Partial Evaluation of Pointcuts

Karl Klose[1], Klaus Ostermann[1], and Michael Leuschel[2]

[1] Darmstadt University of Technology, Germany
{klose,ostermann}@st.informatik.tu-darmstadt.de
[2] University of Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de

Abstract. In aspect-oriented programming, 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 shadow and check computation are specific for every pointcut designator. This makes it very tedious to extend the pointcut language.
We propose the use of declarative languages, together with associated analysis and specialisation tools, to implement powerful and extensible pointcut languages. More specifically, we propose to *synthesize* (rather than program manually) the shadow and dynamic check algorithms. With this approach, it becomes easier to implement powerful pointcut languages efficiently and to keep pointcut languages open for extension.

1 Introduction

Aspect-oriented programming (AOP) eases the modularization of crosscutting concerns in a single module called an *aspect*. *Pointcuts* are used to describe at which point in the execution an aspect affects the execution of the basic program. The points that can be selected by a pointcut are called *joinpoints*. 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. Pointcuts are often used to control the execution of *advice*. An advice is executed at every point in the execution which triggers the associated pointcut. Although this is currently the primary usage of pointcuts, they can be used for a wide range of purposes, such as reverse engineering [6], detection of application errors [14], or flexible instrumentation of applications [7].

The first pointcut languages such as those in early versions of AspectJ [10] were static in that pointcuts could be mapped directly to locations in the source code of the underlying program. Recently, there is a trend towards more dynamic pointcut languages which can quantify over dynamic information such as the callstack [20, 17], dynamic argument values [8], the full execution trace of the application [1, 16], the structure of the dynamic heap [16], or even the future of the execution [11]. Such complex dynamic pointcuts cannot easily be mapped to places in the source code.

The most common approach to implement dynamic pointcuts is to identify a set of pointcut *shadows* - places in the code, where the pointcut is *potentially* triggered - and to insert dynamic checks at these places. 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 [8, 4, 16, 5].

The contributions of this paper are as follows: We propose a *generic* approach to finding shadows and generating dynamic checks, where the algorithms for finding shadows and computing dynamic checks are synthesized from the pointcut specification rather than programmed manually. To this end, we propose the use of declarative languages, together with associated analysis and specialisation tools—in particular partial evaluation—to implement powerful and extensible pointcut languages. This is the first work to embed the shadow search and dynamic check generation problem into the framework of partial evaluation. Our measurements show that our approach scales to reasonably large programs and 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. Sec. 5 discusses related work and Sec. 6 concludes.

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.

We have built a converter which transforms Java bytecode into a Prolog representation of the bytecode. Fig. 2 illustrates how the converted example from Fig. 1 looks like.

```
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,
29       int dx, int dy){
30     s.moveBy(dx,dy)
31     l.moveBy(dx,dy);
32     l.getP1().setX(42);
33   }
34 }
```

Fig. 1. 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 method(6,ref('shapes.Line'),
14   'moveBy',public,false,
15   false,false,false,false,false,
16   [prim(int),prim(int)],void).
17  ...
18 def(6,2,21,ref('shapes.Point'),p4,
19   get(ref('shapes.Point'),'p1',
20   ref('shapes.Line'),thisValue)).
21 def(6,3,21,ref('shapes.Point'),p6,
22   get(ref('shapes.Point'),'p1',
23   ref('shapes.Line'),thisValue)).
24 def(6,4,21,prim(int),p7,
25   invokeFunc('getX',ref('shapes.Point'),
26   p6,[],[],prim(int))).
27 def(6,5,21,prim(int),p9,
28   add(p7,param(1))).
29 invokeProc(6,6,21,'setX',
30   ref('shapes.Point'),p4,
31   [prim(int)],[p9]).
32  ...
33 return(6,22,25).
```

Fig. 2. Prolog encoding

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 relationships 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 (which are used to represent intermediate results) 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`. Each bytecode instruction starts with a number identifying the method which contains the instruction and a number denoting the position of this instruction in the method body. The third argument identifies the line number in the source code[3].

2.2 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 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.

In order to describe the matched 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 positions. A location is a pair `loc(MethodNumber,InstrNumber)` which represents the method- and instruction number as given in the bytecode. 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 functor `calls`. 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`:

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

The location `loc(6,6)` in the call to `Point.setX` corresponds to source code line 21 (Figure 1) and to the bytecode instruction at line 29 in Figure 2.

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 bytecode. Values are encoded as pairs which consists of a type and an address or primitive value like boolean or integer. The ι_n expressions are object references in the runtime environment. The representation of

[3] Please note that there can be multiple bytecode instructions for a line of sourcecode.

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 specialisation 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 specialisation process from the pointcut programmer (as well as leading to cleaner code).

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

2.3 The Pointcut Library

So far, we have seen how pointcuts can be formulated using the representation of the bytecode and of the callstack 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 callstack and the byte code.

```
calls( Stack, Location, Receiver, MethodName, Arguments ) :-
 Stack = [calls( Location, Receiver, MethodName, Arguments ) | _ ].
% cflow/2: succeeds if the callstack contains a given event
cflow(Stack, Ev) :- member(Ev,Stack), !.
% cflowbelow/2: Like cflow/2, but excludes the current jointpoint(event)
cflowbelow([_|Cs], Ev) :- cflow(Cs, Ev).
% directSubtype/2: A is a direct subtype of B
directSubtype( A, B ) :- class(_, A, _, _, _, B) ; interfaces(A,B).
% subtype/2: transitive closure of directSubtype/2
subtype(A,B):-directSubtype(A,B) ; (directSubtype(A,C),subtype(C,B)).
% subtypeeq/2: reflexive closure of subtype/2
subtypeeq(A,B) :- A=B ; subtype(A,B).
% instanceof relations use subtype relation
instance_of(Val, Type) :- dtype(Val,T), subtypeeq(T,Type).
withinMethod( Location, MethID ) :- Location = loc(MethID,_),
 method(MethID,_,_,_,_,_,_,_,_,_,_,_),
 methodInvokation(Location,_,_,_,_,_,_).
```

Fig. 3. Excerpt from the pointcut library

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. An excerpt is given in Fig. 3. 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 callstack; `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 in terms of existing predicates and the bytecode representation. 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 specialiser 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.

2.4 Example Pointcuts

Fig. 4 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. 1. The pointcut `change` in line 1 describes the points in the execution, where the display should be updated and the advice in line 8 specifies that a call to `display.update` should be executed *after* such a modification (specified by `change`).

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

Fig. 4. Display updating in AspectJ

```
1 (calls(Stack,Loc,Target,setX,_),
2  stype(Target,'shapes.Point') );
3 (calls(Stack,Loc,Target,setY,_),
4  stype(Target,'shapes.Point') );
5 (calls(Stack,Loc,Target,moveBy,_),
6  instance_of(Target,'shapes.Shape') ),
7 \+ cflowbelow(Stack,calls(_,_,moveBy,_))
```

Fig. 5. Pointcut in Prolog

The first two conditions (Lines 2 and 3) 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 4 selects calls to the `moveBy` method with two integer arguments defined in the type `Shape` or any of its subtypes. This is expressed by the `+` sign appended to `Shape`. These conditions are combined by `||` meaning `or`, which selects any point that satisfies one of these 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 the `call` joinpoint in the control flow. This pointcut is combined with the other three by the `&&` operator meaning *and* (or intersection). Fig. 5 shows how the same pointcut can be expressed in our pointcut language.

In order to illustrate the effect of specialisation, we will now consider a few pointcuts and the result of their specialisation, without talking yet about how the specialisation actually works.

- `calls(Stack,Loc,_,setX,_),withinMethod(Loc, MethID),method(MethID,_,_,public,_,_,_,_,_,_)`
  Shadows: (21,`true`), (23, `true`)
- `calls(Stack,Loc,R,moveBy,_), instance_of(R,` 'shapes.Line')
  Shadows: (30, `dtype(R,T), subtypeeq(T,`'shapes.Line')), (31 `true`)
- `calls(Stack,Loc,_, setX, _), cflow(Stack, calls(_,_,moveBy,_))`
  Shadows: (21, `true`), (23, `true`), (32, `cflow(Stack,calls(_,_,moveBy,_))`)

Fig. 6. Example pointcuts and their shadows and dynamic checks

Fig. 6 shows a few sample pointcuts and the result of specialising them with the example program from Fig. 1. Shadows are given as pairs (line number from Fig. 1[4], residual check).

The first pointcut selects all calls of a `setX` method within a public method. The relation between the method and the call is expressed in terms of the location (`Loc`) of the 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`.

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 specialisation. 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 callstack during specialisation. Whereas the first shadow requires a dynamic check, the second (and third) 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 callstack is an important parameter for the specialisation in computing residual pointcuts.

[4] In the actual implementation, the method/instruction indexes from the bytecode are used for this purpose.

3 The Specialisation Framework

Our specialisation 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 part is given by the representation of the program, which determines the possible static contexts in which the pointcut may be evaluated. Specialisation is then 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*, i.e. they can be (safely) evaluated at specialisation time.

In this section we present the partial evaluation of pointcut queries with respect to the program source.

3.1 The Specialiser

Program specialisation is a technique to specialise a given general purpose program for certain specific application area. Partial evaluation [9] 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) and leaving a residual program that only contains the dynamic checks. The partial evaluation (or *specialiser*) tool used throughout this work is based on the core of the offline specialiser presented in [12] and is thus similar to the core of LOGEN [13][5]. To control the behavior of the specialiser, an annotated form of the program has to be provided. We use the following three annotations of those described in [12]: `call` evaluates the goal using the prolog interpreter, `rescall` leaves the goal in the residual program and `unfold` replaces the goal by the residual program obtained from specialising the (annotated) body of the predicate.

There are basically two alternatives to obtain the annotations for a clause: *online specialisers* generate the annotations on the fly while *offline specialisers* use annotations provided by the user or a generator.

Although being based on an offline specialiser, our system does not require the programmer to annotate most of her pointcuts manually, but we rather use a set of rules for the standard predicates of the pointcut library. These rules are used to perform the annotation automatically before specialisation. For predicates that do not have a corresponding rule in the database the `rescall` annotation is used by default. Only in the case where these annotations are not optimal from the programmer's view, should he annotate the program himself, for example, when introducing user-defined predicates.

3.2 Approximation of Runtime Entities

In the scenario of pointcut specialisation, only the static part of the program is available, i.e., the class, interface and field declarations and a set of bytecode

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

instructions. However, our pointcut language allows to quantify over runtime conditions. The easiest way to handle runtime values like the actual types of values is to generate all possible instantiations and explore them by backtracking. Because this approach does not scale well for large programs, we use approximations of dynamic entities instead. 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.

The values of variables are not accessible at specialisation time. Nevertheless specialisation should be able to benefit from the *static information* about the variables that can be retrieved from the programs bytecode. We use an approach 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 most general (class) type of this set. To make this abstraction compatible with unification, we encode this most general type of the variable as an *open list* containing all its super classes[6]. 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` which is a subtype of `shapes.Line` would be encoded as `['java.lang.Object','shapes.Line','shapes.Arrow'|_]` and the unification of both would 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. The predicate `abstractValue(V,Class)` is used to bind a variable `V` to an abstract value of type `Class`.

For the approximation of the callstack, we use the notion of *static events* for approximations of the real runtime events which have the same structure as dynamic events, but contain variables or approximations for the runtime information. Using the static event, we approximate the callstack at a given location by a list containing the static event as first element. Furthermore, the second element of the callstack must be a call to the method containing that location. For the example callstack in the last section we can thus give the following approximation:

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

Better approximations that contain more elements or more precise type information can be generated by using the call and control flow graph of the program. As the construction of the application's callgraph can be very costly, it is desir-

[6] The approximation of interfaces is simply a variable as they lack a common base interface.

able to be able to control the amount of approximation. In our framework this can easily be accomplished by modifying the predicate which produces the stack approximation.

3.3 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.

The following code listing shows the description of the `instance_of` and the `cflow` predicate:

```
instance_of(Var,Cls) :- abstractValue(Var,Cls),dtype(Var,DT),subtypeeq(DT,Cls).

cflow(S,Ev) :- S = [Ev|Cs], (\+ var(Ev), Ev = calls(L,R,M,A),calls(S,L,R,M,A),! ;cflow(Cs,Ev)).
```

(call: `abstractValue`, `dtype`; rescall: `subtypeeq`; call: `S = [Ev|Cs]`, `var(Ev)`, `Ev = calls(L,R,M,A)`; call: `calls(S,L,R,M,A)`; rescall: `cflow(Cs,Ev)`)

The first subgoal of `instance_of` is evaluated at specialisation time and checks if the variable can be unified with the abstract type of `Cls`; otherwise the instance check can be refuted at specialisation time. The second subgoal binds `DT` to the variable for the dynamic type of `Var` to be used in the subtype check which is left as residual program by the third subgoal.

The first clause of the `cflow` description checks (at specialisation), if the event `Ev` is at the top of the stack. In this case, no residual program is necessary. Otherwise, for example, if the head of the list is a variable, a call to the `cflow` predicate is left as residual program by the second clause.

3.4 Example Specialisations

After introducing the specialisation and approximation techniques, we demonstrate the specialisation process using example pointcuts. We use the program given in Fig. 1 in Sec. 2.

We will discuss three pointcuts (`pc1`-`pc3`), accessible via `pointcut/2` and the result of their specialisation. These examples show a statically determinable shadow, a pointcut leaving a residual type check and an example for the results of specialising the `cflow` predicate.

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 show the access to the pointcut predicates and the result of specialisation:

```
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. 1, 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 are selected that go to an instance of '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 call at location `(2,2)` requires a runtime check (via `subtypeeq`) to determine, if the receiver is an instance of `shapes.Point`.

The location `(2,3)` is not a shadow of this modified pointcut, as the static type of the receiver is `shapes.Line` and its abstract type thus cannot be unified with the abstract type of `shapes.Point` used in the pointcut.

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

```
7 ?- pointcut(pc3,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. 1, `(6,6)` and `(6,16)` to line 21 and 22, respectively.

The specialisation of the three example pointcuts is quite fast (about 0.1 ms), which is no surprise given the size of the program. To demonstrate the feasibility of our approach for larger 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. We used some quite general pointcuts which return a large number of shadows to test the performance of our specialisation tool: `callStringMethod` matches each call to a method of the class `java.lang.String`, `ctor` matches all invocations of a constructor, `ctorRec` matches all invocations of a constructor inside another constructor, and `ctorNotRec` matches all invocations of a constructor *not* inside another constructor. Fig. 7 shows the results of specialising these pointcuts[7].

[7] Tests performed with SWI-Prolog on a 2.8GHz Windows XP machine.

Pointcut	*Shadows*	*Time*
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. 7. Specialisation runtime

3.5 Language Extension

An important feature of our framework is the extensibility of the pointcut language. This is a necessary property to write aspects on an abstract level, as stated in [16]. Extensions to the language can be written by the programmer to adapt the language to a single program or implemented as domain-specific pointcut library to be used within a whole class of applications.

Extending the pointcut language requires the follow steps: 1) its implementation must be added to the pointcut library to make it available to predicates that call it at runtime, 2) the annotation of its body has to be provided as a rule for unfolding and 3) an `unfold`-annotation for the predicate has to be added to the annotation database, which is used by the rule generator.

As an example, we extend our pointcut language with a predicate to detect loops in the callstack. A loop is the re-occurence of a method call to the same method on the same object and with the same argument values. Below is the annotated implementation of this predicate.

```
loop_detect(S,L) :- calls(S,L,Rec,Method,Args), cflowbelow(S,calls(_,Rec,Method,Args)).
                    \_____call_____/            \_____________unfold_____________/
```

To integrate this predicate, the predicate definition without the annotations has to be added to the pointcut library and the annotated form has to be stored into the annotation database (we omit the technical details for brevity).

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:

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. 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., [3], [22]), so this is a definitely a feasible solution. In order to produce efficient Java code, there are in principle several possible avenues. A first approach is to produce code in a special subset of Prolog that can be efficiently translated to Java. For example,

one could try and ensure that all the residual code is in a form similar to Mercury [18] which can be compiled into efficient imperative code. 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 [21] for more details and a worked out case study.

Certain parts of the residual program, for example predicates that refer to entities which are present in the Java virtual machine, like the callstack, or argument values, could be treated in a special way. 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 for an augmented virtual machine. The analysis of such techniques and their efficient implementation is part of ongoing research.

5 Related Work

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 [15]. 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. 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 [16] 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. An implementation approach based on abstract interpretation of pointcut queries is presented, which aims primarily at the reduction of space usage. Our work goes beyond [16] in that we give a realistic approach to implement (a subset) of such an expressive pointcut language in the context of Java, a non-toy programming language.

Walker and Viggers [19] discuss *temporal pointcuts*, called *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. Although more information about the computation history is available, the expressiveness of the pointcut language is very limited in comparison to our approach.

In [1], Allan et al. discuss the extension of the AspectJ language to be able to express sequences of "classic" AspectJ pointcuts. The extended language allows a sequencing pattern of ordinary AspectJ pointcuts to be considered as a pointcut and to bind values to variables which are unified on later occurrence. The implementation of shadow computation and optimization remains hand-coded, which is the main difference to the approach we presented.

Goldsmith et al. [7] present *PARTIQLE*, a framework to automatize 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 for each event. The relations between events are expressed using *JOIN*s and SQL logical connectives. The difference between PARTIQLE 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.

Another work targeting at detection of statically or dynamically wrong behavior, is discussed in [14] by Martin et al. The PQL language has a Java-like syntax which allows to define named queries and to use them to build more complex and even recursive queries. 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 comparison to our language - limited in its expressiveness.

6 Conclusions

We have presented a generic and extensible framework for finding pointcut shadows in Java programs using 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 modifying existing ones, new program models and pointcut predicates can be added to provide the programmer with a more domain specific language and 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. Furthermore, as we have demonstrated, the performance of our framework scales reasonable with program size.

Acknowledgements This work was partly supported by the feasiPLe project financed by the German Ministry of Education and Research (BMBF).

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 *Proceedings of OOPSLA 2005*, pages 345–364, New York, NY, USA, 2005. ACM Press.
2. AspectJ Home Page. http://www.eclipse.org/ aspectj/.
3. M. Calejo. Interprolog: Towards a declarative embedding of logic programming in java. In *Proceedings of JELIA 2004*, pages 714–717, 2004.

4. S. Chiba and K. Nakagawa. Josh: an open aspectj-like language. In *Proceedings of AOSD 2004*, pages 102–111, New York, NY, USA, 2004. ACM Press.
5. M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *Proceedings of APLAS 2004*. Springer LNCS, 2004.
6. 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.
7. S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of OOPSLA 2005*, pages 385–402. ACM Press, 2005.
8. K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *AOSD 2003 Proceedings*, pages 60–69. ACM Press, 2003.
9. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
10. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, pages 327–353. Springer-Verlag, 2001.
11. 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.
12. 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*. Springer Verlag, 2004.
13. 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.
14. M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of OOPSLA 2005*, pages 365–383, New York, NY, USA, 2005. ACM Press.
15. H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of CC 2003*. Springer, 2003.
16. K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of ECOOP 2005*. Springer LNCS, 2005.
17. D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of AOSD'03*. ACM, 2003.
18. 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.
19. R. J. Walker and K. Viggers. Communication history patterns: Direct implementations of protocol specifications. Technical report, University of Calgary, 2004.
20. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS 2004*, 26(5):890–910, 2004.
21. Q. Wang, G. Gupta, and M. Leuschel. Towards provably correct code generation via Horn logical continuation semantics. In *Proceedings of PADL 2005*, pages 98–112, 2005.
22. Q. Zhou and P. Tarau. Garbage Collection Algorithms for Java-Based Prolog Engines. In *Proceedings of PADL 2003*, pages 304–320, New Orleans, USA, 2003. Springer, LNCS 2562.