Towards a Type System for Detecting Never-Matching Pointcut Compositions

Tomoyuki Aotani
Graduate School of Arts and Sciences
University of Tokyo
aotani@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences
University of Tokyo
masuhara@acm.org

ABSTRACT

Pointcuts in the current AspectJ family of languages are loosely checked because the languages allow compositions of pointcuts that never match any join points, which developers are unlikely to intend, for example, `set(* *)&&get(* *)`. We formalize the problem by defining well-formedness of pointcuts and design a novel type system for assuring well-formedness. The type of pointcuts is encoded by using record, union and the bottom types.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of program constructs—*Type structure*

General Terms

Design, languages, theory

Keywords

AOP, pointcut compositions, records

1. INTRODUCTION

Join point selection mechanisms, i.e., pointcuts, play an important role in AspectJ family of languages such as AspectJ [13] and JBoss AOP. While there have been studies targeting many facets of those languages, such as expressiveness and robustness [1,5,9,12,17], safe pointcut compositions have been less investigated [4, 16]. The property becomes more important the more aspects use composed pointcuts.

This position paper focuses on safe pointcut composability so that composed pointcuts can match at least one join point in *some* program. We call such a pointcut *well-formed*. By checking well-formedness of every pointcuts in aspect definitions, developers can notice unintended pointcut compositions before applying aspects to programs. This property helps programmers to avoid pointcuts that never have any match by merely examining aspect definitions. It is particularly important for separate compilation of aspects. In other words, we are interested in detecting *never-matching* pointcuts compositions whose resulting pointcuts match no join point in *any program*. Note that current AspectJ compiler implementations (abc [2]) can report, only after weaving aspects into classes, that an advice declaration is not woven into any join point shadows. When we separately compile aspects (which is also required by the load-time weaving techniques), the current compilers silently pass never-matching pointcut compositions.

To detect such never-matching pointcut compositions, we are going to develop a type system that guarantees the *well-formedness* of pointcuts, defined as follows:

DEFINITION 1 (WELL-FORMED POINTCUT). *Let U be the set of all well-typed base programs and $JP(b)$ be the set of join points in any execution of a well-typed base program b. Pointcut p is well-formed when it satisfies:*

$$\exists b \in U. \exists j \in JP(b). match(p, j)$$

In other words, a pointcut p is well-formed when there exists a well-typed program that generates a join point matching p.

The rest of the paper is organized as follows. Section 2 explains the problems we address. Section 3 shows a sketch of our type system. Section 4 discusses related work. Section 5 concludes the paper.

2. NEVER-MATCHING POINTCUT COMPOSITIONS

In order to clarify the problems that we address, we present an example in which a pointcut never matches any join point.

AspectJ compilers allow meaningless pointcut compositions that never match join points in any program.

In AspectJ, one can compose any two pointcuts with `&&` (`and`) and `||` (`or`) pointcut designators. For example, one can capture both get and set join points by composing a get and set pointcuts with an `or` operator; i.e., `get(* *)||set(* *)`.

On the other hand, the composition of these two pointcuts with an `and` operator, i.e., `get(* *)&&set(* *)` is meaningless because no join point is get and set at the same time.

3. A SKETCH OF OUR TYPE SYSTEM FOR POINTCUTS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.
Copyright 2007 ACM ISBN 1-59593-671-4/07/03 ...$5.00.

Table 1: Elaborated pointcut primitives.

	get	set
instance variables	`mget`	`mset`
class variables	`sget`	`sset`

This section gives a sketch of our type system that types pointcuts with respect to the well-formedness. We formalize the system by adding a pointcut and advice mechanism to Featherweight Java [10].

The key idea is to represent the type of join points as a record type. The type of pointcuts is also a record type because a pointcut can be seen as a set of matching join points.

Because the paper focuses on pointcut compositions, we explain only the pointcut types below.

3.1 Overview of Pointcut Types

The pointcut type P is defined as follows.

$$\begin{array}{lcl} P & ::= & \{l_i : {Y_i}^{i\in 1\cdots n}\} \mid P + P \mid \bot \\ Y & ::= & k \mid T^g \mid [{T_i}^{i\in 1\cdots n}] \mid \bullet \\ k & ::= & \texttt{mset} \mid \texttt{mget} \mid \texttt{sset} \mid \texttt{sget} \mid \cdots \\ g & ::= & \bullet \mid \epsilon \end{array}$$
$$T \in \mathit{idpats}$$

Basically, a pointcut type is encoded into a record type $\{l_i : {Y_i}^{i\in 1\cdots n}\}$ that contains most attributes of the `JoinPoint` object in AspectJ such as the kind of the matching join point. The label l corresponds to the attribute of a join point including `this`, `args` and `kind`, and is associated with an attribute type Y. The attribute type Y is either the kind of matching join points k, an element of *idpats* with runtime value availability tag g, or a sequence of the elements of *idpats*. $P + P$ denotes the union of two pointcut types and $\bot$ denotes pointcuts never matching any join point. We assume that $\bot + P$ and $P + \bot$ are equivalent to P. The set *idpats* is the union of the three sets: the singleton set of an $*$, the set of names of primitive types such as `int` and `boolean`, and the set of valid identifiers with respect to the Java Language Specification [8] such as `Object`, `List` and `width`. $[{T_i}^{i\in 1\cdots n}]$ represents a comma-separated sequence of the elements of *idpats*. The single $\bullet$ denotes absence. For example, $\{\texttt{args} : \bullet\}$ denotes that matched join points never have `args` values. Meta-variable k ranges over the kinds of join points such as `get` and `set`. Meta-variable g ranges over runtime value availability tags. When a label l is associated with $T^\bullet$, it denotes that there is no runtime value for the attribute l. For example, $\{\texttt{target} : T^\bullet\}$ represents that the `target` attribute of matching join points is constrained to have the type T but has no runtime value. For readability, we omit the availability tag when it is ϵ, so we simply write T rather than T^ϵ.

3.2 Pointcut Sublanguage and Typing Rules

Since we are still working on details of the type system, the paper demonstrates how pointcuts are typed by merely using `set`, `get`, `args`, `||` and `&&` pointcuts. The pointcut sublanguage is defined as Figure 2. The `args` pointcut does not bind any variable because we are only interested in the pointcut compositions. Instead, an `args` pointcut limits the types and numbers of arguments of matching join points.

We divide `set` and `get` pointcuts into the four pointcuts as is shown in Table 1 so that they explicitly distinguish

$$\begin{array}{lcl} \mathit{pc} & ::= & \mathit{prm}(T\ C.f) \mid \texttt{args}({T_i}^{i\in 1\cdots n}) \mid \mathit{pc}\&\&\mathit{pc} \mid \mathit{pc}||\mathit{pc} \\ \mathit{prm} & ::= & \texttt{mset} \mid \texttt{sset} \mid \texttt{mget} \mid \texttt{sget} \end{array}$$

Figure 2: Pointcut sublanguage ($T, C, f \in \mathit{idpats}$).

$$\frac{\mathit{pc}_1 : P_1 \quad \mathit{pc}_2 : P_2 \quad P_1 \otimes P_2 \rightsquigarrow P}{\mathit{pc}_1\&\&\mathit{pc}_2 : P}$$

$$\frac{\mathit{pc}_1 : P_1 \quad \mathit{pc}_2 : P_2}{\mathit{pc}_1||\mathit{pc}_2 : P_1 + P_2}$$

Figure 3: Typing rules $\mathit{pc} : P$ for pointcut compositions.

whether matching join points access the `static` fields or not. This is because the join points related to class fields have no `target` value.[1]

The types of `mset`, `sset`, `mget`, `mset` and `args` pointcuts are shown in Figure 1. For example, the type of the pointcut `mset(int Point.x)`, which matches `p.x = 3` assuming `p` is an instance object of a `Point` class, becomes $\{\texttt{target} : \texttt{Point}, \texttt{args} : \texttt{int}, \texttt{kind} : \texttt{mset}, \texttt{name} : \texttt{x}, \texttt{ret} : \bullet\}$.

The typing rules for pointcut compositions (i.e., $\mathit{pc}_1\&\&\mathit{pc}_2$ and $\mathit{pc}_1||\mathit{pc}_2$) are shown in Figure 3. Composing two pointcuts with an `or` pointcut, the resulting type becomes simply the union of the two pointcut types. Composing with an `and` pointcut, the resulting type becomes a common subtype of the two pointcut types, intuitively. The common subtype is calculated using the rules in Figure 4.

As we can see, we need to define the type subsumption ($<:$) on pointcut types only for the cases that the right hand side is a record type. It is simply defined as follows.

$$\bot <: \{l_i : {T_i}^{i\in 1\cdots n}\}$$

$$\frac{P_1 <: \{l_i : {T_i}^{i\in 1\cdots n}\} \quad P_2 <: \{l_i : {T_i}^{i\in 1\cdots n}\}}{P_1 + P_2 <: \{l_i : {T_i}^{i\in 1\cdots n}\}}$$

We employ the standard record type subsumptions (i.e. subsumptions on record widths, depths and permutations [18]).

$$\frac{n \le m \quad \forall i \in 1\cdots n.\exists j \in 1\cdots m.Y_i <: Y'_j}{\{l_i : {Y_i}^{i\in 1\cdots n}\} <: \{l'_i : {Y'_i}^{i\in 1\cdots m}\}}$$

For the elements of *idpats*, say T_1 and T_2, the subsumption is defined as follows. $T_1 <: T_2$ if

- T_2 is $*$, or
- T_1 and T_2 is the same identifier.

And the subsumptions on sequences of elements and tagged elements of *idpats* are defined as follows.

$$\frac{n = m \quad \forall i \in 1\cdots n.T_i <: T'_i}{[{Ti}^{i\in 1\cdots n}] <: [{T'_i}^{i\in 1\cdots m}]}$$

$$\frac{g_1 = g_2 \quad T_1 <: T_2}{T_1^{g_1} <: T_2^{g_2}}$$

[1]Similar elaboration can be applied to pointcuts related to methods, i.e., `call` and `execution` pointcuts.

$$
\begin{array}{rcl}
\texttt{mget}(T\ C.f) & : & \{\texttt{target} : C, \texttt{args} : \bullet, \texttt{kind} : \texttt{mget}, \texttt{name} : f, \texttt{ret} : T\} \\
\texttt{sget}(T\ C.f) & : & \{\texttt{target} : C^{\bullet}, \texttt{args} : \bullet, \texttt{kind} : \texttt{sget}, \texttt{name} : f, \texttt{ret} : T\} \\
\texttt{mset}(T\ C.f) & : & \{\texttt{target} : C, \texttt{args} : [T], \texttt{kind} : \texttt{mset}, \texttt{name} : f, \texttt{ret} : \bullet\} \\
\texttt{sset}(T\ C.f) & : & \{\texttt{target} : C^{\bullet}, \texttt{args} : [T], \texttt{kind} : \texttt{sset}, \texttt{name} : f, \texttt{ret} : \bullet\} \\
\texttt{args}({T_i}^{i\in 1\cdots n}) & : & \{\texttt{args} : [{T_i}^{i\in 1\cdots n}]\}
\end{array}
$$

Figure 1: Typing rules $pc : P$ for `mset`, `sset`, `mget`, `mset` and `args` pointcuts ($T, C, f \in idpats$).

$$\bot \otimes P \rightsquigarrow \bot \qquad P \otimes \bot \rightsquigarrow \bot$$

$$\frac{P <: \{l_i : {T_i}^{i\in 1\cdots n}\} \quad P <: \{l'_i : {T'_i}^{i\in 1\cdots n}\}}{\{l_i : {T_i}^{i\in 1\cdots n}\} \otimes \{l'_i : {T'_i}^{i\in 1\cdots n}\} \rightsquigarrow P}$$

$$\frac{P_1 \otimes P_3 \rightsquigarrow P'_1 \quad P_2 \otimes P_3 \rightsquigarrow P'_2}{(P_1 + P_2) \otimes P_3 \rightsquigarrow P'_1 + P'_2}$$

$$\frac{P_1 \otimes P_2 \rightsquigarrow P'_2 \quad P_1 \otimes P_3 \rightsquigarrow P'_3}{P_1 \otimes (P_2 + P_3) \rightsquigarrow P'_2 + P'_3}$$

Figure 4: Type calculation rules $P \otimes P \rightsquigarrow P$

Note that `ArrayList` $<:$ `Object` is not available in our definition. This is mainly because we want to check pointcut compositions without any base programs.

The subsumption on kinds $k_1 <: k_2$ holds only when k_1 and k_2 are the same.

3.3 Typing Examples

This section demonstrates that our type system can successfully accept the well-formed pointcuts and detects never-matching pointcuts.

Never-matching pointcut compositions is typed as $\bot$.

Our type system can successfully type `get(* *)&&set(* *)` as $\bot$ without any base programs. For simplicity, we show this by using elaborated pointcuts, i.e., `mget(* *.*)&&mset(* *.*)`[2]. As shown in Table 1, each pointcuts are typed as follows.

`mset(* *.*)`:
$\{\texttt{target} : *, \texttt{args} : [*], \texttt{kind} : \texttt{mset}, \texttt{name} : *, \texttt{ret} : \bullet\}$

`mget(* *.*)`:
$\{\texttt{target} : *, \texttt{args} : \bullet, \texttt{kind} : \texttt{mget}, \texttt{name} : *, \texttt{ret} : *\}$

The type of the composed pointcut `mget(* *.*)&&mset(* *.*)` becomes a common subtype of the two pointcut types as mentioned in Section 3.2, and we find $\bot$, which is the only possible type because there is no common subtype of $\bullet$ and $*$, nor of `mget` and `mset`. Thus our type system types `mget(* *.*)&&mset(* *.*)` as $\bot$, and can conclude that it is a never-matching pointcut.

The union of never-matching pointcuts and well-formed pointcuts is not typed as $\bot$.

Composing a never-matching pointcut and a well-formed pointcut with an `or` pointcut (`||`), we get a well-formed pointcut. In our type system, the fact is rephrased that composing an pointcut typed as $\bot$ and another pointcut not typed as $\bot$ with an `or` pointcut, the type of the resulting pointcut is not $\bot$. This property is satisfied in our system clearly following to the typing rule for the `or` pointcut compositions.

For example, the pointcut

```
(mset(* *.*)||mget(* *.*))&&args(int)
```

is well-formed and matches all assignments to any object fields. Because get join points have no argument, none of them is selected.

Reducing the bottom types by using the assumptions $P + \bot = P$ and $\bot + P = P$, the type of the pointcut becomes

$$\{\texttt{target} : *, \texttt{args} : [*], \texttt{kind} : \texttt{mset}, \texttt{name} : *, \texttt{ret} : \bullet\}$$

in our type system and it successfully reflects the fact that we mentioned just before.

4. RELATED WORK

Our work is not the first attempt to detect never matching pointcuts. Douence et al. defined the alphabet analysis for their pointcut language for control-flow. An alphabet is a set of join point shadows [15] that can generate matching join points. They also suggested that when the alphabet becomes empty, the pointcut never matches any join points and such pointcut definitions are erroneous. Program Description Language (PDL) [16] is a domain specific language for checking design rules such as the Law of Demeter [14]. Its pointcut language is similar to the one of AspectJ, and has a type system that assures that typed pointcuts have at least one matching join point. The typing rule and semantics of `not` pointcuts are very interesting, although the semantics differs from the one of AspectJ.

Aspect FGJ (or shortly AFGJ) [11] is an aspect-oriented calculus which extends Featherweight GJ [10] with forms for advice declaration and for proceeding to the next declared advice. Though the language have a execution pointcut primitive `exe` and two operators for pointcut compositions `&&` and `||`, the pointcut logic can successfully reject pointcut compositions of two different execution pointcuts such as[3]

```
exe(int Point.getX())&&exe(int Point.getColor()).
```

We think this work may be a good starting point of our formalization task.

MiniMAO$_1$ [3] is another core aspect-oriented calculus of AspectJ-like aspect-oriented programming languages based on Classic Java [6] to investigate the semantics of proceed and the soundness over advice weavings. Types of pointcuts are similar to ours but the approach does not detect never-matching pointcuts.

[2]The `get` and `set` pointcuts shall be encoded by disjunctions of the `mget` and `sget`, and the `mset` and `sset` pointcuts, respectively.

[3]Though AFGJ has a different syntactic format like `exe int Point.X()`, we use AspectJ-like format for readability.

AspectML [19] is a polymorphic aspect-oriented functional programming language. Pointcuts are first-class values and typed, but the language merely has `execution` join points as far as we know.

5. CONCLUSIONS AND FUTURE WORK

We pointed out that the AspectJ family of languages allow compositions of pointcuts that never match join points in any program, and that such compositions should be detected from aspect definitions alone. We showed a sketch of our type system to detect such never-matching compositions of pointcuts. Our key idea is to encode types of pointcuts and join points with record types. In the type system, the type of mutually exclusive pointcut compositions, such as `set(* *)&&get(* *)`, becomes $\bot$, which denotes never matching pointcuts.

We are currently working on the details of the type system based on Featherweight Java [10]. One of the major difficulties we are facing now is the `!` (`not`) operator. One possible solution would be to use negation (or complement) types, whose semantics is based on sets [7].

Our future work includes proof of type soundness; i.e., for any non $\bot$-typed pointcuts there exits a join point that matches the pointcut. An interesting direction of our future work is to extend the languages with generics so that we can verify correctness of the design and implementation of pointcuts in AspectJ5.

Acknowledgments

We would like to thank the anonymous reviewers for their encouraging and thoughtful suggestions. We also thank Jan Hanneman, Kohei Sakurai, Kazunori Kawauchi, Tsutomu Kumazawa and other members of the TM seminar at the University of Tokyo, and Atsushi Igrashi and other members of the Kumiki project for their valuable advice and discussions on this study.

6. REFERENCES

[1] T. Aotani and H. Masuhara. Scope: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *Proceedings of the 6th International Conference on Aspect-oriented software development*, pages 161–172. ACM Press, 2007.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM Press, 2005.

[3] C. Clifton and G. T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.

[4] R. Douence and L. Teboul. A pointcut language for control-flow. In *Proceedings of 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 95–114. Springer, 2004.

[5] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems*, pages 366–381. Springer, 2004.

[6] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269. Springer, 1999.

[7] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society, 2002.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[9] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 60–69. ACM Press, 2003.

[10] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[11] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 2006. to apper.

[12] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 501–525, 2006.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, et al. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[14] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: checking the law of demeter with AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 40–49. ACM Press, 2003.

[15] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 46–60, 2003.

[16] C. Morgan, K. D. Volder, and E. Wohlstadter. A static aspect language for checking design rules. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 63–72. ACM Press, 2007.

[17] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 214–240. Springer, 2005.

[18] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[19] G. Washburn and S. Weirich. Good advice for type-directed programming aspect-oriented programming and extensible generic functions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 33–44. ACM Press, 2006.