options to weave the remaining dynamic checks into the program.

The remainder of this paper is structured as follows: Sec. 2 gives an overview of our approach by means of small examples and describes the encoding of source code in Prolog and the design of the pointcut language. The use of partial evaluation and the approximation of runtime entities in our framework is explained in Sec. 3. Different possibilities to weave residual pointcuts into a program are described in Sec. 4. Related work is discussed in Sec. 5. Sec. 6 concludes and gives an outlook to future work.

2 Overview

In this section we give a quick overview of our approach without going into the details of the partial evaluation process itself. We limit our elaborations to pointcut queries over Java programs in this work, but other languages can be handled in a similar manner, with appropriate changes to the encoding of the program and the type system related predicates.

2.1 Prolog representation of the bytecode

Pointcut queries in our language are Prolog predicates. To enable these Prolog predicates to reason about the program's static structure and execution, these must be represented in the Prolog database.

In our work, we use a representation of the Java bytecode. We have built a converter which transforms Java bytecode into a Prolog representation of the byte code. Fig. 4 illustrates how the converted example from Fig. 3 looks like.

The Prolog representation contains the declarations and definitions of all classes in one database file. There a two kinds of facts in this database: information about classes, interfaces, methods and fields and their relationship, and facts describing the bytecode instructions which form the body of the methods.

For each class there is a fact called `class`, which includes (in the order of appearance) the package and class name, the modifiers (`public`, `abstract`, etc.), the super class and the implemented interfaces. The class name is wrapped in a `ref` term to indicate that it denotes a reference type (in contrast to a primitive type) and can be used to access the methods defined in that class.

Methods are represented by `method` facts. Each method is identified by a unique number (as its first argument) and the name of its enclosing class. The remaining arguments are the method name, flags for the method modifiers, the return type and the list of argument types, in that order.

The remaining facts in the representation encode the different types of bytecode instructions: `get` and `put` for field access instructions, `return`s, and `invoke` for method returns and calls, respectively. Assignments to local variables (denoted by `p1`, `p2` etc.) are encoded as `def` facts. A local variable declaration includes an initializing instruction, which may be either method calls which return a value (`invokeFunc`), reading field access (`get`) or object creation via `new`[3]. Each bytecode instructions starts with a number identifying the method which contains the instruction and a number denoting the position of this instruction in the list of instructions forming the method body. The third argument identifies the line number in the source code that corresponds to this instruction.

2.2 Example pointcuts

Fig. 1 shows an aspect in the language AspectJ for keeping a display showing graphical shapes up to date. The base program defining the shapes hierarchy is given in Fig. 3. The pointcut `change` in line 4 describes the points in the execution, where the display should be updated and the advice in line 10 specifies that a call to `display.update` should be executed *after* such a modification (specified by `change`).

The display to keep up-to-date is stored in a field of the aspect. The aspect itself is similar to a class, with the exception that it cannot be instantiated by the programmer but exists as a singleton object [4].

The first two conditions (line 5 and 6) of the pointcut select calls to a method called `setX` resp. `setY` of an object of static type `Point` with exactly one parameter of type `int`. The condition in line 7 selects calls to the `move` method with two

[3] object creation does not include the constructor call

[4] Aspects can also be specified to exist not only once per virtual machine but once in a thread or once for each object of a specific class or once for a control flow of a method

```
1  aspect UpdateSignaling {
2    Display display;
3
4    pointcut change():
5       (  call(void Point.setX(int))
6       || call(void Point.setY(int))
7       || call(void Shape+.moveBy(int, int)) )
8       && !cflowbelow(call(void Shape+.moveBy(int, int)));
9
10   after() returning: change() {
11     display.update();
12   }
13 }
```

Fig. 1. Display updating in AspectJ

integer arguments the type `Shape` or any of its subtypes. This is expressed by the use `+` appended to `Shape`. These conditions are combined by `||` meaning `or`, which selects any point that satisfies one of these conditions (this corresponds to the union of the sets of points selected by the three conditions).

The last condition excludes (this is expressed by the negation operator `!` in front of the pointcut) any joinpoint which is *in the control flow* of a call to `Shape+.moveBy(int,int)` but not such a call itself. The control flow of a call (expressed by `cflow`) comprises all joinpoints which appear while executing this call, including the call joinpoint itself. The pointcut `cflowbelow` excludes this call joinpoint from the set, selecting only joinpoints below this the call joinpoint in the control flow. This pointcut is combined with the other three by the `&&` operator meaning *and* (or set intersection when thinking about joinpoint sets).

The use of control flow dependent joinpoints illustrated the need for dynamic checks in the woven code, as the control flow can only be determined at runtime.

In our pointcut language, this pointcut can be expressed as follows:[5]

```
(calls(Stack,Loc,Target,setX,_), stype(Target,'shapes.Point') );
(calls(Stack,Loc,Target,setY,_), stype(Target,'shapes.Point') );
(calls(Stack,Loc,Target,moveBy,_), instance_of(Target,'shapes.Shape') ),
\+ cflowbelow(Stack,calls(_,_,moveBy,_))
```

In order to illustrate the effect of specialization, we will now consider a few pointcuts and the result of their specialization, without talking yet about how the specialization actually works. Note that our pointcut language is based on Prolog and thus uses the conventions that uppercase identifiers denote variables, lower case or identifiers quoted by 's are atomic values and _ denotes anonymous variables. Basic predicates are combined by , meaning *and* and ; meaning *or*.

Fig. 2 shows a few sample pointcuts and the result of specializing them with the example program from Fig. 3. We use the line numbers from Fig. 3 to denote the shadows; in reality, the method/instruction indexes from the byte code are used for this purpose.

[5] One could imagine providing an alternate syntax, e.g. inspired by AspectJ, for those users less well versed in Prolog.

```
– calls(Stack,Loc,_,setX,_), withinMethod(Loc, MethID), method(MethID,_,_,public,_,_,_,_,_,_)
  Shadow Residual pointcut
  21,23   true
– calls(Stack,Loc,R,moveBy,_), instance_of(R, 'shapes.Line')
  Shadow Residual pointcut
  29      Stack = [calls(Loc,R,moveBy,_)|_],
          dtype(R,T), subtypeeq(T, 'shapes.Line')
  30      true
– calls(Stack,Loc,_, setX, _), cflow(Stack, calls(_,_,moveBy,_))
  Shadow Residual pointcut
  31      cflow(Stack,calls(_,_,moveBy,_))
  21,23   true
```

Fig. 2. Example pointcuts and their shadows and dynamic checks

The first pointcut selects all calls of a `setX` method (`calls(Stack,Loc,_,setX,_)`) within (`withinMethod(Loc, MethID)`) a public method (`method(MethID,_,_,public,_,_,_,_,_,_)`). In our language, the relation between the method and the call is expressed in terms of the location (`Loc`) of an instruction and the identifier of the method (`MethID`). The predicate `withinMethod` binds `Loc` to all locations in the code which are lexically contained in the method identified by `MethID`.

This first pointcut illustrates that all predicates over static data, like `withinMethod` and `method`, and the static part of the method call are already evaluated during specialization, hence the residual pointcut is just `true`[6].

The second pointcut (all calls of `moveBy` where the receiver object is an instance of class `Line` at runtime) illustrates how static type information is incorporated into the specialization. At the first shadow, the static type of the receiver is `Shape`, hence a dynamic check is required whether the receiver is actually a `Line`. At the second shadow, however, the statically known receiver type is already `Line`, hence no dynamic check is necessary.

The third pointcut (all calls of a `setX` method in the control flow of a `moveBy` method) illustrates the effectiveness of the static approximation of the call stack during specialization. Whereas the first shadow requires a dynamic check, the second shadow has no dynamic check because it is known statically that the `setX` calls in lines 21 and 23 are in the control flow of a `moveBy` call. We will see that the design of the static approximation of the call stack is an important parameter for the specialization in computing residual pointcuts.

2.3 Programming model

Pointcut queries in our language can refer to the static structure of the program and a well-defined subset of the dynamic runtime properties. Based on this

[6] Actually, the residual program is not completely empty but binds the variables from the pointcut. For clarity, we have omitted residual programs that only bind variables but do not contain dynamic checks

information, arbitrary calculations can be used to decide whether or not the pointcut matches the current state of execution (and thus decide whether an aspect is applicable or not).

The runtime information that can be used in pointcut queries is not limited to the current joinpoint (or event), but comprises the whole callstack. The callstack is represented as a list containing all calls to methods that are currently in execution, i.e. have not yet finished.

Pointcuts are predicates that match certain points in the program execution, called joinpoints. In order to describe these joinpoints, pointcuts need to refer to the context in which they are evaluated. This context comprises — in our model — the current callstack, the current lexical position and the program. This context information can be kept implicitly available, as it is the case in AspectJ's pointcut language, or given as parameters to the pointcut query. In our case, we decided to make the callstack and the lexical position explicit parameters of the pointcut queries, whereas the program is implicitly available as a global set of facts in the Prolog database.

We use the variable names `Stack` for callstacks and `Loc` for lexical position. A location is a pair `loc(MethodNumber,InstrNumber)` which represents the method- and instruction number as given in the byte code. A callstack is represented by a list of stack frames, where each but the top frame must be a method call, represented by terms using the function symbol `calls/4`. The current instruction is at the top of the stack.

The following listing gives an example callstack as it may look like when modifying the field `x` in the method `Point.setX`, which was called by `Line.moveBy`:

```
1 [
2  set(loc(10,5), value(ref('shapes.Point'),ι2), x, 42),
3  calls(loc(6,11), value(ref('shapes.Point'),ι2), setX, [value(prim(int),42)]),
4  calls(loc(21,3), value(ref('shapes.Line'),ι1), moveBy, [value(prim(int),1), value(prim(int),1)])
5 ]
```

The location `loc(6,11)` in the call to `Point.setX` corresponds to sourcecode line 21 (Figure 3) and to the bytecode instruction at line 30 in Figure 4.

The first parameter in each stack frame denotes the location of the corresponding instruction in the program - it is hence a pointer into the Prolog representation of the byte code. Values are encoded as pairs which consists of a type and an address or primitive value like boolean or integer. The expressions ι_ns are object references in runtime environment. The representation of values is hidden from the pointcut programmer, however; the static type, dynamic type and the value (address for reference values like objects, otherwise the int, bool etc.) must instead be retrieved with the getter predicates `stype(V,T)`, `dtype(V,T)` and `value(V,A)`, respectively. The reason is that the static representation of values during specialization is different from the representation of runtime values, and hiding the representation by means of getters is an easy way to hide details of the specialization process from the pointcut programmer (as well as leading to cleaner code).

Depending on the weaving strategy, such call stacks may never be explicitly reified as physical data, but should mainly be seen as the data model upon which pointcuts are expressed.

2.4 The pointcut library

So far, we have seen how pointcuts can be formulated using the representation of the byte code and of the call stack directly. The real power of the approach lies in the fact that we can easily extend the pointcut language by means of Prolog predicates on top of the raw representation of the call stack and the byte code.

The predicates which form the pointcut language are defined as Prolog predicates themselves, which use the Prolog encoding of the program. The implementation of these predicates defines the connection between the semantics of the pointcut language and that of the bytecode language. For instance, in the definition of `instance_of` the `subtype` relation is used, which is directly extracted from the inheritance relation exposed by the bytecode representation. Similar to the corresponding AspectJ pointcut designators, the `cflow` predicate checks whether a particular entry can be found in the call stack; `cflowbelow` checks all but the first stack frame.

The pointcut library is the extension point of the pointcut language: new pointcut predicates can be introduced by defining them in the pointcut library. New pointcut predicates can be built in terms of existing predicates and the bytecode representation. As we will see, the specialization process is independent of the predicates defined in the pointcut library; hence every pointcut language that operates over the program and the callstack can easily be encoded by implementing corresponding predicates.

Furthermore it can be of interest to add new descriptions of the program - for example, the complete trace of the application or profiling information - and to use these descriptions in the definition of new pointcut predicates, thus providing the programmer with access to the new model. If the added descriptions are static (e.g., representations of configuration files), the specializer will automatically compile all references to the static data away. If the added descriptions are dynamic, a corresponding static approximation of the dynamic data has to be provided. We will discuss this point later.

3 The specialization framework

Program specialization is a technique to specialise a given general purpose program for certain specific application area. Partial evaluation [13] is a well-established technique that obtains a specialised program by pre-computing parts of the original source program that only depend on some given part of the input (called the static data).

Our specialization framework performs the task of computing shadows and the respective residual programs for pointcut queries. This is achieved by partially evaluating the pointcut query w.r.t. the static part of the input. This static

```
1  package shapes;
2
3  interface Shape {
4    public void moveBy(int dx, int dy);
5  }
6  class Point implements Shape {
7    private int x, y;
8    public int getX() { return x; }
9    public int getY() { return y; }
10   public void setX(int x) { this.x = x; }
11   public void setY(int y) { this.y = y; }
12   public void moveBy(int dx, int dy) {
13     x += dx; y += dy;
14   }
15 }
16 class Line implements Shape {
17   private Point p1, p2;
18   public Point getP1() { return p1; }
19   public Point getP2() { return p2; }
20   public void moveBy(int dx, int dy) {
21     p1.setX(p1.getX()+dx);
22     p1.setY(p1.getY()+dy);
23     p2.setX(p2.getX()+dx);
24     p2.setY(p2.getY()+dy);
25   }
26 }
27 class GraphicApp {
28   public void test(Shape s, Line l, int dx, int dy){
29     s.moveBy(dx,dy)
30     l.moveBy(dx,dy);
31     l.getP1().setX(42);
32   }
33 }
```

Fig. 3. The shape example

```
1  class('shapes',ref('shapes.Line'),
2    default,false,false,false,
3    ref('java.lang.Object')).
4  interfaces(ref('shapes.Line'),
5    ref('shapes.Shape')).
6  field(ref('shapes.Line'),'p1',
7    private,false,false,
8    false,false, false,
9    ref('shapes.Point')).
10 field(ref('shapes.Line'),'p2',
11   private,false,false,false,false,
12   false,ref('shapes.Point')).
13
14 method(6,ref('shapes.Line'),
15   'moveBy',public,false,
16   false,false,false,false,false,
17   [prim(int),prim(int)],void).
18  ...
19 def(6,7,22,ref('shapes.Point'),p11,
20   get(ref('shapes.Point'),'p1',
21   ref('shapes.Line'),thisValue)).
22 def(6,8,22,ref('shapes.Point'),p13,
23   get(ref('shapes.Point'),'p1',
24   ref('shapes.Line'),thisValue)).
25 def(6,9,22,prim(int),p14,
26   invokeFunc('getY',ref('shapes.Point'),
27   p13,[],[],prim(int))).
28 def(6,10,22,prim(int),p16,
29   add(p14,param(2))).
30 invokeProc(6,11,22,'setY',
31   ref('shapes.Point'),p11,
32   [prim(int)],[p16]).
33  ...
34 return(6,22,25).
```

Fig. 4. Excerpts of Prolog encoding

part is given by the representation of the program, which determines the possible static contexts in which the pointcut may be evaluated.

In order to make more static information available to the specialization process, we will later introduce static approximations of runtime entities like values and the callstack.

Specialization is performed by a partial evaluator for Prolog. The behavior of this tool is controlled by a description of the pointcut primitives and predicates in the pointcut library which marks certain parts of the pointcut program as *callable*. That means that they can be (safely) evaluated at specialisation time.

After a short introduction to our partial evaluator, we present the partial evaluation of pointcut queries with respect to the program source.

3.1 Partial evaluation

The partial evaluation (or *specialiser*) tool used throughout this work is based on the core of the offline specialiser presented in [17]. It is thus similar to the core of LOGEN [18].[7] An offline partial evaluator is one which works in two phases: first

[7] Albeit being an offline partial evaluator rather than a compiler generator.

the source program is annotated, partitioning the instructions of the program into those that can be evaluated by the partial evaluator and those that cannot. In the second phase—the specialisation phase proper—the specialised code is generated by following the annotations. In the following we suppose some basic familiarity with offline partial evaluation.

The core of this evaluator is quite short and is given in Fig. 5. The `body/2` predicate takes the annotated clause as its first argument and binds it second one to the residual program. while the first two clauses simply deal with the extension of the predicate over the connectives , and ;, the last three clauses define the whole partial evaluator, as we use it in this work.

The partial evaluator uses *annotations* to decide which part of the program can be evaluated. In our system, a query is annotated by wrapping it into the functor which represents the annotation. For example, `call(f(X))` means that `f(X)` can be evaluated at specialization time. The partial evaluator has been extended to automatically annotate un-annotated source code; using a set of rules on how the standard pointcut predicates should be treated.

We will now give an overview of the annotations available in our system:

- In case of a `call` annotation, the residual code is `true`, if the evaluation of the query succeeds. If there is no answer to the query, no residual program is generated, which is equivalent to the residual program `fail`. However, if the evaluation of the query yields multiple answers, the specialiser will return multiple residual clauses, which can be regarded as alternative solutions. In the case of infinitely many answers, the partial evaluator will, like the base program, not terminate.
- Handling of the `rescall` annotation is quite simple: the wrapped query is left as residual program, without any further evaluation.
- Handling the `unfold` annotation is similar to the normal evaluation of a Prolog query, which searches for a clause whose head unifies with the term and evaluates the body of this rule. As the partial evaluator works on annotated programs, which is retrieved by the predicate `rule/2`. As with normal Prolog evaluation, multiple applicable rules create backtracking in the partial evaluator, which leads to multiple residual clauses.

Evaluator control and Automatic Annotation Offline specialization tools require an annotated rule for each predicate that must be unfolded. A rule, like a normal Prolog clause, has a head and a body, but the body consists of annotated Prolog code. Normally, the program is annotated once and then used with different dynamic data, for example when compiling a program by specializing an interpreter w.r.t. the source program.

Our system does not require the programmer to annotate her pointcuts manually, but we rather use a standard set of rules to perform this annotation automatically before specialisation. Only in the case where these annotations are not optimal from the programmer's view, should he annotate the program himself. A typical example for this case is the introduction of user-defined predicates. As

the specialiser normally leaves calls to these predicates as residual code, it can be useful to annotate the predicates to benefit from partial evaluation.[8]

Whenever the specialiser needs a rule for unfolding a predicate term `P`

1. it checks the rule database `rule_db/2` for a predefined rule.
2. if no predefined rule is found, the body of the clause is examined and for each subquery in the body the annotation database (`annotation_db/2`) is queried for a predefined annotation. If no such annotation is present, the default annotation `rescall` is used.

The predefined annotations for the predicates in the pointcut library is given in Fig. 6.

```
1 body((A;B),Body)
  :- body(A,Body) ; body(B,Body).
2 body((A,B),(CA,CB))
  :- body(A,CA), body(B,CB).
3 body(call(C),true) :- call(C).
4 body(rescall(C),C).
5 body(unfold(X), B) :- unfold(X,B).
6 unfold(X,Code)
  :- rule(X,B), body(B,Code).
7
8 specialise( pointcut([S,L],Body), pointcut([S,L],ResBody) ) :-
9  create_rule_body(Body, AnnotatedBody), body( (call(joinpoint(S,L)), AnnotatedBody), ResBody).
```

Fig. 5. The specialiser core and its interface

Predicate	*Annotation*
`within/3`	call
`instance_of/2`	unfold
`stype/2`	call
`subtype/2`	call
`subtypeeq/2`	call
`calls/5`	unfold
`get/5`	unfold
`set/5`	unfold
`cflowbelow/2`	unfold
`cflow/2`	unfold

Fig. 6. Predefined annotations

In cases were a built-in should not be annotated `call`, the user must provide an annotated rule-body for the enclosing rule or add an annotation entry to database, which makes this annotation a global choice for this built-in call.

3.2 The partial evaluation of pointcuts

The basic concept to use partial evaluation for shadow matching is to think of the pointcut query as a logic program and the state of the software system (i.e., the program representation, typing information, heap, callstack, etc) as its input. The static part of this input comprises the static type information and the program's bytecode.

Shadow finding now means to specialise the pointcut w.r.t. to the static data. If the query returns solutions, the binding of the pointcut's location parameter can be used to identify the shadows and the residual program can be woven at this shadow.

[8] An alternative would have been the adaptation of an automatic binding-time analysis such as [6].

The specialization is performed by the predicate `specialise/2` (Fig. 5), which takes a pointcut as its first argument and binds its second argument to the residual pointcut (observe that we wrap the pointcut query itself inside the `pointcut/2` predicate).

The first subquery of `specialise` generates an annotation of the pointcut query. A call to the `joinpoint` predicate is then put in front of this annotated body and the resulting term is specialised by the partial evaluator. The call to the `joinpoint` predicate ensures that each solution of the partial evaluation actually denotes the static part of a joinpoint, that is an instruction in the bytecode together with its static context.

3.3 Approximation of runtime entities

In the scenario of pointcut specialisation, only the static part of the program is available. The static view onto the program consists of the class, interface and field declarations and a set of bytecode instructions. In contrast, the pointcut language allows the user to specify crosscuts based on runtime data, for example parameter and receiver values, the elements of the callstack and the actual type of a value.

In the case of values, the only solution is to leave checks as residual programs, as there is no information on actual values available from the static part of the program. The easiest way to handle actual types is to generate all possible instantiations and explore them by backtracking. If, for example, a pointcut constraints a variable to be a subtype of a class, the partial evaluator may try all possible subclasses. The problem with this implementation is that it does not scale well for programs with a large number of classes. A better solution is to use *abstract values* as approximations of runtime entities.

We will now describe the approximation of the actual type of a value and the elements of the callstack and how they are used in the specialisation process.

Approximation of the callstack The static event, i.e. the statically known part of the joinpoint event, for a location can be used to construct an approximation of the runtime callstack at this location. The static event is an approximation of the first element of the callstack, which contains the current event. Furthermore, the second element of the callstack must be a call to the method that contains the current location. The stack approximation is constructed by the predicate `staticStack`. For the example callstack in the last section we can thus give the following approximation:

```
1 [
2  set( loc(10,5), value(['java.lang.Object', 'shapes.Point' | _],_,_), x, value([prim(int)],_,_) ),
3  calls( _,     value(['java.lang.Object', 'shapes.Point' | _],_,_), setX, [value([prim(int)],_,_)] ),
4 ]
```

Better approximations that contain more elements or more precise type information can be generated by using the callgraph of the program . As the construction of the application's callgraph can be very costly, it is desirable to be able to control the amount of approximation. In our framework this can easily

be accomplished by modifying the `staticStack` predicate (to produce more/other information).

Representation of values In pointcuts, variables can be used to bind values, for example by providing them as arguments to event predicates. These *value variables* can be used in two different contexts: as the actual value (for instance in comparisons) or as the actual type (in `subtype`-checks). However, this dynamic information is not accessible at specialisation time. Nevertheless specialisation should be able to benefit from the *static information* about the variable that can be retrieved from the programs bytecode.

To solve this problem, we introduce an abstraction of values. It is based on the idea to associate with each variable the set of all classes whose instances the variable can possibly hold. In the context of Java single inheritance, we can describe the set of all possible types of a variable by the `:<`-maximal (i.e. the most general) type of the set. To make this abstraction compatible with unification, we encode the most general type of the variable as an *open list* containing all its super classes[9]. In this form, two encodings can be unified if one is the prefix of the other list, which means that it encodes a super type of the other list.

For example, the list presentation of the class `shapes.Line` from our example is `['java.lang.Object', 'shapes.Line' | _]`. A class `shapes.Arrow :< shapes.Line` would be encoded as `['java.lang.Object', 'shapes.Line', 'shapes.Arrow' | _]` and the unification of both will yield the latter list as required.

To associate the abstract type with the variable for the dynamic type and argument, type variables are bound to a term `value(AbsType,DynType,DynValue)`, where `AbsType` is the encoding of the possible types of this variable and `DynType` and `DynValue` denote the dynamic type and value and are variables in the specialisation phase.

3.4 Description of Pointcut Predicates

To take advantage from the approximation of runtime values and the callstack, we provide *descriptions* of the pointcut predicates defined in the pointcut library: a description of a pointcut library predicate does not only provide the necessary annotations for the partial evaluator, but also includes additional calls to handle the approximations of dynamic entities. Fig. 7 shows the description of some of the pointcut library predicates.

3.5 Example specialisations

After introducing the specialisation and approximation techniques, we demonstrate the specialisation process using example pointcuts. We use the program given in Fig. 3 in Sec. 2.

[9] There is no list representation for interfaces as they lack a common base interface. The abstraction of an interface is simply a variable. In this case, no information about the static type can be exploited in the specialization.

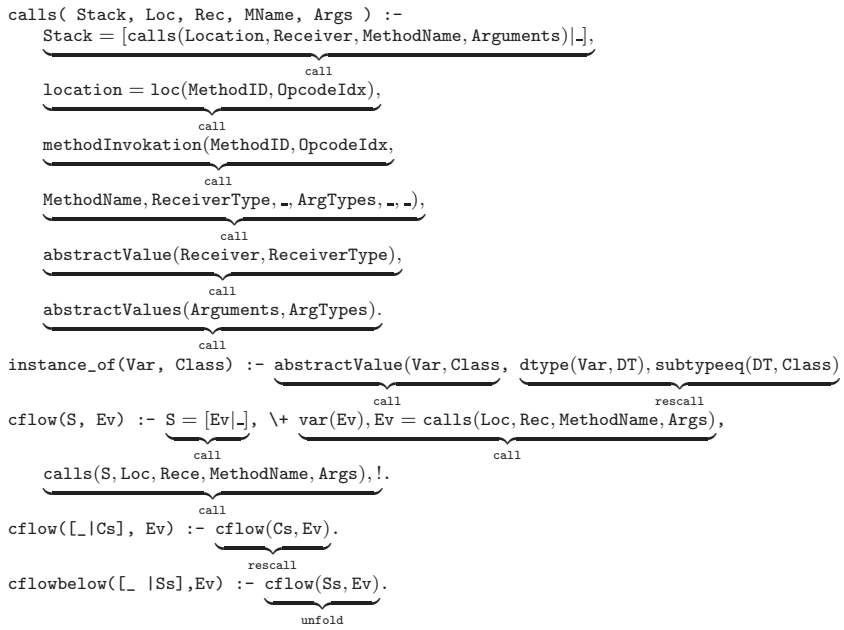

Fig. 7. Description of the `calls` predicate

The pointcuts that we want to discuss are named `pc1-pc3` and are stored in the Prolog source by means of `pointcut/2` facts. This predicate binds its second argument to the pointcut term identified by the name given as first argument.

The first pointcut we want to discuss is `pc1` = `calls(S,L,Rec,moveBy,_)`, selecting all method call joinpoints to a method called `moveBy`. The following two interpreter invocations shows the access to the pointcut predicates and the result of specialisation:

```
2 ?- pointcut(pc1,P).
P = pointcut([_G332, _G335], calls(_G332, _G335, _G340, moveBy, _G342)) ;
3 ?- specialisePointcut(pc1,Result).
Result = pointcut([ [calls(loc(2, 2), _G394, moveBy, [prim(int), prim(int)]),
 calls(loc(_G608, _G609), _G604, test, _G606)|_G529], loc(2, 2) ], true) ;
Result = pointcut([ [calls(loc(2, 3),
 value([ref('java.lang.Object'), ref('shapes.Line')|_G644], _G620, _G621),
 moveBy, [prim(int), prim(int)]), calls(loc(_G608, _G609), _G604, test, _G606)|_G529],
 loc(2, 3) ], true)
```

The lengthy output is a result of the partial instantiation of the callstack parameter and the binding of values to type abstractions. The shadows location and the residual pointcut are marked with a frame in both results. Both residual pointcuts are `true`, meaning that there is no dynamic check required at the shadow. The locations `(2,2)` and `(2,3)` refer to lines 29 and 30 in Fig. 3, respectively.

In the next example we show the effect of constraining the set of possible types of a variable. In the pointcut `pc2`, only calls to a `moveBy` method on a value, which is an instance of a `'shapes.Point'` at runtime. Calculating the shadows gives

```
5 ?- pointcut(pc2,P).
P = pointcut([_G338, _G341], (calls(_G338, _G341, _G349, moveBy, _G351),
                              instance_of(_G349, 'shapes.Point')))
6 ?- shadows(pc2,S).
S = [ (loc(2, 2), subtypeeq(_G383, ref('shapes.Point')))]
```

The location `(2,3)` is not a shadow of this modified pointcut, as the static type of the receiver is `shapes.Line`. The call at location `(2,2)` requires a runtime check to determine, if the receiver is an instance of `shapes.Point`.

In our last example, calls to `setX` in the control flow of a call to the method `test` are selected.

```
7 ?- pointcut(pc4,P).
P = pointcut([_G335, _G338], (calls(_G335, _G338, _G346, setX, _G348),
                              cflow(_G335, calls(_G353, _G354, test, _G356)))) ;
8 ?- shadows(pc3,S).
S = [ (loc(2, 5), true),
  (loc(6, 6), cflow([calls(loc(_G427, _G428), _G423, moveBy, _G425)|_G420],
                    calls(_G430, _G431, test, _G433))),
  (loc(6, 16), cflow([calls(loc(_G396, _G397), _G392, moveBy, _G394)|_G389],
                     calls(_G399, _G400, test, _G402))) ]
```

The location `(2,5)` corresponds to line 31 of Fig. 3, `(6,6)` and `(6,16)` to line 21 and 22, respectively.

Fig. 8 gives some figures for the speed of specialisation. The first three lines show the performance of specialising the example pointcuts. As these examples show, specialisation for small programs is quite efficient. The last four lines rows of To give an outlook on how our approach scales for medium sized programs, we tested specialisation of pointcuts on a bytecode toolkit project called *BAT* with about 800 types (classes+interfaces) and a bytecode size of about 2,25 MB.

Pointcut	*Shadows*	*Time*
pc1	2	0.08 ms
pc2	1	0.12 ms
pc3	3	0.13 ms
callStringMethod	6,655	0.30 sec
ctor	3,187	0.32 sec
ctorRec	1,313	0.55 sec
ctorNotRec	1,874	0.50 sec

Fig. 8. Specialisation runtime

4 Weaving residual programs

Hitherto we have only tackled the problems of finding shadows and computing efficient residual pointcut programs. However, this is only one part of the weaving

process. What remains is to insert the residual pointcut checks into the bytecode. We identified the following possibilities to process the residual Prolog programs:

Using a Prolog interpreter. Under the assumption that a Prolog interpreter is part of the runtime environment, Java code can be inserted which calls this interpreter for the residual pointcut query, checks the solutions and possibly calls the advice. Alas, this solution would also require the whole pointcut library, the user defined pointcut predicates and the Prolog representation to be accessible at runtime. Although this approach is quite simple, the overhead of keeping a Prolog interpreter and the libraries available for the virtual machine may not be tolerable in practice. Still, there are many tools for embedding Prolog within Java (e.g., [4], [29]), so this is a definitely a feasible solution.

Transforming the Prolog program. In order to produce efficient Java code, there are in principle several possible avenues.

1. A first approach is to produce specialized Prolog code in a special subset of Prolog that can be efficiently translated to Java. E.g., one could try and ensure that all the residual code is in a form similar to Mercury [24] which can be compiled into efficient imperative code.
2. Another solution is to ensure that the specialized code is close to *abstract machine code* or assembly code. This can be achieved by threading the environment of the interpreter via definite clause grammars; see [27] for more details and a worked out case study.

Certain parts of the residual program, however, could be treated in a special way. Residual predicates that refer to entities which are present in the Java virtual machine, like the callstack, argument values of a call or the actual type of a value. It is a promising idea to include a way to make this information *directly* accessible from the Java virtual machine. Calls to those predicates could then be translated directly into special bytecode instructions which have to be executed by an augmented virtual machine. The analysis of such techniques and their efficient implementation is part of ongoing research.

5 Related work

This is not the first work to use partial evaluation techniques in the context of pointcuts. Masuhara et al. have proposed a model where an aspect-oriented compiler is generated from a Scheme interpreter of the AO language using partial evaluation of Scheme programs [21]. Hence this work assumes that an interpreter for the whole base language is available. Also, the execution speed of a partially evaluated interpreter cannot keep up with today's optimizing compilers and virtual machines. Hence our work takes a different approach which does not require an interpreter for the language and with which programs can still be executed on optimizing virtual machines.

Ostermann, Mezini and Bockisch [22] present Alpha, a prototype language with a very expressive logic-based pointcut language. Alpha's pointcut language served as the base of our pointcut language.

The work discusses ALPHA pointcuts in comparison with their equivalent AspectJ pointcuts w.r.t. to robustness under changes in the structure of the software system. It is argued that expressiveness of ALPHA pointcuts allows the programmer to express pointcuts on a higher abstraction level, which lowers their coupling to actual syntax elements.

In addition, an approach based on abstract interpretation of pointcut queries is presented, which aims primarily at the reduction of space usage. This is achieved by collection all references to events used in the pointcut queries and restrict event generation to the corresponding expressions.

Our work goes beyond [22] in that we give a realistic approach to implement (a subset) of such an expressive pointcut language in the context of Java, a commercial programming language.

Walker and Viggers [25] discuss *temporal pointcuts*, called history patterns or *tracecuts*, to enrich the AspectJ [2] pointcut language with the ability to reason about former calls and their temporal relations. Moreover, data that has been passed as an argument can be accessed by the advice as it could be done via variable binding in our language. *Tracecuts* are patterns that are matched against a *history* of calls by a finite automaton. The implementation translates a program with tracecuts into AspectJ source code. Although more information about the computation history is available, the expressiveness of the pointcut language is very limited in contrast to our approach.

In [1], Allan et al. discuss the extension of the AspectJ language to be a able to express sequences of "classic"' AspectJ pointcuts. The language has been implemented as part of the Aspect-Sandbox (ASB) project, which aims at easing the implementation and testing of new features in aspect oriented languages. The extended language allows a sequencing pattern of ordinary AspectJ pointcuts to be considered as a pointcut and, additionally, allows the use of variables to bind and constraint terms very similar to those in logic based pointcut languages: when a variable is bound to a certain value at one event in the sequence, any other use of this variable at a later event must point to the same value, otherwise the whole sequence does not match. The implementation of shadow computation and optimization remains hand-coded, which is the main difference to the approach we presented.

Goldsmith et al. [9] present a framework to automize the instrumentation of source code to find static and dynamic pattern in programs. The language *PQTL* they introduce is basically a subset of SQL which operates on a database representing the program trace. In the database, each type of event is represented as a table, include timing information (much like the timestamps used in our work) for each event. Since the query language is built on SQL, dependencies between events must be expressed in terms of *JOIN*s and SQL logical connectives. This makes queries harder to understand as one has to consider the table layout and the SQL syntax, which is less predicative than Prolog.

The *PARTIQLE* compiler for PQTL uses timing graphs and a set of rules to determine the instrumentation sites in the application based on the queries and constructs minimal runtime data structures for the event database based on the

properties and events used in the query. The program is then instrumented at byte code level with instructions to add the required event data to the database and to check for event data that can be deleted. By the use of timing graphs for the partial order of events, so called *retention checks* are generated which prevent events to be added to the database which, at that point in the execution, cannot match any *join* in the query.

The difference between the PARTIQLE system and our approach lies in the expressiveness and extensibility of the pointcut language: PQTL can recognize patterns formulated in a very limited and fixed language, whereas in our language arbitrary predicates over the callstack can be expressed and user-defined pointcuts can be added to the pointcut language.

A work that also targets at detection of wrong behavior, may it be static or dynamic, is discussed in [20] by Martin et al. The PQL language has a more Java-like syntax to specify query which allows to define named queries and to use them to build more complex and even recursive queries, which provide a kind of *Kleene-star operator*. PQL queries are composed of the primitives method call, field access, object creation and the end of the program as well as negation, matching another query and partial-order matching of events.

Although the language can match context-sensitive patterns over the execution trace, the pattern language is fixed and is - in contrast to our language - very limited.

6 Conclusions and Future Work

We have presented a generic and extensible framework for finding pointcut shadows in Java programs. For this we have used logic programming, together with associated analysis and specialisation tools.

The framework is extensible at different points: the joinpoint model can be extended by adding new events or the modification of existing ones. Furthermore, new program models and pointcut predicates can be added to provide the programmer with a more domain specific language. This flexibility has the advantage, that the weaving and optimization layer of compilers does not have to be re-implemented when the pointcut language changes. The level of abstraction used in the approximation of the runtime behavior can be varied to switch between fast compile-test cycles and more accurate - but slower - compilation.

As we have demonstrated, the performance of our framework scales reasonable with program size. However, there are plenty of promising goals for future work: the integration into real compiler architectures and the weaving for special virtual machines are the most important problems which have to be solved.

Furthermore, there are different optimisation options, which have not yet been integrated: Static analysis techniques based on the Prolog bytecode representation can be used to compute better approximations of values and the callstack. Closer approximations do not only speed up the evaluation process but can also result in much more optimized residual pointcuts. Static information, for instance the subtyping relation, static events and abstract values, can

be memoized. That is, after the first computation the results are cached and reused in subsequent calls. Tabled Prolog systems - like XSB [28] - have internal support for this kind of optimization. Finally, the use more advanced technologies, like faster Prolog systems instead of SWI-Prolog and the full-fledged LOGEN partial evaluation system are likely to speedup the optimisation.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM Press.
2. AspectJ Home Page. `http://www.eclipse.org/aspectj/`.
3. C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.
4. M. Calejo. Interprolog: Towards a declarative embedding of logic programming in java. In J. J. Alferes and J. A. Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 714–717. Springer, 2004.
5. S. Chiba and K. Nakagawa. Josh: an open aspectj-like language. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111, New York, NY, USA, 2004. ACM Press.
6. S.-J. Craig, J. Gallagher, M. Leuschel, and K. S. Henriksen. Fully automatic binding-time analysis for Prolog. In S. Etalle, editor, *Proceedings LOPSTR 2004*, LNCS 3573, pages 53–68. Springer-Verlag, August 2004.
7. M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *Second ASIAN Symposium on Programming Languages and Systems (APLAS)*. LNCS, 2004.
8. M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. In B. Werner, editor, *Eleventh Working Conference on Reverse Engineering*, pages 182–191, Delft, Netherlands, November 2004. IEEE Computer Society.
9. S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 385–402, New York, NY, USA, 2005. ACM Press.
10. K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *AOSD 2003 Proceedings*, pages 60–69. ACM Press, 2003.
11. S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving. In *Proc. AOSD 2004*. ACM Press, 2004.
12. R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proc. European Conf. Object-Oriented Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427. Springer-Verlag, 2003.
13. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference*

on Object-Oriented Programming, pages 327–353, London, UK, 2001. Springer-Verlag.
15. K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Foundations of Aspect-Oriented Languages workshop (FOAL'05), Chicago, USA, 2005.*, 2005.
16. R. Lämmel. A semantical approach to method-call interception. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55, New York, NY, USA, 2002. ACM Press.
17. M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specializing interpreters using offline partial deduction. In K.-K. L. Maurice Bruynooghe, editor, *Program Development in Computational Logic*, pages 341–376. Springer Verlag, LNCS 3049, November 2004.
18. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.
19. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press.
20. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press.
21. H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC2003), LNCS 2622*. Springer, 2003.
22. K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP'05: European Conference on Object-Oriented Programming*. Springer LNCS, 2005.
23. D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of AOSD'03*. ACM, 2003.
24. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.
25. R. J. Walker and K. Viggers. Technical report.
26. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.
27. Q. Wang, G. Gupta, and M. Leuschel. Towards provably correct code generation via Horn logical continuation semantics. In *PADL'05, Springer LNCS 3350*, pages 98–112, 2005.
28. XSB Home Page. `http://xsb.sourceforge.net/`.
29. Q. Zhou and P. Tarau. Garbage Collection Algorithms for Java-Based Prolog Engines. In V. Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003*, pages 304–320, New Orleans, USA, Jan. 2003. Springer, LNCS 2562.