# Aspect-Aware Points-to Analysis

Qiang Sun
Department of Computer Science
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
sun-qiang@sjtu.edu.cn

Jianjun Zhao
Department of Computer Science
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
zhao-jj@sjtu.edu.cn

## Abstract

*Points-to analysis is a fundamental analysis technique whose results are useful in compiler optimization and software engineering tools. Although many points-to analysis algorithms have been proposed for procedural and object-oriented languages like C and Java, there is no points-to analysis for aspect-oriented languages so far. Based on Andersen-style points-to analysis for Java, we propose flow- and context-insensitive points-to analysis for AspectJ. The main idea is to perform the analysis crossing the boundary between aspects and classes. Therefore, our technique is able to handle the unique aspectual features. To investigate the effectiveness of our technique, we implement our analysis approach on top of the ajc AspectJ compiler and evaluate it on nine AspectJ benchmarks. The experimental result indicates that, compared to existing Java approaches, the proposed technique can achieve a significant higher precision and run in practical time and space.*

## 1. Introduction

Points-to analysis is a useful technique which is widely used in compiler optimization and program analysis tasks. The goal of points-to analysis is to compute a points-to relation between variables of pointer types and allocation sites. The recent work [12, 13, 15] for Java has shown that flow- and context- insensitive points-to analysis can be efficient and practical for large Java systems.

Aspect-oriented programming (AOP) has been proposed as a technique for improving separation of concerns in software design and implementation [1, 3, 10]. AOP works by providing explicit mechanisms for capturing the structure of crosscutting concerns such as exception handling, synchronization, performance optimizations, and resource sharing, which are usually difficult to express clearly in source code using existing programming techniques. AOP can also control the code tangling problem, making the underlying concerns more apparent, and enable the software more easy to develop, maintain, and evolve.

AspectJ [5], one of the most widely used AOP languages, is a seamless aspect-oriented extension to Java by adding some new constructs such as *join point*, *advice*, and *aspect*. An AspectJ program can be divided into two parts: *base code* which includes classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modeling crosscutting concerns in the program. With the inclusion of join points, an aspect woven into the base code is solely responsible for a particular crosscutting concern, which raises the system's modularity.

Since the executable code of AspectJ programs is pure Java bytecode produced by an AspectJ compiler, an obvious approach is to apply directly the existing Java analysis to the bytecode of AspectJ programs, and then map the results back to the source code. However, as pointed out in [19], there is a significant discrepancy between the AspectJ source code and the woven Java bytecode, which makes the analysis results quite imprecise. In our experimental study, this naive approach typically produces 2.5 times larger result set on average.

An alternative approach used in our proposed technique is to perform source-code-level analysis for AspectJ programs. However, the source-code-level analysis is complicated by AspectJ semantics. Since an AspectJ program consists of not only classes but also aspects that can crosscut many classes, any points-to analysis should deal with not only classes, but also aspects that may affect these classes. To this end, analysis starts from an entry point of a class and if there is an aspect that may affect the class being analyzed, the analysis should move into the aspect to compute the points-to information occurred due to effect of the aspect. Moreover, because aspects in AspectJ are transparent to classes, that is, if you just look at the source of a class, you can not know which aspect should be woven into the class. Therefore, in order to perform points-to analysis for AspectJ programs correctly and precisely, a new points-to analysis technique which can handle the unique aspectual features is needed.

In this paper, we propose a flow- and context-insensitive points-to analysis for AspectJ programs based on an Andersen-style points-to analysis for Java [15]. We show how the analysis can cross the boundary between a class and its related aspects, and therefore produces fine precise results as for AspectJ programs. The main features of our approach are to first get information related to those statements that use join points, and then achieve the information of aspects that will be attached to these statements through the join points.

The main contributions of our work are threefold.

- We define a general-purpose points-to analysis for AspectJ which extends the semantics of Rountev et al.'s points-to analysis for Java [15] which is originally derived from Andersen's points-to analysis for C [6].
- We implement a points-to analysis tool for AspectJ programs based on the ajc compiler.
- We perform an empirical evaluation on nine AspectJ benchmarks. The experimental result indicates that a significant higher precision can be achieved compared to the existing Java approach.

The rest of the paper is organized as follows. Section 2 briefly introduces Andersen's points-to analysis for Java. Section 3 discusses the problems when applying existing points-to analysis to AspectJ. Section 4 defines our points-to analysis for AspectJ. Section 5 presents the details of our analysis steps. Section 6 evaluates our analysis tool by experimental results. Section 7 discusses some related work. Concluding remarks are given in Section 8.

## 2. Flow- and Context-Insensitive Points-to Analysis for Java

Several flow- and context-insensitive points-to analysis for Java have been proposed [12–15, 17], which are topically based on similar analysis for C. In this section, we focus on one analysis proposed by Rountev et al. [15] (we call it RMR analysis for short), which is derived from Andersen's points-to analysis for C [6].

To perform points-to analysis for Java programs, RMR analysis [15] defines three sets. The first is set $R$ which contains all reference variables in the analyzed program and also includes the static fields in classes. The second is set $O$ which contains the names of all objects that are created at object allocation sites; for each allocation site $s_i$, object name $o_i \in O$ is unique in whole program. The third is set $F$ which contains instance fields in the classes, but not includes the static fields. There are two types of edges in the points-to graphs. Edge $(r, o_i) \in R \times O$ represents that reference variable $r$ points to object $o_i$ and Edge $(\langle o_i, f\rangle, o_j) \in (O \times F) \times O$ represents that field $f$ of object $o_i$ points to object $o_j$.

Points-to graph is a directed multi-graph which can be used to represent the points-to information in a program. As an example, Figure 1 shows a sample program and Figure 2 shows its points-to graph. In Figure 2, there are nodes representing variables and objects and edges representing points-to relationships. $o_1$ and $o_2$ mean the objects created in the sample program. We represent the variables in whole program using the form of "$xx@xx@xx$". For example, $r@void\ set(Y\ r)@X$ means variable $r$ in the method

```
public class Y {} public class X {
    Y f,g,h;
    void set(Y r)
    {
        this.f = r;
    }
    public static void main(String[] args) {
        X p = new X();  // creating O1
        Y q = new Y();  // creating O2
        Y l;
        p.set(q);
        p.h = q;
        l = p.f;
    }
}
```

**Figure 1. A Java program.**

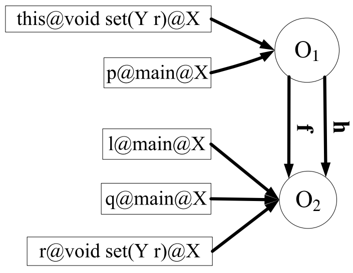

**Figure 2. A points-to graph for the Java program in Figure 1.**

$void\ set(Y\ r)$ of class $X$. $void\ set(Y\ r)$ represents the signature of the method which contains the method name, the formal parameter list and the return type of the method. The directed edges without any properties represent the points-to relationship from a variable to an object. The edges from an object to another means the field of one object points to another object, whose property contains the field information.

As summarized in [15], the statements kinds that can be handled by RMR analysis are listed below. In addition to these statements, RMR analysis can also deal with other kinds of statements such as calls to constructors and static methods by a similar way.

- Direct assignment: $l = r$
- Instance field write: $l.f = r$
- Instance field read: $l = r.f$
- Object creation: $l = new\ C$
- Virtual invocation: $l = r_0.m(r_1, \ldots, r_k)$

The semantics of RMR analysis can be defined according to a group of transfer functions [15]. These functions add new edges to points-to graphs during analysis. Each transfer function represents the semantics of a program statement. The function sets for different statements are shown in Figure 3 in the format $f(G, s) \Rightarrow G'$, where $s$ is a statement, $G$ is an input points-to graph, and $G'$ is the

- $f(G, l = newC) = G \cup \{(l, o_i)\}$
- $f(G, l = r) = G \cup \{(l, o_i) | o_i \in Pt(G, r)\}$
- $f(G, l.f = r) = G \cup \{(\langle o_i, f\rangle, o_j) | o_i \in Pt(G, l) \wedge o_j \in Pt(G, r)\}$
- $f(G, l = r.f) = G \cup \{(l, o_i) | o_j \in Pt(G, l) \wedge o_i \in Pt(G, \langle o_j, f\rangle)\}$
- $f(G, r_0.m(r_1, \ldots, r_n)) = G \cup \{resolve(G, m, o_i, r_1, \ldots, r_n) | o_i \in Pt(G, r_0)\}$
- $resolve(G, m, o_i, r_1, \ldots, r_n) =$ let $m_j(p_0, p_1, \ldots, p_n, ret_j) = dispatch(o_i, m)$ in $\{(p_0, o_i)\} \cup f(G, p_1 = r_1) \cup \ldots \cup f(G, l = ret_j)$

**Figure 3. Points-to effects of program statements for Andersen's analysis [15].**

resulting points-to graph. $Pt(G, x)$ represents the points-to set of $x$ in graph $G$. The solution computed by the analysis is a points-to graph that is the closure of the empty graph under the application of all transfer functions for program statements.

There are some obvious effects on the points-to graph for most statements. For instance, it will generate some new points-to edges from $l$ to all objects pointed to by $r$ for statement $l = r$, and for virtual call sites, we should compute the results for every receiver object pointed to by $r_0$. The $resolve$ function defined by the **let** expression (For example, let $x = t_1$ in $t_2$, means that the expression $t_1$ is evaluated and the name $x$ is bound to the resulting value while evaluating $t_2$ ), computes the points-to information at the call site. Function *dispatch* uses the class of the receiver object pointed to and the compile-time target of the call to get the actual method $m_j$ invoked at run-time. $p_0, \ldots, p_n$ are variables which represent the formal parameters of the method. For instance, $p_0$ is used to represent the implicit parameter *this* and $ret_j$ is used to represent the return values of $m_j$. All values returned by the method are assigned to an auxiliary variable, which makes each method have a unique return variable.

## 3. Motivating Example

We next discuss the problems that may occur, when applying RMR analysis to AspectJ programs.

Consider the program presented in Figure 4. When we perform points-to analysis for it using RMR analysis, the analysis does not care about the effect from the aspect `A` though it has potential impact on changing the points-to graph of the analyzed program. If we don't process the after advice in this example, $p.h$ will only points to $o_2$. However, the points-to set of $p.h$, in fact, contains $o_2$ and $o_3$, due to the effect of the after advice. This problem is caused by the so-called "obliviousness principle" of AOP languages [9], i.e., the class being analyzed does not know which aspects may or may not be woven into it. So if the analysis starts from a class, it will not consider the effects from aspects, and therefore, may lead to imprecise or even incorrect points-to information.

```
public aspect A {
    pointcut p1(X x, Y y):
        call(void X.set(Y))&&target(x)&&args(y);
    pointcut p2(X x):
        set(Y X.h)&&target(x)&&!within(A);
    before(X x,Y y):p1(x,y){
        x.g = y;
    }
    after(X x):p2(x)
    {
        Y l = new Y();  // creating O3
        x.g = l;
    }
}

public class Y {} public class X {
    Y f,g,h;
    void set(Y r)
    {
        this.f = r;
    }
    public static void main(String[] args) {
        X p = new X();  // creating O1
        Y q = new Y();  // creating O2
        Y l;
        p.set(q);
        p.h = q;
        l = p.f;
    }
}
```

**Figure 4. An AspectJ program.**

The semantics of the join points will affect the RMR model directly. The join points that should be taken into consideration are shown in Table 1.

| Join Point Kind | May Effect |
|---|---|
| constructor call | object creation |
| constructor execution | |
| method call | virtual invocation |
| method execution | |
| get | instance field read |
| set | instance field write |

**Table 1. Join point kinds effecting points-to analysis for AspectJ.**

At the join point with the form of constructor call or execution, advice implicitly invoked may do some operations which change the points-to graph. For example, there is an instruction $this.f = r$ in the advice, which means when an object is created, its $f$ field is written into object referenced by variable $r$. Comparing with the program without aspects, this object's $f$ field may point to more objects. The advices match the join points with the form of $get$ or $set$ may also effect the points-to graph. In such case, an advice can change the object written in or read from the field of object.

Another problem is that some new constructs such as advice, introduction, various kinds of pointcuts, and inter-type declarations are added to AspectJ. These constructs are different in nature from those in Java, and therefore should also be handled differently by the analysis.

In the rest of the paper, we propose our extension of the semantics of RMR analysis to perform points-to analysis for AspectJ programs.

## 4. A Semantics for Points-to Analysis of AspectJ

The semantics of our points-to analysis extends that of RMR analysis [15]. We extend this semantics to handle the special issues of AspectJ programs.

In order to keep backward-compatible with existing Java application. Our analysis for AspectJ is also defined in terms of three sets, i.e., $R$, $O$, and $F$. $R$ is the set of all reference variables in the analyzed program, including static fields both in classes and aspects. Each reference variable is represented in whole program. $O$ is the set of names for all objects created at object allocation sites in both aspects and classes; for each allocation site $s_i$, there is a unique object name $o_i \in O$ in whole program. $F$ is the set of all instance fields in both aspects and classes, not including static fields. Our analysis builds points-to graphs which contain two types of edges. Edge $(r, o_i) \in R \times O$ represents that reference variable $r$ points to object $o_i$. Edge $(\langle o_i, f \rangle, o_j) \in (O \times F) \times O$ represents that field $f$ of object $o_i$ points to object $o_j$. Figure 4 and Figure 5 show a sample program and its points-to graph respectively.

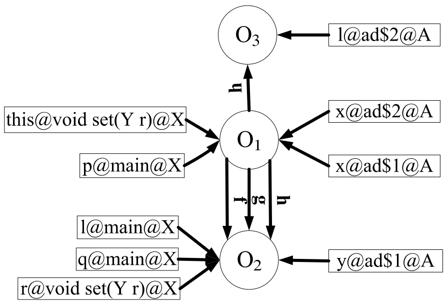

**Figure 5. A points-to graph for AspectJ program in Figure 4.**

The points-to graph of an AspectJ program is very similar with that of a Java program. The difference in the points-to graph of an AspectJ program is that it contains more variables and objects related with aspects. Since the implicit invocation of advice, the advice doesn't have a name. So in order to distinguish them, we use the sequence number of defined order to identify the advice within one aspect. For example, for $l@ad\$2@A$, $l$ is variable name, $A$ means aspect $A$, and $ad\$2$ represents the second advice of aspect $A$ in defined order.

In addition to the issues mentioned above, our analysis can also appropriately handle the language constructs that contribute to aspects in AspectJ. These constructs are:

- Pointcuts: $this(), target()$, and $args()$
- Advice: $after\ returning$ and $around$

The semantics of our analysis is also defined in terms of transfer functions that add new edges to points-to graphs during analysis. Each transfer function represents the semantics of a program statement. The functions for different statements are shown in Figure 3 in the format $f(G, s) \Rightarrow G'$, where s is a statement, $G$ is an input points-to graph, and $G'$ is the resulting points-to graph. $Pt(G, x)$ denotes the points-to set of $x$ in graph $G$. The solution computed by the analysis is a points-to graph that is the closure of the empty graph under the application of all transfer function for program statements.

Compared to the RMR analysis, our analysis considers the source of both classes and aspects. In order to deal with aspects in AspectJ, the analysis semantics should be extended for the following cases:

- There is a declaration of class inheritance for an aspect
- The context references of pointcuts and advice, and
- The return value for a piece of advice can be used as a variable to return

In the following, we will show how to handle these cases in our analysis.

### 4.1. To Handle Pointcut

We first define the parameter list which will be used later.

$$\begin{aligned} tList &::= T\ v, tList | T\ v \\ vList &::= v, vList | v \\ v &::= variable \\ T &::= type \end{aligned}$$

$tList$ represents the formal parameter list containing both variables and their types, while $vList$ means the list only includes the variables without their types.

When using pointcuts $this$, $target$, and $args$, there are some similar situations like parameter passing between methods for aspects. In such a case, some edges should be added to the points-to graph just like how we handle the relationships between formal and actual parameters between methods. Set $S_{cs}$ contains all the call sites in the source code. $site_i \in S_{cs}$, $site_i$ has the form as follows:

$$site_i : l = r_0.m(r_1, \ldots, r_n)$$

The pointcut for the places that the matched advices will be woven. A simple form of pointcut is defined.

$PC$ ::= **pointcut** $pcname(tlist)$ : $MD\&\&CD$
$D$ ::= **this**$(v)$|**target**$(v)$|**args**$(vlist)$
$CD$ ::= $D\&\&CD|D$
$MD$ ::= method-call or -execution designator

We define the actual parameter set, delivered by the call site, as set $S_m$.

$$S_m = \{r_1, \ldots, r_n\} \cup \{r_0\} \cup \{this\} \quad (1)$$

From above, $r_0$ means receiver object. When method $m$ is static, $\{r_0\}$ and $\{this\}$ are empty sets. We build up a formal parameter set $S_p$ for the call site's pointcut. $S_p$ contains all variables appearing in $tList$ of the pointcut declaration. According to the semantic of AspectJ language, there is a mapping $\varphi_1$ from $S_p$ to $S_m$, which will be illustrated as follows:

$$\varphi_1 : S_p \mapsto S_m \quad (2)$$

From the form (2) just mentioned, we get $dom(\varphi_1) = S_p$, $rank(\varphi_1) \subseteq S_m$. Through the mapping above we can make clear the relationships between actual parameters of call site and formal parameters of pointcut.

For example, in Figure 4, at the call site $p.set(q)$, the actual parameter set $S_m$ is $\{p, q, this\}$. We consider the pointcut

$pointcut\ p1(X\ x,\ Y\ y) : call(voidX.set(Y))$
$\&\&target(x)\&\&args(y);$

then we can get the $S_p$ and $\varphi_1$:
$S_p = \{x, y\}$,$\varphi_1(x) = p, \varphi_1(y) = q$.

## 4.2. To Handle Advice

Unlike method calls, in which formal parameter values are explicitly passed by the caller, an advice gets the context information from the environment where the call site as a join point is selected by the pointcut. In other words, the formal parameter values are provided by the pointcut. We define the advice signature form as follows:

$A$ ::= $AT(tList) : pcname(vList')$
$AT$ ::= **before**|**after**|**around**|**after returning**
$vList'$ ::= $v, vList'|T, vList'|T|v$

In the case that the advice only uses a subset of the parameter set provided by the matched pointcut, the locations of the unused parameter variables in $vList$ are occupied by the corresponding types, so we define the form $vList'$. We make an $S_a$ set for the formal parameters of advice. $S_a$ contains all variables appearing in $tList$ of the advice signature. Through the semantic of AspectJ language, we can get a mapping, $\varphi_2$, from $S_a$ to $S_p$:

$$\varphi_2 : S_a \mapsto S_p \quad (3)$$

From the form (3) just mentioned, we get $dom(\varphi_2) = S_a$, $rank(\varphi_2) \subseteq S_p$. From the product of two mappings, we directly have $\varphi$, the mapping relationship, getting from formal parameters of advice and actual parameters of call site.

$$\varphi = \varphi_1 \cdot \varphi_2 \quad (4)$$

For example, there is an after advice matching the pointcut $p1$ in Figure 4. The advice signature is as follows:

$$after(X\ xx, Y\ yy) : p1(xx, yy)$$

After computing $S_a$, $\varphi_2$ and $\varphi$ for this advice, we can get that:
$S_a = \{xx, yy\}, \varphi_2(xx) = x, \varphi_2(yy) = y.$
$\varphi(xx) = p, \varphi(yy) = q.$

We can get the mapping relationship of formal parameters of advice and actual parameters of call site. We use this approach to improve the precision of analysis. For example, in Figure 6, two before advices match the combined pointcuts which share the same part $p(c)$ in aspect $AS$. If we transfer the variables $l1$ and $l2$ from method $main()$ to two before advices through $pointcut\ p(C\ c)$, i.e. $l1 \rightarrow c, l2 \rightarrow c, c \rightarrow c1, c \rightarrow c2$, we can get that both $l1$ and $l2$ will be transferred to $c1$ and $c2$. If we compute the mapping from the formal parameters of before advice to actual parameters of call site, variables $l1$ and $l2$ will be directly assigned to $c1$ and $c2$ respectively.

```
public aspect AS {
    pointcut p(C c):call(* *.foo(*))&&args(c);
    before(C c1):p(c1)&&target(A)
    {
        // to do something ...
    }
    before(C c2):p(c2)&&target(B)
    {
        // to do something ...
    }
}

public class A {
    public void foo(C c)
    {
        // to do something ...
    }
}
public class B {
    public void foo(C c)
    {
        // to do something ...
    }
}
public class C {
    public static void main(String[] args) {
        C l1 = new C();
        C l2 = new C();
        A a = new A();
        B b = new B();
        a.foo(l1);
        b.foo(l2);
    }
}
```

**Figure 6. Parameter transfer effecting precision.**

As a result, for handling advice, we should add the following rules:

- $f(G, r_0.m(r_1, \ldots, r_n)) = G \cup \{resolve(G, m, s, a, o_i, o_j, r_1, \ldots, r_n) | o_i \in Pt(G, r_0) \wedge o_j \in Pt(G, this)\}$
- $resolve(G, m, s, a, o_i, o_j, r_1, \ldots, r_n) =$
  let $advice(b_0, b_1, p_1, \ldots, p_k) \ = \ dispatch(s, a)$

in $\{(b_0, o_i)\} \cup \{(b_1, o_j)\} \cup$
$f(G, p_1 = \varphi(p_1)) \cup \ldots \cup f(G, p_k = \varphi(p_k))$

where $m$ is a method name, $s$ is a statement, $a$ is the aspect to be woven, $o_1$ is a receiver object, $o_j$ is *this*, $r_i$ is a actual parameter, and $l$ is the reference variable that receives a return value. The formal parameters used by advice are $b_0, b_1, p_1, \ldots, p_k$. From the definition of $\varphi$, we get $\varphi(p_i) \in \{r_1, \ldots, r_n\}(1 \leq i \leq k, k \leq n)$. Also, we suppose that the transferred aspects have been determined. The above rules are used as an extension of the rules presented in Section 2, but do not replace them.

Extensions are also needed for after returning and around advice. Since an after advice can access the return value of a method, an edge for after returning advice should be added to the points-to graph during analysis. For the return value of the identifier $ret$ of the form $m(p_0, p_1, \ldots, p_n, ret)$, we should add an edge to the points-to graph according to the following rule:

$$f(G, ar = ret)$$

Where $ar$ is a variable of the after returning advice that can access the return value. For around advice, it determines by whether the advice contains the $preceed()$ call or not. If it contains a $proceed()$ call, we need to add some edges for the woven statements as well as the statements for the target aspects. If it contains no $proceed()$ call, we just need to add some edges for the statements of the target aspects. Moreover, if there is a return value, we should add an edge for the return value and the reference variables that receive it with the following rule:

$$f(G, l = aret)$$

Here $aret$ is the return value of around advice. Moreover, in addition to those edges mentioned above, some edges within the advice should also be created, we can use the same way to create this kind of edges as we do for before and after advice.

### 4.3. Advice Call Tree

To be more specific, when multiple advices match a join point, an advice call tree as shown in Figure 7 is constructed. Considering the advice precedence issue, we travel this tree in preorder, in which before advice is visited first which is followed by around advice after which stands after advice. There we can get these advices in the execution order of run-time. For example, there is a method call as follows:

$$Site_i : l = r_0.m(r_1, \ldots, r_n).$$

Call point represents the $Site_i$ statement operated by the program. Execution point indicates that the program has already reached the method $m$ to be executed. The pointcut $p$ contains the method-call designator $call(*.m(..))$, which will select the join point of call type (call point) at $Site_i$, afterwards three types of advices, before, around, after, are all matched by $p$. What's more, there is a $proceed()$ call in every around advice. Each tree level has at most one around advice, which calls advices of the next lower level of the tree by $proceed()$ call. At the execution point, there is a pointcut $p'$, which contains the method-execution designator $execution(*.m(..))$. The advices match to the join point with the execution type (execution point) by $p'$, whose execution after and before hand is quite similar. When we arrive the final $proceed()$ call of around advice, it calls the method $m$ through the $proceed()$ call.

Intra-aspect precedence rules are a bit complex. If two advices are defined in the same aspect, their precedence is determined by their defined orders and types. There are two main rules:

- **One of the pieces of advice is after advice**. In this case, the advice defined later in the file takes precedence .
- **Neither advice is of the after type**. In this case, the advice defined earlier in the file takes precedence.

Within an aspect, the types of advices that match the same join point lead to a sequence which can be represented by a regular expression:

$$after^*(before + around)^*after^*.$$

Since there is no precedence circularity, we can spread the discussion into inter-aspect from which the conclusion is also available.

We deal with the after advice in a particular way according to the advice precedence rules. The after advices which match to the same join point are called in two situations. Take the after advices matching the call point for example in Figure 7. One situation is that after the first around advice (around advice 1) is executed, the after advice $11, 12, \ldots, 1l$ will be called; the other situation is that the after advices are called by the $proceed()$ call of the last around advice (around advice p), which are after advice $21, 22, \ldots, 2m$. It is impossible for them to be called by the $proceed()$ calls in the middle levels of the advice call tree.

## 5. Performing Points-to Analysis for AspectJ

We next present our implementation for the points-to analysis technique for AspectJ using pointer assignment graph [12].

### 5.1. Analysis Approach

Referred to Section 2, the result of our points-to analysis is also represented as points-to graphs. We begin to take the empty points-to graph $G$ as input, and meanwhile to construct a pointer assignment graph, the illustration of transfer functions, in the process of analysis. Through the procedures of traveling pointer assignment graph in which transfer functions are iteratively used, the points-to information is added up to the graph $G$ till the end of changes from which we get the final points-to graph $G'$.

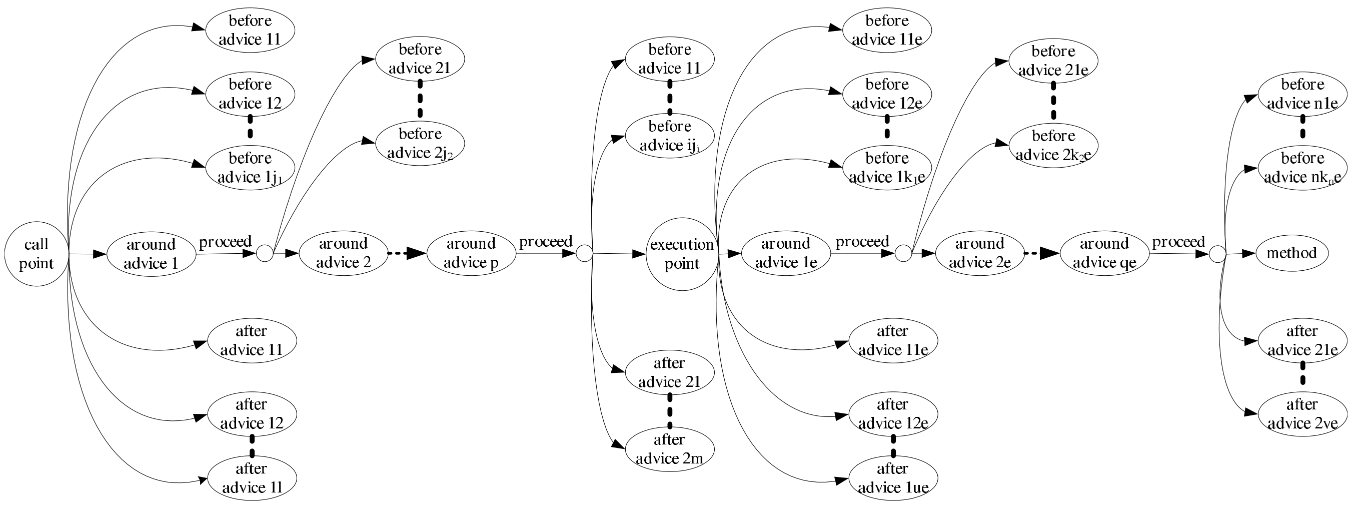

**Figure 7. Advice call tree.**

Our analysis is divided into two stages. One is to construct the call graph and the pointer assignment graph, and the other is to build up points-to graph based on the former stage. In the following, we discuss these issues in details.

## 5.2. Call Graph Construction

We use the call graph for the inter-procedural points-to analysis. Because of polymorphism, in static analysis, it is hard to fix a method called in run-time at a call site. Call graph can construct an approximation set of methods called at a call site. Procedure calls in Java is explicit, such as a method call, constructor call, etc. But in AspectJ, advices are called implicitly. Some advice match the certain pointcuts, the pointcuts select the join point in program, so at the join point these advices should be called. So the call graph for an AspectJ program should represent not only the virtual call in the base code, but also the implicit call of advice in the aspect code. In this work, we extend the RTA [8] algorithm, originally developed for Java, to construct the call graph for an AspectJ program.

## 5.3. Data Structures for Analysis

Before performing the analysis, our analysis needs a table called *aspect table* which can be used to store the aspect-class relationships. We construct the aspect table before the analysis. Each item in the aspect table presents the relationship between a class and an aspect with the following form:

$$(class\_name, aspect\_name, statement\_number, pointcut, caller, callee)$$

where *class_name* denotes the name of a class, *aspect_name* an aspect name, *statement_number* a statement name, and *pointcut* a pointcut name, *caller* a method name and the *callee* also a method name which is called by caller. If the statement has no call relationship between the methods, the caller and callee are both null. If the statement contains a virtual call, we will consider all the possible callee methods and all the advices involved in the each call path. Each call path can be determined by a pair $(caller, callee)$.

For a pointcut, we can get all advices which match the join points caught by it. In the case of several advices matching the same join point, in order to build advice call tree (Figure 7), we also need to compute their precedence. For an around advice $a$, if there is no $proceed()$ statement in it, the advices which have a lower precedence than $a$ would not be executed. If $a$ has a $proceed()$ statement, the advices associated with the $proceed()$ statement of $a$ will be looked up from the advice call tree discussed in Section 4.

For the dynamic pointcuts such as $cflow$, the matching for the shadows and the advices is nondeterministic at compiling time. So these pointcuts may also influence the precision of our points-to analysis for AspectJ. One possible solution is to add a branch for each possible matching of dynamic pointcuts as Xu described in [19] which uses "maybe" to cover the case of "must" and "never". Another technique proposed by [7] can minimize or eliminate the overhead of $cflow$ using both intra- and inter-procedural analyses.

## 5.4. Pointer Assignment Graph

Our pointer assignment graph consists of three types of nodes and four types of edges. The types of nodes include the allocation site nodes, simple variable nodes, and field dereference nodes. Allocation site nodes model the allocation sites in both classes and aspects and are used to represent the objects in run-time. Variable nodes model local variables in methods and advices, the method and advice parameters including formal parameters and actual parameters, return values, implicit parameter *this*, and static fields. Field dereference nodes model the instance fields being accessed in the program; a variable node and a field node pair represents the variable's field access. Field dereference node as the property of the field write edge or field read edge is included in the edges of these two types.

| Allocation | Assignment | Field Write | Field Read |
|---|---|---|---|
| $s : l = newC$ | $l = r$ | $l.f = r$ | $l = r.f$ |
| $o_s \rightarrow l$ | $r \rightarrow l$ | $r \xrightarrow{write[f]} l$ | $r \xrightarrow{read[f]} l$ |
| $f(G, l = newC)$ | $f(G, l = r)$ | $f(G, l.f = r)$ | $f(G, l = r.f)$ |

**Table 2. The four types pointer assignment graph edges.**

Four types of edges are illustrated in Table 2. The first line is the edge type names which are followed by the statements to be processed in the second line. The third line shows the edge forms. The last line corresponds to the transfer functions according to each edge type respectively.

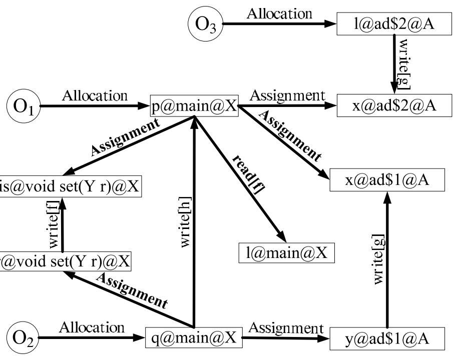

**Figure 8. A pointer assignment graph for AspectJ program in Figure 4.**

Figure 8 shows a pointer assignment graph for the sample program in Figure 4. The variable and object nodes of the pointer assignment graph share the same representation with that of the points-to graph. The types of edges correspond to the illustration in Table 2. For example, the edge $read[f]$ is a field read edge and $f$ is a field dereference node.

## 5.5. Points-to Graph Construction

Once a pointer assignment graph has been constructed, we can construct the points-to graph according to the transfer functions. We use the iterative algorithm and worklist algorithm [12] to construct the points-to graph.

The iterative algorithm is a naive and baseline algorithm which can be used to check the correctness of the results of the more complicated algorithms. First, as the input, an empty points-to graph is initialized according to allocation edges. Second, the algorithm iteratively applies every transfer function $f(G, l = r)$, $f(G, l.f = r)$, and $f(G, l = r.f)$ to the points-to graph according to assignment edges, field write edges, and field read edges until there is no change in the points-to graph.

Comparing with the iterative algorithm, worklist algorithm is better but more complex. At the beginning, the algorithm builds up allocation relationships and initializes *worklist* maintained by solver, which contains variable nodes whose points-to objects need to be propagated. After that, the algorithm deals with worklist and all the field write and read edges iteratively to obtain the final solution. When a variable node $p$ is included by worklist, the algorithm applies the transfer functions to the points-to graph along the pointer assignment graph edges. These edges have the form of $p \rightarrow q$, $p \xrightarrow{write[f]} q$, $q \xrightarrow{write[f]} p$, and $p \xrightarrow{read[f]} q$. If the points-to relationship of variable $q$ has changed during these operations, $q$ should be added to the worklist.

## 5.6. Analysis Implementation

We implement a points-to analysis tool for AspectJ on top of ajc 1.5.4 AspectJ compiler. In our implementation, we manipulate the abstract syntax tree and join point matching information created by ajc to build the pointer assignment graph, and then construct the points-to graph.

## 6. Evaluation

**Subject Programs.** We use nine widely used AspectJ benchmarks to evaluate our analysis shown in Table 3. The first seven programs are selected from AspectJ Benchmarks (AJBenches) [1] and the rest two are taken from the AspectJ example package [2]. Table 3 provides the detailed information of our benchmark programs. For each program, the number of lines of code, classes, aspects, methods, advices and call sites are shown in Table 3. The experiment is conducted on a DELL C521 PC with 1.80 Ghz AMD Sempron(tm) Processor and 1.00 Gb memory, under the environment of Sun JDK 1.5.0.10.

**Procedure.** In our experiment, we compare the precision between the existing byte-code-level Java approach and our source-code-level approach. For each benchmark, we compile it into bytecode using the abc compiler [1] and then use the Dava decompiler tool in Soot [4] to decompile the woven code back to Java source code. We employ the existing Java approach [15] to perform points-to analysis on the Java source and our approach to perform analysis on the AspectJ source code.

**Result.** The experimental results are shown in Tables 4 and 5. For the experiment data of each benchmark in Tables 4, the first row is the result using our analysis, and the second row is the result using the existing Java approach.

| Program | #LOC | #Class | #Aspect | #Method | #Advice | #CallSite |
|---|---|---|---|---|---|---|
| bean | 121 | 2 | 1 | 20 | 2 | 48 |
| cona1 | 1942 | 21 | 9 | 181 | 46 | 665 |
| cona2 | 291 | 2 | 1 | 21 | 10 | 135 |
| dcm | 1668 | 29 | 4 | 174 | 8 | 543 |
| figure | 94 | 5 | 1 | 22 | 1 | 29 |
| nullcheck | 1474 | 23 | 1 | 156 | 1 | 480 |
| qsort | 72 | 2 | 1 | 8 | 4 | 21 |
| telecom | 248 | 8 | 2 | 31 | 4 | 98 |
| spacewar | 1537 | 22 | 9 | 161 | 24 | 627 |

**Table 3. Subject Programs.**

In each cell of Tables 5, a slash / separates the result using our analysis and the existing Java approach. Clearly, in most cases, our analysis approach outperforms the existing Java approach.

**Precision Analysis.** We first analyze the precision of deference sites and call sites for each benchmark. The dereference sites consider all occurrences of field reference, $\langle o, f \rangle$, in which $o$ means an object and $f$ means a field of $o$. The percentage of dereference sites are given with 0, 1, 2, 3-10, and more than 10 elements in their points-to sets. Dereference sites with 0 items in the set mean that no object is written in the field of object. We consider all virtual calls and report the percentage of such call sites with 1, 2, and more than two targets. More than one target methods are found using the class of the receiver object. For instance, $o$ is the receiver object for a method call, $o.m()$, all the methods with the form of $m()$ defined in all subclasses of $o$'s class will be found. Calls with 1 target indicate that they are monomorphic calls at run-time.

| Program | Deference Sites (% of total) | | | | | Call Sites (% of total) | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3-10 | 10+ | 1 | 2 | 3+ |
| bean | 55.6 | 44.4 | 0.0 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| | 64.3 | 35.7 | 0.0 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| cona1 | 95.1 | 4.0 | 0.0 | 0.9 | 0.0 | 99.4 | 0.6 | 0.0 |
| | 85.3 | 13.8 | 0.0 | 0.9 | 0.0 | 99.0 | 0.5 | 0.5 |
| cona2 | 86.3 | 13.7 | 0.0 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| | 73.3 | 26.7 | 0.0 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| dcm | 94.1 | 4.9 | 0.0 | 1.0 | 0.0 | 98.5 | 1.5 | 0.0 |
| | 72.7 | 26.2 | 0.5 | 0.6 | 0.0 | 99.8 | 0.2 | 0.0 |
| figure | 64.1 | 35.9 | 0.0 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| | 62.5 | 37.5 | 0.0 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| nullcheck | 83.0 | 13.2 | 1.4 | 2.2 | 0.2 | 98.3 | 1.7 | 0.0 |
| | 86.8 | 11.8 | 0.3 | 0.9 | 0.2 | 99.5 | 0.5 | 0.0 |
| quicksort | 82.4 | 17.6 | 0.0 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| | 70.6 | 29.4 | 0.0 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| telecom | 72.3 | 21.5 | 3.1 | 3.1 | 0.0 | 100.0 | 0.0 | 0.0 |
| | 64.1 | 34.2 | 1.7 | 0.0 | 0.0 | 100.0 | 0.0 | 0.0 |
| spacewar | 85.0 | 11.7 | 3.2 | 0.0 | 0.0 | 97.2 | 0.5 | 2.3 |
| | 76.1 | 23.0 | 0.9 | 0.0 | 0.0 | 98.8 | 0.8 | 0.4 |

**Table 4. Precision.**

Since our approach is flow- and context- insensitive, the analysis result is safe which means the result of points-to relationships contains all the possibilities that may occur at run-time. The smaller the points-to object set is, the more useful the result is. However, the precision required is highly dependent on the application to be analyzed, so we can not get an absolute measure of precision. Table 4 compares the dereference sites and call sites analyzed by our tool from the source code and those from the woven bytecode.

For most cases, the percentage of deference sites with one item or more items is significantly lower. For some simple benchmark program such as benchmark **bean**, our approach produces the higher percentage of dereference sites with one or more items. We find that in these benchmarks some advice bodies, inlined at their shadows by the weaving process, have only a few statements. In these cases, inlining eliminates inter-procedure parameters transfer, which reduces the percentage of deference sites with one or more items.

From Table 4, we can see the percentage of call sites with one target from analyzing woven bytecode is a little higher than those from analyzing source code. Because after weaving a lot of advices become to the Java methods, which are not polymorphic.

**Performance Analysis.** We also study the performance of our analysis algorithm. We use the number of nodes and edges to represent the scale of pointer assignment graph. The time for building the pointer assignment graph is measured for each benchmark. In Table 5, **BGT** means the time

| Program | #Node | #Edge | BGT(ms) |
|---|---|---|---|
| bean | 78/108 | 32/54 | 63/78 |
| cona1 | 1088/1107 | 414/546 | 468/593 |
| cona2 | 164/233 | 52/140 | 62/125 |
| dcm | 696/3283 | 417/2066 | 984/39594 |
| figure | 61/86 | 31/44 | 0/15 |
| nullcheck | 558/1978 | 248/2907 | 531/11968 |
| quicksort | 44/59 | 18/36 | 0/15 |
| telecom | 166/204 | 62/118 | 47/109 |
| spacewar | 773/1739 | 271/1563 | 735/4078 |

**Table 5. Performance.**

of building the pointer assignment graph. From Table 5, we can see that the pointer assignment graphs built from source code are smaller than those built from bytecode, and the building time for the former is shorter than that for the latter. The reason is that after weaving, some advices are inlined into the Java source code, which makes Java source code contains more statements than that before waving.

**Conclusions.** In addition to reducing the cost in space and time, our analysis is also independent from the particular weaving techniques used to generate the final Java bytecode. Such feature has some important advantages in performing bytecode level analysis. The aspect-aware points-to analysis proposed in this paper, is practical to construct, easy to understand, contains significantly fewer nodes and edges, enables pre-weaving analysis of interactions between classes and aspects.

## 7. Related Work

We next discuss some related work in the area of points-to analysis for Java in particular and the flow-insensitive and context-insensitive points-to analysis in general. To the best of our knowledge, our points-to analysis algorithm presented in this paper is the first one for AspectJ programs.

In the past several years, points-to analysis [6, 16] has been an active research field. A very good overview of the current state of algorithms and metrics for points-to analysis are given by Hind [11]. How well the algorithms scale to large programs is an important issue. Trade-offs are made between efficiency and precision by various points-to analysis. On one hand, with less precise results equality-based analysis [16] runs in almost linear time. On the other hand, with cubic worst-case complexity subset-based analysis [6] produces more precise results.

The points-to analysis used for C to Java [12, 15] is adapted by several groups. Andersen's analysis is extended by RMR analysis [15] to efficiently represent and solve systems of annotated inclusion constraints. The annotations play two roles in our analysis of which method annotations are used to model the semantics of virtual calls precisely and efficiently, and field annotations allow us to distinguish between different fields of an object. In addition, RMR analysis keeps track of all reachable methods in order to avoid analyzing irrelevant library code. Lhotak and Hendren [12] use a framework called *SPARK*, which allows experimentation with many variations of points-to analysis for Java to implement points-to solvers that are more efficient in time and space than the other reported work, including that of Whaley and Lam [18]. They make it the most efficient Java points-to analysis solver of which we have ever known. SPARK is implemented as the part of Soot bytecode analysis, optimization, and annotation framework which uses the Jimple intermediate representation as the input, rather than the Java source code directly. However, our analysis tool uses the AspectJ source code as input.

## 8. Concluding Remarks

In this paper, we propose a flow- and context-insensitive points-to analysis for AspectJ. Our analysis adopt some successful ideas from the points-to analysis for Java. We showed how our analysis can be extended to cross the boundary between a class and its related aspects, and therefore the analysis produces fine precise analysis results as for Java programs. The main features of our approach are to get information related to those statements that use join points and the information of aspects that will be attached to these statements through the join points. By doing so, the analysis can correctly handle aspects efficiently.

There are several plans in our minds. First, since user code and library code have intricate interconnections, we will extend existing model to consider both the user code and library code. Second, we will improve our analysis approach to deal with AspectJ features better. Third, we also plan to do the flow- and context-sensitive points-to analysis for AspectJ programs.

## Acknowledgements

This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Sai Zhang, Si Huang and Cheng Zhang for their discussion on this work.

## References

[1] The AspectBench Compiler. `http://abc.comlab.ox.ac.uk/`.

[2] AspectJ compiler 1.5, May 2005. `http://www.eclipse.org/aspectj/`.

[3] AspectJ Development Tools (AJDT). `http://www.eclipse.org/ajdt/`.

[4] Soot. *http://www.sable.mcgill.ca/soot*.

[5] The AspectJ Team.The AspectJ Programming Guide, 2002.

[6] L. Andersen. Program analysis and specialization for the C programming language. Technical Report 94-19, University of Copenhagen, 1994.

[7] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *In PLDI*, pages 117–128, 2005.

[8] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. pages 324–341.

[9] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.

[10] G.Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.Videira, J.M.Loingtier, and J.Irwin. Aspect-oriented programming. In *In Proc. of ECOOP 1997*, 2002.

[11] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[12] O. Lhotak and L. Hendren. Scaling Java points-to analysis using SPARK. *In Proc. Conf. Compiler Construction*, 2622:153–169, 2003.

[13] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. pages 73–79, 2001.

[14] C. Razafimahefa. A study of side-effect analyses for Java. In *Master's thesis, McGill University, Dec. 1999.*, 1999.

[15] A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java based on annotated constraints. Technical Report DCS-TR-424, Rutgers University, Nov 2000.

[16] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[17] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, University Passau, nov 2000.

[18] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, September 2002.

[19] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *In Proc. of the 29th International Conference on Software Engineering*, pages 65–74.