FOAL 2005 Proceedings
Foundations of Aspect-Oriented Languages
Workshop at AOSD 2005

Curtis Clifton, Ralf Lämmel, and Gary T. Leavens (editors)

TR #05-05
March 2005

Keywords: Aspect-oriented programming languages, aspects, pointcuts, advice, AspectJ, formal semantics, static analysis, behavior, method call interception, program transformation, loops, LoopsAJ, context exposure, Real-Time Specification for Java, RTSJ, scoped memory, program slicing, separation of concerns, crosscutting concerns, temporal relations between events, predicates over execution traces, Aspectual Caml, type inference, polymorphism, curried functions, around advice, proceed, type soundness.

2003 CR Categories: D.1.3 [*Programming Techniques*] Concurrent Programming—parallel programming; D.1.m [*Programming Techniques*] Miscellaneous—aspect-oriented programming, real-time programming; D.2.4 [*Software Engineering*] Software/Program Verification—formal methods, correctness proofs; D.3.1 [*Programming Languages*] Formal Definitions and Theory—semantics; D.3.2 [*Programming Languages*] Language classifications—object-oriented languages, AspectJ; D.3.3 [*Programming Languages*] Language Constructs and Features—control structures, polymorphism; D.3.4 [*Programming Languages*] Processors—code generation, compilers, memory management; F.3.1 [*Logics and Meaning of Programs*] Specifying and verifying and reasoning about programs—logics of programs; F.3.2 [*Logics and Meaning of Programs*] Semantics of programming languages—operational semantics, program analysis;

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Contents

Preface

Aspect-oriented programming is a paradigm in software engineering and programming languages that promises better support for separation of concerns. The fourth Foundations of Aspect-Oriented Languages (FOAL) workshop was held at the Fourth International Conference on Aspect-Oriented Software Development in Chicago, USA, on March 14, 2005. This workshop was designed to be a forum for research in formal foundations of aspect-oriented programming languages. The call for papers announced the areas of interest for FOAL as including: semantics of aspect-oriented languages, specification and verification for such languages, type systems, static analysis, theory of testing, theory of aspect composition, and theory of aspect translation (compilation) and rewriting. The call for papers welcomed all theoretical and foundational studies of foundations of aspect-oriented languages.

FOAL logos courtesy of Luca Cardelli

The goals of this FOAL workshop were to:

- Make progress on the foundations of aspect-oriented programming languages.
- Exchange ideas about semantics and formal methods for aspect-oriented programming languages.
- Foster interest within the programming language theory and types communities in aspect-oriented programming languages.
- Foster interest within the formal methods community in aspect-oriented programming and the problems of reasoning about aspect-oriented programs.

The workshop was organized by Gary T. Leavens (Iowa State University), Ralf Lämmel (Microsoft Research), and Curtis Clifton (Iowa State University). The program committee was chaired by David Walker (Princeton University) and included Walker, Lämmel, Leavens, Jonathan Aldrich (Carnegie Mellon University), John Tang Boyland (University of Wisconsin, Milwaukee), Sophia Drossopoulou (Imperial College), Tzilla Elrad (Illinois Institute of Technology), Kathleen Fisher (AT&T Labs–Research), Pascal Fradet (INRIA), Sam Kamin (University of Illinois), Shmuel Katz (Technion–Israel Institute of Technology), Pertti Kellomäki (Tampere University of Technology), Shriram Krishnamurthi (Brown University), David Lorenz (Northeastern University), Dave MacQueen (University of Chicago), Hidehiko Masuhara (University of Tokyo), Oege de Moor (Oxford University), Greg Morrisett (Harvard University), Peter D. Mosses (University of Wales, Swansea), James Riely (DePaul University), Henny Sipma (Stanford University), Kevin Sullivan (University of Virginia), and Mitchell Wand (Northeastern University). We thank the organizers of AOSD 2005 for hosting the workshop.

Message from the Program Committee Chair

This volume contains the papers presented at FOAL ’05, the Workshop on Foundations of Aspect-Oriented Languages. Beginning in 2002, the FOAL series of workshops has been a forum for discussion of the theory and principles behind aspect-oriented programming language design and implementation. This year the conference was held in Chicago, USA, on Monday, the 14th of March.

The call for papers solicited long papers (10 pages), short papers (6 pages) and very short papers (3 pages). This year we received a total of 19 submissions of which 8 were long submissions, 10 were short submissions and 1 was a very short submission. Each paper was reviewed by a minimum of three reviewers and many papers received four or five reviews. After the initial reviews were submitted, the program committee discussed each paper during a 4-day online program committee meeting held between February 6th and 9th. Papers for which a member of the program committee was an author were held to a slightly higher standard than other papers. The final program includes 5 long submissions and 3 short submissions. The authors of long submissions spoke for 30 minutes with the audience engaging in a 15 minute discussion afterwards. The authors of short submissions spoke for 20 minutes with the audience engaging in a 10 minute discussion afterwards. In addition, we had a 45-minute panel session on ”Aspect-Oriented Programming Languages and Modularity” with three panelists, selected by the FOAL organizers and myself, each presenting different perspectives on the topic.

I am very grateful to the program committee for their diligent work reviewing all the submissions. I am also very grateful to Shriram Krishnamurthi and Pete Hopkins for giving us access to the Brown Continue Server which we used to administer the reviewing process. The Continue Server was a pleasure to use and it helped to relieve the burden of managing reviews manually. Finally, the program organizers, Curtis Clifton, Ralf Lämmel and Gary T. Leavens, made my job as program chair simple and straightforward. I thank them for all their hard work organizing the details that made this workshop possible.

Sincerely,

David Walker
FOAL 05 Program Chair
Princeton University

Proving aspect-oriented programming laws

Leonardo Cole[*]
lcn@cin.ufpe.br

Paulo Borba[†]
phmb@cin.ufpe.br

Alexandre Mota
acm@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
P.O. Box 7851 - 50.732-970 Recife PE, Brazil

ABSTRACT
The proof of the behaviour-preserving property of programming laws is not trivially demonstrated. It is necessary to show that the programs, before and after the transformation, have the same behaviour. In this paper we show how it is possible to prove that an aspect-oriented programming law preserves behaviour; an operational semantics for Method Call Interception is used. An equivalence relation stating that two programs have the same behaviour is defined. We use these concepts and discuss soundness for the law *Add-Before Execution*.

Categories and Subject Descriptors
D.1 [**Software**]: Programming Techniques—*Aspect-Oriented Programming*; D.3.2 [**Programming Languages**]: Language Classifications—*AspectJ*

General Terms
Languages

Keywords
Refactoring, AspectJ, Aspect-Oriented Programming, Separation of concerns

1. INTRODUCTION
In order to explore the benefits of refactoring [7, 17, 18], aspect-oriented developers are identifying common transformations for aspect-oriented programs [16, 14, 10, 12], mostly in AspectJ [13], a general purpose aspect-oriented extension to Java [9]. However, they lack support for assuring that the transformations preserve behaviour and are indeed refactorings.

[*]Supported by CAPES.

[†]Partially supported by CNPq.

FAOL 2005 Chicago, USA

It is possible to use AspectJ programming laws [5] to derive or create behaviour preserving transformations (refactorings) for a subset of this language. Programming laws [11] define equivalence between two programs, given that some conditions are respected. By applying and composing those laws, one can show that an AspectJ transformation is a refactoring. A refactoring denotes a behaviour preserving transformation that increases code quality. Contrasting with a refactoring, a law is bi-directional and it does not always increase code quality, it is part of a bigger strategy that does. Besides, the laws are much simpler than most refactorings because they involve only localized changes, and each one focuses on one specific AspectJ construct. The laws form a basis for defining refactorings with confidence that they preserve behaviour. Hence, soundness of the laws with respect to a formal semantics is a necessary property.

This paper shows one way to prove that those aspect-oriented laws indeed preserve behaviour. We use the semantics of an aspect-oriented language [15] in which we can represent part of the laws. This language is not as expressive as AspectJ, but provides mechanisms to define some kinds of AspectJ advices with a well defined semantics. It allows us to explore notions of semantic equivalence between aspect-oriented programs. This increases the confidence that the transformations applied by the laws preserve behaviour. However, some hypothesis must be satisfied in order to enable the laws proof. For instance, the programs can not use reflection and can not be concurrent. Those hypothesis are also considered for object-oriented programming laws [4].

A limitation to our current work is a consequence of being able to represent only part of the laws with the chosen semantics. As the chosen language is not as powerful as AspectJ, we can represent Laws 3 - *Add before-execution*, 4 - *Add before-call*, 7 - *Add after-execution returning*, 13 - *Merge advices*, 15 - *Remove target parameter*, and 14 - *Remove this parameter*. It would be necessary to define another language (or extend the one we used) to prove the remaining laws. Nevertheless, we can use this subset of the laws to show that some important refactorings indeed preserve behaviour, for instance, the *Extract Method Calls* [14].

This paper is organized as follows. Section 2 discusses the semantics used here and our notion of equivalence between two programs. Section 3 introduces the laws, showing their structure, preconditions and intent. Section 4 shows a formal argumentation about soundness of one law. Then, we discuss related work in Section 5 and conclude in Section 6.

2. SEMANTICS OF METHOD CALL INTERCEPTION (MCI)

Semantics for aspect-oriented languages is still en emerging field. The aspect-oriented languages used today still do not have an associated formal semantics where it is possible to reason about programs. However, there are several approaches [3, 1, 20, 15, 19, 6, 2] that try to solve this problem. In this section we discuss an aspect-oriented semantics based on Method Call Interception (MCI) [15].

The MCI semantics was chosen because it allows us to represent several of the advice types offered by AspectJ. Hence, allowing us to reason about programming laws involving those kinds of advice. Moreover, the MCI semantics is described as an extension to an object-oriented one, similarly to the way AspectJ extends Java. Therefore, providing an easier comprehension of how the semantics change from the object-oriented language to its aspect-oriented extension. This semantics only deals with advices, which we consider as a core concept in aspect-orientation. However, other AspetcJ constructs, such as inter-type declarations, are also important and the proof for laws involving them should consider a different or extended language.

Lämmel starts defining the semantics for a small java-like object-oriented language called μO^2 [15]. He describes an operational semantics and defines the rules for this language. Although Lämmel describes both static and dynamic semantics, we consider only the dynamic semantics because we want to compare behaviour of programs. The static semantics is useful to verify if the programs are well constructed according to the type system. Hence, the static semantics would be necessary to proof that the laws relate valid programs, this is regarded as a future work.

After defining the semantics for μO^2, Lämmel extends this language to incorporate the new construct `superimpose`, which allows the definition of an advice intercepting a method. However, the first definition for the `superimpose` construct is very simple and was extended in two ways. First he introduces interactivity, allowing advices to expose and use variables from the method's execution context. Second, he extends the language definition including quantitative mechanisms, allowing a single advice to intercept several methods. The syntax for the resulting aspect-oriented language can be seen in Figure 1. The MCI extension starts at the `caller` definition.

prog	=	*cdef** *cn.mn*
cdef	=	`class` *cn* `extends` *cn* {*field** *mdef**}
field	=	*type fn*
mdef	=	*type mn* (*arg**) *body*
type	=	*cn* | `void`
arg	=	*type vn*
body	=	*exp* | `abstract`
cn	=	class names
fn	=	field names
mn	=	method names
vn	=	variable names
exp	=	`null`
	|	`this`
	|	*vn*
	|	`view` *type exp*
	|	*exp.fn*
	|	*exp.vn* = *exp*
	|	*exp.mn* (*exp**)
	|	`super`.*mn* (*exp**)
	|	`let` *vn* : *type* = *exp* `in` *exp*
	|	*exp;exp*
	|	`while` (*exp*) *exp*
	|	`caller`
	|	`callee`
	|	`superimpose` *exp* `on` *eve*
eve	=	*mci loc* | *eve* `within` *loc*
mci	=	`dispatch` | `enter` | `exit`
loc	=	*
	|	`object` *exp*
	|	`class` *cn*
	|	`subclass` *cn*
	|	`method` *mn*
	|	`result` *type*
	|	`argument` *type vn*
	|	*loc* && *loc*
	|	*loc* || *loc*
	|	!*loc*

Figure 1: MCI syntax

The `superimpose` construct defines that some code (*exp*) is to be executed on the occurrence of an event (*eve*). Comparing to AspectJ, the *exp* can be regarded as the advice body, and *eve* can be regarded as the pointcut expression. The description of an event defines when and where a method interception occurs. A method can be intercepted at three distinct points (*mci*): `dispatch`, before its arguments evaluation; `enter`, after the arguments evaluation but before the method's execution; and `exit`, after the method's execution. Those *mci* points are analogous to the `before-call`, `before-execution` and `after-returning-execution` from AspectJ. The other component of an event (*loc*) describes the location of the method interception, which is an expression that matches methods based on its name, class, arguments, return type, etc. An event can also be constrained to occur only within another location.

Lämmel defines an operational semantics for this language [15]. He defines several rules to show how an expression should be evaluated. Each rule shows the return value of the evaluated expression and shows how the state changes. Some rules may depend on the the execution of other rules to achieve its result. Hence, the evaluation of a program can be represented as a tree showing several evaluation rules.

The domains for a rule consist of a method code table (T), which links method names with its parameters and body. An object store (Σ) that holds references to objects and its field values. This object store also hold the advice registry. There is also a reference to the executing object (θ) and an environment for the program variables (η). The expression $\Pi_i(t)$ denotes the *ith* projection of a tuple t.

Figure 2 shows the evaluation rule [15] for the `superimpose` construct. This rule states that evaluating a `superimpose` declaration returns a `null` reference (0 is the meaning of a null expression) and updates the object store (Σ''). The superimpose evaluation consists of three steps: first, we evaluate the event expression (1), which yields the event description ($\overline{k}$) and an updated object store (Σ'); second, we create the advice, represented by α (2); finally, we call the `register` helper function (3), which updates Σ' by registering the event and advice from the previous evaluations.

$$\begin{array}{lr} T, \Sigma, \theta, \eta \vdash \mathit{eve} \Rightarrow \overline{k}, \Sigma' & (1) \\ \wedge\ \alpha = ((\Pi_{cn}(\theta), \Pi_{mn}(\theta)), \mathit{exp}) & (2) \\ \wedge\ \overline{\texttt{register}}(\Sigma', \overline{k}, \alpha) \Rightarrow \Sigma'' & (3) \\ \hline T, \Sigma, \theta, \eta \vdash \texttt{superimpose}\ \mathit{exp}\ \texttt{on}\ \mathit{eve}\ \Rightarrow 0, \Sigma'' & (4) \end{array}$$

Figure 2: `superimpose` evaluation rule

We do not show all the evaluation rules, more details can be found elsewhere [15]. As mentioned before, one of the reasons to choose the MCI semantics is that it shows an object-oriented semantics and extends it to introduce MCI. This description allows us to see exactly how the semantics change when we introduce aspect-oriented features to the language. As the superimpose construct affects only method calls, the only rule changed during the MCI extension is the method `call` evaluation rule.

Originally a method call is evaluated according to the rule listed in Figure 3. First, we evaluate the expression that yields the object on which the method is being called (5). Second, we search the environment for the method definition (6). Then, it is necessary to evaluate the expressions representing the arguments values (7-9). Finally, an environment is mounted with the evaluated arguments (10) to execute the method's body (11).

$$\begin{array}{lr} T, \Sigma_0, \theta, \eta \vdash \mathit{exp} \Rightarrow \rho, \Sigma_1 & (5) \\ \wedge\ \Pi_1(T) \bullet (\rho, \mathit{mn}) = ((\mathit{vn}_1, ..., \mathit{vn}_n), \mathit{exp}') & (6) \\ \wedge\ T, \Sigma_1, \theta, \eta \vdash \mathit{exp}_1 \Rightarrow v_1, \Sigma_2 & (7) \\ \wedge\ ... & (8) \\ \wedge\ T, \Sigma_n, \theta, \eta \vdash \mathit{exp}_n \Rightarrow v_n, \Sigma_{n+1} & (9) \\ \wedge\ \eta' = \perp [\mathit{vn}_1 \mapsto v_1, ..., \mathit{vn}_n \mapsto v_n] & (10) \\ \wedge\ T, \Sigma_{n+1}, \eta' \vdash \mathit{exp}' \Rightarrow v, \Sigma_{n+2} & (11) \\ \hline T, \Sigma_0, \theta, \eta \vdash \mathit{exp}.\mathit{mn}(\mathit{exp}_1, ..., \mathit{exp}_n) \Rightarrow v, \Sigma_{n+2} & (12) \end{array}$$

Figure 3: Object-oriented `call` evaluation rule

A general object reference is represented by ρ. The function application is denoted as $f \bullet x$, and the entirely undefined function is denoted as $\perp$. The evaluation of a method call yields its value (v') and an updated object store (Σ'_{n+2}).

With the MCI extension, the `call` rule is changed to verify at certain points, if there is a registered event that should be executed. Figure 4 shows the `call` rule with the MCI extension. The lookup for registered events matching this method's execution is done through the helper functions `dispatch` (15), `enter` (20), and `exit` (22).

An event can be registered using the `superimpose` construct. The lookup functions showed in the MCI call rule, search the

$$\begin{array}{lr} T, \Sigma_0, \theta, \eta \vdash \mathit{exp} \Rightarrow \rho, \Sigma_1 & (13) \\ \wedge\ \Pi_1(T) \bullet (\rho, \mathit{mn}) = ((\mathit{vn}_1, ..., \mathit{vn}_n), \mathit{exp}') & (14) \\ \wedge\ \texttt{dispatch}(T, \Sigma_1, \theta, (\rho, \mathit{mn})) \Rightarrow \Sigma'_1 & (15) \\ \wedge\ T, \Sigma'_1, \theta, \eta \vdash \mathit{exp}_1 \Rightarrow v_1, \Sigma_2 & (16) \\ \wedge\ ... & (17) \\ \wedge\ T, \Sigma_n, \theta, \eta \vdash \mathit{exp}_n \Rightarrow v_n, \Sigma_{n+1} & (18) \\ \wedge\ \eta' = \perp [\mathit{vn}_1 \mapsto v_1, ..., \mathit{vn}_n \mapsto v_n] & (19) \\ \wedge\ \texttt{enter}(T, \Sigma_{n+1}, \theta, (\rho, \mathit{mn}), \eta') \Rightarrow \Sigma'_{n+1} & (20) \\ \wedge\ T, \Sigma'_{n+1}, ((\rho, \mathit{mn}), \perp) \vdash \mathit{exp}' \Rightarrow v, \Sigma_{n+2} & (21) \\ \wedge\ \texttt{exit}(T, \Sigma_{n+2}, \theta, (\rho, \mathit{mn}), \eta', v) \Rightarrow v', \Sigma'_{n+2} & (22) \\ \hline T, \Sigma_0, \theta, \eta \vdash \mathit{exp}.\mathit{mn}(\mathit{exp}_1, ..., \mathit{exp}_n) \Rightarrow v', \Sigma'_{n+2} & (23) \end{array}$$

Figure 4: MCI `call` evaluation rule

environment to see if the registered event matches the executing method. If there is a match, the registered expression is executed. Note that the `superimpose` must be evaluated before the method call for the advice to take effect. Any method calls made before the `superimpose` evaluation will behave according to the μO^2 rule because the environment will not have a registered event. This feature allows us to dynamically introduce advices, which is not possible in AspectJ.

As we want to map the MCI semantics to AspectJ, we need to constrain the language to ensure that all `superimpose` expressions are evaluated before the program starts executing. This can be achieved by allowing `superimpose` declarations only at the beginning of the main method (method called to initiate the program execution according to the language grammar, see *prog* in Figure 1).

It is possible to represent part of the advice types provided by AspectJ using the `superimpose` construct. In fact, we can represent `before-call`, `before-execution` and `after-returning-execution` advices. The first type maps to a `superimpose on dispatch` construct, the other two can be mapped to `superimpose on enter` and `superimpose on exit` constructions, respectively.

Other AspectJ constructs, including pointcuts, inter-type declarations, and other kinds of advice, can not be represented with the MCI semantics. This limitation enables us to reason only about Laws 3 - *Add before-execution*, 4 - *Add before-call*, 7 - *Add after-execution returning*, 13 - *Merge advices*, 15 - *Remove target parameter*, and 14 - *Remove this parameter*. In Section 4 we discuss the soundness of Law 3 - (*Add Before-Execution*). To enable the proof of the other laws, it would be necessary to extend the showed language, or to define a completely new one. This is regarded as a future work.

2.1 MCI Program Equivalence

We want to use the MCI semantics to reason about aspect-oriented programs and verify whether two programs behave the same. Thus, it is necessary to define an equivalence relation between them. This equivalence relation can be difficult to define. For instance, if we choose an equivalence relation that compares two environments (states) resulting from programs execution, it would fail to compare programs that behave the same but use different data structures. Different data structures may result in different environments at the end of a program execution. For example, consider two stack implementations: the first uses an array to represent the stack, and the second uses a linked list. Both implementations may behave as a stack, but their final states are different because their data structures are different. In this case, it would be necessary to isolate input and output variables from the environment and compare only those variables.

As the programming laws we are willing to proof, with the MCI semantics, do not change the data structure, we can establish equivalence by comparing the object stores generated by the evaluation of both programs. Figure 5 shows the object store (Σ) domain to evaluate an expression [15]. This domain has three components: a function that associates data locations with their values ($\delta \rightarrow_{fin} v$), a function that associates object references with their types ($\rho \rightarrow_{fin} cn$), and the advice registry ($\overline{\Delta}$). Our equivalence notion only uses the first component of the object store comparing the field values and how they change, as stated by Definition 1.

$$\boxed{\begin{array}{lcll}
\Sigma & = & \delta \rightarrow_{fin} v & \text{(Object store)} \\
 & \times & \rho \rightarrow_{fin} cn & \text{(Runtime type information)} \\
 & \times & \overline{\Delta} & \text{(Advice registry)} \\
\delta & = & \rho \times fn & \text{(Data locations)} \\
\rho & & & \text{(Object references)}
\end{array}}$$

Figure 5: Object Store

Definition 1 (Program Equivalence). *Let P and Q be two MCI programs. P is equivalent to Q ($P \equiv Q$) iff, for all valid input, the fields and their values from the resulting object store of P equals that of Q.*

We are only interested in the first component from the object store, which maps field locations to their values. Thus, after the programs evaluation we can compare the values of their fields and state that two programs behave the same if all their fields and values are equal. The runtime type information is not relevant to our relation, it is part of the object store to allow the evaluation of expressions like type casts. The advice registry is expected to change because we intend to introduce new `superimpose` commands to the program.

This equivalence notion is rather strong. It may distinguish two programs even if they have the same behaviour. The stack implementations using an array or a linked list would be different programs according to our definition. This is not a problem because two programs that are equivalent according to our definition, would also be equivalent using a more precise definition. Besides, our definition is the simplest solution suitable to our goals. Also, note that we are interested on the external behaviour of a program. Hence, our definition deals with closed programs and not with equivalence of classes. For instance, a method never called by a program do not influence the equivalence notion because its behaviour do not contribute to the external program behaviour.

Although we define the equivalence relation for MCI, this notion is independent of programming languages. However, this equivalence relation can only be considered for sequential programs. If the programs are concurrent, the equivalence relation should consider the structure of the evaluation tree as well. Nevertheless, our laws do not deal with those mechanisms.

3. LAWS

Sometimes, modifications required by refactorings are difficult to understand as they might perform global code changes. We use programming laws [5] to increase the confidence that an AspectJ transformation preserves behaviour. The laws are much simpler than most refactorings because they involve only localized changes, and each one focuses on one specific AspectJ construct. The laws form a basis for defining refactorings with confidence that they preserve behaviour.

In this section we describe a simple law, showing its intent, structure, and preconditions. The laws establish the equivalence of AspectJ programs given that some restrictions are respected. Therefore, the structure of each law consists of three parts: left-side, right-side and preconditions. The first two are templates of the equivalent programs. The third part indicates conditions that must hold to ensure the equivalence is valid. For example, the following law is useful to extract code from the beginning of a method into an aspect. If the extracted code is spread through several methods, we would apply the law several times to isolate this code. Afterwards, we would use another law to merge the resulting advices, increasing reuse.

Law 3. Add Before-Execution

```
ts
class C {
  fs
  ms
  T m(ps) {
    body';
    body
  }
}
paspect A {
  pcs
  bars
  afs
}
```

=

```
ts
class C {
  fs
  ms
  T m(ps) {
    body
  }
}
paspect A {
  pcs
  bars
  before(context) :
   exec(σ(C.m)) &&
   bind(context) {
    body'[cthis/this]
  }
  afs
}
```

provided

($\rightarrow$) *body*$'$ does not declare or use local variables; *body*$'$ does not call `super`;

($\leftarrow$) *body*$'$ does not call `return`;

($\leftrightarrow$) A has the lowest precedence on the join points involving the signature $\sigma(C.m)$;

The laws basically represent two transformations, one applying the law from left to right and another one in the opposite direction. Each law has preconditions to ensure that the program is valid after the transformation and preconditions to ensure that the transformation preserves behaviour. When applied from left to right, this law moves part of a method's body into an advice that is triggered before method execution.

We denote the set of class and aspect declarations by *ts*, and the set of field declarations and method declarations by *fs* and *ms*, respectively. We also abstract the `privileged` modifier from AspectJ as `priv`. The set of pointcut declarations is denoted as *pcs*. Note that the advices can not be considered as a set, since order of declaration dictates precedence of advices. According to the AspectJ semantics, if two advices are *after*, the one declared later has precedence, in every other case, the advice declared first has precedence. Thus, we divide the list of advices in two. The first part (*bars*) contains the list of all `before` and `around` advices, while the second part contains only `after` advices (*afs*). This separation ensures that `after` advices always appear at the end of the aspect. It also allows us to define exactly the point where the new advice should be placed to execute in the same order in both sides of the law. Additionally, for advices declared in different aspects, precedence depends on their hierarchy or their order in a `declare precedence` construct.

Inside advices, we can access variables in the context of the captured join point. The law always expose the maximum context available, in this case, the executing object (`this`(*cthis*)) and the method parameters (`args`(*ps*)). The expression *bind(context)* includes those pointcut designators for exposing context. We omit visibility modifiers, `throws` clauses and inheritance constructs for simplicity. However, there are similar laws that include the variations of visibility modifiers, exceptions and inheritance constructs.

Examining the left hand side of Law 3, we see that $body'$ executes after all `before` advices declared for this join point. It also executes after all the `around` advices, intercepting this join point, call `proceed`. This means that the new advice on the right hand side of the law should be the last one to execute, preserving the order in which the code is executed in both sides of the law. Thus, the `before` advice should be placed after the list of `before` and `around` advices, but before the list of `after` advices. Moreover, to ensure that the new advice created with Law 3 is the last one to execute, we have a precondition stating that aspect A has the lowest precedence over other aspects defined in *ts*. This precondition must hold in both directions.

As we move $body'$ to the aspect, its visible context changes. Hence, it is necessary to constrain the context dependencies in order to guarantee that the law relates valid AspectJ programs. Therefore, we impose conditions on accessing private members, local variables (not including the methods arguments) and calls to `super`. While the last two are forbidden, access to private members is allowed if done through `this`. This is necessary to enable the mapping of accesses to the object referenced by `this`, to the object exposed as the executing object on the advice (*cthis*). The mapping is denoted by the expression $body'[cthis/\texttt{this}]$, where we substitute all occurrences of `this` for the variable *cthis* in $body'$.

However, there are other implications that must be considered. Changes to the method execution flow (calls to `return`) are generally not allowed because the advice cannot implement it, or it would increase complexity. This precondition is necessary to ensure that the law preserves behaviour.

Other laws are similarly defined in terms of transformations and preconditions, and establish properties of other constructs besides `before` advice. Table 1 shows a summary of the laws. More details about AspectJ programming laws can be found elsewhere [5].

4. SOUNDNESS OF THE ADD BEFORE-EXECUTION LAW

In this section we show that the Law 3 (*Add Before-Execution*) is sound using the semantics we chose. We interpret both sides of the law according to the semantics. Then we compare the resulting environments according to our equivalence notion to see whether the two sides of the law have the same meaning.

Following, we show the Law 3 written in terms of the MCI syntax. Thus, we map the `before-execution` advice from AspectJ to a `superimpose on enter` construct from the MCI language (see Section 2). Also, we constrain the language allowing only declarations of the `superimpose` construct at the beginning of the main method. Moreover, the MCI language does not have any modular concept similar to an aspect. Thus, the aspect simulation is also accomplished by the use of a main method with `superimpose` declarations at the beginning. As a consequence, changes made to the aspect are represented as changes made to the main method and its superimposes. Note that, similarly to the AspectJ law, we have to substitute the `this` keyword for the `callee` keyword when using $body'$ on the right hand side of the law.

Law 3. Add Before-Execution (MCI)

```
ts
class C ext T {
 fs
 ms
 Type m(ps) {
   body';
   body
 }
}
class M ext T {
 void main() {
   sis;
   mainBody
 }
}
```

=

```
ts
class C ext T {
 fs
 ms
 Type m(ps) {
   body
 }
}
class M ext T {
 void main() {
   superimpose body'
     on enter
       class C &&
       method m &&
       argument ps;
   sis;
   mainBody
 }
}
```

Table 1: Summary of laws

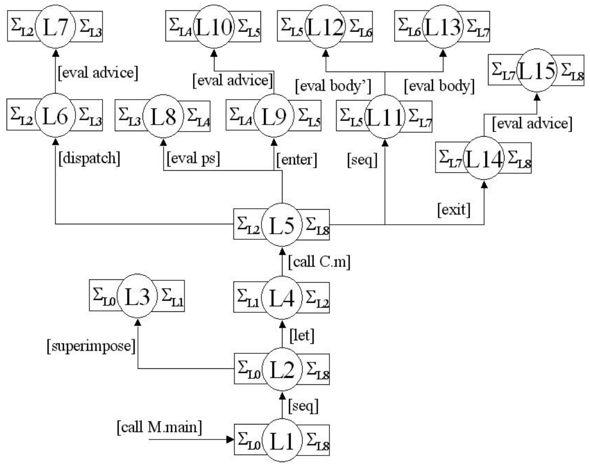
Law	Name	Law	Name
1	Add empty aspect	16	Remove argument parameter
2	Make aspect privileged	17	Add catch softened exception
3	Add before-execution	18	Soften exception
4	Add before-call	19	Remove exception from throws clause
5	Add after-execution	20	Remove exception handling
6	Add after-call	21	Move exception handling to aspect
7	Add after-execution returning successfully	22	Move field to aspect
8	Add after-call returning successfully	23	Move method to aspect
9	Add after-execution throwing exceptions	24	Move implements declaration to aspect
10	Add after-call throwing exceptions	25	Move extends declaration to aspect
11	Add around-execution	26	Extract named pointcut
12	Add around-call	27	Use named pointcut
13	Merge advices	28	Move field introduction up to interface
14	Remove `this` parameter	29	Move method introduction up to interface
15	Remove `target` parameter	30	Remove method implementation

There is also the advice ordering problem discussed in Section 3. According to our understanding from the MCI semantics, advices declared later have precedence, no matter the kind of MCI. Thus, we do not need to separate advices as we do with AspectJ. It is only necessary to declare the new `superimpose on enter`, just before all the other `superimpose` declarations (*sis*) to ensure that the new one is the last to be executed. If we were dealing with Law 7 (*Add after-execution returning successfully*), the new `superimpose` declaration should be placed after all the existing ones to ensure that the `after` advice should be the first to execute. We assume that the kind of rewriting discussed so far, does not change the semantics of Law 3.

In Section 2 we showed that there is just one evaluation rule that changes with the MCI extension. Thus, our soundness discussion involves only the `call` rule. A complete proof would involve all the language constructs and use induction on the structure of *mainBody*. The base case would consider each single command that can appear in *mainBody*, while the induction step would consider every composition of those commands. This complete proof is regarded as a future work, here we provide a formal argumentation to show that Law 3 is sound.

Our argumentation is based on a case where the *mainBody* represents a single call to method m of class C (note that we need to create an object, using the `let` construct, to call a method). This comes directly from the fact that the `superimpose` only affects the method call semantics. Any other simple construction for *mainBody* would trivially preserve behaviour because the other language constructs are not affected by the `superimpose`.

Figure 6 shows the evaluation tree for the left hand side of the law, considering that *mainBody* is the command: `let` $c : C =$ `new` C `in` $c.m(ps)$. Every node consists of a program state. The transitions represent applications of transition rules according to the semantics. Thus, each transition is labeled after the applied rule. Also, the left square represent the input object store and the right square represents the output object store for each rule applied. The nodes are numbered according to the execution order, with label L1 being the first.



Σ_{L2} L7 Σ_{L3}
Σ_{L4} L10 Σ_{L5}
Σ_{L5} L12 Σ_{L6}
Σ_{L6} L13 Σ_{L7}
Σ_{L7} L15 Σ_{L8}
[eval advice]
[eval advice]
[eval body']
[eval body]
Σ_{L2} L6 Σ_{L3}
Σ_{L3} L8 Σ_{L4}
Σ_{L4} L9 Σ_{L5}
Σ_{L5} L11 Σ_{L7}
[eval advice]
[dispatch]
[eval ps]
[enter]
[seq]
Σ_{L7} L14 Σ_{L8}
[exit]
Σ_{L2} L5 Σ_{L8}
[call C.m]
Σ_{L0} L3 Σ_{L1}
Σ_{L1} L4 Σ_{L2}
[let]
[superimpose]
Σ_{L0} L2 Σ_{L8}
[seq]
[call M.main]
Σ_{L0} L1 Σ_{L8}

Figure 6: Evaluation tree for the left hand side.

The left hand side consists in evaluating a sequential composition (L2), which leads to the evaluation of the `superimpose` declarations present in *sis* (L3) and the evaluation of the `let` command (L4). The let updates the store and calls method m of class C (L5). The method call evaluation occurs as showed in Figure 4. First, events registered for `dispatch` MCI are executed (L7). Next we evaluate the method's parameters (L8). Then, events registered for `enter` MCI are executed (L10). Following we evaluate the method's body, which is a sequential composition (L11) of body' (L12) and body (L13). Finally, events registered for `exit` MCI are executed (L15). As we want to compare the execution of two programs, we do not expand execution nodes that are equal for both. For instance, the evaluation of *body*, *body*$'$, *ps*, `dispatch` and `exit` advice nodes are the same for both programs.

Next, Figure 7 shows the evaluation tree for the right hand side of the law. In this case, there is a sequential composition(R2) that first evaluates another sequential composition (R3), which includes our new `superimpose` (R4) and the old ones (R5). Then it starts the program similarly to the left hand side. The evaluation of the `superimpose` command updates the registry located on the object store by registering *body′* to be executed when entering the method m with arguments *ps* of class C. As a result, the evaluation of the `enter` helper function (R11) performs a lookup in the registry for events registered for this method and finds that *body′* should be executed (R12). Another difference is that the evaluation of the method's body now includes only *body* (R14).

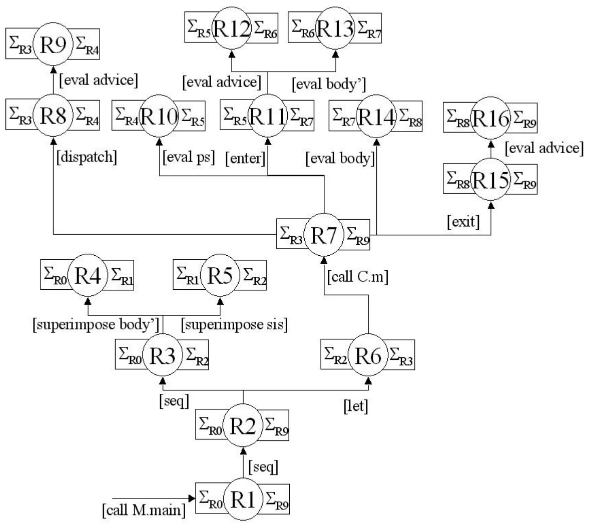


Figure 7: Evaluation tree for the right hand side.

Proof. (Sketch) According to the equivalence notion established in Section 2.1, we are interested on the nodes that may update the first component of the object store (field values). First, the `let` command may update the object store by adding a new object and the values of its fields. The second way to update the field values in the object store is through an assignment. Assignments can appear in any expression and thus, we look for the nodes able to evaluate expressions.

On the left hand side, the nodes related to the evaluation of expressions are: `let` (L4), `dispatch` (L7), *ps* (L8), `enter` (L10), *body′* (L12), *body* (L13), and `exit` (L15). Similarly, the nodes we are interested on the right hand side are: `let` (R6), `dispatch` (R9), *ps* (R10), `enter` (R12), *body′* (R13), *body* (R14), and `exit` (R16).

Analyzing the equivalent nodes from both programs (i.e. L4 and R6, L7 and R9, etc) we can see they are syntactically equal, and thus have an equivalent evaluation. The only factor that may result in different field values at the end of the program execution is the order in which the nodes are evaluated. In both Figures 6 and 7, the number inside the node represent the order of evaluation, which is the same in both programs. During the evaluation, the field values are supposed to be equal after the evaluation of nodes L13 and R14. Thus, according to our equivalence notion, and considering that the programs are sequential, we can conclude that the programs have the same behaviour. ■

4.1 Soundness of Other Laws

This proof could be similarly extended for Laws 4 (*Add before-call*), and 7 (*Add after returning successfully*). As they only differ by the kind of advice (MCI) used. Law 4 would use the `superimpose on dispatch` construct and Law 7 would use the `superimpose on exit` construct. For this reason, we consider that this two laws are also sound.

The right hand side of Law 4 would generate an evaluation tree where *body′* is evaluated before some other method call. This means that *body′* is evaluated even before the arguments of the method to be called. The evaluation tree for the left hand side would place *body′* above the `dispatch` node, ensuring that it is also evaluated before the arguments of the considered method.

The proof for Law 7 is almost equal to the proof for Law 3. The only difference is that on the evaluation tree for the left hand side, *body′* appears after *body*, and on the right hand side, *body′* appears above the `exit` node. This also ensures that *body′* is evaluated after *body* in both sides of the law.

However, Laws 13, 14, and 15 should be considered differently. The proof for Law 13 would rely on the composition of MCI locations (|| operand on event locations) to ensure that a registered event matches two or more join points. As the only difference between the left hand side and right hand side is the `superimpose` declarations (consequently the registry), both evaluation trees would be equal. According to the MCI semantics, the evaluation of the || operator is the same as evaluating its first operand an then its second operand. Both evaluations register the same piece of code to execute at different events.

The proof for Laws 15 and 14 would rely on removing the `callee` and `caller` constructs respectively. In the MCI semantics, these constructs only bind variables to be used by the advice, they do not constrain the types as occurs with `this` and `target` in AspectJ. As type restrictions are apart from variable binding, we can remove the variable binding given that the variable is not used inside the advice.

We do not discuss the remaining laws formally. As most laws are very simple and intuitive, since each one deals with one construct at a time, their description provides informal arguments describing why the two sides of the laws are equivalent. Hence, we generally described how to map an AspectJ construct to its corresponding Java implementation. Moreover, some laws when applied from right to left, perform a transformation very similar to the transformation applied by the AspectJ compiler to weave aspects and classes.

5. RELATED WORK

This paper uses an existing operational semantics for Method Call Interception [15] to represent aspect-oriented programming laws and reason about them. It seemed appropriate to choose this semantics because of its simplicity, its model of extending an object-oriented language, and its capacity to represent several types of advices from AspectJ.

However, there are other approaches for reasoning about aspect-oriented programs. It would be difficult to represent the laws using most of them. Douence et. al. [6] define a domain-specific language, along with its semantics, to define crosscuts based no execution monitoring. His system is based on events similarly to the Observer [8] pattern.

Andrews [2] presents process algebras as a formal basis for aspect-oriented languages. He uses a subset of CSP tailored to his purpose, representing join points as synchronization sets. He also defines an equivalence notion between programs and uses it to show the correctness of his weaving process. He uses an imperative language. We use the MCI semantics because it is much simpler and extends the semantics of an object-oriented language just as AspectJ extends Java.

Wand et. al. [19] define a semantic model for dynamic join points. This is not appropriate to our purpose because we needed a semantics in which we could represent AspectJ features. Xu et. al. [20] use a reduction strategy to transform aspect-oriented programs to implicit invocation. This transformation allows them to reason about the programs using already defined semantics for implicit invocation. Aldrich [1] discusses the problem of modular reasoning about aspect-oriented programs. He defines an aspect-oriented language and associated semantics where modular reasoning is possible. Finally, Barzilay et. al. [3] examine call and execution semantics in AspectJ and their interaction with inheritance.

There is also a related work [4] that includes the definition of object-oriented programming laws, an associated semantics, an equivalence notion, and soundness of the laws. Besides, they also prove the relative completeness of their set of laws by defining a normal form and a reduction strategy to transform any program into the normal form.

Hanenberg, Oberschulte and Unland [10] propose some preconditions to apply an object-oriented refactoring in the presence of aspects. Those conditions guarantee a mapping of join points during refactoring, therefore preserving behaviour. They also propose modifications to refactorings such as *Extract Class* [7] in order to make them aspect-aware and therefore respect the preconditions. The second part of Hanenberg, Oberschulte and Unland's research regards refactorings to AspectJ. In fact, they propose some new refactorings from Java to AspectJ. However, they only discuss the refactoring as a whole and the conditions to apply the refactoring. We not only define preconditions but we are able to prove that our transformations preserve behaviour. We also derived the proposed refactorings using the laws, showing that they preserve behaviour.

Analogously, Iwamoto and Zhao [12] proposes modifications to existing refactorings in order to make them aspect-aware. However, it is a superficial discussion. They only show some examples and give some guidelines on how to avoid the aspect effects on the object-oriented refactorings. They also show examples of refactorings from Java to AspectJ. Although, there is no argumentation about necessary conditions to apply the refactorings to ensure that they preserve behaviour. We used the suggested refactorings and derived them as a composition of laws. Hence, we were able to state in which conditions we can apply the refactorings as well.

Finally, there is a related work [14] that discusses aspect-oriented refactorings showing problems when applying object-oriented refactorings in the presence of aspects. It proposes several complex and interesting refactorings and shows clear and easy to understand examples. The laws we are able to prove with the discussed semantics are enough to prove some of his refactorings, for instance the *Extract Method Calls* refactoring.

6. CONCLUSIONS

This paper is a complement to another work on aspect-oriented programming laws [5]. The previous work relied on the simplicity of the laws, which involve only local changes and deal with one AspectJ construct each. Here we show that the laws can be proved sound according to a formal semantics. We show that specifically Law 3 (*Add Before-Execution*). However, other five laws could be chosen.

For that, we use an operational semantics for Method Call Interception [15] where we could represent some of the laws. We also defined an equivalence relation stating the conditions in which two programs behave the same. The proof is based on the evaluation of both programs and then, the analysis of the resulting environments comparing the values of object fields.

However, we can not prove all the laws using this semantics. The MCI semantics is able to represent only `before-call`, `before-execution` and `after-execution returning` advices from AspectJ. Thus, we can only reason about the laws related to those advices. To enable the proof of the remaining laws, we should define a completely new language, along with its semantics, including all the AspectJ constructs covered by the laws. Another solution would be to extend an existing language (i.e MCI) to incorporate the missing constructs. Our current solution allows the proof of Laws 3 - *Add before-execution*, 4 - *Add before-call*, 7 - *Add after-execution returning*, 13 - *Merge advices*, 15 - *Remove target parameter*, and 14 - *Remove this parameter*. The proof of other laws is regarded as a future work.

7. ACKNOWLEDGMENTS

We would like to thank Rohit Gheyi and the other Software Productivity Group members who contributed with comments and suggestions. We also would like to thank the anonymous referees for making several suggestions that significantly improved our paper. This work was supported by CAPES and CNPq, both are Brazilian research agencies.

8. REFERENCES

[1] J. Aldrich. Open Modules: A proposal for Modular Reasoning In Aspect-Oriented Programming. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, Mar. 2004.

[2] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, pages 187–209. Springer-Verlag, Sept 2001.

[3] O. Barzilay, Y. Feldman, S. Tyszberowicz, and A. Yehudai. Call and Execution Semantics in AspectJ. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, Mar. 2004.

[4] P. H. M. Borba, A. C. A. Sampaio, A. L. C. Cavalcanti, and M. L. Cornelio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, January 2004.

[5] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, Mar. 2005. ACM Press. To appear.

[6] R. Douence, O. Motelet, and M. Sudholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192, pages 170–186. Springer-Verlag, Sept 2001.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison–Wesley, second edition, 2000.

[10] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35, Erfurt, Germany, Sept. 2003.

[11] C. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[12] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kandé, D. Stein, P. Tarr, and A. Zakaria, editors, *The 4th AOSD Modeling With UML Workshop*, 2003.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[14] R. Laddad. Aspect-Oriented Refactoring Series. TheServerSide.com, Dec. 2003.

[15] R. Lämmel. A Semantical Approach to Method-Call Interception. In G. Kiczales, editor, *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, Apr. 2002. ACM Press.

[16] M. Monteiro and J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Chicago, USA, Mar. 2005. ACM Press.

[17] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[18] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, Urbana-Champaign, IL, USA, 1999.

[19] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 1–8. Department of Computer Science, Iowa State University, Apr. 2002.

[20] J. Xu, H. Rajan, and K. Sullivan. Aspect Reasoning by Reduction to Implicit Invocation. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004; Technical Report CS Dept., Iowa State Univ.*, Mar. 2004.

A join point for loops in AspectJ

Bruno Harbulot
bruno.harbulot@cs.man.ac.uk

John R. Gurd
jgurd@cs.man.ac.uk

Centre for Novel Computing, School of Computer Science,
University of Manchester, Oxford Road, Manchester M13 9PL, UK

ABSTRACT

The current AspectJ join points represent locations in the code that are at the interface of the Java objects. However, not all the "things that happen"[1] happen at the interfaces. In particular, loops are a key place that could be advised for parallelisation. Although parallelisation via aspects can be performed in certain cases by refactoring the Java code, it is not always possible or desirable. This article presents a model of loop join point, which allows AspectJ to intervene directly in loops.
The approach used for recognising loops is based on a control-flow analysis at the bytecode level; this avoids ambiguities due to alternative forms of source-code that would effectively produce identical loops. This model is also embellished with a mechanism for context exposure, which is pivotal for giving a meaning to the use of this join point. This context exposure is particularly useful for writing pointcuts that select specific loops only, and the problem of loop selection is also presented in the paper.
Finally, *LoopsAJ*, an extension for the `abc` compiler that provides AspectJ with a loop join point, is presented. It is shown how to use this extension for writing aspects that parallelise loops.

1. INTRODUCTION

When parallelising code in order to improve performance, loops are the natural places to make changes. There are sometimes several alternative ways of parallelising the same loop, depending on various parameters, such as the nature of the data being processed, or the architecture on which the application is going to be executed. In certain cases, it is possible to use aspects for parallelising loops, in particular for choosing a method of parallelisation [3]. However, since there is currently no join point for loops in AspectJ [5], the method proposed in [3] resorts to refactoring the base-code. In order to eliminate this inconvenience, this paper proposes a loop join point model for AspectJ which allows direct parallelisation of loops, without refactoring of the base-code.

[1](to use the AspectJ guide phrasing for introducing the concept of join point).

Foundations Of Aspect-oriented Languages workshop (*FOAL 2005*), held in conjunction with *AOSD 2005* in Chicago, USA.

Section 2 presents a formal definition of the loop join point model. This includes the definition of a loop and the way it can be identified. Although this approach is based on Java and AspectJ, the model can potentially be applied to other languages. Section 3 embellishes the loop join point model with a relation to the data handled by the loops. Section 4 explains the specific requirements for loop selection, and describes the associated difficulties, compared with other kind of join points. Section 5 introduces *LoopsAJ*, a prototype implementation of a weaver capable of handling the loop join point model, based on `abc` [2].Section 6 shows how to write aspects for parallelisation using the loop join point. Section 7 describes some of the problems related to base-code containing exceptions. Section 8 briefly introduces ideas about other potential fine-grained join points: a "loop-body" join point and an "if-then-else" join point. Finally, Section 9 concludes.

2. LOOP JOIN POINT MODEL

This section presents the definition of a loop join point model. It could be applied to various aspect-oriented systems, but the presentation focusses on the approach used in AspectJ. Section 2.1 describes the general approach used to recognise loops in the code. Section 2.2 gives a summary of compiler theory related to loop recognition. Finally, in Section 2.3, the definition of a loop is progressively restricted in order to build a model suitable for a join point.

The first step is to identify what the *shadow* of the loop join point is. The shadow of a join point is defined as follows: "*[A] join point is a point in the dynamic call graph of a running program [...]. Every [such] dynamic join point has a corresponding static shadow in the source code or bytecode of the program. The AspectJ compiler inserts code at these static shadows in order to modify the dynamic behavior of the program*" [4]. The main elements required for a join point shadow are:

- a weaving point for *before-advice*,
- a weaving point (or maybe several points) for *after-advice*, and
- the eventual ability to weave *around-advice*.

Then, for the dynamic part, the model should make it possible to extract information regarding the execution context at the join point.

2.1 From source or from bytecode

The first decision to be made is whether the join point is recognised at source code level or at bytecode level. The way loops are programmed in Java is not necessarily directly reflected in the generated bytecode. For example, instinctively, most Java programmers would consider the body of a for-loop to be the lines of code within the curly brackets following the `for(;;)` statement. However, a loop with the same effect can also be written in different ways, for example as a while-loop, or with some of the `for` statements displaced, as shown in Figure 1.

```
for (int i = 0 ; i < MAX ; i++) {
   /* A */
}

int j = 0 ;
int STRIDE = 1 ;
for ( ; j < MAX ; j += STRIDE ) {
   /* A */
}

int k = 0 ;
while (k < MAX) {
   /* A */
   k++ ;
}
```

Figure 1: Simple examples of equivalent loops.

In addition, the main conditional expression of a loop may encompass several instructions, in particular if it involves a call to a method or a complex expression, as shown in Figure 2. Although the condition may not seem to be part of the loop body, it could always be refactored so as to be part of it (for example through a temporary `boolean` variable). Moreover, the compiled code does not necessarily reflect the way a complex expression has been written in the source code.

```
int i = 0 ;
while (condition(i) || (i>10)) {
   /* A */
   i++ ;
}

int j = 0 ;
boolean ok = condition(j) || (j>10) ;
while (ok) {
   /* A */
   j++ ;
   ok = condition(j) || (j>10) ;
}
```

Figure 2: Loop with more complex conditions.

Since the main concern is to recognise the behaviour of the code, rather than the way it was written, the choice was made to base the representation of loops at the bytecode level rather than at the source code level. As a result, the representation is more robust to variations in programming style. However, this choice introduces limitations regarding (a) the potential specific handling of abrupt exit (see Section 2.4), and (b) the nature of the control-flow graphs. Indeed, as explained in more detail in Section 7, the model expects a reducible (or well-structured [1, 10]) graph. When exceptions are not used, Java source-code produces bytecode with reducible control-flow graphs, but this is not necessarily the case for bytecode produced by other means.

2.2 Dominators, back edges and natural loops

The initial approach for finding loops in the control-flow graph follows the method described in [1, 10]. This method is based on finding *dominators* and *back edges*, defined as follows: "*Node* d *of a flow graph* dominates *node* n *[...] if every path from the initial node of the flow graph to* n *goes through* d. *[... The] edges in the flow graph whose heads dominate their tails [are called]* back edges. *(If* a $\rightarrow$ b *is an edge,* b *is the* head*,* a *is the* tail*.) [... Also,* a *is a* predecessor *of* b*, and* b *is a* successor *of* a *...] Given a* back edge n $\rightarrow$ d*, we define the* natural loop *of the edge to be* d *plus the set of nodes that can reach* n *without going through* d*. Node* d *is called the* header *of the loop*" [1].

Figures 3(a) and 3(b) represent, respectively, the (block-level [2]) control-flow graph and the associated dominator tree for the simple `for`-loop shown in Figure 1. In this example, the only back edge is 3 $\rightarrow$ 2, and its natural loop comprises blocks (nodes) 2 (which is the header) and 3.

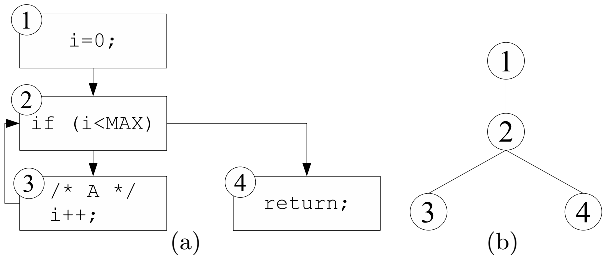


Figure 3: Control-flow graph (a) and dominator tree (b) for a simple `for`-loop.

Natural loops can be confusing because there could be several loops with the same header. As shown in Figure 4, what appears to be a single loop actually corresponds to two natural loops sharing the same header. In such a case, defining the points immediately *before* or *after* a natural loop would be ambiguous. Therefore, instead of using natural loops for the join point model, the union of all the natural loops sharing the same header is considered as a single *combined loop*. To avoid ambiguous cases, implementations should consider a node containing only an unconditional `goto` as the same node as its successor node. In the remainder of this article, the term "*loop*" will be used to mean a "combined loop", unless otherwise stated.

Following this style, an *inner loop* is a loop whose blocks are all contained within another loop, but do not share the latter's header. This also happens to match the natural definition of inner loops at the source level.

In the following sections, three categories of loops are presented, together with their characteristics pertinent to possible use as join points. The categories introduce increasing

[2]i.e., the nodes of the control-flow graph are *basic blocks* [1] of code statements.

```
int i = 0 ;
while (i<MAX) {
   if (cond(i++)) {
      /* A */
   } else {
      /* B */
   }
}
```

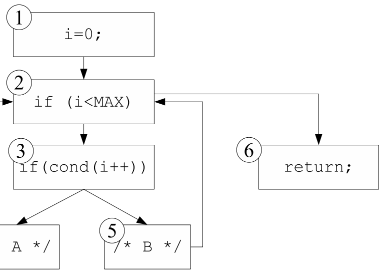


Figure 4: Two natural loops with the same header.

degrees of constraint which affect their ability to weave the three forms of advice: *before*, *after* and *around*.

2.3 Loops in the general case

A loop always has a unique entry point, namely its header. *Before-advice* can therefore be woven in a *pre-header*, that is, a node (block) inserted before the header to which the jumps from outside the considered loop are redirected, but the jumps from inside it are not (see Figure 5).

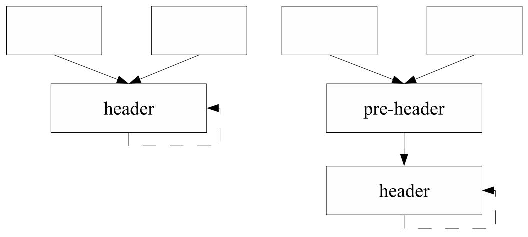


Figure 5: Insertion of a pre-header.

Without further constraint, it cannot be guaranteed that there is a unique point in the control flow that is executed immediately after execution of a loop. In order to introduce appropriate constraints, the following definitions are added. A node in a loop is an *exit node* if it can branch outside that loop. A node outside a loop which has predecessors inside that loop is termed a *successor node* of the loop.

Typically, a non-nested loop which contains a `break` statement has two exit nodes and one successor node, while a double loop nest with a `break` statement in the inner loop that branches outside the outer loop has two exit nodes and two successor nodes. For example, Figure 6 shows the source code and the corresponding (block-level) control-flow graph for a doubly nested loop:

- The inner loop consists of blocks 4, 5 and 6; its exit nodes are blocks 4 and 5; its successor nodes are blocks 7 and 8.
- The outer loop consists of blocks 2, 3, 4, 5, 6 and 7; its exit nodes are blocks 2 and 5; its (sole) successor node is block 8.

```
int i = 0 ;
outside:
while (i < maxI) {
   int j = 0 ;
   while (j < maxJ) {
      if (c(i,j))
         break outside ;
      j++ ;
   }
   i++ ;
}
/* A */
```

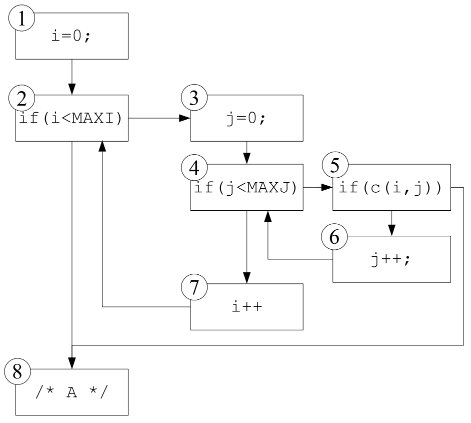


Figure 6: Two nested loops and break statement jumping outside outer-loop.

In this case, "after" loop $\{4, 5, 6\}$ is both on the transitions between blocks 4 and 7, and between blocks 5 and 8.

In such cases, where there are several successor nodes, weaving an *after* piece of advice would require replication of the woven code at all edges between exit nodes and their successor nodes. Although it is, in principle, possible to achieve this, some aspect-oriented tools do not allow this kind of weaving.

2.4 Loops with a unique successor node

The problem of having multiple exit nodes only occurs when there are nested `break` or `continue` statements that branch outside the inner-most loop to which they belong. The default case (of a `break` statement with no label specified) corresponds to an exit node that branches outside the loop, but to the same successor node as the normal exit would go. In this case, weaving an *after* piece of advice could be done either at the end of each exit node (possibly at multiple points, as described previously) or at the beginning of the (unique) successor node (which thus guarantees a single weaving point). Weaving an *after-advice* (at a single weaving point) therefore consists of inserting a *pre-successor*, i.e., a new node inserted prior to the successor node, to which the jumps from the exit nodes to the (unique) successor node are redirected.

A loop with a unique successor node can also be reduced to a single node in the control-flow graph. This then makes it possible to weave an *around-advice* at the join point for the loop.

Just as there are two different constructs for writing after-advice depending on whether the execution returns normally or throws an exception[3], so might be abrupt exits be handled differently (due to `break` statements). However, there are cases where it is not possible to tell from the bytecode how such exits would differ from those due to the main condition of the loop evaluating to false. This is a limitation that might have been avoided if a source-code representation had been used, but it does make the model robust to changes in programming style, as illustrated by the code in Figure 7. The two loops in the figure might well produce the same bytecode and control-flow graph, in which case the use of `break` would not be distinguishable from the use of the "`&&`" operator. It would thus be impossible to treat an exit from the loop due to the `break` statement any differently than an exit from the loop due to `b` evaluating to `false`.

```
while (a && b) {
   /* Do something */
}

while (a) {
   if (!b)
      break ;
   /* Do something */
}
```

Figure 7: Considered special handling of `break` statements.

2.5 Loops with a unique exit node

The full potential of a loop join point can only be exploited if its model comprises information regarding the behaviour of the loop. In particular, it can be useful to predict as far as possible that the loop iterates over a specific range of integers or over an `Iterator` (see Section 3). However clever such a prediction may be, the programmer of an aspect dealing with loops might want to handle cases where there is no possibility of an abrupt exit (*i.e.*, there is no `break` statement in the loop). As shown in Figure 7, this case may also exclude loops with complex conditions (in particular expressions comprising *and* operations, which may create several exit points).

2.6 Summary

Three categories of loops have been identified, with increasing degrees of constraint. All three forms could be implemented by a different pointcut, each with different meaning and weaving capabilities. The more general form (several successor nodes possible) would only allow the weaving of *before*-advice, and possibly *after*-advice if the implementation of the weaver allows multiple weaving points. The intermediate form (unique successor node possible) and the restricted form (only one exit node and one successor node) would allow the weaving of *before*-, *after*- and *around*-advice. The latter also guarantees that there is a single condition for exit from the loop. This information is summarised in Table 2.6; it will be used for context exposure in Section 3.

3. CONTEXT EXPOSURE

Although loops do not have arguments in the same way as other join points (such as method calls), they often depend on contextual information to which programmers may want access. In particular, two forms of contextualised loops are frequently found:

- loops iterating regularly over a range of integers (presented in Section 3.1), and
- loops iterating over an `Iterator` (presented in Section 3.2).

[3] "`after() returning(...):`" executes the advice after a normal execution, "`after() throwing(...):`" executes the advice if an exception has been thrown, and "`after():`" executes the advice in both cases.

	Before	After	Around
several successor nodes	√	multiple weaving points	×
several exit nodes, 1 successor node	√	√	√
1 exit node, 1 successor node	√	√	√

Table 1: Different loop types and their weaving capabilities.

Knowing that a loop is of one of these forms allows one to predict the execution behaviour of the loop in some detail. In order to make the resulting predictions meaningful, only loops with unique exit points and unique successors are considered in this section. This prevents loops which have any potential abrupt exits (e.g., using `break` statements) from consideration; a potential use of `break` would make the finding of a range of integers or of an `Iterator` less useful, since the loop might exit before the predicted end.

3.1 Loop iterating over a range of integers

Loops iterating over a range of integers, following an arithmetic sequence, are one of the most frequent forms of loops. They consist of: initialising an integer local variable before the loop; incrementing this value by a constant (the stride) at the end of each iteration; and exiting the loop when the value reaches a given maximum value. This form of loop follows the pattern shown in Figure 1.

As explained in [3], exposing the iteration space is essential to make it possible to write aspects for parallelisation. The initial value, the stride and the final value will be available in the execution context of the loop join point model, when possible. Since these values are parameters ruling the execution of the loop, they could be considered, in aspect-oriented models such as AspectJ, as "arguments" of the loop.

Predicting what the range of integer values is going to be at the time of execution is not always possible. In order to be exposed to the join point model, these values have to be determinable before the join point is encountered. The availability of these values will depend on the capabilities of the implemented static analysis in the shadow matcher. Determination of these values ought to be implemented in a conservative way, discarding the cases where it cannot be certain that these values will not change during the execution of the loop.

3.2 Loop iterating over an Iterator

Another frequent form of loop (found in particular in Java programs) is that conducted by an `Iterator`. In a manner similar to that presented in Section 3.1, the instance of `Iterator` controlling the loop can be seen as an "argument" to be included in the join point context.

3.3 Parallel with Java 5 for-construct

Java 5 offers a new way to write for-loops iterating over all the elements of an array or `Collection`, similar to "for-each" constructs in certain other languages; this is shown in Figure 8.

```
/* Before Java 5 */
Collection c ;
for (Iterator it = c.iterator() ; it.hasNext(); ) {
   Object obj = it.next() ;
   /* Do something with obj */
}

/* Since Java 5 */
Collection<Object> c ;
for (Object obj: c) {
   /* Do something with obj */
}
```

Figure 8: Example of new Java 5 for-loops.

For iterating over the elements of an array or of a `Collection`, the for-loop construct before Java 5 relies on the abstraction provided by an array index or, respectively, by an `Iterator`. Java 5 gives a new abstraction, more meaningful in terms of data representation. The data guiding the loop execution is directly and explicitly included in the way the for-loop is written in the source code.[4] This is a useful piece of information regarding this kind of loop, and the loop join point model should also be able to expose it, wherever possible. It can also be useful for loop selection, as described in Section 4, and for certain forms of parallelisation, as described in Section 6.

4. LOOP SELECTION

This section analyses and proposes solutions to the problem of writing pointcuts for loops. In particular, the aim is to determine which characteristics can be used for making a selection. In aspect-oriented systems such as AspectJ, the means of selection for a join point is, in most cases, ultimately based on the naming of some source element characterising the join point, possibly by using a regular expression. For example, to advise a method call or a group of methods, the pointcut has to contain an explicit reference to some names characterising the method signatures, whether it be a pattern matching the name of the methods, or a pattern matching the parameter types. Since loops cannot be named, it is impossible to use a name-based pattern to write a pointcut that would select a particular loop.

Neither loop labels, nor Java 5 (or C#) metadata, can be used to identify a particular loop in a method. Firstly, the loop labels will not be kept in the bytecode (and, in any case, they are rarely used, unless motivated by a `break` statement branching outside an inner loop). Secondly, Java 5 metadata cannot be applied to statements (apart from variable declarations).

If it is known for certain that all the loops within a method are to be advised, it would be possible, in AspectJ, to use pointcut constructs such as `withincode` or `cflow` to restrict the pointcut to all the loops contained in the methods traditionally picked up by those constructs. However, selecting only one of several loops within the same method turns out to be impossible without any further mechanism.

In order to solve this problem, it is proposed that selection of loops is made to rely on the data being processed, as well as the method in which the join point's shadow is located. In this case, the context —or what was called the "arguments" of the loop in Section 3— can be used to refine the selection. For example, the programmer might want to write a pointcut that would only select loops iterating over a specified range of integers, over a particular array, or over a particular `Collection`. Such an example is shown in Figure 11 (Section 6): the parallelising advice only applies to arrays of `byte`s.

More speculatively, there might be a potential application for metadata, which could be introduced in the declarations of the local variables that refer to the arrays, `Collection`s or `Iterator`s utilised as "arguments" to certain loops.

5. IMPLEMENTATION IN abc

Although the loop join-point model could potentially be implemented in various aspect-oriented tools, based for example on Java or C#, the focus has been put on a model integrable into AspectJ. The implementation uses `abc`,[5] an alternative AspectJ compiler, for two main reasons:

- extensibility was at the core of the `abc` design [2]; and
- `abc` relies heavily on the Soot framework [12], which provides most of the infrastructure for performing the analyses, in particular those described in Section 2.2.

This section describes an extension for `abc`, known as *LoopsAJ*, which implements a loop join point for AspectJ and subsequently provides the `loop()` pointcut. The latter picks out loops with unique exit points (as described in Section 2.5) and provides contextual information where possible. Other pointcuts for the other forms of loops could also be provided (by lowering the degree of constraint imposed in the shadow matcher).

5.1 Shadow matching

The Soot framework, and subsequently `abc`, use Jimple, which is a three-address representation of bytecode. This makes it possible to look for loops at bytecode level (as described in Section 2.1). The shadow matcher and all pre- or post-transformations operate on this representation.

LoopsAJ extends the method that finds the shadows in each method, so that it looks for loops as well. For each method processed, the control-flow graph and its corresponding dominator tree are built using the Soot framework toolkits. Then combined loops are identified, as described in Section 2.2.

`abc` provides two kinds of classes representing a shadow-match: `BodyShadowMatch` and `StmtShadowMatch` (both extend `ShadowMatch`). The former is utilised when the shadow

[4] This is solely a source-code enhancement; the bytecode still contains `Iterators` (for `Collections`) or temporary variables (for arrays).

[5] `http://www.aspectbench.org/`

is the whole method body; for example, when a method-execution pointcut is used. The latter is used for pinpointing a specific statement (or group of statements) in the method; for example, when a method-call pointcut is used.

One of the requirements of `abc` is to insert `nop` operators in the shadow, at the points where *before* and *after* [6] pieces of advice might be woven. Given this, most of the `abc` infrastructure can already handle loop shadows if they are treated like `StmtShadowMatch` for *before* and *after* pieces of advice.

However, handling *around* pieces of advice requires a few modifications in the `abc` around-weaver [6]. One of the cases where a group of statements is used is the constructor-call shadow match. In this case, two consecutive statements are included in the shadow-match. However, loop shadows are not necessarily formed by consecutive statements. Indeed, at bytecode or Jimple level, the blocks forming a given loop may be spread across the method, with jumps from one block to another leaving blocks that do not belong to the loop interleaved between blocks that do. For this reason, `StmtShadowMatch` has been extended by `NonContiguousStmtGroupShadowMatch`, for which the around weaver has been modified in order to utilise its new type.

5.2 Transformations for context exposure

Exposing the context, as described in Section 3, depends on the cleverness of analysis and on the feasibility of certain transformations. For the context exposed to make sense, it has to be constant during execution of the join point.

In order to ensure this, as long as it is possible to predict that the transformation will not change the meaning of the loop, loop-invariant assignments are moved to the pre-header (before the shadow matching takes place), using a scheme inspired by [1, Ch 10.7].

5.2.1 Exposing the boundaries or the `Iterator`s

Further, in the case of loop iterating over a range of integers, if the context values are numerical constants, temporary variables are introduced and initialised in the pre-header, in order to make it possible to modify these values via calls to `proceed(...)` within an *around-advice*. An example transformation is shown in terms of source-code in Figure 9.

```
/* Moving the invariants outside */
int i = 0 ;
while (i < 10) {
   /* ... */
   int stride = 3 ;
   i = i + stride ;
}
// ------------------------------------------------
/* First step: moving the invariants outside */
int i = 0 ;
int stride = 3 ;
while (i < 10) {
   /* ... */
   i = i + stride ;
}
// ------------------------------------------------
/* Second step: storing the boundaries in
   temporary variables */
int stride = 3 ;
int minimum = 0 ;
int maximum = 10 ;
int i = minimum ;
while (i < maximum) {
    /* ... */
    i = i + stride ;
}
```

Figure 9: Code-motion example.

The part of the implementation that determines the feasibility of these transformations uses the dataflow analysis facilities provided by Soot; these have also been used to implement a code-motion method and a reaching-definition analysis [10, 1].

5.2.2 Exposing the originating array or `Collection`

It is not always possible to find an array to which the range of integers corresponds (i.e. when `minimum=0`, `stride=1`, and `maximum` is the length of the array). For example, if the boundaries and the array are passed as arguments to the containing method, finding the array that was the origin of these values might require much more complex, cross-methods and points-to, analysis. The current implementation requires at least the statement initialising `maximum` to the length of the array to be within the same method as the loop.

Similarly, a `Collection` will only be exposed if the `Iterator` used for the loop comes from a call to `Collection.iterator()` and `Iterator.next()` is not called before the beginning of the loop.

5.2.3 Writing pointcuts

For loops iterating over a range of integers, the boundary values are passed via the `args` construct of AspectJ, to which `int` values are bound (for `minimum`, `maximum` and `stride`). Also, an extra argument will be bound to the originating array, if it has been found.

For loops iterating over an `Iterator`, the first argument of `args` will be bound to the corresponding instance of `Iterator`. Also, an extra argument will be bound to the originating `Collection`, if it has been found.

In cases where the originating array or `Collection` do not matter, it is recommended to use the double-dot wildcard notation ("`..`") [5], to make the argument optional. For example:

- `loop() && args(min, max, stride)` will match only loops iterating over an arithmetic sequence of integers for which the compiler was unable to find an array (although it may exist);
- `loop() && args(min, max, stride, ..)` will match all the loops iterating over a particular arithmetic sequence of integers; and
- `loop() && args(min, max, stride, array)` will match all the loops iterating over an array, a refer-

[6]It is not always possible to insert *after-advice*, as described in Section 2.

ence to which will be bound to pointcut parameter "`array`".

5.3 Limitations

The limitations in the current implementation of *LoopsAJ* divide into two categories: limits of analysis and predictability; and limits due to features not yet written in this preliminary version.

5.3.1 Analysis and predictability

One of the main limitations is the predictability on which the invariant code-motion is based. Although code-motion is currently done successfully in most useful cases, it will not be performed in cases where an invariant is not spotted by the data analysis. The implementation of such transformations ought to be conservative, that is to say, it should not be done unless it is certain that the resulting code will be equivalent.

Another limitation is the lack of points-to analysis in respect of `Iterator`s. Indeed, even though an `Iterator` instance may look like it is being iterated regularly in the loop (*i.e.* there is one and only one call to `next()` per iteration), nothing guarantees that no other thread is holding a reference to the same `Iterator` and is calling `next()` concurrently. In this respect, the exposure of `Iterator`s is probably not conservative enough. There may be a solution to this problem if a form of whole-program analysis were to be used. (This concurrency problem does not occur for loops iterating over a range of integers since `int` is a primitive type and the `int` values are local variables that cannot be modified by another thread.)

More generally, there could be further dependency analysis to provide safeguards in case of concurrent execution of a join point. Again a whole-program analysis may be required to be sure that a loop can be executed in parallel. Such analyses can be much more complicated, and are beyond the scope of this article.

Whatever the implementation of a weaver capable of handling loop join points is, it should be stated clearly by the implementors how conservative their implementation is, in particular, how certain it is that a specified `Collection` is at the origin of an `Iterator`.

5.3.2 Future work

The *LoopsAJ* implementation is still being worked on. In its current state, only cases where the loop termination condition is of the form $i < max$ and where the increment is of the form $i = i + ...$ are handled. Eventually, other conditions, using $\leq$, $>$ or $\geq$, will be handled as well.

Also, the *around* weaving has only been implemented in cases that do not generate closures (see [6] for further details). One of the difficulties being currently addressed is to keep the block graphs up-to-date after weaving an *around*-advice. At the time of writing, the implementation works only partially on loop nests.

Moreover, the handling of traps is not always updated, which can lead to the generation of bytecode with incorrect exception tables.

6. ASPECTS FOR PARALLELISATION

This section shows an application of the `loop()` pointcut, namely parallelisation of loops.

The example advice shown in Figure 10 executes in parallel (using cyclic loop scheduling) all the loops contained in class `LoopsAJTest` which are recognised as iterating over a range of integers. As shown, the `loop()` pointcut combines ideally with the "*worker object creation pattern*" [7], which creates new `Runnable`s to execute join points on separate threads.

```
void around(int min, int max, int step):
  within(LoopsAJTest)
  && loop() && args (min, max, step, ..) {
     int numThreads = 4 ;
     Thread[] threads = new Thread[numThreads] ;
     for (int i = 0 ; i<numThreads ; i++) {
        final int t_min = min+i ;
        final int t_max = max ;
        final int t_step = numThreads*step ;
        Runnable r = new Runnable () {
           public void run() {
             proceed(t_min, t_max, t_step) ;
           }
        } ;
        threads[i] = new Thread(r) ;
     }
     for (int i = 1 ; i<numThreads ; i++) {
        threads[i].start() ;
     }
     threads[0].run() ;
     try {
       for (int i = 1 ; i<numThreads ; i++) {
         threads[i].join() ;
       }
     } catch (InterruptedException e) {  }
  }
```

Figure 10: Loop parallelisation using Java Threads.

The aspect shown in Figure 11 is slightly more complex. It executes in parallel, using MPI for Java,[7] the loops working on a array of `byte`s that are in method `LoopsAJTest.test`. The original array, `a`, is exposed to the pointcut. It is then sliced into an array `p` per MPI task. Then `proceed` uses array `p` instead of `a`, so the loop in each MPI task only iterates over its local portion of `a`.

When using these kinds of aspects, the programmer needs to make sure that the loops that are going to be executed in parallel can actually be parallelised. As explained in Section 5.3, no inter-dependency analysis is currently performed.

7. ISSUES RELATED TO EXCEPTIONS

This model and the way the loops are recognised do not work properly if exceptions are used in the methods advised. Firstly, exceptions handlers are activated according to position between two bytecode instructions. Weaving may insert code within the range of an exception handler when this may not be intended. Secondly, combined loops correspond approximately to loops written in the source-code, as long as the graph is reducible (or well-structured). This is the case for bytecode produced by Java source-code when the graph does not contain edges due to the potential handling of exceptions. However, taking the exceptions into account adds

[7]`http://www.hpjava.org/mpiJava.html`

```
import mpi.* ;

aspect MPIParallel {
  int rank ;
  int nthreads ;

  void around(String[] arg):
      execution(void LoopsAJTest.main(..))
      && args(arg) {
    try {
      MPI.Init(arg);
      rank = MPI.COMM_WORLD.Rank();
      nthreads = MPI.COMM_WORLD.Size();

      proceed(arg) ;

      MPI.Finalize();
    } catch (MPIException e) {
      e.printStackTrace() ;
    }
  }

  void around(int min, int max, int stride, byte[] a):
        loop() && args(min, max, stride, a, ..) &&
        withincode(* LoopsAJTest.test(..)) {
    try {
      MPI.COMM_WORLD.Barrier();
      int slice_length = a.length / nthreads ;
      byte[] p = new byte[slice_length] ;
      if (rank == 0) {
        for (int i = 0 ; i < slice_length ; i++) {
           p[i] = a[i] ;
        }
        for (int k = 1; k < nthreads; k++) {
           MPI.COMM_WORLD.Ssend(a, k*slice_length,
                        slice_length, MPI.BYTE, k, k) ;
        }
      } else {
        MPI.COMM_WORLD.Recv(p, 0, slice_length,
                    MPI.BYTE, 0, rank) ;
      }
      proceed(0, slice_length, 1, p) ;
      MPI.COMM_WORLD.Barrier();
    } catch (MPIException e) {
       e.printStackTrace() ;
    }
  }
}
```

Figure 11: Loop parallelisation using mpiJava.

```
public int foo (int i, int j) {
   while (true) {
      try {
         while (i < j)
           i = j++/i ;
      } catch (RuntimeException re) {
         i = 10 ;
         continue ;
      }
      break ;
   }
   return j ;
}
```

Figure 12: Example of loops involving exceptions.

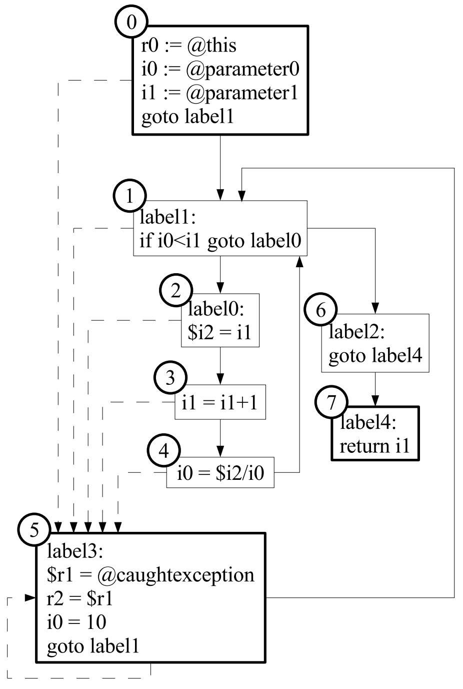


Figure 13: Complete block-level control flow graph.

extra edges to the graph, which may make the graph non-reducible. The main characteristics of non-reducible graphs are that: (a) loops may have several headers; and (b) there are still cycles in the graph after all the back edges have been removed.

To illustrate this problem, Figure 12 shows an example of code that involves loops and exceptions (taken from [9]). Figure 13 shows the corresponding complete block-level control-flow graph (including exceptions, shown as dashed lines) using the Jimple intermediate representation for this example, as produced by the control-flow graph viewer included in the Soot framework. The edges due to traps are dashed only in the illustration; in the system they are treated as regular edges. Without entering into the details of the syntax of Jimple, in this example, `i0` and `i1` represent `i` and `j`, respectively, in the Java source-code.

The back edges found using the method described in Section 2.2 are $4 \rightarrow 1$ and $5 \rightarrow 5$. The graph is not reducible because, after these back edges have been removed, a cycle made of nodes 1 and 5 exists. This gives a loop comprising nodes 1, 2, 3 and 4 —which corresponds to "`while (i<j) i=j++/i;`" in the source-code— and another loop comprising node 5 (which handles the exception in the source-code) only. Although the first loop is meaningful, and corresponds to what would be naturally expected by looking at the source-code, the second would cause *before-advice* to be inserted just before the exception is caught, and *after-advice* just before "`continue`" (without even dealing with the correctness of trap handling). This effect would not necessarily be meaningful or useful for advising this loop.

Moreover, such code is not robust to changes of compilation strategy. For example, a different compiler might insert an extra, "useless" `goto` statement between nodes 0 and 1 in this graph, yielding the control-flow graph shown in Figure 14. In this case there is a third back edge ($5 \rightarrow 8$), which gives a natural loop that could be assimilated into the outer "`while(true) { ... }`" loop in the source-code. The method used so far is not suitable for such cases involving exceptions, since the loop model should depend as little as possible on the compilation strategy utilised.

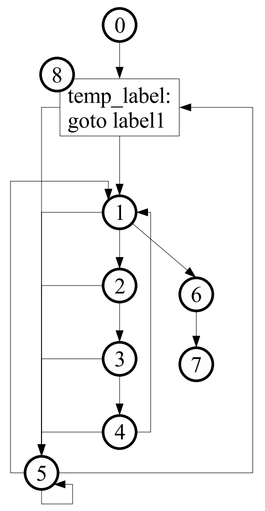


Figure 14: Another possible control-flow graph.

The problem with exceptions lies in the edges they add to the graph. In particular, the edges between the predecessors of the first node that could throw an exception and the node catching the exceptions distort the dominator tree when trying to find the back edges. Because these edges do not actually come from the predecessors, but from a point just before the nodes that could throw an exception, a possible solution would be to change this representation and to introduce separate nodes for throwing exceptions. For each node A that could potentially throw an exception represented as an edge from A to B, a new node E_A would be inserted before A, so that all the edges pointing to A would be redirected to E_A, and an extra edge $E_A \rightarrow B$ would be added. The resulting control-flow graph for the example in Figure 12 is sketched in Figure 15. This is similar to the graph in Figure 13, but contains extra nodes E_1, E_2, E_3 and E_4, which preceed nodes 1, 2, 3 and 4, respectively, and represent the cases where an exception would be thrown in one of these nodes, thus preventing the operations in that node from being performed. This representation now gives two back edges ($4 \rightarrow E_1$ and $5 \rightarrow E_1$) corresponding to a single combined loop. To avoid ambiguity, chains of unconditional `goto`s should be considered as a single node if they can all throw exceptions to the same catching blocks. This approach has not yet been implemented in our prototype.

8. RELATED TOPICS

This section explores two related potential fine-grained join points (i.e. join points that recognise complex behaviour within a method and not only at the interface of the object), namely a loop-body join point (Section 8.1), and an "if-then-else" join point (Section 8.2).

8.1 "Loop-body" join point

The model of loop join point presented thus far takes an outside view of the loop; the points *before* and *after* the loop are not within the loop itself. As a consequence, however many iterations there may be for a given loop, *before* and *after*-advice will be executed only once. For some applications, for example for inserting a piece of advice before each iteration, it might be desirable to advise the loop body. However, the semantics would be difficult to define.

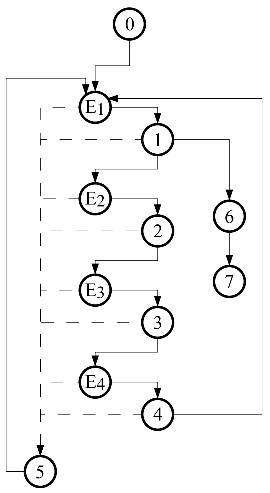


Figure 15: Control-flow graph with special nodes for exceptions.

Even in the source-code, there is ambiguity about where to weave *before* and *after* advice in such a case. For example, is the termination condition in the loop-body or not? (see Figure 16). This question is even more relevant for complex conditions that may include calls to methods.

```
int i = 0;
while (i<2) {
   /* Is ``before'' the loop-body right here, or
      should it be before (i<2) is evaluated?     */
   System.out.println("i: "+i) ;
   i++ ;
   /* Is ``after'' the loop-body here? Would ``i++''
      be included in the loop-body of the equivalent
      for-loop?       */
}

i = 0 ;
do {
   /* Before the loop-body */
   System.out.println("i: "+i) ;
   i++ ;
   /* Is ``after'' here, or should it be after (i<2)
      has been evaluated?    */
} while (i<2) ;
```

Figure 16: Loop-body join point: where are "before" and "after"?

Again, a basic-block control-flow approach may solve the problem. It may be possible to define that "before" the loop-body is the point at the begining of the header, included in the loop, and that "after" the loop-body is a point inserted on the back edge of the natural loop. If there were several back edges in the corresponding combined loop, an equivalent of the "pre-header" could be inserted between the back edges and the header, in order to keep a single weaving point. In the case of a `while`-loop or a `for`-loop, "before" the loop-body would also be before the evaluation of the condition.

Without any enhancement, such a model would not comprise any contextual information (or "arguments" to the loop-body).

8.2 “If-then-else” join point

Why stop at loops? Similar techniques could be applied so as to provide aspect-oriented languages such as AspectJ with a model for an “if-then-else” join point.

At source-code level, there is again the question of whether the evaluation of the condition should or should not be included in the “if-then-else” join point.

A basic-block control-flow approach may help to define a model. A possible way to find the shadows of “if-then-else” constructs might be in the combined use of dominators and *postdominators*. “*[We] say that node* p postdominates *node* i *[...] if every possible execution path from* i *to [the exit] includes* p” [10]. Given a node a that branches conditionally to other nodes (unconditional branching presents no interest), the smallest subgraph G of the control-flow graph that contains another node b such that a dominates all the nodes in G and b postdominates a, would represent an area of conditional execution, starting from a and joining back at b. Since a would dominate all the nodes in G, it would be the unique entry node to G. Since b would postdominate a, b would be the unique exit node from G. Just before the conditional jump in a would be the *before* weaving point, and just before b (for edges coming from inside G) would be the *after* weaving point.

Again, it is unclear what kind of contextual information could be included in such a model. Without it, such an “if-then-else” join point would represent areas of code that will only be partially executed (for a given (dynamic) join point).

However, going a step further by making it possible to advise `goto` statements directly in the bytecode, may break modularity and consistency, even within a method, which would counteract the benefits of using aspects.

Apart from the usual debugging and tracing applications of such join points, another successful approach for defining fine-grained join points (including conditional `if` blocks) has been applied to code-coverage analysis [11].

9. CONCLUSIONS

The paper demonstrates that it is possible to provide AspectJ (and perhaps other aspect-oriented systems) with a loop join point, which can be applied, in particular, to loop parallelisation.

The two main remaining difficulties are: (a) the cleverness of the analysis for context exposure; and (b) the mechanisms for loop selection. The context analysis is mostly implementation dependant. But the loop selection problem is more fundamental, especially because loops cannot be named or tagged. Loop selection based on contextual data can work, but is also limited with the current AspectJ join points. A possible way forward would be to use dataflow pointcuts, as presented in [8]. An extension of this pointcut that would predict the dataflow [8] would perhaps make it possible to determine at compile time which loops should be advised by a parallelising aspect, therefore reducing the overhead of run-time `cflow` (or `dflow`) checks.

[8]A `pdflow` pointcut could be imagined in a similar way as the `pcflow` pointcut mentioned by G. Kiczales in his keynote talk at AOSD’2003.

10. ACKNOWLEDGMENTS

B. Harbulot acknowledges the support of the Engineering and Physical Sciences Research Council (EPSRC) during his doctoral studies.

11. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[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 (to appear).* ACM Press, 2005.

[3] B. Harbulot and J. R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 122–131. ACM Press, 2004.

[4] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP ’01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[6] S. Kuzins. Efficient implementation of around-advice for the AspectBench Compiler. Master’s thesis, Oxford University, UK, September 2004.

[7] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning, 2003.

[8] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. *Lecture Notes in Computer Science 2895, Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS’03).*, pages 105–121, 2003.

[9] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proceedings of CC’02*, 2002.

[10] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[11] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *Proceedings of the 4th international conference on Aspect-oriented software development (to appear).* ACM Press, 2005.

[12] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, Sable Group, McGill University, Montreal, Canada, July 1998.

How to Compile Aspects with Real-Time Java

Pengcheng Wu

College of Computer & Information Science
Northeastern University
Boston, Massachusetts 02115 USA
wupc@ccs.neu.edu

ABSTRACT

The Real-Time Specification for Java(RTSJ) uses a special memory model based on scoped memory areas to address the unpredictability of Java's Garbage Collection mechanism, which makes Java unsuitable to write real-time applications. It has been widely believed that Aspect-oriented Programming (AOP) is helpful for implementing distributed computing applications where a lot of crosscutting concerns exist, including real-time concerns. While it is tempting to use AspectJ with RTSJ programs, both of the AspectJ compilers cannot handle complication in the RTSJ setting correctly, since they didn't take into account the special memory model of the RTSJ. This paper reports our exploration in this area and proposes a compilation approach that takes into account of the memory model of the RTSJ.

1. INTRODUCTION

Although Java [1] has been successfully used for developing complex enterprise-level softwares, it has hardly been used for developing real time applications. Real time systems have stringent time constraints that Java is unsuitable to implement. One of the major obstacles is Java's Garbage Collection (GC) mechanism. While the GC provides important software engineering benefits by freeing developers from error-prone manual memory deallocation tasks, its unpredictable object allocation performance (due to the unpredictable behaviors of the garbage collector) makes it impossible to write real time programs.

The Real-Time Specification for Java (RTSJ) [9] was proposed and released to address this problem and it promises to make Java suitable to construct large scale real-time systems. One official reference implementation of the RTSJ has been provided by TimeSys Corp. [10], and several other open source implementations [8, 7] are being developed as well.

One of the most significant features offered by the RTSJ is a new memory management model based on scoped memory areas. A real-time thread can enter a scoped memory area. When it does so, all subsequent object allocation requests (using the **new** operator) until the thread exits from the scoped memory will allocate objects in the scoped memory area, which is not interfered by the GC, and the whole memory area will be freed once all real time threads have exited from it. Thus the time needed to allocate an object in a scoped memory is predictable. Scoped memory areas can be nested. To keep the safety of Java programming, some important object reference rules are set and enforced by RTSJ-compliant JVMs. For example, objects allocated in outer memory scopes must *not* refer to objects allocated in inner scopes to prevent dangling references.

On the other hand, Aspect-oriented Programming (AOP) [3] was proposed as a new programming paradigm to modularize crosscutting concerns and AspectJ [2, 6], as an AOP extension to Java, is the most widely used AOP language. It is widely believed that distributed system applications have many crosscutting concerns and thus are ideal working platforms for AOP techniques. One of the common concerns in distributed system applications is to implement real time requirements. So it would be tempting to use AspectJ on RTSJ systems to see how it could improve implementations of distributed applications.

However, current compilation approaches used by two major AspectJ compilers (one is Eclipse AspectJ team's AspectJ compiler [6] and the other one is the AspectBench Compiler for AspectJ or abc [5].) fail to work in the RTSJ settings, because neither of them took into account the RTSJ's special memory management schema.

This position paper reports our recent experience of using AspectJ in RTSJ programs and our exploration why the current compilation approaches fail to work with them. Based on the findings, we propose the correct compilation strategy with the RTSJ's memory management schema taken into account.

The rest of the paper is organized as follows: Section 2 provides an overview of the RTSJ's memory management model and show how the current AspectJ's compilation approaches fail in this setting; Section 3 proposes a new compilation approach to address those problems; Section 4 discusses future work.

2. ASPECTJ COMPILATION AND RTSJ MEMORY MODEL

2.1 RTSJ's Memory Model

To address the unpredicatability of Java's GC mechanism, the RTSJ extends the Java memory model by providing memory areas other than the heap. Memory areas are divided into three categories, i.e., `ImmortalMemory`, `HeapMemory` and `ScopedMemory` as shown in Figure 1.

`ImmortalMemory` is a singleton memory area and objects allocated in it have the same lifetime of the JVM, i.e., they are never reclaimed and the GC will never interfere with them. Objects allocated in `HeapMemory` are just like regular Java objects that are subject to GC's reclaim.

A `ScopedMemory` area provides guarantees on object allocation time. A real-time thread can enter a scoped memory and when it does so, all subsequent object allocation requests using the **new** operator until the thread exits from it will allocate objects in the scoped memory area. Objects allocated in a scoped memory area are not reclaimed by the GC, instead, the whole memory area will be freed once all real time threads have exited from it. Thus the time needed to allocate an object in a scoped memory is predictable. `LTMemory` and `VTMemory` provide linear time and a variable amount of time allocation respectively.

Scoped memory areas can be nested. Each real-time thread is associated with a scope stack that defines its allocation context and the history of the scoped memory areas it has entered [4]. The RTSJ also provides APIs for programmers to explicitly specify in which memory area an object should be allocated (not just in the most recently entered scoped area). Due to the special characteristics of the RTSJ's memory model, to keep the safety of Java programming, some important object reference rules are set and enforced by RTSJ-compliant JVMs. One of the most important rules is that objects allocated in outer memory scopes must *not* refer to objects allocated in inner scopes to avoid dangling references. At run time, if such kind of references are ever detected by RTSJ-compliant JVMs, an `IllegalAssignmentException` will be thrown and the whole execution will be stopped. It is programmer's responsibility to make sure that object references obey those rules.

To give readers some intuitions, Listing 1 is a short RTSJ program for an aircraft detection system as presented in [11] . Class `App` implements a real time thread. It creates a scoped memory (line 9) and runs a task (implemented by class `Runner`) in the context of that memory area. The `Runner` then creates another memory area (line 16) and allocates a `Detector` object in the area referred by `mem`(line 17). Then the thread enters a loop in which the detector continuously receives position frames from aircrafts and stores those frames so that it can determine, for example, whether two aircrafts are too close each other. Note that the `run` method (line 24) of class `Detector` is called in the dynamic extent of the execution of method `cdmem.enter(...)`(line 19) and thus all of the **new** requests associated with that `run` method will allocate objects in the memory area referred by `cdmem`.

Listing 1: A RTSJ program

```
   class App extends RealtimeThread {
 2   public static void main(String[] args) throws Exception {
       MemoryArea mem = ImmortalMemory.instance();
 4     App app = (App) mem.newInstance(App.class);
       app.start();
 6   }

 8   public void run() {
       ScopedMemory mem = new LTMemory( ... );
10     mem.enter(new Runner());
     }
12 }

14 class Runner implements Runnable {
      public void run() {
16     LTMemory cdmem = new LTMemory( ... );
       Detector cd = new Detector( ... );
18     while(true)
         cdmem.enter(cd);
20    }
   }
22
   class Detector implements Runnable {
24    public void run() {
       Frame frame = receiveFrame();
26     //get a frame and stores it into a table
       ...
28    }
   }
```

2.2 AspectJ's Compilation Approaches Break RTSJ Memory Model

We want to deploy aspects to RTSJ programs to improve implementations of crosscutting concerns. However, the AspectJ 's compilation approaches (both the official AspectJ compiler and the AspectBench Compiler for AspectJ) do not take into account the RTSJ's special memory model and object reference rules, so the compiled code fail to run on RTSJ compliant JVMs. The following subsections present the cases where the compiled code may fail (they may not be all the cases, instead, they are just the cases we have explored).

2.2.1 Instance-based Aspect Instantiation

In the AspectJ language, aspect instantiation is always implicit. Although programmers can specify how aspect should be instantiated, they have no control when the instantiation should happen, neither can they explicitly instantiate aspects. When a programmer defines an aspect, she can specify how the aspect should be instantiated by using **perthis** , **pertarget**, **percflow** keywords or just without specifying anything, which indicates there will be only a singleton instance of the aspect during the program execution. We call **perthis** and **pertarget** *instance-based aspect instantiation*, because for each **this** (and **target** respectively) object of the corresponding join point (as specified by a pointcut designator(PCD)), there is a separate instance of that aspect associated with the object and the advice will be executed on the particular aspect instance corresponding to the **this** (or **target**) object of a join point. Interested readers are referred to the AspectJ language manual [6] for the details.

For an instance-based aspect, the AspectJ compilers gener-

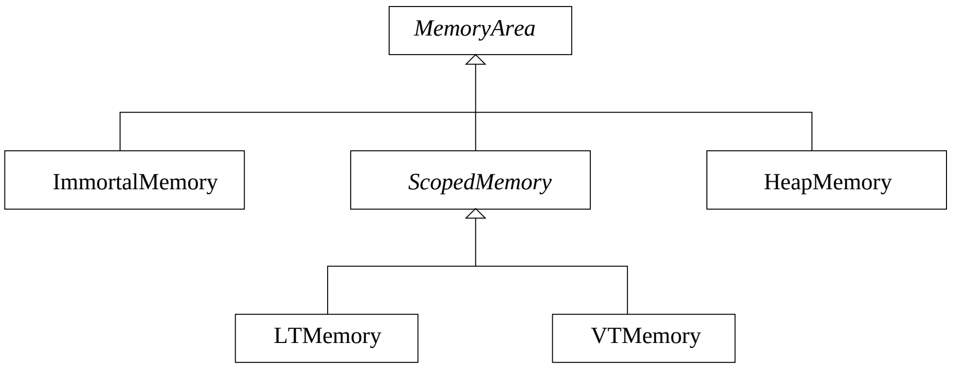


Figure 1: The RTSJ Memory Areas

ate code such that each of those object instances will maintain a reference to its corresponding aspect instance and thus the aspect instance looking up has little runtime overhead. And the aspect instantiation is a "lazy" procedure in that the instantiation (and the reference assignment) only occur when an join point matched with the PCD actually is reached at runtime, instead of whenever an object of such types is created. This approach avoids unnecessary aspect instantiations if no join point matched with the PCD is reached at runtime.

However, in the settings of the RTSJ, the scoped memory area in which an object instance is created is not necessarily the same memory area in which the first join point matched with the PCD occurs. And in such a scenario, the aforementioned object reference rule of the RTSJ may be violated. For example, we want to deploy an aspect as defined in Listing 2 to the RTSJ base program as defined in Listing 1 so that the detector won't do busy polling about the positions of the aircrafts. Instead, it asks those positions periodicly, say every 2 seconds. Aspect `PeriodicalPoll` has to be declared as **perthis**, since there may be many `Detector` objects in the system and each detector has to maintain its own state about when the last polling was.

As expected, an `IllegalAssignmentException` is thrown at runtime (running on the RTSJ official JVM [10] and the code generated by both of the AspectJ compilers show the same behavior). The reason is that the `Detector` instance in this example is allocated (line 17 of Listing 1) in the scoped memory area referred by `mem`, while the `run` method on this detector is first executed in the context of the scoped memory area referred by `cdmem`, where the `PeriodicalPoll` instance corresponding to the `Detector` instance is created and associated with it, and memory area `mem` is an outer scope of memory area `cdmem`. Thus the object reference rule has been violated.

While it is reasonable for a programmer to obey the object reference rules in her own code (e.g, code to implement classes or advice), she has *no* way to fix the problem reported here, since aspect instantiation is implicit and beyond the control of her. It is not just a bug, instead, it is a systematic issue, since similar problems occur on other aspect constructs as well, as presented later. A different compilation approach must be proposed to take into account the RTSJ's memory model.

Listing 2: An aspect applied on the program

```
//Make a detector do periodic polling, instead of busy
//polling.
aspect PeriodicPoll perthis(p()) {
  Time lastTimePolled;
  pointcut p(): execution(* Detector.run(..));

  around(): p() {
    if(it has not yet been 2 seconds since last polling)
      getCurrentThread().yield(); //don't do polling
    else {
      //update the time
      lastTimePolled = System.getCurrentTime();
      proceed(); //do polling
    }
  }
}
```

2.2.2 *CFLOW-based Aspect Instantiation*

If we change the aspect declaration in Listing 2 to be a **percflow** aspect, the same exception will be thrown when the program is run on the RTSJ JVM. When an aspect is declared as **percflow**, each time the execution enters the dynamic extent of a join point matched with the PCD on which the **percflow** is defined, there is an instance of the aspect created and associated with that extent and the aspect program always operates on the innermost aspect instance. Again, interested readers are referred to the AspectJ language manual [6] for the details.

For a **percflow** aspect, the AspectJ compilers will generate code such that there is a global stack to simulate the execution's entering and leaving the extents of join points, and to store the aspect instances in the stack. The global stack is created in the class loading time, and thus is allocated in the heap memory, while the **percflow** aspect instances may be allocated in scoped memory areas, depending on the current thread's memory context. So the object reference rule of RTSJ may be violated.

2.2.3 *CFLOW Pointcut with Bindings*

When an aspect has a **cflow** pointcut with bindings, the program may also throw out an `IllegalAssignmentException` when running on the RTSJ JVM. The AspectJ compilers generate code using a similar stack based approach as for **percflow** aspects. And the reason for the exception is also similar.

2.2.4 *Singleton Aspect and Reflective Access to thisJoinPoint*

Singleton aspect instantiation and the reflective access to **thisJoinPoint** are other two cases where the code generated by the AspectJ compilers will create instances that have interactions with the base program. By analyzing the compilation approaches, we expect those two cases won't violate the rules of the RTSJ memory model. And this expectation has been consistent with our experiences of running AspectJ programs on the RTSJ JVM.

3. PROPOSED COMPILATION APPROACH

We propose a different compilation approach with the RTSJ's special memory model and object reference rule taken into account. We do a case by case explanation.

3.1 Instance-based Aspect Instantiation

There are several options to address the instance-based aspect instantiation problem. Let's list and discuss them here.

- Always allocate instance-based aspect instances in the heap memory. The `IllegalAssignmentException` problem will go away with this approach, since objects allocated in scoped memory areas may have references to heap objects. However, this approach is contradictory to one of the original goals of the RTSJ, which is to remove the unpredicatabilities of Java's GC system. Allocate an aspect instance in the heap memory may trigger the GC thread and make the thread be suspended infinitely. Worse, the RTSJ supports a special yet very useful thread kind, `NoHeapRealtimeThread`, which disallows any access to heap objects. So the heap-allocated aspect instance approach cannot work with `NoHeapRealtimeThreads`.
- Allocate instance-based aspect instances in the immortal Memory. The `IllegalAssignmentException` problem will also go away with this approach, since objects allocated in `ImmortalMemory` have the lefetime as the JVM and objects allocated in scoped memory areas may have references to them. But we view `ImmortalMemory` precious resources (because the memory cannot be reclaimed, even when an aspect instance is no longer reachable), so this approach should at least be discouraged so that `ImmortalMemory` can be saved for necessary cases.
- Allocate instance-based aspect instances in the same memory area as the host objects. This approach will make the `IllegalAssignmentException` problem go away, while avoids all of the problems of the previous two approaches. In addition, this approach is feasible, since the RTSJ supports APIs to let the application allocate objects in any accessible memory area.

After analyzing all the possibilities, we propose to use option 3, i.e., allocate instance-based aspect instances in the same memory area as the host objects.

3.2 CFLOW-based Aspect Instantiation

Cflow-based aspect instantiation is more subtle to handle with than the instance-based aspect instantiation, due to the stack structure of a program execution. A simple approach would be to allocate the cflow-based aspect instances in the `ImmortalMemory` (the heap memory is definitely not an option, as discussed before.), but it is not an optimal solution since we want to save the `ImmortalMemory`.

With that in mind, we propose an approach that exploits the tree structure of scoped memory areas and the fact that there is a special communicating `portal` object associated with each scoped memory area. Each of such a `portal` object will maintain a map from threads to stacks, which are similar to the global stack used in the AspectJ compilers. Each of the stacks stores the aspect instances associated with the dynamic extent of the join point occurring in the current scoped memory area in the corresponding thread. When looking up a **percflow** aspect instance, the system will first look it up in the stack (corresponding to the current executing thread) stored in the `portal` object of the current scoped memory area; if it cannot find one, then it will climb up the scoped memory area tree hierarchy and in each of those scoped memories, look it up in the corresponding stack of the `portal` object until it finds one, or it has reached the root where we can determine there is no such an instance.

3.3 CFLOW Pointcut with Bindings

Our proposed approach for this case would be similar to the approach for the previous case. We will exploit the `portal` object and associate a stack to it and make use of the tree structure of the scoped memory areas.

4. FUTURE WORK

We plan to implement the proposed compilation approach in one of the AspectJ compilers and test the compiler on some real RTSJ benchmarks. In addition, based on the semantics of the RTSJ memory model and the semantics of the AspectJ (we need to give a new one to incorporate the instantiation respects and the cflow-related stuff), we are aiming to give a formal proof that under this new compilation approach, it is guaranteed that there will be no object reference violation due to the instantiations introduced by the compiler and the singleton aspect instantiation or the reflective access to **thisJoinPoint** won't violate those rules either.

5. CONCLUSION

This paper addresses the issues of compiling aspects in the settings of the Real-Time Specification for Java. We have identified and analyzed the cases where the current compilation approaches will fail due to the fact that the special memory model of the RTSJ are not taken into account. Based on the analysis, we propose a new compilation approach to address this problem.

6. ACKNOWLEDGEMENTS

We would like to thank Shriram Krishnamurthi, Karl Lieberherr and Mitchell Wand for their helpful discussion. We are grateful to anonymous reviewers for their valuable comments.

7. REFERENCES

[1] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley, 2000. Second edition.

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mike Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, pages 327–353, Budapest, 2001. Springer Verlag.

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer Verlag, 1997.

[4] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2004.

[5] Programming Tools Group at Oxford University and the Sable Research Group at McGill University. The AspectBench Compiler for AspectJ. http://abc.comlab.ox.ac.uk/.

[6] AspectJ Team. AspectJ home page. http://www.eclipse.org/aspectj. Continuously updated.

[7] The jRate Team. The jRate Project. http://jrate.sourceforge.net/index.html.

[8] The Ovm Project Team. The Ovm Project. http://www.ovmj.org/.

[9] The Real-Time for Java Expert Group. The Real-Time Specification for Java. https://rtsj.dev.java.net/.

[10] TimeSys Corp. Reference Implementation for RTSJ. http://www.timesys.com.

[11] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, 2004.

Slicing AspectJ Woven Code

Davide Balzarotti, Antonio Castaldo D'Ursi, Luca Cavallaro
Politecnico di Milano
Dip. di Elettronica e Informazione
Via Ponzio 34/5, I-20133 Milano, Italy

Mattia Monga
Università degli Studi di Milano
Dip. di Informatica e Comunicazione
Via Comelico 39/41, I-20135 Milano, Italy,

ABSTRACT

The AspectJ programming language allows for the expression, in a compact way, of computations that affect several points in a program (join points), even without knowing where these point exactly are. This is claimed to ease the separation of cross-cutting code. However, it is not clear how real the separation is. In fact it might be difficult to figure out the behavior of the whole system. In order to analyze how an aspect affects the system, one has to consider the slices of the system affected by aspectual computations. However, the expressive power of AspectJ constructs makes difficult to implement slicing algorithms that are both precise and produce useful, i.e., small enough, slices. In this paper we describe our approach to slice AspectJ programs, based on the analysis of the woven code.

1. INTRODUCTION

Well organized software systems are partitioned in modular units each addressing a well defined concern. Such parts are developed in relative isolation and then assembled to produce the whole system. A clean and explicit separation of concerns reduces the complexity of the description of the individual problems, thereby increasing the comprehensibility of the complete system [15].

The notion of aspect-oriented programming was introduced by Kiczales et al. in [10]. Their approach was successfully implemented in AspectJ [1] by Xerox PARC. AspectJ aims at managing tangled concerns at the level of Java code. AspectJ allows for definition of first-class entities called `aspects`. These constructs are reminiscent of the Java `class`: it is a code unit with a name and its own data members and methods. In addition, aspects may *introduce* an attribute or a method in existing classes and *advise* that some code is to be executed `before` or `after` a specific event occurs during the execution of the whole program. Aspect definitions are *woven* into the traditional object-oriented (Java) bytecode at compile-time. Nevertheless, events that can trigger the execution of aspect-oriented code are run-time events: method calls, exception handling, and other specific points in the control flow of a program.

The AspectJ way to provide support for encapsulating otherwise cross-cutting concerns is based on:

1. a syntactic extension to Java for isolating aspect-oriented code;
2. a language for identifying *join points* where advice code should be introduced; set of join points are called *pointcuts*
3. a compile time *weaver* responsible to mix aspects with the rest of the code in order to produce the running system.

The constructs provided by AspectJ show up to be very convenient to express cross-cutting concerns. A typical AspectJ advice can be something like "before any call to the division function, check if the divisor is not zero"; in a very economical way it is possible to affect all the divisions[1] in the code, even without knowing where these divisions will occur. However, it is not clear how real the separation is. In fact, even though a "no division by zero" aspect would be a isolated code unit, it might be difficult to figure out the behavior of the whole system: every time the division function is called, one has to consider that also the aspect oriented code is executed. In general, aspects, while coded in a separate unit, do not enable a true modular reasoning[5, 16]. Moreover, it is still not clear how to cope with the difficult problem of aspect interaction (see [7, 9, 3, 12] for some work in progress and discussion). In order to asses the resulting complexity of an aspect oriented program, we tried to apply well known techniques of program comprehension, namely static analysis and program slicing, to AspectJ. In this paper we describe our effort for building a slicer able to identify which part of an AspectJ program is affected by a specific aspect.

The paper is organized as follows: in Section 2 we present the challenge of slicing AspectJ programs, in Section 3 we describe our approach to the problem, in Section 4 we sketch the implementation of our tool, in Section 5 we show a simple example, and finally in Section 6 we draw some conclusions.

2. SLICING AO PROGRAMS

Program slicing is a program analysis technique introduced by Weiser in the first half of the '80s [21]. A backward (or forward) *slice* of a program consists in all the statements that may influence (or may be influenced by) a given set of statements, called the *slicing criterion.*

[1]Unfortunately this example cannot be implemented in AspectJ as far as integer constants are concerned, for built-in operations are not advisable. The case to be considered should use some *number* classes with a division method

We will focus our attention on backward slicing based only on static information, i.e., without making any hypothesis about input data. Slicing techniques were initially proposed for procedural programs, however they have been widely studied and applied also to object oriented programs [13]. In [14] Liang and Harrold approached the slicing of object oriented programs as a graph reachability problem: each method in an object oriented program is represented by a directed graph (*method dependence graph*, MDG) in which every statement is a node and edges represent control and data dependences among them. All MDGs are then merged in a system dependence graph (SDG), a directed graph that represents the whole analyzed program. On this graph, slices can be computed by exploiting the algorithm introduced by Horwitz, Binkley, and Reps [8].

Unfortunately, the techniques developed for object oriented languages cannot be used as they are with aspect oriented languages, due to some specific aspect oriented features present in most modern aspect oriented languages. In fact, AspectJ provides powerful constructs that, while giving great power to programmers, pose a number of problems during static analyses of the code.

A first issue is the use of "inter-type declarations". In fact, aspects can modify a type by introducing a member in a class or even by manipulating the type hierarchy. This might be useful when one wants to adapt an existing class to a given interface. The use of these constructs, though handy in most cases, has the evil effect to force any analysis to be "holistic", because every analysis needs a closed world assumption. If one wants, for instance, to decide if between two classes `A` and `B` an inheritance relationship holds (a critical information needed when examining polymorphic calls), one has to analyze all the aspects, because any of them could declare such a relationship. Similarly, point-cuts may be defined by using wildcards. This flexibility forces the analysis to take into account all the code of the system.

Another issue is the use of dynamic properties in pointcut definitions. For example, Figure 1 shows a pointcut definition that depends on the value of the function `If.getInputFromUser()`.

```
aspect Trace{
    before(): call(void System.exit(int))
        && if(If.getInputFromUser()){
         System.out.println(thisJoinPoint);
    }
}
```

Figure 1: A pointcut definition that uses dynamic properties

Thus, specific slicing techniques for AspectJ programs were proposed. Zhao and Rinard proposed an algorithm for building a system dependence graph specific for AspectJ programs [22]. They consider each advice like a method and associate an MDG to each of them. Two cases are considered for inter-type declarations. If an inter-type declaration introduces a method it is represented using a module dependence graph. If it introduces a field in a class it is considered as an instance variable of both the aspect, that introduced the field, and the class in which it is introduced.
A pointcut is represented with a *join point vertex*. A *weaving arc* connects the point in the Java part of the AspectJ program picked up by the pointcut to the join point vertex. The join point vertex is connected to the module dependence graph entry vertex associated with it.
Eventually, the whole aspect is represented by an *aspect dependence graph*, a directed graph whose entry vertex is connected by an *aspect membership arc* to the join point vertices and the module dependence graphs declared by the represented aspect. The aspect dependence graph represents the parameters, eventually passed or used by advices or inter-type declared methods, with formal in and formal out vertices, just like formal vertices used in Liang and Harrold's system dependence graph.

3. AO SLICING OF WOVEN CODE

Slicing AspectJ programs by considering methods and advice code as first-class entities [4, 22, 18] is conceptually appealing, since it does not depend on the actual implementation of the AspectJ weaver, and, more fundamentally, it enables the use of aspects as first-class entities in the resulting model. However, building a working tool is far from trivial, because it needs to be able to manage several AspectJ syntax details. In particular, the AspectJ pointcut definition language allows programmers to characterize pointcuts on a wide range of abstraction levels:

- Lexical (`withincode`, regular expression on identifiers, etc.)
- Statically known interfaces (**void** `*.func`(**int**), etc.)
- Run time events (`call`, `execution`, `set`, **if**, etc.)

For example, [22] does not take into account wildcards, changes in class hierarchy, and dynamic pointcuts. Also whether it would be possible to manage all these characteristics with ad-hoc (and not easy to implement) solutions, the resulting program should implement a lot of features currently implemented by the AspectJ compiler.

Instead, one can try to analyze the woven program, i.e., plain Java bytecode, by applying existing techniques and map the results on the original structure of the program.

Thus, in order to build as quick as possible a tool for experimenting with AspectJ programs, we adopted a more pragmatic strategy:

1. Compile classes and aspects using the AspectJ compiler.
2. Weave aspects into an executable program.
3. Apply existing slicing algorithms (we built upon the Soot static analysis framework [19]) to the resulting byte-code.
4. Obtain a slice, as a set of byte-code statements.
5. Map the results onto the original aspect oriented source code.

The advantage in adopting such an approach is twofold. First, it is not necessary to translate aspects into classes because this task is done (in a better way) by AspectJ itself. Second, this approach does not neglect any detail related to AspectJ syntax and it does not need any modification in case of changes in some AspectJ functionalities.

Working at the level of Java byte-code could appear not appropriate because any distinction among classes and aspects may seem to be lost. The AspectJ weaver translates aspects in classes, advices in methods, and join points in methods invocation. Thanks to this approach, it is not difficult to map every statement to its original aspect (or class). However, a tool based on byte-code slicing has to be changed when the AspectJ weaver modifies its implementation strategy.

Thus, the strategy we implemented in our tool starts by inspecting the Java byte-code, then the call graph is computed and the "def-use" analysis performed. Eventually, an SDG of the woven Java program is built, and, by exploiting standard algorithms proposed by the program analysis community during the last 20 years, static slices are computed. Finally, slices can be mapped backwards to the AspectJ code, leveraging on the information about aspects that is still encoded in the byte code. In the following section we describe how the tool was implemented and the limitations of the current prototype.

4. SLICER IMPLEMENTATION

4.1 Strategy and limitations

Notwithstanding the deep research work done in the slicing field, only a few products able to do slicing of real object-oriented programs exist: for example, the Bandera tool [6] has a component aimed at slicing Java programs in order to ease model checking of properties of multi-threading programs. Bandera operates at the bytecode level, thus one could imagine its use also in an AspectJ context (remember that AspectJ programs are eventually woven in plain bytecode). However, all our attempts to use it for slicing programs generated by the AspectJ compiler failed, since the code, while publicly available under a GPL license, is hard to understand and evolve. In fact, Bandera's slicing component is targeted to slice synchronization constructs, thus, it should be enhanced to deal with generic slices. Recently, Bandera's research group released the Indus program slicer [2]. We did not test the new tool, yet. However, using the Indus tool required a licence agreement conflicting with our goal of producing an open source tool.

However, both Bandera and Indus are based on a Java program analysis framework called Soot [19][2]. Therefore, we decided to build our slicer directly on the Soot set of libraries. The Soot framework uses an intermediate representations called *Jimple*, that simplifies the analysis of the bytecode. The algorithm used to compute slices is the one proposed by Horwitz, Binkley and Reps [8]. This algorithm works on the SDG of the program that is built by putting together the MDGs of all methods. It is worth noting that the advice code is represented by plain methods in the woven code, thus normal techniques apply. In order to build the SDG, each method is analyzed after the methods it calls. For each analyzed method the following graphs are computed

- the control flow graph (CFG), representing the control flow among statements: a statement a is connected to b if b can be executed immediately after a;
- the control dependence graph (CDG), representing statement dependences from conditional statements;
- the data flow graph (DFG), representing data dependences: a statement a is connected to b if b uses a variable defined by a.

[2]Soot itself is LGPL, thus there is no contradiction in using it in a proprietary tool as Indus is.

The main purpose of our work was to show the feasibility of our slicing approach based on bytecode analysis. We would like to have a tool as quick as possible to start experimenting with AspectJ examples of increasing complexity. In order to build a working tool in a reasonable amount of time, the current prototype has the following limitations:

- variables are not of array types;
- no exception handling mechanism is used;
- there are no inner classes;
- there are no static members;
- there are no recursive calls;
- the program has a single thread of control;
- there are no inter-procedural aliases.

The tool produces correct slices (i.e, slices that contain all the statements that might be part of the minimal slice, that is not computable [20]) for any program that satisfies the above limitations. We intend to remove all these limitations in the forthcoming versions of the tool, but their introduction was useful to build quickly a proof-of-concept prototype. Our tool is available on request under a GPL license agreement.

4.2 Building the graphs

In order to build the CFG and the CDG of each method we relied on the Soot framework. To each CFG we added a *Start* node to represent the method's entry point and a *Stop* node to represent the method's exit point. Then the CDG was built following the approach used by the Bandera tool [6].

Since a method (or advice code) often calls another method (or advice code) their MDGs must be connected at the call sites. An example showing how call sites are connected is shown in Figure 2.

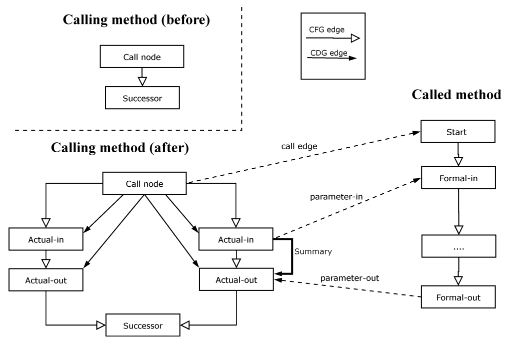


Figure 2: Call site modification for a polymorphic call

Due to polymorphism, the piece of code actually executed could be decided only at run-time, thus we created a new branch of control flow for each possible call target. In each branch, we put actual-in and actual-out nodes depending on

how parameters are used by the called method.
We say that a method *uses* one of its parameters if it reads the parameter value or if it defines its value. We say that a method *defines* one of its parameters if it defines the parameter value. We put an actual-in node for each parameter used by the callee and an actual-out node for each parameter that may be modified by the callee.

We chose to analyze each method after the methods it calls, so we know which parameters are used or defined and we can determine which actual nodes we have to create.
To take into account dependencies among actual and formal parameters, we added some edges to the SDG. We put a parameter-in edge from an actual-in node to the corresponding formal-in node in the called method. We put a parameter-out edge from a formal-out node in the called method to the corresponding actual-out node in the calling method.

Actual-in and actual-out nodes in each branch are control dependent on the method call instruction, so we add control dependence edges from the call node to actual-in and actual-out nodes. To take into account the dependence of called method on its caller, the call node itself is linked to the entry vertex of called method with a method call edge. Since method call edges are interprocedural edges, they are only put in the SDG.
Next we add summary edges from actual-in nodes to actual-out nodes. A summary edge is added from actual-in A to actual-out B if and only if the value of A may affect the value of B. Again, these dependencies have been computed during the analysis of the called method.
The last node of each branch is eventually connected with a control flow edge to the original call node successor in the CFG. Figure 2 shows a modified call in case there is a single parameter and its value is modified by the called method.

4.2.1 Formal Parameters and Return Value Representation

After the creation of actual nodes, we have to build formal-in and formal-out nodes to represent formal parameters of the method under analysis. Since Jimple representation already contains instructions representing the assignment of parameter values to local variables, we use these instructions as formal-in nodes.

To create formal-out nodes, we analyze method instructions one by one, searching for instructions that modify reference-type parameters. Non-reference parameters (primitive types as **int**s) cannot be modified, since their redefinition is not returned to the calling method, so we do not create formal-out nodes for them. For each parameter, we add a formal-out node in the method if and only if there is at least one instruction that can modify the parameter. Formal-out nodes are placed sequentially before the Stop node in the CFG of the called method. To handle multiple return statements, we link each one of them to the first formal-out node (if there are no formal-out nodes, they are linked to the Stop node).

4.3 Dataflow analysis

DFGs are built following a slightly modified version of the algorithm proposed in [17]. This algorithm requires to associate six sets (*use*, *def*, *gen*, *kill*, *in*, and *out*) to each node. The *def* set contains the variables defined in a given node. In our implementation the *gen* set (generated from the *def* one) contains strings in the form 'node-number.variable-defined'. For example, if node 7 defines variable 'foo', we put in node 7 *gen* set the string '7.foo'. When we find a killing definition of 'foo', we put in the killing node *kill* set the string '*.foo'. This means that every other definition of 'foo' has to be killed. We also use the character '*' to express datamember killing. When we find a killing definition of the reference variable 'bar', we put in the killing node kill set the string '*.bar.*', instead of explicitly killing all its data-members. This allows us to represent killing definitions for object data-members without knowing the inner structure of classes.

Computation of reaching definitions needs comprehension of intra-procedural alias information. We don't describe here the algorithms we use to compute intraprocedural aliases, since they are performed by the Soot framework. Since aliases affect variables uses and definitions in method nodes, we have to modify *gen*, *def* and *use* sets for each node.

For each used variable in the node we build a graph to express the use of its 'containers' and its aliases. The 'container' of a class datamember is the object the datamember belongs to. An example of this graph is shown in figure 3.

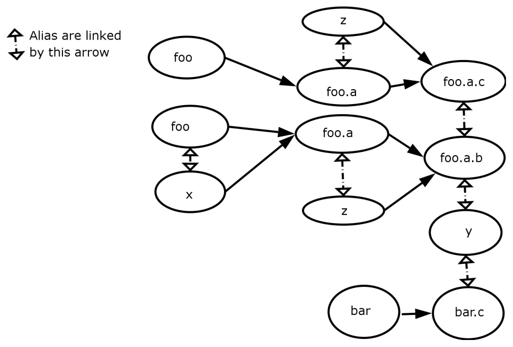


Figure 3: *Alias graph example*

We start building this graph by adding the used variable to it. Next we add to the graph its aliases. Then, for each graph head, we add to the graph its 'container', linking the container with the head. When we add a 'container' to the graph, we also add its aliases and we link them with the head we are examining. We go on until no more containers can be added to the graph.

From this graph we can extract the node *use* set. We examine the graph heads one by one. We follow graph edges until we reach one of the tails. For each node we come across, we take the last part of its name and concatenate it with others. The obtained variables are added to the node use set. Then we remove the examined head from the graph.
Referring to figure 3, we can examine what happens when examining the head named 'x'. First, 'x' is added to the node use set. Then we follow the edge and we come across 'foo.a'. We add 'x.a' to the node use set. Next we come across 'foo.a.b', and we add 'x.a.b' to the node use set. Since we have reached a tail node, we remove the 'x' node from the graph.
If we are examining a variable being defined by the node, we build the graph in the same way, except that we do not add variables alias. However containers' aliases are still added to the graph.

Reaching definitions are then computed using the same algorithm proposed in [17], with some minor modifications needed to make it work with our representation of killing definitions.

The last step in the method analysis consists in calculating dependencies of formal-out nodes and return value node from formal-in nodes. Starting from each formal-out node, we backwards follow summary, control and data dependence edges, looking for formal-in nodes. Any formal-in node found in this intraprocedural slice affects the formal-out node we examined.

4.4 Mapping the Slice to the AspectJ Code

Once the slice of code that affects a given criterion is computed, we have to map it back onto the original AspectJ source code. This is accomplished analyzing source code information found into the bytecode instructions. Since bytecode contains information about original source code lines corresponding to each bytecode instruction, the mapping is performed extracting source code line numbers from bytecode. In this way method and advice statements are easily identified. Instead, pointcut declarations do not normally contain executable statements. However, when they do (as in the example shown in Fig. 1), the mapping is solved correctly.

Currently, we are not able to correctly map inter-type declarations. In fact, AspectJ weaver documentation does not precisely describe the weaving of inter-type declaration (that part is prone to heavy optimization, and it is likely to change in different releases). Inter-type declarations are implemented by direct bytecode manipulation, without preserving any information about their source, therefore by analyzing only bytecode it is impossible to spot them correctly.

5. AN EXAMPLE

Figure 4 contains a piece of code that shows a simple case of aspect interference. Class `T` represents an hypothetical boiler controller. Suppose the programmer wants to add two different aspects. The first one (`LockAspect` in the code) introduces a locking mechanism in order to assure that only one object at a time can modify the boiler status. The second aspect (`TInvariant` in the code) checks that the boiler temperature can never be set to a value greater than 100 Celsius degrees and shuts down the boiler otherwise. Both the aspects work properly if they are independently applied to the program but if they are applied together the invariant aspect can in some case interfere with the locking mechanism leading the system to a deadlock status.

Applying our tool to the weaved bytecode it is possible to construct the SDG and then calculate the slices using the two aspects as slicing criteria. The whole graph contains 236 nodes and around 650 edges. [3]

The slice built using `TInvariant` as slicing criterion does not contain any node that belongs to the `LockAspect` code. That means that the locking mechanism does not interfere with the invariant property. On the contrary, the slide built starting from `LockAspect` contains nodes of the `invariant` code. This is not a proof that the two aspects are incompatible, but it represents a useful information for the programmer since it points out that the `TInvariant` aspect affects the behavior of the locking aspect.

[3]A machine equipped with a 1.6GHz processor, 512MB of RAM, GNU/Linux, a Blackdown Java VM, spent 2.1s to produce the graph and 1ms to compute the slice, including I/O

6. CONCLUSIONS

Our tool, while quite limited in the current preliminary version, shows that AspectJ programs analysis can be actually performed analyzing woven bytecode. Moreover, since every Java program is also a valid AspectJ program, the tool can also be used to analyze plain Java programs, provided that the limitations described in Section 4.1 are satisfied.

Mapping the computed slice onto the source code is currently possible only for statements that are part of methods and advice code. The main open problem is still about the inter-type declarations: currently they are not correctly mapped, because it is not easy to understand whether class files have been modified during the weaving process. However, we can correctly analyze effects of inter-type declarations in the program. Mapping of inter-type declarations would be easily implemented if AspectJ compiler could mark intertype declarations using Java annotations, provided by Java 1.5.

We plan to remove most of the limitations of the tool in the following releases. We will be able soon to analyze arrays introducing in our code the opportune representation. Soot already provides a representation for arrays and we are going to use it. We will also implement a simplyfied exception analysis, that will be able to deal with intraprocedural exceptions, using tools provided by Soot.

The further step will be dealing with direct and mutual recursion along the lines sketched in [17]. Morever, static fields will be analyzed by exploiting the techniques described in [11].

Our final goal is to understand how large is the impact of using aspects on the comprehension of the whole program. In fact, if the slice associated to an aspect would be too big (at worst the whole program), this is a hint that the separation of the aspect code from the base one is only syntactical, since in the worst case no compositional invariant can be taken for granted.

7. REFERENCES

[1] Aspectj. http://www.aspectj.org.

[2] Indus. website:http://indus.projects.cis.ksu.edu/.

[3] Davide Balzarotti and Mattia Monga. Using program slicing to analyze aspect-oriented composition. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, pages 25–29, Lancaster (UK), March 2004. Iowa State University.

[4] Lynne Blair and Mattia Monga. Reasoning on AspectJ programmes. In *Proceedings of Workshop on Aspect-Oriented Software Development*, pages 45–50, Essen, Germany, March 2003. German Informatics Society.

[5] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy,. Technical Report TR03-01a, Iowa State University, January 2003. presented at SPLAT 2003.

[6] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun

```
public class T{
  int temperature;
  public T(){
   this.temperature = 0;
  }

  public void set_temp(int t){
   System.out.println("Setting temperature to "+t);
   this.temperature = t;
  }

  public void shutdown(){
   System.out.println("Shutting down...");
  }
}

public class Main {
  public void method1(T t, int x){
   t.set_temp(x);
  }

  public static void main(String[] argc){
   Main m = new Main();
   T t = new T();
   m.method1(t, Integer.parseInt(argc[0]));
  }
}
```

```
public aspect LockAspect {
  public void T.get_lock(){
   System.out.println("Lock acquired");
  }
  public void T.release_lock(){
   System.out.println("Lock released");
  }
  before(T t): target(t) && (call(void set_temp(int))){
   t.get_lock();
  }
  after(T t): target(t) && (call(void set_temp(int))){
   t.release_lock();
  }
  before(T t): target(t) && (call(void shutdown())){
   t.get_lock();
  }
  after(T t): target(t) && (call(void shutdown())){
   t.release_lock();
  }
}

public aspect TInvariant {
  before(T t, int newval):
   set(int T.temperature) && args(newval) && target(t){
    if (newval > 100) t.shutdown();
   }
}
```

Figure 4: Example of interacting aspects

Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

[7] Remi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse, and interaction analisys of stateful aspects. In *Proceedings of the 3rd international conference of aspect-oriented software development*, Lancaster, UK, March 2004. ACM.

[8] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[9] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Gary T. Leavens, Ralf Lämmel, and Curtis Clifton, editors, *Foundations of Aspect-Oriented Languages*, March 2004.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[11] Gyula Kovács, Ferenc Magyar, and Tibor Gyimóthy. Static slicing of Java programs.

[12] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *Proceedings of SIGSOFT'04/FSE-12*, Newport Beach, CA, USA, November 2004. ACM.

[13] L. Larsen and M.J. Harrold. Slicing object-oriented software. In *In Proceedings of the 18th International Conference on Software Engineering*, pages 45–50. Association for Computer Machinery, March 1996.

[14] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *ICSM*, pages 358–367, 1998.

[15] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[16] Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of SIGSOFT'04/FSE-12*, pages 147–158, Newport Beach, CA, USA, 2004. ACM.

[17] Christoph Steindl. *Slicing for Object-Oriented programming languages*. PhD thesis, Johannes Kepler University Linz, 1999.

[18] Maximilian Stoerzer. Analysis of AspectJ programs. In *Proceedings of 3rd German Workshop on Aspect-Oriented Software Development*, March 2003.

[19] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[20] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, March 1981.

[21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[22] Jianjun Zhao. Slicing aspect-oriented software. In *Proceedings of the 10th IEEE International Workshop on Programming Comprehension*, pages 251–260, June 2002.

Back to the Future: Pointcuts as Predicates over Traces

Karl Klose
Darmstadt University of Technology
D-64283 Darmstadt, Germany
Karl.Klose@gmx.de

Klaus Ostermann
Darmstadt University of Technology
D-64283 Darmstadt, Germany
ostermann@informatik.tu-darmstadt.de

ABSTRACT
Pointcuts in aspect-oriented languages can be seen as predicates over events in the computation of a program. The ability to express temporal relations between such events is a key feature towards more expressive pointcut languages. In this paper, we describe the design and implementation of a pointcut language within which pointcuts are predicates over the complete execution trace of the program. In particular, pointcuts may refer to events that will happen in the future. In this model, advice application is an iterative process that stops once a fixed-point is reached. On the negative side, we do not have a "killer example" for these kinds of pointcuts, there are still some serious limitations, and our implementation strategy is not suitable for a practical language. However, we think that considering pointcuts as predicates over the whole computation and advice application as a fixed point problem is an interesting new perspective on pointcuts for the FOAL audience.

1. INTRODUCTION
In aspect-oriented programming, dynamic join points are points in the execution of a program, and pointcuts are predicates over join points. In the past, aspect-oriented programming has concentrated on pointcuts that are only predicates over the current join point and can thus efficiently be implemented by means of static weaving. However, these pointcuts are not powerful enough to express *relations* between different join points.

AspectJ has one special construct to relate different join points in the form of the **cflow** pointcut designator [7]. More recent works try to make more information about history of the computation available for describing pointcuts [3, 10].

In this paper we want to increase the expressiveness of pointcuts even further and consider pointcuts as predicates over the whole execution trace. We have designed and implemented a prototypical aspect-oriented languages within which the execution trace is reified as a deductive database in Prolog [9] and pointcuts are queries over this database.

Advice application is a fixed point problem in this language. Our implementation computes a solution to the fixed point problem with an iterative process, within which from all advices that are applicable for a given trace, the advice that starts at the smallest point `T` in time is executed by resetting the application to the state at time `T`, and executing the advice and the remainder of the application. This process is iterated until a fixed-point is reached.

This model allows pointcuts to refer to future events in the computation, and also allows sophisticated interactions between advices. However, the power that comes with this model is also not without disadvantages. For example, it is easy to create examples with paradox aspects where the aforementioned iterative process does not have a fixed-point.

The remainder of the paper is structured as follows. Sec. 2 presents our language GAMMA. Sec. 3 presents some example programs. Sec. 4 discusses the problem of semantics and some preliminary results on the applicability of domain theory and static analysis techniques to guarantee that iterative advice-application is well-behaved. Our prototype implementation is presented in Sec. 5. Sec. 6 discusses related work. Sec. 7 concludes.

2. AN OVERVIEW OF THE GAMMA LANGUAGE
Our prototype language GAMMA is an aspect-oriented language on top of a minimal object-oriented core language. This object-oriented core language is based on the teaching calculus L2 from Sophia Drossoupolou [4], which is similar to Featherweight Java [6], but also has an object store and hence supports assignments, aliasing etc. A formal syntax, operational semantics, and type system are described in [4]. Here we will use and describe this core only informally.

GAMMA supports classes and single inheritance. The only primitive type is `bool`. Methods have a return type and always one argument whose name is always `x`. If no argument is required, we add a dummy argument of type `bool`.

GAMMA is expression-oriented, so the body of a method is an expression. All names have to start with a lower-case letter. This makes interoperability with Prolog a little bit easier because then every name in this language can directly be used as a Prolog atom. Minimal I/O is available via a `print` expression that can print strings or objects.

In addition to fields and methods the class `main`[1] can contain aspects, which consist of a keyword indicating the kind (`before` or `after`), a pointcut term (a pattern in the execution trace) and *advice*, code whose execution is triggered by the pointcut. A pointcut is basically a Prolog query (with some restrictions) in which at least one predicate should have the variable `Now` as its first argument. The value of this variable determines at which point in the computation the advice should be executed.
An advice is similar to a normal method but it can use unified variables of the query as expressions. If `this` is used inside an advice body it always refers to the first `main` instance that has been created.

The execution trace is represented as a collection of facts in a Prolog database, where each fact corresponds to one step of the interpreter. The number of the current step, or timestamp, is always stored as the first argument of a fact.

The following tables give an overview of the facts used in traces and their arguments:

Fact	Meaning
get(T, C, A, F, V)	reading field access
set(T, C, A, F, V)	writing field access
calls(T, C, A, M, V)	method call
endCall(T, B, R)	end of method call
newObject(T, C)	object creation

Arguments	Meaning
A	Address of target object
B	Timestamp of begin of call
C	Class of target/to create instance of
F	Name of field
M	Name of method
R	Return value
T	Timestamp
V	Current/new value of field

The pointcut language is closely connected with the Prolog syntax. In fact every pointcut is a Prolog query. Pointcut terms consist of predicates that can be combined by commas (a comma means `and` in Prolog). A predicate can contain variables (which have to start with a capital letter, like `Var`), anonymous variables (_), predicates or atoms (which start with a lower letter, like `main`. To negate a prolog term, the predicate `not`[2] can be used. The special variable `Now` identifies the timestamp where the pointcut matches, i. e. the point where advice should be inserted. Variables that have been used in the pointcut can be used as expressions in the advice. Fig. 1 shows a short program with an aspect which demonstrates how variables used in the query (here `Address`) can easily be accessed from within the advice.

To compare variables used for timestamps, the predicates `pred(T1,T2)` and its transitive closure `isbefore(T1,T2)` are available. The pointcut `not(set(T,_,Addr,Field,_)), get(Now,_,Addr,Field,_), isbefore(T, Now)` for example matches read access to any field of an object that has not been set before. In contrast, `set(Now,_,Addr,Field,_), set(T,_,Addr,Field,_), pred(Now,T)` matches any assignment to a field immediately followed by another assignment to the same field. If timestamp variables are not related, they can match *any* point in the trace.

[1]Due to space limitations and for the sake of clarity we simplified our language: only the class `main` can have aspects and `after` as well as `around` advices have been omitted

[2]In Prolog one should rather use `\+` than `not`, but this syntax is easier to parse.

```
class main extds Object {
 bool var;
 before set(Now,_,Address,_,_) {
  print(Address)
 }
 bool main(bool x){
   this.var := true
 }
}
```

Figure 1: Class with aspect

More complex predicates, like the well-known `cflow`, can be easily formulated as rules:

```
% T2 is in the control flow of the call at T1
cflow(T1, T2) :-
  calls(T1,_,_,_,_),
  endcall(T3,T1,_),
  isbefore(T1,T2),
  isbefore(T2,T3).
```

Similarly the content of the store and the call stack at each time can be reconstructed from the `set` resp. `calls` facts using `isbefore`.

3. APPLICATION EXAMPLES

Consider an environment where a set of graphical objects are potentially manipulated by some `operation`. There is a display, which should only be updated if at least one element has been changed and the number of updates should be minimal. Fig. 2 shows a solution in GAMMA. The pointcut matches at the end of the execution of `main.operation` if a call to `point.setpos` lies in its control flow.

```
before calls(T1, main, _, operation, _),
        cflow(T1, T2),
        calls(T2, point, _, setpos),
        endCall(Now, T1, _) {
    this.display.update(true)
   }
```

Figure 2: A display example

This example shows how the pointcut languages allows us to simply refer to the history of the execution, making the aspect both short and easy to understand. In other pointcut languages, one would have to manually store parts of the history together with complicated imperative logic in order to achieve the same effect.

The pointcut in Fig. 2 only refers to past events. This is different in the next example. Fig. 3 illustrates a kind of *eager authentication*, which performs authentication *before*, but only if, a call to a *protected* **database** function is made inside the control flow of **server.execute**. Such an aspect may appear in a scenario where a complex command has to be executed and it is necessary to be logged in if *any* of the subcommands will require authentication during its execution.

```
before calls(Now,server,_,execute,_),
        cflow(Now,T),
        calls(T,database,_,protected,_) {
   this.db.authenticate(true)
  }
```

Figure 3: An authentication example

By the usage of logical programming and predicates as the base of our pointcut language we gain a great flexibility to express temporal related pointcuts without making the language too complex to use and understand.

4. PARADOX ASPECTS

In the preceding examples it was intuitively clear what the semantics of the programs should be, even in the case that a pointcut refers to the future. But we can easily construct examples where it is hard to say how the semantics of the program should be defined. We will discuss some of these examples and different strategies to solve the problems these examples impose.

As mentioned before, the execution of advice can enable pointcuts at *every* other position in the computation. This can easily produce a phenomenon that is similar to the "grand-mother paradoxon" in time travel: an aspect whose pointcut is enabled by the base program uses its advice to change the control flow in such a way, that the pointcut is not being enabled. Fig. 4 gives an example of such an aspect. The problem is that the trace of this program is not consistent in any of both cases: if the advice is not executed, its aspects pointcut is enabled but if it is executed, its pointcut is not enabled.

```
class main extds Object{
   bool create;
   before calls(Now,_,_,foo,_),
           newObject(T,a),
           isbefore(Now,T) {
     this.create := false
   }
   bool foo(bool x){
     if this.create
       then (new a; true)
       else false
   }
   bool main(bool x){
     this.create := true;
     this.foo()
   }
}
```

Figure 4: A paradox aspect

4.1 Properties of advice application

We can view advice application in our language as a non-deterministic transition system on traces, whereby $t \to t'$ means that the trace t' is the result of applying an advice to the trace t.

An activation point of a trace is a position in the trace, where advice has to be inserted due to an enabled pointcut or inserted advice has to be removed because the corresponding pointcut is not enabled any longer. For convenience we write $AP(t)$ for the *activation points of a trace* t. In general, several pointcuts may be applicable to a trace (i.e., there is more than one activation point), hence the transition system is non-deterministic. Unfortunately, this transistion system does not enjoy the confluence property, meaning that the final result depends on the non-deterministic choice of the next advice to apply. It also does not have a standardization property, informally meaning that there is no "best" choice for the next advice.

4.2 Traces as domains

Domain theory provides a general setting within which recursive equations have a proper solution. When we look at the process of advice application as a function on the set of all traces of a program P, $\mathcal{F} : \mathcal{T}_P \to \mathcal{T}_P$, one way to reason about termination is to apply the fixpoint theorem, a result of domain theory. As mentioned before, there may be several strategies to define such a function, so we will discuss those functions in general. A specific selection strategy is presented in Sec. 5.

If there is a partial order $\sqsubseteq$ that makes the set $(\mathcal{T}_P, \sqsubseteq)$ a **cpo**[3] and if the advice application operator $\mathcal{F}$ is monotonic and Scott-continuous w. r. t. $\sqsubseteq$, then the fixpoint theorem garantees that $\mu(\mathcal{F}) = \bigsqcup_{n\in\mathbb{N}} \mathcal{F}^n(\bot)$ exists with $\mathcal{F}(\mu(\mathcal{F})) = \mu(\mathcal{F})$. This fixed point thus can be constructed by repeatedly applying advice. The bottom element $\bot$ is in our case the trace of the base program without any advice. The challenging part is to define $\mathcal{F}$ and $\sqsubseteq$.

One example to construct such an order is as follows. Let $lap(t)$ (least activation point of t) be the first point where advice has to be inserted or must be removed. Furthermore, let $s = (s_0, \ldots, s_{n-1})$, $t = (t_0, \ldots, t_{m-1})$, $a = lap(s)$ and $b = lap(t)$ then consider the transitive and reflexive closure $\sqsubseteq_P$ of the following relation:

$$s \sqsubset_P t \;\Leftrightarrow\; t = (s_0, \ldots, s_{a-1}, \overbrace{u_0, \ldots, u_{k-1}}^{\text{trace of advice}}, v_0, \ldots, v_{l-1}) \quad (1)$$
$$\wedge\; b > a + k + 1 \quad (2)$$
$$\wedge\; n < a + l \quad (3)$$

In words, a trace t is greater than another trace s, if it can be obtained (possibly in more than one step) from s by simply inserting advice at the earliest possible point (1). Conditions (2)+(3) ensures that no advice will be removed, that each new advice is always executed *after* the last one and that the part of the trace after the advice invocation does not get longer.
It is easy to see that this order makes $\mathcal{T}_P$ a **cpo** because every chain starting with $s = (s_0, \ldots, s_{n-1})$ can have at most n distinct traces. However, this condition is very restrictive and hard to check statically. For example, pointcuts cannot

[3]A **cpo** is a partial ordered set where the supremum of each ω-chain is also contained in the set.

change the control flow after advice application in such a way that the trace gets longer (condition (3)).

Now we need to define an advice application operator $\mathcal{F}$ that is monotonous and continuous. Our general strategy is to select always the earliest activation point in the trace whose advice has not yet been executed. However, we need additional restrictions in order to ensure that $t \sqsubseteq_P \mathcal{F}(t)$.

This can be ensured by a (conservative) static analysis of advice bodies. We present the basic ideas for an algorithm to identify programs that meet the desired restriction.

For each pointcut there is a set of shadows, locations in the source code, where it is *possible* that it could match. We say that an aspect A *precedes* another aspect B if the earliest shadow of A is below or equal to a shadow of B in at least one control path. Furthermore an A *affects* B, if the execution of A can affect the pointcut matching of B, i. e. if a shadow of B lies in that part of the program that can be possibly affected by the execution of A. One condition for the class of acceptable programs could be formulated as: **if** A **precedes** B**, then** B **must not affect** A (and therefore A **must not affect itself**) and the *affect* relation should not have any cycles.

The crucial point is clearly to identify the part of a program that can be affected by an aspect. We developed some basic techniques, but they are rather inaccurate and inefficient, so further work on this topic is required.

5. PROTOTYPE IMPLEMENTATION

As a result of the powerful pointcut language and execution model, some new problems arise in determining the semantics of a program. First, as pointcuts can refer to events in the future, we can not make judgments about advice invocations before the complete trace of the execution is available. Therefore we must run the program at least once to see, where advice has to be executed. Another problem is that the execution of advice itself can effect the execution of other aspects at *any point* in the execution trace, after and *even before* the point at which this advice has been inserted. Our solution to these problems is presented in this section.

Our approach is to iterate advice application, beginning with a trace that does not invoke any advice at all, to (hopefully) get better and better approximations of the final execution trace. If more than one pointcut matches, the first one is chosen to be executed. We model this procedure as an "advice-application-operator" $\mathcal{F}$ which takes a trace t and returns another trace t'.

The interpreter we use to run the program creates and stores a copy of its internal state at every step. The interpreter can thus be reset to any point in the execution of the program. This feature is necessary to restart the execution from the first point where changes in the behavior are expected (due to advice insertion or removal).

To find the points in a trace where advice has to be inserted, the pointcuts of all aspects must be evaluated. The queries are passed to the Prolog engine which returns a set of variable bindings for each positive answer. So we can determine the timestamp of the match by looking at the value of the variable `Now`. These timestamps along with the associated advices and the variable bindings describe the activation points in the trace. The set of APs that has been identified in the current trace is called `foundAPs`. By `oldAPs` we denote the set of APs that has been found in the last trace (this set is empty in the first iteration).

Now there are two things that can happen: a pointcut could match a position it did not match before or a pointcut did match a position in the old trace but does not in the actual one. In the first case, the AP is in the set `foundAPs\oldAPs` otherwise it is in `oldAPs\foundAPs`. The earliest such event (`currentAP`) is that $a \in$ `oldAPs`$\triangle$`foundAPs`[4] with the minimal timestamp (the textual order of the aspects is considered, if two APs have the same timestamp).

After determining `currentAP`, the interpreter (and with it the fact database) is reset to the timestamp of that point and the program is executed from this point in the next iteration. The interpreter does not need to considers other APs than `currentAP` because the execution trace of those that were before stays in the database and APs that lie in the future may become invalid due to the execution the advice. When the advice has been inserted, the timestamp of corresponding AP in `oldAPs` must be updated, because the matching point of the pointcut trace has moved due to the trace produced by the advice.

When the advice has been executed, the `currentAP` must be updated, because the timestamp of the matching position in the trace has moved. A fixed-point of the iteration is reached, if both sets, `oldAPs` and `foundAPs`, are the same.

There are two properties of advice application that follow from the procedure described above:

1. Advice application will only terminate if the base program (without advice) terminates.
2. If advice must be removed at any point in the trace, the iteration ends up in a cycle since the resulting trace has been processed before.

The program in Fig. 5 illustrates how a pointcut can refer to future events: it only matches those assignments to `varx` which are (not necessarily immediately) followed by an assignment to `vary`. The iteration process for this example is shown in Fig. 6. It also shows how the AP is updated after inserting the advice.

The first property of our model, namely that advice application will only terminate if the base program does, is indeed very restrictive. The consequence is that certain program parts cannot be modelled as aspects, in particular aspects that force the program to terminate. e. g. the break condition of an algorithm. For the same reason our model cannot capture infinite computations.
We could overcome these limitations by changing the iteration process to execute advice (and maybe reset the inter-

[4]$A \triangle B$ is the symmetric difference of the sets A and B: $x \in A \triangle B \Rightarrow (x \in A \backslash B$ `or` $x \in B \backslash A)$

```
class main extds Object{
 bool varx;
 bool vary;
 before set(Now,_,_,varx,_),
          set(T,_,_,vary,_),
          isbefore(now,T){
  print("in advice")
 }
 bool main(bool x){
  this.varx := true;
  print("between assignments");
  this.vary := true;
  false
 }
}
```

Figure 5: Example for a "clairvoyant" aspect

```
Run #1 starting at 0
   => newObject(0, main)
   => calls(1, main, iota1, main, false)
   => set(3, main, iota1, varx, true)
between assignments
   => set(4, main, iota1, vary, true)
   => endCall(6, 1, false)
old act.points: ()
found act.points: (3)
new act.points:   (3)

Run #2 starting at 3
   => invokeAdvice(3)
in advice
   => endAdvice(4, 3)
   => set(5, main, iota1, varx, true)
between assignments
   => set(6, main, iota1, vary, true)
   => endCall(8, 1, false)
old act.points:   (5)
found act.points: (5)
new act.points:   (5)
Found fixed point after 2 runs.
Result is false
```

Figure 6: Example for an iteration

preter to a former state) directly at the step where its pointcut matched. This of course means to execute all queries at every step of computation. Since efficiency is not our primary consideration this may be tolerable, but it is not clear how and if this process can be described elegantly, for example in terms of domains as done above.

6. RELATED WORK

Walker and Viggers [8] discuss a kind of *temporal pointcuts*, called history patterns or *tracecuts*, to enrich the AspectJ [1] pointcut language with the abiblity to reason about former calls and their temporal relations. Moreover, data that has been passed as an argument can be accessed by the advice as it could be done via variable binding in our language. *Tracecuts* are patterns that are matched against a *history* of calls by a finite automaton. The implementation translates a program with tracecuts into AspectJ source code. However, the model of history patterns is not as rich as ours since it considers only method calls, whereas our approach may refer to almost any event in the computation.

Douence et al describe a pattern matching language based on Haskell [3] which allows pointcuts to relate different points in the execution history. The Java prototype uses an event monitoring system to accomplish pattern matching. The pointcut language used in this approach describes patterns as sequences of events. This is different to our language where the relation of joinpoints can be stated in a predicative way.

The work of Gybels and Brichau [5] is similar to our approach as they use logic programming and unification for pointcut matching. However, since the model behind their pointcut language does not cover the trace, it is not possible to encode pointcuts that relate different points in the execution. Furthermore the language only offers access to the current joinpoint, so it is not possible to access data from the store. Finally, the approach does not cover the usage of bound variables inside the advice.

Static analysis of aspect interaction is discussed by Douence et al [2], but their focus lies on detecting overlapping shadows of different aspects. They argue that aspects should be *orthogonal*, that means not covering the same join points, independent of the base program they are used with.

7. CONCLUSIONS

We have presented a powerful pointcut language, that makes it easy to write pointcuts that can reason about the execution trace and temporal relations between join points (facts) on a very abstract level. Our approach is so general that even referring to future events is possible. However, our results so far have some serious limitations. We need to find less restrictive ways to ensure termination of the advice application process. An efficient implementation and sophisticated tools for static analysis are also part of future work.

8. REFERENCES

[1] AspectJ homepage, 2003. http://aspectj.org.

[2] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*. Springer-Verlag, 2002.

[3] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *LNCS*. Springer-Verlag, 2001.

[4] S. Drossoupolou. Lecture notes on the L2 calculus. http://www.doc.ic.ac.uk/~scd/Teaching/L1L2.pdf.

[5] 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.

[6] A. Igarashi, B. 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, 1999.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[8] K. V. Robert J. Walker. Communication history patterns: Direct implementations of protocol specifications. Technical report, 2004.

[9] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1994.

[10] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, 2004.

Aspectual Caml: an Aspect-Oriented Functional Language

Hideaki Tatsuzawa
Department of Computer Science, University of Tokyo
hideaki@is.s.u-tokyo.ac.jp

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

Akinori Yonezawa
Department of Computer Science, University of Tokyo
yonezawa@is.s.u-tokyo.ac.jp

ABSTRACT

We propose an aspect-oriented programming (AOP) language called *Aspectual Caml* based on a strongly-typed functional language Objective Caml. Aspectual Caml offers two AOP mechanisms, namely the pointcut and advice mechanism and the type extension mechanism, which gives similar functionality to the inter-type declarations in AspectJ. Those mechanisms are not simple adaptation of the similar mechanisms in existing AOP languages, but re-designed for common programming styles in functional languages such as type inference, polymorphic types, and curried functions. We implemented a prototype compiler of the language and used the language for separating crosscutting concerns in application programs, including a type system separated from a compiler of a simple language.

1. INTRODUCTION

Aspect-Oriented Programming (AOP)[7, 16] is a programming paradigm for modularizing *crosscutting concerns*, which can not be well modularized with existing module mechanisms. Although AOP would be useful to many programming languages with module mechanisms, it has been mainly studied in the contexts of object-oriented programming languages such as Java[4, 5, 14, 15], C++[18], and Smalltalk[6, 12].

In this paper, we propose an AOP language called *Aspectual Caml* based on a functional language Objective Caml. The goal of development of Aspectual Caml is twofold. First, we aim to enable *practical* AOP for development of functional programs. Since there have been developed large and complicated application programs in functional languages[13, 20], such as compilers, theorem provers[3] and software verification tools[2], AOP features should be useful to modularize crosscutting concerns also in functional languages. Second, we aim to provide Aspectual Caml as a basis of further theoretical studies on AOP features. Strongly-typed functional languages, such as ML and Haskell, offer many powerful language features based on solid theoretical foundations. Aspectual Caml, which incorporates existing AOP language features into a strongly-type functional language, would help theoretical examination of the features.

Aspectual Caml is an AOP extension to Objective Caml, a dialect of ML functional language. We design its AOP features by adapting the AOP features in AspectJ, including the pointcut and advice mechanism and the inter-type declaration mechanism, for a functional language with polymorphic types and type inference. We also design the AOP features so that they would fit key properties of strongly-typed functional programming including type safety, type inference, and curried functions.

The language is implemented as a translator to Objective Caml by extending the parser and type checker of the Objective Caml compiler.

The rest of the paper is organized as follows. Section 2 introduces the AOP features of Aspectual Caml. Section 3 presents our current implementation. Section 4 shows case studies of modularization of crosscutting concerns in some application programs with Aspectual Caml. Section 5 presents relevant studies. Section 6 concludes the paper.

2. LANGUAGE DESIGN

This section describes the language design of Aspectual Caml. First, we overview problems in introducing AOP features into functional languages and solutions to those problems. Next, we present an example of extending a small program (which is called a *base program* in this paper) with an aspect. We then discuss the design of the AOP features, namely the pointcut and advice mechanism and the type extension mechanism with emphases on the differences from AspectJ.

2.1 Design Issues

Although the AOP features of Aspectual Caml are similar to the ones in AspectJ, the designing those features was not a trivial task. Unique features in the base language (*i.e.,* Objective Caml), compared from Java, such as the higher-order functions, variant records, and polymorphic types, require reconsideration of most AOP features.

Below, we briefly discuss some of the notable issues in the design of AOP features in Aspectual Caml, and our proposed solutions:

- ML (including Objective Caml) and Haskell programs usually omit types in expressions thanks to the type inference system, whereas types are more explicitly written in Java and AspectJ program. Aspectual Caml has a type inference system for pointcut and advice descriptions.

- Strongly typed languages such as ML and Haskell often have polymorphic types. We found that polymorphic types in pointcuts sometimes break programmers' intuition. This is coped with two types of pointcuts, namely *polymorphic* and *monomorphic* pointcuts.
- Functional programs often use curried functions to receive more than one parameters. If the semantics of `call` pointcut were merely capture one application to functions, it would be inconvenient to identify second or later applications to curried functions. To cope with this problem, Aspectual Caml offers *curried pointcuts*.
- Although AOP features similar to the inter-type declarations in AspectJ would be useful, they should be carefully designed because functional programs usually represent structured data by using variant record types, whereas object-oriented programs do by using classes. In particular, the inter-type declarations in AspectJ relies on the type compatibility of classes with additional instance variables and methods, which is not guaranteed for the variant record types. The type extension mechanism in Aspectual Caml therefore has limited scope to preserve type compatibility.

2.2 Example: Extending Simple Interpreter

In this section, we will show an example of a simple program with an aspect. The base program is an interpreter of a small language, which merely has numbers, variables, additions and let-terms. The aspect adds a new kind of terms—subtractions—into the language. Since Aspectual Caml is an extension to Objective Caml, the interpreter is written in Objective Caml.

2.2.1 Interpreter

The interpreter definition begins with definitions for variables which are of type `id`, an identifier type:

```
type id = I of string
let get_name (I s) = s
```

A term is of variant record type `t`, which can vary over number (`Num`), variable (`Var`), addition (`Add`), or let (`Let`) terms:

```
type t = Num of int
       | Var of id
       | Add of t * t
       | Let of id * t * t
```

There are a few functions for manipulating environments, whose definitions are omitted here:

```
let extend = (* env -> id -> int -> env *)
let lookup = (* id -> env -> int *)
let empty_env = (* env *)
```

The interpreter `eval` is a recursive function that takes an environment and a term and returns its value:

```
let rec eval env t = match t with
| Num(n) -> n
| Var(id) -> lookup id env
| Add(t1, t2) ->
     let e = eval env in (e t1) + (e t2)
| Let(id, t1, t2)  ->
     eval (extend env id (eval env t1)) t2
```

For example, the following expression represents evaluation of `let x=3+4 in x+x`, which yields `14`.

```
eval empty_env (Let(I("x"), Add(Num(3),Num(4)),
                Add(Var(I("x")),Var(I("x")))))
```

```
aspect AddSubtraction
  type+ t = ... | Sub of t * t
  pointcut evaluation env t = call eval env; t
  advice eval_sub = [around evaluation env t]
    match t with
    Sub(t1, t2) -> (eval env t1) - (eval env t2)
  | _ -> proceed t
end
```

Figure 1: An aspect that adds subtraction to interpreter

2.2.2 Adding Subtraction to the Simple Language

The code fragment in Figure 1 shows an aspect definition in Aspectual Caml that extends the interpreter to support subtractions. The first line declares the beginning of an aspect named `AddSubtraction`, which spans until keyword `end`. The body of the aspect consists of an extension to the data structure and a modification to the evaluation behavior.

The second line is *type extension* that adds an additional constructor `Sub` to type `t` so that extended interpreter can handle subtraction terms. Within `AddSubtraction` aspect, the type `t` has a constructor `Sub` as well as other constructors defined in the base program. Section 2.4 explains this mechanism in detail.

The third line defines a pointcut named `evaluation` that specifies any application of an environment and a term to `eval` function. The pointcut also binds variables `env` and `t` to the parameters of `eval`. This is also an example of *curried pointcut* that can specify applications to curried functions. Section 2.3.2.4 will explain this in detail.

Lines 4–7 are an advice declaration named `eval_sub` that evaluates subtraction terms augmented above. The keyword `around` on the right hand side at the fourth line specifies that the body of the advice runs instead of a function application matching the pointcut `evaluation`. The lines 5–7 are the body of the advice, which subtracts values of two sub-terms when the term is a `Sub` constructor. Otherwise, it lets the original `eval` interpret the term by applying the term to a special variable `proceed`, which is bound to a function that represents the rest of the computation at the function application.

Note that the pointcut and the body of the advice have no type descriptions, which is similar to other function defini-

Table 1: Kinds of Join Points in Aspectual Caml and AspectJ

in Aspectual Caml	in AspectJ
function call	method call
function execution	method execution
construction of a variant	constructor call
pattern matching	field get

tions in Objective Caml. The type system infers appropriate types and guarantees type safety of the program.

2.3 Pointcut and Advice Mechanism

Aspectual Caml offers a pointcut and advice mechanism for modularizing crosscutting program behavior. The following three key elements explains the mechanism:

- *join points* are the points in program execution whose behavior can be augmented or altered by advice declarations.
- *pointcuts* are the means of identifying join points, and
- *advice declarations* are the means of effecting join points.

The design is basically similar to those in AspectJ-like AOP languages. We mainly explain the notable differences below.

2.3.1 Join Points

Similar to AspectJ-like languages, Aspectual Caml employs a dynamic join point model, in which join points are the points in program execution, rather than the points in a program text. There are four kinds of join points in Aspectual Caml, which are listed in Table 1 with their AspectJ counterparts.

Note that the correspondences between Aspectual Caml and AspectJ are rather subjective as functional programs and Java-like object-oriented programs often express similar concepts in different ways. For example, functional programs often use variant records to represent compound data while object-oriented programs use objects. Therefore we place the pattern matching (which takes field values out of a variant record) and field get join points in the same row. There are no field-set-like join points in Aspectual Caml since variant records are immutable[1].

A join point holds properties of the execution, such as the name of the function to be applied to and arguments. The names of functions are those directly appear in program text. For example, evaluation of `let lookup = List.assoc in lookup var env` generates a function call join point whose function name is `lookup`, rather than `List.assoc`. We believe that programmers give meaningful names to functions even if the higher-order functions make renaming of functions quite easy in functional programming [2].

[1]Many functional programming languages offer *references* for representing mutable data. The operations over references are also the candidates of join points in future version of Aspectual Caml.

[2]In contrast, models of AOP languages should be tolerant with renaming of variables. MiniAML [21], for example, distinguishes between variable names and signatures that pointcuts match by introducing labels into the calculus.

Table 2: Summary of Primitive Pointcuts

syntax	matching join points
`call` N P_1 `;` `...;` P_n	function call
`exec` N P_1 `;` `...;` P_n	function execution
`new` N`(`P_1`,...,`P_n`)`	construction of a variant
`match` P	pattern matching (before selecting a variant)
`within` N	all join points within a static scope specified N

Note that function call join points include those to anonymous functions such as `(fun x -> x+1) 3`.

2.3.2 Pointcuts

A pointcut is a predicate over join points. It tests join points based on the kinds and properties of join points, and binds values in the join point to variables when matches.

2.3.2.1 Primitive Pointcuts

Similar to AspectJ, Aspectual Caml has a sublanguage to describe pointcuts. Table 2 lists the syntax of primitive pointcuts and kinds of join points selected by respective pointcuts. In the table, N denotes a name pattern and P_i denotes a parameter pattern.

A name pattern N is a string of alphabets, numbers, and wildcards followed by a type expression. It matches any function or constructor whose name matches the former part, and whose type matches the latter part. When `call` or `exec` pointcuts use a wildcard in a name pattern N, they match calls or executions of any function including an anonymous function. The type expression can be omitted for matching functions of any type.

A parameter pattern P is a pattern that used to describe a formal parameter of a function in Objective Caml[3]. It is either a variable name, or a constructor with parameter patterns, followed by a type expression. It matches any value of the specified type, or any value that is constructed with the specified constructor and the field values that match respective the parameter patterns. Again, the type expression can be omitted. For example, "`x:int`" matches any integer. "`Add(Num(x),Var(y))`" matches any `Add` term whose first and second fields are any `Num` and `Var` terms, respectively. Note that parameter patterns with constructors are basically runtime conditions. This is similar to `args`, `this` and `target` pointcuts in AspectJ which can specify runtime types of parameters.

Pointcut `within(`N`)` matches any join point that is created by an expression appearing in a lexical element (e.g., a function definition) matching N. In order to specify function definitions nested in other function definitions, the pattern

[3]In Objective Caml, it is simply called a "pattern", but we refer it to as a "parameter pattern" for distinguishing from the name patterns.

N can use a path expression, which is not explained in the paper.

2.3.2.2 Parameter Binding

The parameter patterns in a primitive pointcut also bind parameters to variables. For example, when string `"abc"` is applied to function `lookup` and there is a pointcut `call lookup name`, the pointcut matches the join point and binds the string `"abc"` to the variable `name` so that the advice body can access to the parameter values. When a pattern has an underscore character ("_") instead of a variable name, it ignores the parameter value.

2.3.2.3 Combining and Reusing Pointcuts

Aspectual Caml offers various means of combining and reusing pointcuts similar to AspectJ. There are the operators for combining pointcuts, namely `and`, `or`, `not`, and `cflow`. It also supports named pointcuts. For example, the line 3 in Figure 1 names a pointcut expression (`call eval env; t`) `evaluation`,

```
pointcut evaluation env t = call eval env; t
```

which can be used in a similar manner to primitive pointcuts in the subsequent pointcut expressions, like `evaluation env t` at line 4 in the same figure.

2.3.2.4 Pointcuts for Curried Functions

The `call` and `exec` pointcuts also support curried functions. For example, `call eval env; t` matches the second partial application to function `eval`. Therefore, when an expression `eval empty_env (Num 0)` is evaluated, the pointcut matches the application of `(Num 0)` to the function returned by the evaluation of `eval empty_env`. The pointcut matches even when the partially applied function is not immediately applied. As a result, when `let e = eval env in (e t1) + (e t2)` is evaluated, the applications of `t1` and `t2` to `e` match the above call pointcut.

The following definition gives more precise meaning to `call` pointcuts:

- `call` N P_1 matches evaluation of an expression `(`e_0 e_1`)` when the expression e_0 matches the name pattern N and the expression e_1 matches the parameter pattern P_1.
- `call` N P_1`; ...;` P_n matches evaluation of an expression `(`e_0 e_1`)` when the evaluated value of e_0 is returned from a join point matching to `call` N P_1`; ...;` $P_{(n-1)}$ and the expression e_1 matches the parameter pattern P_N.

Similarly, `exec` pointcuts support curried functions on the callee's side.

Section 3.4 presents how this advice declarations with a curried pointcut can be implemented.

2.3.2.5 Type Inference for Pointcuts

When types are omitted in a pointcut expression, they are automatically inferred from the advice body in which the pointcut is used. This fits with the programming style in Objective Caml, where types can be omitted as much as possible.

For example, the advice `eval_sub` in Figure 1 has no type expressions in the pointcut `evaluation env t`. However, it is inferred from the expressions in the advice body, that the types of the variables `env` and `t` and the return type of the function are the types `env`, `t` and `int`, respectively. As a result, the pointcut, whose definition is `call eval env; t`, matches applications to a function named `eval` and of type `env` $\to$ `t` $\to$ `int`.

The type inference gives the most general types to the variables in the pointcuts. In the following advice definition, the system gives fresh type variables α and β to variables `env` and `t`, respectively:

```
advice tracing = [around call eval env; t]
  let result = proceed t in print_int result; result
```

As a result, the pointcut matches any applications to functions whose type is more specific than $\alpha \to \beta \to$ `int`. As a result, this advice captures applications to `eval` as well as other `eval` functions that takes two parameters and returns integer values.

2.3.2.6 Polymorphic and Monomorphic Pointcuts

Aspectual Caml provides a mechanism that programmers can make the types in a named pointcut either polymorphic or monomorphic. This is useful when there are more than one advice definition that uses the same named pointcut. When a named pointcut is defined with the keyword `concrete`, it is a *monomorphic pointcut* whose type variables can not be instantiated. Otherwise, it is a *polymorphic pointcut* whose type variables are instantiated when the pointcut is used in an advice definition.

For example, the `evaluation` pointcut in Figure 1 is polymorphic. It matches any function applications `eval` of type $\forall\alpha\beta\gamma.\alpha \to \beta \to \gamma$. When `evaluation` used in advice `eval_sub`, the type system instantiates α, β, and γ and then infers the types with respect to the advice body. Therefore, another advice definition that uses `evaluation` with different types do not conflict with the previous advice definition:

```
advice tracing = [before evaluation env t]
  print_string env; print_string t
end
```

This mechanism is quite similar to the let-polymorphism in ML languages.

Although the polymorphic pointcuts are useful to define generalized pointcuts, they are sometimes inconvenient when the programmer wants to specify the same set of join points at any advice that uses the same pointcut. *Monomorphic pointcuts* are useful in such a situation. Consider the following aspect definition that prints messages at the beginning and end of any function application:

```
aspect Logging
  pointcut logged n = call ??$ n
  advice log_entry = [before logged n]
    print_string ("\nenters with "^(string_of_int n))
  advice log_exit = [after logged n]
    print_string "\nexits"
end
```

Since `logged` is a polymorphic pointcut that matches any application to functions of type $\forall\alpha\beta.\alpha \rightarrow \beta$, the first advice matches only functions that take integer values as their parameter, whereas the second matches any function. This is because the types in the pointcut are inferred at each advice definition.

By declaring `logged` pointcut with the keyword `concrete` and type expression to the variables that are used in the advice:

```
concrete pointcut logged n = call ??$ (n:int)
```

`logged` pointcut becomes monomorphic that matches any application to functions of type `int` $\rightarrow \alpha$. With this pointcut definition, the two advice definitions are guaranteed to advise the same set of join points because the types in the pointcut will not be instantiated further.

2.3.3 Advice

Advice, defined with a pointcut, gives behaviors at, before, or after join points, these timing are decided by timing specifiers `around`, `before`, and `after` respectively, specified by the pointcut. In the body of advice, programmers can use all top-level variables, variables bound by the pointcut, and the special function `proceed` (available only in `around` advice). Since `proceed` means the replaced behavior, it restarts the original execution when it takes an argument.

For preserving type safety, the body expression of `around` advice must have the same type as returning values of specified join points. In addition, that of `before` and `after` advice must have the type `unit`. In the example of subtraction extension, the body of `eval_sub` has the type `int` that is the same type as a result value of `eval`.

2.4 Type Extension Mechanism

The type extension mechanism allows aspects to define extra fields or constructors in variant types in a base program. The former mechanism can be seen as a rough equivalent to the inter-type instance variable declarations in AspectJ.

Despite the simplicity of the mechanism, we believe that it is as crucial as the pointcut and advice mechanism. As you can observe in example programs in AspectJ, not a few crosscutting concerns contain not only behavior (which is implemented by the pointcut advice mechanism) but also data structures (which are implemented by the inter-type declarations).

2.4.1 Defining Extra Constructs

One of the abilities of the type extension mechanism is to define additional constructors to existing variant types. A variant type definition `type+` T `= ... |` C adds constructor C to existing type whose type name is T. In Figure 1, we have already seen an example that adds `Sub` constructor to the type `t`.

The constructors added to a variant type by aspects often make pattern matching non-exhaustive. In other words, a base program that originally defined the variant type usually has functions that process for each variant differently (*e.g.,* `eval` in the simple interpreter). Therefore, an aspect that added a constructor to a variant type would also need to advise such functions so as to process the case for the additional constructor. In the example Figure 1, the advice `eval_sub` processes the constructor `Sub` for the function `eval`, which otherwise reports non-exhaustiveness warnings.

2.4.2 Defining Extra Fields

The type extension mechanism can also allow to define additional fields to constructors of existing variant types. A variant type definition

$$\texttt{type+ } T_0 \texttt{ = } C \texttt{ of ... * } T_1\{e_1\} \texttt{ * } \cdots \texttt{ * } T_n\{e_n\}$$

adds fields of type $T_1, \ldots, T_n$ to a constructor C of type T_0. The expressions $e_1, \ldots, e_n$ in the curly brackets specify default values to the respective fields.

For example, assume we want to associate a number (*e.g.,* a line number in a source program) to each variable in the simple language of Section 2.2. A solution with the type extension mechanism is to add an integer field to `ident` type by writing the following definition:

```
type+ ident = I of ... * int{0}
```

As the base program originally defines `ident` type as `type ident = I of string`, a value created by the constructor `I` has a pair of string and integer.

Extended fields are available only in the aspects that define the extension. This means that the type of the constructor look differently inside and outside of the aspect:

- Inside the same aspect, the constructor has the extended type. Therefore, `I ("x",1)` is a correct expression in the aspect.
- Outside the aspect, the constructor retains the original type, and yields a value that has the values of default expressions in the extended fields. Therefore, `I "x"` is a correct expression outside the aspect, which yield a value that has `"x"` and `0` in its string and integer fields, respectively.

3. IMPLEMENTATION

We implemented a compiler, or a weaver of Aspectual Caml as a translator to Objective Caml. Many parts of the compiler are implemented by modifying internal data structures and functions in an Objective Caml compiler as the AOP features deeply involve with the type system.

The compiler first parses a given program to build a parse tree. Then the next five steps process the parse tree:

1. infers types in the base function definitions;
2. infers types in the aspect definitions;
3. modifies variant type definitions in the base program by processing type extensions;

4. simplify advice definitions; and
5. inserts applications to advice bodies into matching expressions.

Finally, it generates Objective Caml program by unparsing the modified parse tree.

Below, those five steps are explained by using the example in Section 2.2.

3.1 Type Inference for Base Functions

The types in the base function definitions are inferred by using the internal functions in the original Objective Caml compiler. After the type inference, all variables in the functions are annotated with types (or type schemes):

```
type id = I of string
let (get_name:id->string) = fun (I(s:string)) -> s
type t =        (* omitted *)
let extend =    (* ibid. *)
let lookup =    (* ibid. *)
let empty_env = (* ibid. *)
let rec (eval:env->t->int) =
  fun (env:env) -> fun (t:t) -> match t with
| Num(n:int) -> n
| Var(id:id) -> lookup id env
| Add((t1:t), (t2:t)) ->
     let (e:t)->int = eval env in (e t1) + (e t2)
| Let((id:id), (t1:t), (t2:t))  ->
     eval (extend env id (eval env t1)) t2
```

3.2 Type Inference for Aspects

The types in aspect definitions are inferred in a similar manner to the type inference for the base functions. Notable points are the treatments of polymorphic/monomorphic pointcuts, and scope of the variables.

The type of a pointcut is a type of join points that can match the pointcut and a type environment for the variables in the pointcut. The type of matching join points is decided by the shapes of primitive pointcuts in the pointcut and the types of the variables. The variables bound by the pointcuts have unique type variables otherwise explicitly specified. For polymorphic pointcuts, those type variables are quantified with universal quantifiers that can be instantiated at the advice definitions. Monomorphic pointcuts use the special type variables that can not be instantiated in the later processes.

For example, `evaluation` pointcut in Figure 1 has, type of $\forall\alpha\beta\gamma.\alpha \rightarrow \beta \rightarrow \gamma$ for the matching join points, and $[\texttt{env} : \beta, \texttt{t} : \gamma]$ for variables.

Note that the type inference of pointcuts does not use the types of function names; *e.g.,* the type of `eval` in the base program. This is because the function names in pointcuts do not necessarily refer to specific functions in the base program, but they rather refer to any function that have matching name.

The type inference of an advice definition is basically similar to the type inference of a function definition, but it takes types of parameters from the types of the pointcut, and gives a type to `proceed` variable that is implicitly available in the advice body. Given an advice definition `advice` a `= [around call` p`]` e where p is a pointcut of join point type $\alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta$ and variable type ρ, the type of e is inferred under the global type environment extended with ρ and $[\texttt{proceed} : \alpha_n \rightarrow \beta]$.

For example, type inference of `eval_sub` advice uses a global type environment extended with $[\texttt{proceed} : \beta \rightarrow \gamma, \texttt{env} : \alpha, \texttt{t} : \beta]$, and assigns types as follows:

```
advice eval_sub
 = [around evaluation (env:env) (t:t)]
  (* let proceed:t->int *)
    match t with
      Sub((t1:t), (t2:t)) ->
        (eval env t1) - (eval env t2)
    | _ -> proceed t
```

Note that the types of `eval` and `Sub` are taken from the global type environment, which eventually instantiates the types of other variables including those in the pointcut.

3.3 Reflect Type Modifications in Base Programs

In this phase, type extensions are reflected in the base programs. The definition of types are changed according to the aspects. Additionally, the default values are added to expressions whose fields are extended by the aspects.

3.4 Simplify Advice Definitions

The next step is to transform the advice definitions into simpler ones in order to make the later weaving process easier.

First, it transforms every `before` and `after` advice definition into `around` advice, by simply inserting an application to `proceed` at the beginning or end of the advice body.

Second, it transforms an advice declaration that uses curried pointcuts so that all `call` or `exec` pointcuts takes exactly one parameter. The next is a translated advice definition from `eval_sub` (inferred types are omitted for readability):

```
advice eval_sub = [around call eval env]
  let proceed = proceed env in
    fun t -> match t with
        Sub(t1, t2) -> (eval env t1) - (eval env t2)
      | _ -> proceed t
```

When an environment is applied to `eval`, the transformed advice runs and returns a function that runs the body of the original advice when it takes a term. In other words, `eval` is advised to return a function that runs the original advice body.

Generally, it transforms an advice definition with a curried pointcut by iteratively removing the last parameter in the curried pointcut by using the following rule that transforms an advice definition:

`advice` a `= [around call` f v_1`;` $\cdots$`;` v_n`]`
  e

into the next one:

```
advice a = [around call f v1; ···; vn-1]
  let proceed = proceed vn-1 in
    fun vn -> e
```

There is a subtle problem with this approach when curried pointcuts are used with a disjunctive (or) operator, which is left for future research. For example, the following advice causes the problem:

```
advice trace_eval_or_e =
  [around (call eval _; t) || (call e t)]
  print_string "eval"; proceed t
```

When we evaluate `eval empty_env (Add(Num(0), Num(1)))` with this advice, evaluation of each subexpression of `Add` is advised twice if the advice declaration is translated by following the above rule. This is because the advice is translated into the following two advice declarations:

```
advice trace_eval_or_e_1 = [around (call eval env)]
  let proceed = proceed env in
  fun t -> print_string "eval"; proceed t
advice trace_eval_or_e_2 = [around (call e t)]
  print_string "eval"; proceed t
```

Since `eval` has a subexpression `let e = eval env in (e t1) + (e t2)`, the first advice modifies the value of `e` to run the body of the advice, and the second advice runs the body of the advice at `e t1` and `e t2`, respectively. Consequently, evaluation of `e t1` and `e t2` runs the body of advice twice.

3.5 Weave Advice Definitions

The last step is to insert expressions that runs advice bodies at appropriate times in the base functions. It first transforms each advice definition into a function definition. It then walks through all expressions (*i.e., join point shadows*) in the function definitions, and inserts an application to an advice function when it matches the pointcut of the advice.

Given an advice definition, the first step is to simply generate a recursive function that takes `proceed` parameter followed by the parameters to the advice. For example, it generates the following function for `eval_sub` advice (again, types are omitted for readability):

```
let rec eval_sub proceed env =
  let proceed = proceed env in
    fun t -> match t with
        Sub(t1, t2) -> (eval env t1) - (eval env t2)
        | _ -> proceed t
```

The second step is to rewrite the bodies of the base functions[4] so that they call advice functions at appropriate places. By traversing the expressions in the given program, for each expression type of function application, lambda abstraction, constructor application, or pattern matching for structured values, it looks for advice definitions that have the respective kind of primitive pointcuts. When the name pattern of the pointcut matches the name in the expression, and the type of the pointcut is more general than the type of the expression, it replaces the expression with an application to the advice function.

[4]Precisely, the base functions also include the advice bodies. This enables to advise execution of advice as well as execution of function.

For example, `eval` function in the base program has a subexpression `(eval env)` where `eval:env->t->int` and `env:env`. This application sub-expression matches the `call` pointcut in `eval_sub` as the types of the join point and the pointcut are the same. In this case, it replaces the expression with `(`⟨*eval_sub*⟩ `eval env)` where `(`⟨*eval_sub*⟩ is an expression that references the advice function (explained below).

It is a little tricky to define and reference advice functions due to recursiveness introduced by advice. An advice definition has a global scope; it can advise any execution in any module and it also can use global functions defined in any module. Consequently, advice definitions can introduce recursion into non-recursive functions in the original program. For example, the following code fragment recursively computes factorial numbers by advising the non-recursive function `fact`[1]:

```
let fact n = 1
aspect Fact
  advice realize = [around exec fact n]
    if n=0 then proceed n else n*(fact (n-1))
end
```

In order to allow advice to introduce recursion, we proposed two solutions:

- Define advice functions in a *recursive module*[17] in Objective Caml. As recursive modules allow mutual recursion between functions across modules, this would directly solve the problem.
- Reference advice functions via mutable cells. In this solution, the translated program begins with definitions of mutable cells that hold advice functions. The subsequent function definitions run advice functions by dereferencing from those mutable cells. Finally, after defined advice functions, the program stores the advice functions into the mutable cells.

Although the latter solution is trickier, our current implementation uses it since the recursive modules are not available in official Objective Caml implementations as far as the authors know.

After finished above processes, the compiler generates the following translated code for the example program:

```
(* define mutable cells for advice functions *)
let eval_sub_ref = ref (fun _ -> failwith "")
(* definitions for id, t and env are omitted *)
let rec eval env t = match t with
| Num(n) -> n
| Var(id) -> lookup id env
| Add(t1, t2) ->
    let e = !eval_sub_ref eval env in
    (e t1) + (e t2)
| Let(id, t1, t2)  ->
    !eval_sub_ref
     eval
     (extend env id (!eval_sub_ref eval env t1))
     t2
```

```
(* advice function *)
let rec eval_sub proceed env =
  let proceed = proceed env in
  fun t -> match t with
      Sub(t1, t2) ->
        (!eval_sub_ref eval env t1) -
        (!eval_sub_ref eval env t2)
    | _ -> proceed t
(* store advice function into mutable cell *)
let _ = eval_sub_ref := eval_sub
```

Note that all applications to `eval` function, including those in the advice body, are replaced with applications to `!eval_sub_ref eval`. The `eval_sub_ref` is defined at the beginning of the program with a dummy value, and assigned `eval_sub` function at the end of the program.

3.6 Implementation Status

Thus far, we developed a prototype implementation[5] of Aspectual Caml. Although some of the features discussed in the paper are not available yet, it supports essential features for validating our concept, including the type extension, around advice, and most kinds of primitive pointcuts except for wildcarding. In fact, the next section introduces an example that can be compiled by our prototype implementation.

The current implementation has approximately 24000 lines of Objective Caml program (including 2000 lines of our modified and additional parts), including the parser and type inference system that are modified from the ones in the original Objective Caml compiler. Although it would be theoretically possible to directly pass the translated parse tree to the back-end Objective Caml compiler, our compiler generates source-level program by unparsing the parse tree. This is mainly for the ease of development and for debugging.

4. APPLICATION PROGRAMS

Among several small application programs that we have written in Aspectual Caml, we briefly sketch two of them.

The one is, as we have seen thought the paper, to augment an interpreter of a simple language with additional kinds of terms, such as subtraction. Although it is a very small program, the aspect illustrates its usefulness for pluggable extension; since the aspect does not require to change the original interpreter definitions, we can easily fall back to the original language.

The second application program is larger. It extends a compiler of an untyped language to support static type system. The base part of the program define types for the parse trees of the source and intermediate languages and functions that translate the parse tree in the source language into the intermediate language called K-normal forms. The aspects extend the type of the source parse tree with type information, and modifies the transformation functions to carry out type inference during the transformation.

The aspects in the program can improve comprehensibility of the compiler implementation in particular educational purposes. Since the translation rules in the original can be complicated by the types, separating the compiler into the one for untyped language and the extension for types would clarify both the core translation rules and the interaction between translations and type system.

The second program, which consists of approximately 100 lines of base program and 100 lines of aspect definitions, is shown in Appendix A.

5. RELATED WORK

AspectJ[14, 15] is the first AOP language that offers both the pointcut and advice and inter-type declaration mechanisms. Aspectual Caml is principally designed by following those mechanisms. However, we see AspectJ-family of languages might be too complicated to theoretically study the AOP features as they primarily aim practical languages. For example, AspectJ 1.2 compiler type checks the following advice declaration:

```
Object around() : call(Integer *.*(..))
{ return new Float(0); }
```

even though it could cause a runtime error if applied to an expression like `Integer.decode("0").intValue()`. A simpler language that yet has a notion of polymorphism would help to reason about such a situation.

There are several proposals of theoretical models of AOP features. As far as the authors know, most work merely on the pointcut and advice mechanism. Aspect SandBox[22] describes a semantics of an dynamically-typed procedural language with a pointcut and advice mechanism. Tucker and Krishnamurthi presented a pointcut and advice mechanism in dynamically-typed functional languages[19]. MiniAML is a core calculus for expressing the pointcut and advice mechanism in strongly-typed functional languages[21]. Such a calculus would be suitable to describe the language design of Aspectual Caml, which is currently explained at the source language level. AspectML is an AOP extension to Standard ML with the pointcut and advice mechanism[21]. The semantics of AspectML is defined as a translation into MiniAML. TinyAspect is a model of pointcut and advice mechanism for strongly-typed languages with ML-like modules[1]. It proposes a module system for aspects so as to protect join points in a module from aspects outside the module.

From the application programmers' viewpoint, Aspectual Caml has several unique language features that can not be found in those theoretical models, including polymorphism in pointcuts and advice, various kinds of pointcuts other than function calls, and the type extension mechanism. On the other hand, those models are theoretically sound; i.e., they come with static and dynamic semantics with proven type soundness. Those properties of Aspectual Caml are to be shown in future.

There are several studies for adding fields or constructors into existing types, but not in the context of aspect-oriented programming. Type-safe update programming provides a means of extending existing data types[8], which inspired the type extension mechanism in Aspectual Caml. Poly-

[5]Available at `http://www.yl.is.s.u-tokyo.ac.jp/~hideaki/acaml/` .

morphic variants[10] allow to define functions that manipulate variant records without prior declaration of the variant type. This can improve code re-usability of a program when it uses polymorphic variants instead of ordinary variants[11]. Since there have been many programs that developed with ordinary variants, we believe that the polymorphic variants and our type extension mechanism would complement each other.

6. CONCLUSION

This paper presented the design and implementation of Aspectual Caml, an AOP functional language. The language design aims at developing practical applications by adapting many AOP features in existing AOP languages. In order to fit for the programming styles in strongly-typed functional languages, we reconsidered AOP features, including type inference of aspects, polymorphism in pointcuts, and type extension mechanisms. We believe that those features would serve a good basis for further theoretical development of AOP features such as type safety.

A compiler of an Aspectual Caml subset is implemented as a translator to Objective Caml. It is capable to compile non-trivial application programs in which base and aspect definitions deeply interact. Those application programs would also demonstrate that AOP is as useful in functional programming as in object-oriented programming.

We plan to work more on the design and implementation of Aspectual Caml. In particular, a module system for aspects that would nicely work with the ML module system would be needed. We also consider further polymorphism in advice bodies so as to easily define type universal aspects like tracing. One idea is to integrate the language with G'Caml[9] so that advice can use functions that can examine values in different types.

7. ACKNOWLEDGMENT

We would like to thank Jun Furuse for his valuable advice to the language design and for his expert knowledge on the implementation Objective Caml compiler. We would also like to thank the members in the Yonezawa's research group and the Programming Language Principle group at University Tokyo for their helpful discussion. We are grateful to anonymous reviewers for their beneficial comments.

8. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In R. L. Curtis Clifton and G. T. Leavens, editors, *FOAL2004*, Technical Report TR#04–04, Department of Computer Science, Iowa State University, Mar. 2004.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference record of Symposium on Principles of Programming Languages*, pages 1–3, 2002.

[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[4] J. Bonér. What are the key issues for commercial aop use: how does aspectwerkz address them? In *Proc. of AOSD '04*, pages 5–6. ACM Press, 2004. Invited Industry Paper.

[5] B. Burke and A. Brok. Aspect-oriented programming and JBoss. Published on The O'Reilly Network, May 2003. `http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html`.

[6] B. de Alwis and G. Kiczales. Apostle: A simple incremental weaver for a dynamic aspect language. Technical Report TR-2003-16, Dept. of Computer Science, University of British Columbia, 2003.

[7] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, Oct. 2001.

[8] M. Erwig and D. Ren. Type-safe update programming. In *ESOP 2003*, volume 2618 of *LNCS*, pages 269–283, 2003.

[9] J. Furuse. *Extensional Polymorphism: Theory and Application.* PhD thesis, Université Denis Diderot, Paris, Dec. 2002.

[10] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

[11] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, Nov. 2000.

[12] R. Hirschfeld. Aspects - AOP with squeak. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, Oct. 2001.

[13] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 2001.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97*, number 1241 in LNCS, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[17] X. Leroy. A proposal for recursive modules in Objective Caml. `http://cristal.inria.fr/~xleroy/publi/recursive-modules-note.pdf`.

[18] O. Spinczyk, A. Gal, and W. Schroder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proc of TOOLS2002*, pages 18–21, Sydney, Australia, Feb. 2002.

[19] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proc. of AOSD2003*, pages 158–167. ACM Press, 2003.

[20] P. Wadler. Functional programming: An angry half-dozen. *SIGPLAN Notices*, 33(2):25–30, 1998.

[21] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP2003*, 2003.

[22] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In R. Cytron and G. T. Leavens, editors, *FOAL2002*, Technical Report TR#02–06, Department of Computer Science, Iowa State University, pages 1–8, Enschede, The Netherlands, Apr. 2002.

APPENDIX

A. AN EXAMPLE OF COMPILER WRITTEN IN ASPECTUAL CAML

A.1 Base Program: A Simple Compiler

```
(*type of identifiers*)
type ident = I of string
let ppI (I(x)) = x
(*type of immediate values*)
type imm =
  | Int of int
  | Float of float
(*type of terms*)
type syntax =
  | S_Let of ident * syntax * syntax
  | S_Var of ident
  | S_LetRec of s_fundef list * syntax
  | S_App of syntax * syntax list
  | S_NegInt of syntax
  | S_SubInt of syntax * syntax
  | S_IfLEInt of syntax * syntax * syntax * syntax
  | S_Imm of imm
(*mutually recursive functions*)
and s_fundef = { s_name : ident;
 s_args : ident list;
 s_body : syntax }
(*type of terms after K normalizing*)
type knormal =
  | K_Let of ident * knormal * knormal
  | K_Var of ident
  | K_LetRec of k_fundef list * knormal
  | K_App of ident * ident list
  | K_NegInt of ident
  | K_SubInt of ident * ident
  | K_IfLEInt of ident * ident * knormal * knormal
  | K_Imm of imm
and k_fundef = { k_name : ident;
 k_args : ident list;
 k_body : knormal }
(*return a fresh identifier*)
let fresh_knormal =
  let r = ref 0 in
    fun () -> (incr r;
       I(("_knormal_" ^ (string_of_int !r))))
(*K normalizing for the constructor LetRec*)
let rec knormal_letrec fundef = match fundef with
    [] -> []
  | {s_name = ident;
     s_args = ident_list;
      s_body = exp}::tl ->
      {k_name = ident;
       k_args = ident_list;
       k_body = (knormal exp)}::(knormal_letrec tl)
(*K normalizing for the constructor App*)
and knormal_app exp explist =
(*omitted for limited space*)
(*K normalizing*)
and knormal = function
    S_Var(x) -> K_Var(x)
  | S_NegInt(exp) ->
      let f x = K_NegInt(x) in
      insert_let (knormal exp) f
  | S_SubInt(exp1, exp2) ->
      insert_let (knormal exp1)
                 (fun x ->
                   insert_let (knormal exp2)
                              (fun y -> K_SubInt(x, y)))
  | S_IfLEInt(exp1, exp2, exp3, exp4) ->
      insert_let (knormal exp1)
                 (fun x ->
                   insert_let (knormal exp2)
                              (fun y ->
                               K_IfLEInt(x, y,
                                         knormal exp3,
                                         knormal exp4)))
  | S_Let(ident, exp1, exp2) ->
      K_Let(ident, knormal exp1, knormal exp2)
  | S_LetRec(fl, exp) ->
      K_LetRec(knormal_letrec fl, knormal exp)
  | S_App(exp, explist) -> knormal_app exp explist
  | S_Imm(i) -> K_Imm(i)
and insert_let e c = match e with
    K_Var(x) -> c x
  | exp -> let fresh = fresh_knormal () in
      K_Let(fresh, exp, c fresh)
(*K normalizing main*)
let knormal_main s_exp = knormal s_exp
```

A.2 Aspect: Addition of Typing

```
aspect AddType
(*type of types for expressions*)
type typ =
    Tint
  | Tfloat
  | Tvar of typ option ref
  | Tfun of typ list * typ
(*type extension of identifiers for types*)
type+ ident = I of ... * typ{Tvar(ref None)}
(*returns fresh variables with types*)
let fresh_knormal_with_type =
  let r = ref 0 in
    fun typ -> (incr r;
              I("_knormal_" ^ (string_of_int !r), typ))
(*find out concrete type to short cut constructors Tvar*)
let rec get_type = function
    Tvar({contents = Some(t)}) -> get_type t
  | Tfun(t_list, t) -> Tfun(List.map get_type t_list,
                            get_type t)
  | t -> t
(* occur check *)
let rec occur r1 = function
    Tint | Tfloat -> false
  | Tfun(t2s, t2') -> List.exists (occur r1) t2s
                   || occur r1 t2'
  | Tvar(r2) when r1 == r2 -> true
  | Tvar({ contents = None }) -> false
  | Tvar({ contents = Some(t2) }) -> occur r1 t2
exception Unify of typ * typ
(*unification of two types*)
let rec unify t1 t2 =
(*omitted for limited space*)
(*typing after K normalizing*)
let rec id_typing k_exp t_env = match k_exp with
    K_Var(I(x, typ)) -> begin try
      let typ1 = List.assoc x t_env in
        unify typ typ1; typ
    with Not_found ->
       failwith ("unbound_variable: " ^ x) end
  | K_NegInt(I(_, typ)) -> unify typ Tint; typ
  | K_SubInt(I(_, typ1), I(_, typ2)) ->
```

```
       unify typ1 Tint; unify typ2 Tint; Tint
  | K_Let(I(name, typ), k_e1, k_e2) ->
      let typ1 = id_typing k_e1 t_env in
        unify typ1 typ;
        id_typing k_e2 ((name, typ1)::t_env)
  | K_LetRec(k_fundef_list, k_e) ->
      let new_t_env =
        id_typing_letrec k_fundef_list t_env in
      id_typing k_e new_t_env
  | K_App(id, id_list) ->
      id_typing_app id id_list t_env
  | K_IfLEInt(I(_, typ1), I(_, typ2), k_e1, k_e2) ->
      unify typ1 Tint; unify typ2 Tint;
      let ke1_typ = id_typing k_e1 t_env in
      let ke2_typ = id_typing k_e2 t_env in
        unify ke1_typ ke2_typ;
        ke1_typ
  | K_Imm (Int _) -> Tint
  | K_Imm (Float _) -> Tfloat
(*typing after K normalizing
 for the constructor LetRec *)
and id_typing_letrec k_fundef_list t_env =
(*omitted for limited space*)
(*typing after K normalizing for the constructor App*)
and id_typing_app fun_id arg_ids t_env =
(*omitted for limited space*)
(*a flag to judge if a call to knormal is recursive*)
let rec flag = ref false
(*adding typing expression after K normalizing*)
advice knormal_with_typing =
  [around (call knormal s_exp)]
    if not !flag
    then
      let _ = flag := true in
      let k_exp = proceed s_exp in
      let _ = id_typing k_exp [] in
      let _ = flag := false in k_exp
    else
      let k_exp = proceed s_exp in k_exp
end
```

MiniMAO: Investigating the Semantics of Proceed[*]

Curtis Clifton and Gary T. Leavens
Dept. of Computer Science
Iowa State University
229 Atanasoff Hall
Ames, Iowa 50010-1041
{cclifton,leavens}@cs.iastate.edu

ABSTRACT

This paper describes the semantics of MiniMAO$_1$, a core aspect-oriented calculus. Unlike previous aspect-oriented calculi, it allows `around` advice to change the target object of an advised operation before proceeding. MiniMAO$_1$ accurately models the ways AspectJ allows changing the target object, e.g., at `call` join points. Practical uses for changing the target object using advice include proxies and other wrapper objects.

In addition to accurate modeling of bindings for `around` advice, MiniMAO$_1$ has several other features that make it suitable for the study of aspect-oriented mechanisms, such as those found in AspectJ. Like AspectJ, the calculus consists of an imperative, object-oriented base language plus aspect-oriented extensions. MiniMAO$_1$ has a sound static type system, facilitated by a slightly different form of `proceed` than in AspectJ.

1. INTRODUCTION

This paper describes *MiniMAO$_1$*, a core aspect-oriented [13] calculus. MiniMAO$_1$ is designed to explore two key issues in reasoning about operations in aspect-oriented programs:

— when advice may change the target object of the operation, possibly affecting dynamic method selection, and

— when advice may change or capture the arguments to, or results from, the operation.

MiniMAO$_1$ is sufficiently expressive to encode key aspect-oriented idioms. But by minimizing the set of features, we arrive at a core language that is sufficiently small as to make tractable formal proofs of type soundness and—in planned extensions—proofs of desired modularity properties and verification conditions.

In this paper we describe the dynamic semantics of MiniMAO$_1$ and an interesting portion of its type system. We also state its soundness theorem. We assume that the reader is familiar with the basic concepts of aspect-oriented programming as embodied in the AspectJ programming language [12]. Because of space limitations, we refer interested readers to a companion technical report [4] for details that we omit here. We also leave the study of reasoning issues to future work.

For clarity, we begin with a core object-oriented calculus with classes. We then extend this object-oriented calculus with aspects and advice binding.

[*]Supported by NSF grants CCF-0428078 and CCF-0429567.

$$
\begin{aligned}
P &::= decl^*\ e\\
decl &::= \texttt{class}\ c\ \texttt{extends}\ c\ \{\ field^*\ meth^*\ \}\\
field &::= t\ f\\
meth &::= t\ m\texttt{(}form^*\texttt{)}\ \{\ e\ \}\\
form &::= t\ var,\ \text{where } var \neq \texttt{this}\\
e &::= \texttt{new}\ c\texttt{()} \mid var \mid \texttt{null} \mid e\texttt{.}m\texttt{(}e^*\texttt{)} \mid\\
&\quad\ e\texttt{.}f \mid e\texttt{.}f\ \texttt{=}\ e \mid \texttt{cast}\ t\ e \mid e\texttt{;}\ e
\end{aligned}
$$

$$
\begin{aligned}
c,d &\in \mathscr{C}, \text{ the set of a class names}\\
t,s,u &\in \mathscr{T}, \text{ the set of types}\\
f &\in \mathscr{F}, \text{ the set of field names}\\
m &\in \mathscr{M}, \text{ the set of method names}\\
var &\in \{\texttt{this}\} \cup \mathscr{V}, \text{ where } \mathscr{V} \text{ is the set of variable names}
\end{aligned}
$$

Figure 1: Syntax of MiniMAO$_0$

2. THE BASE LANGUAGE: MiniMAO$_0$

In this section we introduce *MiniMAO$_0$*, an imperative object-oriented calculus with classes, derived from Classic Java [8]. Following the lightweight philosophy of Featherweight Java [9], we eliminate interfaces, super calls, and method overloading. We drop `let` expressions and instead use e_1`;` e_2 to sequentially evaluate e_1 and then e_2. We adopt Featherweight Java's technique of treating the current program and its declarations as global constants. This avoids burdening the formal semantics with excess notation.

To allow later modeling of method call and execution join points, we also separate call and execution in the semantics.

2.1 Syntax of MiniMAO$_0$

The syntax for MiniMAO$_0$ is given in Figure 1. A program consists of a sequence of declarations followed by a single expression. Running a program consists of evaluating this expression.

In MiniMAO$_0$ the declarations are all of classes. We omit access modifiers, which would only add gratuitous complexity; hence all methods and fields are globally accessible. MiniMAO$_0$ also omits constructors. All objects are created with their fields set to `null`.

The set of types is denoted by $\mathscr{T}$. MiniMAO$_0$ includes just one built-in type, `Object`, the top of the class hierarchy. `Object` contains no fields or methods. For MiniMAO$_0$, $\mathscr{T} = \mathscr{C}$, the set of valid class names. $\mathscr{C}$ is left unspecified, but we use Java identifier conventions in examples. We follow the same convention for the sets $\mathscr{F}$, $\mathscr{M}$, and $\mathscr{V}$ used in Figure 1.

Most expressions in MiniMAO$_0$ have a meaning like that in Java, but there are some differences. The expression `new C()` creates an instance of the class named `C`, setting all of its fields to the default `null` value. For syntactic clarity, we follow Classic Java in using a non-Java syntax, `cast` t e, to represent the Java cast `(` t `)` e.

2.2 Operational Semantics of MiniMAO$_0$

We describe the dynamic semantics of MiniMAO$_0$ using a structured operational semantics [7, 15, 18]. The semantics is quite similar to that for Classic Java. There are two main differences: a stack (used for aspect binding in MiniMAO$_1$) and the separation of method call and execution into separate primitive operations.

For the operational semantics we add two expressions that do not appear in the user-visible syntax.

$$\begin{aligned} e &::= \ldots \mid loc \mid (\, l\ (\, v \ldots \,)\,) \\ l &::= \texttt{fun}\ m\langle var^{*}\rangle.e : \tau \\ \tau &::= t \times \ldots \times t \to t \\ v &::= loc \mid \texttt{null} \\ loc &\in \mathscr{L}, \text{ the set of store locations} \end{aligned}$$

One can think of locations, $loc \in \mathscr{L}$, as addresses of object records in a global heap, but we just require that $\mathscr{L}$ is some countable set. The application expression form is used to model method execution independently from method calls. In these expressions, l is a (non-first-class) `fun` term that represents a method and `(` $v\ldots$ `)` is an operand tuple that represents the actual arguments. The application expression thus records information from method dispatch, but before execution of the method body. The `fun` term carries type information—a function type, τ. This type information is not used in evaluation rules, but is helpful in the subject-reduction proof. The use of the application expression form in the operational semantics is described in more detail below.

As is typical in an operational semantics, we consider a subset of the expressions, denoted by v, to be irreducible values. The values in MiniMAO$_0$ are the locations and `null`. Evaluation of a well-typed MiniMAO$_0$ program will produce either a value or an exception.

Evaluation contexts are denoted by $\mathbb{E}$. The definition of evaluation contexts below serves both to define implicit congruence rules and to define a left-to-right evaluation order:

$$\begin{aligned} \mathbb{E} ::= - \mid \mathbb{E}.f \mid \mathbb{E}.f = e \mid v.f = \mathbb{E} \mid \texttt{cast}\ t\ \mathbb{E} \mid \mathbb{E}\,;\,e \mid \\ \mathbb{E}.m(\, e\ldots \,) \mid v.m(\, v\ldots\mathbb{E}e\ldots \,) \mid (\, l\ (\, v\ldots\mathbb{E}e\ldots \,)\,) \end{aligned}$$

The evaluation context for the application expression form only recurses on the arguments and not on the method body expression in the `fun` term of the form. Evaluation of the method body does not take place until the substitution of actuals for formals has been done by the appropriate evaluation rule.

The relation, $\hookrightarrow$, describes evaluation steps:

$$\hookrightarrow : \mathscr{E} \times \mathit{Stack} \times \mathit{Store} \to (\mathscr{E} \cup \mathit{Excep}) \times \mathit{Stack} \times \mathit{Store}$$

This relation takes an expression $e \in \mathscr{E}$, a stack, and a store and maps this to a new expression or an exception, plus a new stack and a new store. Exceptions are elements of

$$\mathit{Excep} = \{\texttt{NullPointerException}, \texttt{ClassCastException}\}.$$

For MiniMAO$_0$, the evaluation relation on the stack is identity, so we leave the set *Stack* undefined for now. The set *Store* contains maps from locations to object records, where an object record has the form $[t\,.\{f \mapsto v \cdot f \in dom(\mathit{fieldsOf}(t))\}]$.

Although suppressed in the evaluation relation, the declarations of the program are used to populate a global *class table*, CT, that maps class names to their declarations.

Evaluation of a MiniMAO$_0$ program begins with the triple consisting of the main expression of the program, a stack, and an empty store. The $\hookrightarrow$ relation is applied repeatedly until the resulting triple is not in the domain of the relation. This terminating condition can arise because the resulting triple contains either an irreducible value or an exception. If the resulting triple contains an irreducible value, then that value, interpreted in the resulting store, is the result of the program. There is no guarantee that this evaluation terminates.

The $\hookrightarrow$ relation is defined by a set of mutually disjoint rules. Except for the CALL and EXEC, these rules are standard and are omitted here. The CALL rule is:

$$\begin{aligned} &\langle \mathbb{E}[loc\,.m(\, v_1,\ldots,v_n \,)], J, S\rangle && \textsc{Call} \\ &\quad\hookrightarrow \langle \mathbb{E}[(\, l\ (\, loc, v_1,\ldots,v_n \,)\,)], J, S\rangle \\ &\quad\text{where } S(loc) = [t\,.F] \text{ and } \mathit{methodBody}(t,m) = l \end{aligned}$$

This says that a method call expression, where the target is a location bound in the store, is evaluated by looking up the body of the method and constructing an application form with a function term, l, recording the formal parameters and method body and an argument tuple recording the actual arguments. The interesting part in the definition of the method lookup function is:

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \mathit{field}^{*}\ \mathit{meth}_1\ldots\mathit{meth}_p\ \} \\ \exists i \in \{1..p\} \cdot \mathit{meth}_i = t\ m(\, t_1\ var_1,\ldots,t_n\ var_n \,)\ \{\ e\ \} \\ \tau = c \times t_1 \times \ldots \times t_n \to t \end{array}}{\mathit{methodBody}(c,m) = \texttt{fun}\ m\langle \texttt{this}, var_1,\ldots,var_n\rangle.e : \tau}$$

Another part recursively searches in superclasses when a method is not found. This models inheritance of methods.

The application form produced by the CALL rule is evaluated by the EXEC rule:

$$\begin{aligned} &\langle \mathbb{E}[(\, \texttt{fun}\ m\langle var_0,\ldots,var_n\rangle.e : \tau\ (\, v_0,\ldots,v_n \,)\,)], J, S\rangle && \textsc{Exec} \\ &\quad\hookrightarrow \langle \mathbb{E}[e\{\!|v_0/var_0,\ldots,v_n/var_n|\!\}], J, S\rangle \end{aligned}$$

This rule replaces `this` and the formal parameters in the body with the appropriate values. (The notation $e\{\!|e'/var|\!\}$ denotes the standard capture-avoiding substitution of e' for var in e.)

An example showing the CALL and EXEC rules is given in Section 3.2.6. The companion technical report [4] contains the complete operational semantics. It also contains a separate static semantics and soundness theorem for MiniMAO$_0$.

3. MiniMAO$_1$: ADDING ASPECTS

In this section we add advice binding to MiniMAO$_0$, producing the aspect-oriented core calculus MiniMAO$_1$. Continuing with our minimalist philosophy, the join point model of MiniMAO$_1$ is quite simple. The model only includes `call` and `execution` join points, the parameter binding forms `this`, `target`, and `args`, and the operators for pointcut union, intersection, and negation. We intensionally omit temporal join points, such as `cflow`; the techniques for dealing semantically with such join points are well understood [17], and such temporal join points do not substantially affect the typing rules for aspects.

MiniMAO$_1$ accurately models AspectJ's semantics for around advice [12], in that it allows advice to change the target object of a method call or execution before proceeding with the operation. Moreover, as in AspectJ, changing the target object at a call join point affects method selection for the call, but changing the target object at an execution join point merely changes the self object of the already selected method. Changing the target object is useful for such idioms as introducing proxy objects. Such proxy objects can be used in aspect-oriented implementations of persistence or for redirecting method calls to remote machines.

$$\begin{aligned}
\mathit{decl} &::= \ldots \mid \texttt{aspect}\ a\ \{\ \mathit{field}^*\ \mathit{adv}^*\ \} \\
\mathit{adv} &::= t\ \texttt{around(}\mathit{form}^*\texttt{)}\ \texttt{:}\ \ \mathit{pcd}\ \{\ e\ \} \\
\mathit{pcd} &::= \texttt{call(}\mathit{pat}\texttt{)} \mid \texttt{execution(}\mathit{pat}\texttt{)} \mid \\
&\qquad \texttt{this(}\mathit{form}\texttt{)} \mid \texttt{target(}\mathit{form}\texttt{)} \mid \texttt{args(}\mathit{form}^*\texttt{)} \mid \\
&\qquad \mathit{pcd}\ \texttt{\&\&}\ \mathit{pcd} \mid \texttt{!}\mathit{pcd} \mid \mathit{pcd}\ \texttt{||}\ \mathit{pcd} \\
\mathit{pat} &::= t\ \mathit{idPat}\texttt{(..)} \\
e &::= \ldots \mid e\texttt{.proceed(}e^*\texttt{)}
\end{aligned}$$

$a \in \mathscr{A}$, the set of aspect names

$\mathit{idPat} \in \mathscr{I}$, the set of identifier patterns

Figure 2: Syntax Extensions for MiniMAO$_1$

MiniMAO$_1$ does depart from AspectJ's semantics for around advice in two ways: it does not allow changing the `this` (i.e., the caller) object at a `call` join point and it uses a different form of `proceed`, which syntactically looks like the advised method call rather than the surrounding advice declaration as in AspectJ. These differences are discussed more below.

One motivation for the design of MiniMAO$_1$ is to keep pointcut matching, advice execution, and primitive operations in the base language as separate as possible. This goal causes us to use more evaluation rules that are strictly necessary. One way to think of MiniMAO$_1$ is as an operational semantics for an aspect-oriented virtual machine, where each primitive operation may generate a join point that may trigger other rules for advice matching. Our approach increases the syntactic complexity of the calculus, but we find that it actually simplifies reasoning. The approach keeps separate concepts in separate rules that can be analyzed with separate lemmas.

No previous work on formalizing the semantics of an aspect-oriented language deals with the actual AspectJ semantics of argument binding for `proceed` expressions and an object-oriented base language. Our calculus is motivated by the insight of Walker et al. [16] that labeling primitive operations is a useful technique for modeling aspect-oriented languages. However, to handle the run-time changing of the target object and arguments when proceeding from advice, we replace their simple labels with more expressive join point abstractions. Also, rather than introduce these join point abstractions through a static translation from an aspect-oriented language to a core language, we generate them dynamically in the operational semantics. The extra data needed for the join point abstractions (versus the simple static labels) is more readily obtained when they are generated dynamically. (This dynamic generation is also adopted by Dantas and Walker [5].) Also, directly typing the aspect-oriented language, instead of just showing a type-safe translation to the labeled core language, seems to more clearly illustrate the issues in typing advice, though this is a matter of taste. Our type system is motivated by that of Jagadeesan et al. [11]. We discuss this and other related work in more detail in Section 4.

3.1 Syntax of MiniMAO$_1$

Figure 2 gives the additional syntax for MiniMAO$_1$. To the declarations of MiniMAO$_0$ we add aspects. For a MiniMAO$_1$ program the set of types, $\mathscr{T}$, is $\mathscr{C} \cup \mathscr{A}$, where $\mathscr{A}$ is the set of aspect names. An aspect declaration includes a sequence of field declarations and a sequence of advice declarations.

We only include `around` advice in MiniMAO$_1$. Operationally, `around` advice can be used to model both `before` and `after` advice. (As noted by Jagadeesan et al. [11], the typing rules necessary for soundness may be less restrictive for `before` or `after` advice.)

An advice declaration in MiniMAO$_1$ consists of a return type, followed by the keyword `around` and a sequence of formal parameters. The pointcut descriptor that follows specifies the set of join points—the *pointcut*—where the advice should be executed. A *join point* is any point in the control flow of a program where advice may be triggered. The pointcut descriptor for a piece of advice also specifies how the formal parameters of the advice are to be bound to the information available at a join point. The final part of an advice declaration is an expression that is the advice body.

MiniMAO$_1$ includes a limited vocabulary for pointcut descriptors. The `call` pointcut descriptor matches the invocation of a method whose signature matches the given pattern. Similarly, the `execution` pointcut descriptor matches the join point corresponding to a method execution. In both of these, we restrict method patterns to a concrete return type plus an identifier pattern that is matched against the name of the called method. We choose not to include matching against target or parameter types here because that is just syntactic sugar for the `target` and `args` pointcut descriptors.

We leave the set $\mathscr{I}$ of identifier patterns underspecified. Generally, one can think of $\mathscr{I}$ as a class of regular expression languages such that all members of $\mathscr{M}$ are elements of a language in $\mathscr{I}$.

The `this`, `target`, and `args` pointcut descriptors correspond to the parameter-binding forms of these descriptors in AspectJ; they bind the named formal parameters to the corresponding information from the join point. To simplify the operational semantics, the syntax requires a type and a formal parameter. For example, where one could write `this(n)` in AspectJ, one must write `this(Number n)` in MiniMAO (where `Number` is the type of the formal parameter `n` in the advice declaration). While this type could be inferred, including it in the syntax clarifies the formalism. Another simplification versus AspectJ is that the `args` pointcut descriptor in MiniMAO$_1$ binds all arguments available at the join point; such bindings do not allow matching of methods with differing numbers of arguments, unlike those in AspectJ. MiniMAO$_1$ does not include any wildcard or subtype matching for `this`, `target`, or `args` pointcut descriptors.

The final three pointcut descriptor forms represent pointcut negation (`!`*pcd*), union (*pcd* `||` *pcd*), and intersection (*pcd* `&&` *pcd*). Pointcut negation only reverses the boolean (match or mismatch) value of the negated pointcut. Any parameters bound by the negated pointcut are dropped. Pointcut union and intersection are "short circuiting"; for example, if pcd_1 in the form pcd_1 `||` pcd_2 matches a join point, then the bindings defined by pcd_1 are used and pcd_2 is ignored.

MiniMAO$_1$ also includes `proceed` expressions, which are only valid within advice. An expression such as $e_0\texttt{.proceed(}e_1,\ldots,e_n\texttt{)}$ takes a target, e_0, and sequence of arguments, $e_1,\ldots,e_n$, and causes execution to continue with the code at the advised join point—either the original method or another piece of advice that applies to the same method. As noted above, the `proceed` expression in MiniMAO$_1$ differs from AspectJ. In MiniMAO$_1$, an expression of the form $e_0\texttt{.proceed(}e_1,\ldots,e_n\texttt{)}$ must be such that the type of the target, e_0, and the number and types of the arguments, $e_1,\ldots,e_n$, match those of the *advised methods*. In AspectJ, the arguments to proceed must match the formal parameters of the surrounding *advice*. This design decision matches our intuition for how `proceed` should work; it has little effect on expressiveness in a language with type-safe around advice. Our design also precludes changing the `this` object at `call` join points. Such changes would only be

$$
\begin{aligned}
J &::= j + J \mid \bullet \\
j &::= (\!| k, v_{opt}, m_{opt}, l_{opt}, \tau_{opt} |\!) \\
k &::= \texttt{call} \mid \texttt{exec} \mid \texttt{this} \\
v_{opt} &::= v \mid - \\
m_{opt} &::= m \mid - \\
l_{opt} &::= l \mid - \\
\tau_{opt} &::= \tau \mid -
\end{aligned}
$$

Figure 3: The Join Point Stack

$$
\begin{aligned}
e &::= \ldots \mid \texttt{joinpt}\ j(\, e^* \,) \mid \texttt{under}\ e \mid \texttt{chain}\ \bar{B}, j(\, e^* \,) \\
\bar{B} &::= B + \bar{B} \mid \bullet \\
B &::= [\![b, loc, e, \tau, \tau]\!] \\
b &::= \langle \alpha, \beta, \beta^* \rangle \\
\alpha &::= var \mapsto loc \mid - \\
\beta &::= var \mid - \\
b &\in \mathscr{B}, \text{the set of advice parameter bindings}
\end{aligned}
$$

Figure 4: Expression Forms Added for the Semantics

visible from other aspects, not the base program. Precluding these changes eliminates some possibilities for aspect interference, a useful property for our work on aspect-oriented reasoning. We are not aware of any use cases demonstrating a need to allow changing the `this` object.

3.2 Operational Semantics of MiniMAO$_1$

This section gives the changes and additions to the operational semantics for MiniMAO$_1$. We describe the stack, new expression forms introduced for the operational semantics, the new evaluation rules, pointcut descriptor matching, and give evaluation examples.

3.2.1 The Join Point Stack

The stack in MiniMAO$_1$, as shown in Figure 3, is a list of *join point abstractions*, which are five-tuples surrounded by half-moon brackets, $(\!|\ldots|\!)$. A join point abstraction records all the information in a join point that is needed for advice matching and advice parameter bindings, together referred to as *advice binding*. A join point abstraction also includes all the information necessary to proceed from advice to the original code that triggered the join point. A join point abstraction consists of the following parts (most of which are optional and are replaced with "$-$" when omitted):

— a join point kind, k, indicating the primitive operation of the join point, or `this` to record the self object at method or advice execution (for binding the `this` pointcut descriptor);

— an optional value indicating the self object at the join point;

— an optional name indicating the method called or executed at the join point;

— an optional `fun` term recording the body of the method to be executed at an execution join point; and

— an optional function type indicating the type of the code under the join point (or, equivalently, the type of a `proceed` expression in any advice that binds to the join point).

The code *under* a join point is the program code that would execute at that join point if no advice matched the join point. For example, the code under a method execution join point is the body of the method. The function type includes the type of the target object as the first argument type.

3.2.2 New Expression Forms

The operational semantics relies on three extra expression forms, shown in Figure 4. The first, `joinpt`, reifies join points of a program evaluation into the expression syntax. A `joinpt` expression consists of a join point abstraction followed by a sequence of actual arguments to the code under the join point.

The second expression form that we add for the operational semantics is `under`. An `under` expression serves as a marker that the nested expression is executing under a join point; that is, a join point abstraction was pushed onto the stack before the nested expression was added to the evaluation context. When the nested expression has been evaluated to a value, then the corresponding join point abstraction can be popped from the stack.

The final additional expression form is `chain`. A `chain` expression records a list, $\bar{B}$, of all the advice that matches at a join point, along with the join point abstraction and the original arguments to the code under the join point.

The advice list of a `chain` expression consists of *body tuples*, one per matching piece of advice. For visual clarity, "snake-like" brackets, $[\![\ldots]\!]$, surround each body tuple. A body tuple is comprised of two parts: operational information and type information. The operational information includes: b, a parameter binding term described below, *loc*, a location, and e, an expression. The location is the self object; it is substituted for `this` when evaluating the advice body. The expression is the advice body.

The *binding term*, b, describes how the values of actual arguments should be substituted for formals in the advice body. This substitution is somewhat complex to account for the special binding of the `this` pointcut descriptor, which takes its data from the original join point, and the `target` and `args` pointcut descriptors, which take their data from the invocation or `proceed` expression immediately preceding the evaluation of the advice body.

Structurally, a binding term consists of a variable-location pair, $var \mapsto loc$, which is used for any `this` pointcut descriptors, followed by a non-empty sequence of variables, which represent the formals to be bound to the target object and each argument in order. The "$-$" symbol is used to represent a hole in a binding term. A hole might occur, for example, if a pointcut descriptor did not use `this`. The set of all possible binding terms is $\mathscr{B}$.

The type information in a body tuple is contained in its last two elements. The first of these represents the declared type of the advice, an arrow from formal parameter types to the return type. The second type element, the last element in the body tuple, is the type of any `proceed` expression contained within the advice body. While this type information simplifies the subject-reduction proof, it is not used in the evaluation rules.

3.2.3 Evaluation Rules for MiniMAO$_1$

Next we give an intuitive description of the new evaluation rules in MiniMAO$_1$. We add new evaluation context rules to handle the `joinpt`, `under`, and `chain` expressions.

$$
\begin{aligned}
\mathbb{E} ::= \ldots \mid \texttt{joinpt}\ j(\, v \ldots \mathbb{E} e \ldots \,) \mid \texttt{under}\ \mathbb{E} \mid \\
\texttt{chain}\ \bar{B}, j(\, v \ldots \mathbb{E} e \ldots \,)
\end{aligned}
$$

The semantics replaces `proceed` expressions with `chain` expressions, so we do not need additional rules for handling `proceed`.

We replace the CALL rule of MiniMAO$_0$ with a pair of rules, CALL$_A$ and CALL$_B$ described below, that introduce join points and handle proceeding from advice respectively. We replace the EXEC rule similarly. This division exposes join points for call and execution to the evaluation rules. Just as virtual dispatch is a primitive operation in a Java virtual machine, our semantics models advice binding as a primitive operation on these exposed join points. This advice binding is done by the new BIND rule. The new ADVISE rule models advice execution, and an UNDER rule helps maintain the join point stack by recording when join point abstractions should be popped.

The evaluation of a program in MiniMAO$_1$ does not begin with an empty store as in MiniMAO$_0$. Instead, a single instance of each declared aspect is added to the store. The locations of these instances are recorded in the global *advice table*, *AT*, which is a set of 5-tuples. Each 5-tuple represents one piece of advice. The 5-tuple for the advice t `around(` $t_1\ var_1,\ldots,t_n\ var_n$ `)` : pcd `{` e `}`, declared in aspect a, is $\langle loc, pcd, e, (t_1 \times \ldots \times t_n \to t), \tau\rangle$, where loc is such that $S_0(loc) = [a\,\mathbf{.}\,F]$ is the aspect instance for a in the initial store, S_0. The function type τ is the type of `proceed` expressions in e, derived from pcd.

The global class table, *CT*, is extended in MiniMAO$_1$ to also map aspect names to the aspect declarations.

3.2.4 Splitting the Call Rule

In MiniMAO$_0$, a method call is evaluated by applying the CALL and EXEC rules in turn. In MiniMAO$_1$, each of these steps is broken into a series of steps. The CALL step becomes:

— CALL$_A$: creates a `call` join point

— BIND: finds matching advice

— ADVISE: evaluates each piece of advice

— CALL$_B$: looks up method, creates an application form

A similar division of labor is used for EXEC. We next describe each of these steps in turn.

Create a Join Point. The CALL$_A$ rule is as follows:

$$\begin{array}{ll}\langle \mathbb{E}[loc\,.\,m(\ v_1,\ldots,v_n\ )], J, S\rangle & \textsc{Call}_A\\ \quad\hookrightarrow \langle \mathbb{E}[\texttt{joinpt}\ (\!|\texttt{call}, -, m, -, \tau|\!)(\ loc, v_1,\ldots,v_n\ )], J, S\rangle & \\ \qquad \text{where } S(loc) = [t\,\mathbf{.}\,F], & \\ \qquad\quad methodType(t,m) = t_1 \times \ldots \times t_n \to t', & \\ \qquad\quad origType(t,m) = t_0, \text{ and } \tau = t_0 \times \ldots \times t_n \to t' & \end{array}$$

This says that a method call expression with a non-`null` target evaluates to a `joinpt` expression where the join point abstraction carries the information about the call necessary to bind advice and to proceed with the original call. This information is: the `call` kind, the method name, and a function type, τ, for the method that includes a target type in the first argument position. The function type is determined using a pair of auxiliary functions, the interesting bits of which are:

$$\frac{\begin{array}{c}CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\ field^*\ meth_1 \ldots meth_p\ \}\\ \exists i \in \{1..p\} \cdot meth_i = t\ m(\ t_1\ var_1,\ldots,t_n\ var_n\ )\ \{\ e\ \}\end{array}}{methodType(c,m) = t_1 \times \ldots \times t_n \to t}$$

$$\begin{array}{l} origType(t,m) =\\ \quad \max\{s \in \mathscr{T} \cdot t \preccurlyeq s \wedge methodType(s,m) = methodType(t,m)\}\end{array}$$

The function, *methodType*, is essentially the same as *methodBody*, defined earlier, but it yields a function type instead of a function term. The function, *origType*, finds the type of the "most super" class of the target type that also declares the method m. (The subtyping relation used in *origType* is just the reflexive transitive closure of the `extends` relation on classes, treating aspects as subtypes of `Object`.) The target type included in the `call` join point abstraction generated by CALL$_A$ is this most super class. Using the most super class allows advice to match a call to any method in a family of overriding methods, by specifying the target type as this most super class. We discuss this a bit more when describing the `target` pointcut descriptor below. Finally, the arguments of the generated `joinpt` expression are the target location—again in the first position—and the arguments of the original call, in order.

Find Matching Advice. The BIND rule is the only place in the calculus where advice binding (lookup) occurs. This rule takes a `joinpt` expression and converts it to a `chain` expression that carries a list of all matching advice for the join point. It also pushes the expression's join point abstraction onto the join point stack.

$$\begin{array}{ll}\langle \mathbb{E}[\texttt{joinpt}\ j(\ v_0,\ldots,v_n\ )], J, S\rangle & \textsc{Bind}\\ \quad\hookrightarrow \langle \mathbb{E}[\texttt{under chain}\ \bar{B}, j(\ v_0,\ldots,v_n\ )], j + J, S\rangle & \\ \qquad \text{where } adviceBind(j + J, S) = \bar{B} & \end{array}$$

The rule uses the auxiliary function *adviceBind* to find the (possibly empty) list of advice matching the new join point stack and store.

$$\begin{array}{l} adviceBind(J,S) = \bar{B}, \text{ where } \bar{B} \text{ is a smallest list satisfying}\\ \quad \forall \langle loc, pcd, e, \tau, \tau'\rangle \in AT \cdot ((matchPCD(J, pcd, S) = b \neq \bot)\\ \qquad\qquad\qquad \implies [\![b, loc, e, \tau, \tau']\!] \in \bar{B})\end{array}$$

The *adviceBind* function applies the *matchPCD* function, described in Section 3.2.5, to find the matching advice in the global advice table. (We leave *adviceBind* underspecified. In particular, we don't give an order for the advice in the list. For practical purposes some well-defined ordering is needed, but any consistent ordering, such as the declaration ordering used in our examples, will suffice.)

Having found the list of matching advice, the BIND rule then constructs a new `chain` expression consisting of this list of advice, the original join point abstraction, and the original arguments. The result expression is wrapped in an `under` expression to record that the join point abstraction must later be popped from the stack.

Evaluate Advice. The ADVISE rule takes a `chain` expression with a non-empty list of advice and evaluates the first piece of advice.

$$\begin{array}{ll}\langle \mathbb{E}[\texttt{chain}\ [\![b, loc, e, \_, \_]\!] + \bar{B}, j(\ v_0,\ldots,v_n\ )], J, S\rangle & \textsc{Advise}\\ \quad\hookrightarrow \langle \mathbb{E}[\texttt{under}\ e'\{\!|loc/\texttt{this}|\!\}\{\!|(v_0,\ldots,v_n)/b|\!\}], j' + J, S\rangle & \\ \qquad \text{where } e' = \langle\!\langle e \rangle\!\rangle_{\bar{B},j} \text{ and } j' = (\!|\texttt{this}, loc, -, -, -|\!) & \end{array}$$

The general procedure is to substitute for `this` in the advice body with the location, loc, of the advice's aspect and substitute for the advice's formal parameters according to the binding term, b. We describe below how the binding term is used for the substitution. However, before the substitution occurs the rule uses the $\langle\!\langle - \rangle\!\rangle_{\bar{B},j}$ auxiliary function to eliminate `proceed` expressions in the advice body.

The "advice chaining" auxiliary function, $\langle\!\langle - \rangle\!\rangle_{\bar{B},j}$, is defined for `proceed` expressions as:

$$\begin{array}{l}\langle\!\langle e_0\texttt{.proceed(}\ e_1,\ldots,e_n\ \texttt{)} \rangle\!\rangle_{\bar{B},j}\\ \qquad\qquad = \texttt{chain}\ \bar{B}, j(\ \langle\!\langle e_0 \rangle\!\rangle_{\bar{B},j}, \langle\!\langle e_1 \rangle\!\rangle_{\bar{B},j}, \ldots, \langle\!\langle e_n \rangle\!\rangle_{\bar{B},j}\ )\end{array}$$

For all other expression forms, the chaining operator is just applied recursively to every subexpression. Thus $\langle\!\langle - \rangle\!\rangle_{\bar{B},j}$ rewrites all

$$e\{\!|\langle v_0,\ldots,v_n\rangle / \langle var \mapsto loc, \beta_0,\ldots,\beta_p\rangle|\!\} = e\{\!|loc/var|\!\}\{\!|v_i/var_i|\!\}_{i\in\{0..n\}\cdot\beta_i=var_i} \text{ where } n \le p$$

$$e\{\!|\langle v_0,\ldots,v_n\rangle / \langle -, \beta_0,\ldots,\beta_p\rangle|\!\} = e\{\!|v_i/var_i|\!\}_{i\in\{0..n\}\cdot\beta_i=var_i} \text{ where } n \le p$$

In all other cases, binding substitution is undefined.

Figure 5: Binding Substitution

`proceed` expressions, replacing them with `chain` expressions carrying the remainder of the advice list $\bar{B}$, along with the join point abstraction, j, needed to proceed to the original operation once the advice list has been exhausted. This rewriting is like that used by Jagadeesan et al. [10], though they do not consider the target object to be one of the arguments to `proceed`. Advice chaining is illustrated with an example in Section 3.2.6.

After using the advice chaining function to rewrite the advice body, the ADVISE rule uses variable substitution to bind the formal parameters of the advice to the actual arguments. It substitutes the aspect location, *loc*, for `this` and substitutes the actuals for the formals according to *b*. We overload the usual substitution notation to define substitution for binding terms. Figure 5 gives this definition. The definition says that the variable in the $var \mapsto loc$ pair is replaced with the location, unless there is a hole,"–", in this position of the binding term. (Here *var* is a formal parameter of the advice and *loc* is the location of the calling object at the join point.) Each element, β_i, in the binding term that is not a hole must be a variable. Each such variable is replaced with the corresponding argument, v_i. For example:

$$(\texttt{x.f = y})\{\!|\langle\texttt{loc0,loc1}\rangle / \langle\texttt{x} \mapsto \texttt{loc2, -, y}\rangle|\!\} = (\texttt{loc2.f = loc1})$$

The $\texttt{x} \mapsto \texttt{loc2}$ in the binding term does not use data from the arguments $\langle\texttt{loc0,loc1}\rangle$; the value `loc0` is not used because of the hole in the binding term; and `y` is replaced with `loc1`. The type system rules out repeated use of a variable in a binding term.

After substitution, the ADVISE rule pushes a `this` join point abstraction onto the stack—equivalent to the self reference stored on the call stack in a Java virtual machine—and wraps the result expression in an `under` expression, which records that the join point abstraction should be popped from the stack later.

Finish the Original Operation. Once the list of advice has been exhausted, the result is a `chain` expression with an empty advice list, the original join point abstraction, and a sequence of arguments. If the BIND rule had found no advice, then the arguments will be the target and arguments from the original call. Otherwise, the arguments will be whatever was provided by the last piece of advice. This `chain` expression is used by the CALL_B rule to evaluate the original call.

$$\begin{array}{ll}\langle\mathbb{E}[\texttt{chain }\bullet,(\!|\texttt{call},-,m,-,\tau|\!)\texttt{(}\,loc,v_1,\ldots,v_n\,\texttt{)}],J,S\rangle & \text{CALL}_\text{B}\\ \quad\hookrightarrow\langle\mathbb{E}[\texttt{(}\,l\,\texttt{(}\,loc,v_1,\ldots,v_n\,\texttt{)}\,\texttt{)}],J,S\rangle & \\ \quad\text{where } S(loc)=[t\,\texttt{.}\,F] \text{ and } \mathit{methodBody}(t,m)=l & \end{array}$$

The CALL_B rule looks up the type of the (possibly changed) target object in the store and finds the method body in the global class table. The rule takes the method name from the join point abstraction. The result of the rule is an application expression, just like the result of the CALL rule in MiniMAO_0.

Because both the CALL_A and CALL_B rules use a target location for method lookup, there are corresponding rules for `null` targets. These rules just map to a triple with a `NullPointerException` and are omitted here.

A General Technique. The technique used to convert the CALL rule from the MiniMAO_0 calculus into a pair of rules, with intervening advice binding and execution, is general. The first rule in the new pair replaces the original expression with a `joinpt` expression, ready for advice binding. The second rule in the pair takes a `chain` expression, exhausted of advice, and maps it to a new expression like the result expression of the rule from MiniMAO_0. This is how the two new EXEC rules are generated:

$$\begin{array}{ll}\langle\mathbb{E}[\texttt{(}\,l\,\texttt{(}\,v_0,\ldots,v_n\,\texttt{)}\,\texttt{)}],J,S\rangle & \text{EXEC}_\text{A}\\ \quad\hookrightarrow\langle\mathbb{E}[\texttt{joinpt }(\!|\texttt{exec},v_0,m,l,\tau|\!)\texttt{(}\,v_0,\ldots,v_n\,\texttt{)}],J,S\rangle & \\ \quad\text{where } l=\texttt{fun }m\langle var_0,\ldots,var_n\rangle.e:\tau & \\ \langle\mathbb{E}[\texttt{chain }\bullet,(\!|\texttt{exec},v,m,l,\tau|\!)\texttt{(}\,v_0,\ldots,v_n\,\texttt{)}],J,S\rangle & \text{EXEC}_\text{B}\\ \quad\hookrightarrow\langle\mathbb{E}[\texttt{under }e\{\!|v_0/var_0,\ldots,v_n/var_n|\!\}],j+J,S\rangle & \\ \quad\text{where } l=\texttt{fun }m\langle var_0,\ldots,var_n\rangle.e:\tau \text{ and} & \\ \qquad j=(\!|\texttt{this},v_0,-,-,-|\!) & \end{array}$$

The EXEC_A rule replaces the application expression with a `joinpt` expression. The join point abstraction of this expression includes the `exec` kind, the method name, the `fun` term of the application, and the type of the `fun` term. The abstraction also includes, in the position reserved for `this` objects, the value of the target object from the argument tuple, because `target` and `this` objects are the same at an `execution` join point. The arguments to the `joinpt` expression are the arguments to the original application expression.

The EXEC_B rule takes a `chain` expression that has been exhausted of its advice. It applies the `fun` term from the `chain`'s join point abstraction to the argument sequence, substituting the arguments for the variables in the body of the `fun` term. Like ADVISE, the EXEC_B rule pushes a `this` join point abstraction onto the stack and wraps its result expression in an `under` expression.

It would be straightforward to add pointcut descriptors and join points for any of the primitive operations in the original calculus, such as field assignment. We would have to generalize the data carried in the join point abstractions to accommodate additional information, but the BIND and ADVISE rules would remain unchanged. Because the `call` and `exec` join points are sufficient for our study, we choose not to include join points for the other primitive operations. To do so would just introduce additional notation and bookkeeping.

The Under Rule. The UNDER rule is the simplest of the new evaluation rules.

$$\begin{array}{ll}\langle\mathbb{E}[\texttt{under }v],J,S\rangle\hookrightarrow\langle\mathbb{E}[v],J',S\rangle & \text{UNDER}\\ \quad\text{where } J=j+J'\text{, for some j} & \end{array}$$

It just extracts the value from the `under` expression and pops one join point abstraction from the stack.

3.2.5 Pointcut Matching

Following Wand et al. [17], we define the *matchPCD* function for matching pointcut descriptors to join points using a boolean algebra over binding terms. Our binding terms, as described in Section 3.2.2 above, are somewhat more complex than theirs, since we model `this`, `target`, and `args` pointcut descriptors and faithfully model the semantics of `proceed` from AspectJ with regard to changing target objects in advice. Nevertheless, the basic technique is the same.

The boolean algebra is:

$$\mathscr{B}_\perp = \mathscr{B} \cup \{\perp\} \qquad b \in \mathscr{B} \qquad r \in \mathscr{B}_\perp \qquad b \vee r = b$$

$$\perp \vee r = r \qquad \perp \wedge r = \perp \qquad b \wedge \perp = \perp \qquad b \wedge b' = b \sqcup b'$$

$$\neg\perp = \langle -,- \rangle \qquad \neg b = \perp$$

The terms of the algebra are drawn from the set $\mathscr{B}_\perp = \mathscr{B} \cup \{\perp\}$, where binding terms can be thought of as "true" and $\perp$ as "false". The operators in the algebra are conjunction ($\wedge$), disjunction ($\vee$), and complement ($\neg$). The double complement of an element is not necessarily the original element, unless we consider all binding terms to be isomorphic; the effect of this detail on advice binding is discussed below. The binary operators are short circuiting; for example, $b \vee r = b$, ignoring the value of r. One difference in our algebra, versus Wand et al. [17], is in the conjunction of two non-$\perp$ terms. Our calculus must consider the bindings from both terms, because we have more than one pointcut descriptor that can bind formals. Sometimes these bindings must be combined, for example when both a `target` and `args` pointcut descriptor are used. The bindings are combined using a pointwise join:

$$\begin{aligned}&\langle \alpha, \beta_0, \ldots, \beta_n \rangle \sqcup \langle \alpha', \beta_0', \ldots, \beta_p' \rangle \\ &\quad = \langle \alpha \sqcup \alpha', \beta_0 \sqcup \beta_0', \ldots, \beta_q \sqcup \beta_q' \rangle \\ &\qquad \text{where } q = \max(n,p), \\ &\qquad\quad \forall i \in \{(n+1)..q\} \cdot (\beta_i = -), \text{ and} \\ &\qquad\quad \forall i \in \{(p+1)..q\} \cdot (\beta_i' = -)\end{aligned}$$

The pointwise join operator extends the shorter binding term if the two terms do not have the same number of elements. The join operator, $\sqcup$, on pairs of α or β terms resolves to the term that is not a hole. Collisions in the join operator, where neither binding has a hole at a given position, are resolved in favor of the left-hand term; however, the typing rules for pointcut descriptors ensure that such collisions do not occur in well-typed programs.

The rules defining *matchPCD* are straightforward. If the pointcut descriptor matches the join point stack, then the rules construct the appropriate binding term; otherwise they evaluate to $\perp$. The only complications are to accommodate the multiple parameter binding forms. For example, `this` and `target` matching must be done without information on how many additional arguments might be bound by an `args` pointcut descriptor. Thus, the length of binding terms must be allowed to vary.

Call and Execution. The `call` rule only matches if the most recent join point is of the corresponding kind and the return type and name of the method under the join point are matched by the pattern:

$$\begin{aligned}&\mathit{matchPCD}(\llparenthesis k, \llcorner\!\lrcorner, m, \llcorner\!\lrcorner, t_0 \times \ldots \times t_p \to t \rrparenthesis + J, \\ &\qquad\qquad \texttt{call(}\ u\ \mathit{idPat}\texttt{(..)}\ \texttt{)}, S) \\ &\qquad\qquad = \begin{cases} \langle -,- \rangle & \text{if } k = \texttt{call}, t = u, m \in \mathit{idPat} \\ \perp & \text{otherwise} \end{cases}\end{aligned}$$

Because this pointcut descriptor does not bind formal parameters, a match is indicated by an empty binding term. The `execution` rule is similar.

This. Two rules are used to handle `this` pointcut descriptors:

$$\begin{aligned}&\mathit{matchPCD}(\llparenthesis \llcorner\!\lrcorner, v, \llcorner\!\lrcorner, \llcorner\!\lrcorner, \llcorner\!\lrcorner \rrparenthesis + J, \texttt{this(}\ t\ \mathit{var}\ \texttt{)}, S) \\ &\qquad = \begin{cases} \langle \mathit{var} \mapsto v, - \rangle & \text{if } v \neq \texttt{null}, S(v) = [s\texttt{.}F], s \preccurlyeq t \\ \perp & \text{otherwise} \end{cases}\end{aligned}$$

$$\begin{aligned}&\mathit{matchPCD}(\llparenthesis \llcorner\!\lrcorner, -, \llcorner\!\lrcorner, \llcorner\!\lrcorner, \llcorner\!\lrcorner \rrparenthesis + J, \texttt{this(}\ t\ \mathit{var}\ \texttt{)}, S) \\ &\qquad\qquad = \mathit{matchPCD}(J, \texttt{this(}\ t\ \mathit{var}\ \texttt{)}, S)\end{aligned}$$

Together, these rules find the most recent join point where the optional self object location is provided in the join point abstraction. Once found, if the object record in that location is a subtype of the formal parameter type, then the formal named by the pointcut descriptor is mapped to the location; otherwise the result is $\perp$.

Target. The `target` pointcut descriptor is handled similarly to `this`, but uses the target type from the join point instead:

$$\begin{aligned}&\mathit{matchPCD}(\llparenthesis \llcorner\!\lrcorner, \llcorner\!\lrcorner, \llcorner\!\lrcorner, \llcorner\!\lrcorner, s_0 \times \ldots \times s_n \to s \rrparenthesis + J, \\ &\qquad\qquad \texttt{target(}\ t\ \mathit{var}\ \texttt{)}, S) \\ &\qquad\qquad\qquad = \begin{cases} \langle -, \mathit{var} \rangle & \text{if } s_0 = t \\ \perp & \text{otherwise} \end{cases}\end{aligned}$$

A rule for searching through the join point stack is elided. Unlike the `this` pointcut descriptor, the location to be bound to the formals is not available from the join point abstraction. The location may come from a `proceed` expression to be evaluated later. Also unlike `this`, `target` requires an exact type match. This is necessary for type soundness, as noted by Jagadeesan et al. [11]. If the descriptor were to match when the target type was a supertype of the parameter type, then the advice could call a method on the object bound to the formal that did not exist in the object's class. On the other hand, if the descriptor were to match when the target type was a subtype of the parameter type, then the advice could replace the target object with a supertype before proceeding to a method call. If this supertype did not declare the method, then a runtime type error would result.[1] Thus, for soundness the `target` pointcut descriptor must use exact type matching. If advice were not allowed to change the target object, then less restrictive target type matching could be used.

This restriction to exact type matching is not as severe as it may seem at first. This is because when the CALL$_\text{A}$ rule generates the target type for its join point abstraction, it uses the type of the class declaring the top-most method in the method overriding hierarchy. Thus, the actual target object for a matched call may be a subtype of the target type that was matched exactly. Using the declaring class of this top-most method also means that advice can be written to match a call to any method in a family of overriding methods. Unlike the CALL$_\text{A}$ rule, the EXEC$_\text{A}$ rule creates a join point abstraction using the actual target type. Again, this is necessary for soundness. At an `exec` join point method selection has already occurred and advice cannot be allowed to change the target object to a superclass even if that superclass declared an overridden method.

We are also interested in investigating whether a more elaborate type system might permit more expressive pointcut matching while maintaining soundness. However, this is orthogonal to our concerns with modular reasoning and so we leave it for future work.

Args. The rule for the `args` pointcut descriptor is similar to the one for `target` above. It matches if the argument types of the most recent join point match those of the pointcut descriptor. The resulting binding includes all formals named in the pointcut descriptor in the corresponding positions. As with the `target` pointcut descriptor, only the relative position to be bound, not the actual value, is available until the advice is executed.

The rules for pointcut descriptor operators (which we elide) sim-

[1] Indeed, in AspectJ 1.2, which includes subtype matching for its `target` pointcut descriptor, one can generate a run-time type error in just this way.

```
class Cl extends Object {
    Object m(Cl a) { this; a }
}

new Cl().m(new Cl);
```

Figure 6: A Sample Program Without Aspects

```
aspect A {
    Object around(Cl t, Cl s) :
        call(Object m(..))
            && target(Cl t) && args(Cl s)
    { this }
}
```

Figure 7: Aspect Added to Program of Figure 6

ply appeal to the corresponding operators in the binding algebra: union to disjunction, intersection to conjunction, and negation to complement. The definition of complement implies that $\neg\neg pcd \neq pcd$. Both would match the same pointcut, but the former would not bind any formals while the later might. (This is slightly different than AspectJ, which simply disallows binding pointcut descriptors under negation operators.)

A final rule says that any cases not covered by the other rules evaluates to $\bot$. This just serves to make *matchPCD* a total function, handling cases that do not occur in the evaluation of a well-typed program (such as matching against an empty join point stack).

3.2.6 Example Evaluations in MiniMAO$_1$

This section gives examples of several evaluations.

Calls in MiniMAO$_0$ vs. MiniMAO$_1$. Suppose we have the program declared in Figure 6. This program does not include any aspects and the result of evaluating it is the same in MiniMAO$_0$ and MiniMAO$_1$, though the difference in the steps taken is illustrative. In both cases there is an evaluation step with left hand side:

$$\langle \texttt{L0.m(L1)}, \bullet, S\rangle$$

where the store S maps both L0 and L1 to Cl objects. In MiniMAO$_0$ this evolves by the CALL and EXEC rules:

$$\hookrightarrow \langle \texttt{(fun m}\langle\texttt{this, a}\rangle\texttt{.(this;a):}\tau\ \texttt{(L0,L1))}, \bullet, S\rangle \qquad (\textsc{Call})$$
$$\hookrightarrow \langle \texttt{L0; L1}, \bullet, S\rangle \qquad (\textsc{Exec})$$

where we leave τ as an exercise for the reader. On the other hand, the evaluation in MiniMAO$_1$ is:

$$\begin{aligned}
&\langle \texttt{L0.m(L1)}, \bullet, S\rangle\\
&\hookrightarrow \langle \texttt{joinpt}\ (\!|\texttt{call},-,\texttt{m},-,\tau'|\!)\ \texttt{(L0, L1)}, \bullet, S\rangle \quad (\textsc{Call}_\textsc{A})\\
&\hookrightarrow \langle \texttt{under chain}\ \bullet, (\!|\texttt{call},-,\texttt{m},-,\tau'|\!)\ \texttt{(L0, L1)}, J, S\rangle \quad (\textsc{Bind})\\
&\hookrightarrow \langle \texttt{under}\\
&\qquad \texttt{(fun m}\langle\texttt{this, a}\rangle\texttt{.(this;a):}\tau\ \texttt{(L0,L1))}, J, S\rangle \quad (\textsc{Call}_\textsc{B})\\
&\hookrightarrow \langle \texttt{under} \quad (\textsc{Exec}_\textsc{A})\\
&\qquad \texttt{joinpt}\ (\!|\texttt{exec},\texttt{L0},\texttt{m},l,\tau|\!)\ \texttt{(L0, L1)}, J, S\rangle\\
&\hookrightarrow \langle \texttt{under under} \quad (\textsc{Bind})\\
&\qquad \texttt{chain}\ \bullet, (\!|\texttt{exec},\texttt{L0},\texttt{m},l,\tau|\!)\ \texttt{(L0, L1)}, J', S\rangle\\
&\hookrightarrow \langle \texttt{under under (L0; L1)}, J', S\rangle \quad (\textsc{Exec}_\textsc{B})
\end{aligned}$$

where l is $\texttt{fun m}\langle\texttt{this, a}\rangle\texttt{.(this;a):}\tau$, and τ', J, and J' are left to the reader. Each step in the original evaluation is split into two parts, with intervening advice lookup.

Advice Binding. Suppose we add the aspect declaration of Figure 7 to the program in Figure 6. The presence of this advice changes the result of the first BIND step above (i.e., the one for the call pointcut descriptor). BIND's call to *adviceBind* uses the following application of *matchPCD*:[2]

$$\begin{aligned}
&matchPCD((\!|\texttt{call},-,\texttt{m},-,\tau'|\!), pcd, S)\\
&\qquad\qquad \text{where}\quad \tau' = \texttt{Cl}\times\texttt{Cl}\rightarrow\texttt{Object}, \text{ and}\\
&\qquad\qquad\qquad pcd \text{ is from Figure 7}\\
&= matchPCD((\!|\texttt{call},-,\texttt{m},-,\tau'|\!), \texttt{call(Object m(..))}, S)\\
&\quad \wedge matchPCD((\!|\texttt{call},-,\texttt{m},-,\tau'|\!), \texttt{target(Cl t)}, S)\\
&\quad \wedge matchPCD((\!|\texttt{call},-,\texttt{m},-,\tau'|\!), \texttt{args(Cl s)}, S)\\
&= (\langle -,-\rangle \sqcup \langle -,\texttt{t}\rangle) \sqcup \langle -,-,\texttt{s}\rangle\\
&= \langle -,\texttt{t}\rangle \sqcup \langle -,-,\texttt{s}\rangle\\
&= \langle -,\texttt{t},\texttt{s}\rangle
\end{aligned}$$

Using this matching derivation, the result of the BIND step is:

$$\begin{aligned}
&\langle \texttt{under chain}\ [\![\lfloor\langle -,\texttt{t},\texttt{s}\rangle,\ \texttt{L2},\ \texttt{this},\ \tau',\ \tau'\rfloor]\!],\\
&\qquad (\!|\texttt{call},-,\texttt{m},-,\tau'|\!)\ \texttt{(L0, L1)}, J, S\rangle
\end{aligned}$$

where L2 is the location of A's aspect instance in the initial store. This triple evolves by the ADVISE rule. Because the body of the advice does not proceed to the advised code, the result of this step is the final result of the program, after using UNDER to pop the join point stack:

$$\begin{aligned}
&\hookrightarrow \langle \texttt{under under L2}, J'', S\rangle && (\textsc{Advise})\\
&\hookrightarrow \langle \texttt{under L2}, J, S\rangle && (\textsc{Under})\\
&\hookrightarrow \langle \texttt{L2}, \bullet, S\rangle && (\textsc{Under})
\end{aligned}$$

Advice Chaining. A final example considers advice that proceeds to the advised code and changes the target object. Consider the program in Figure 8. Unlike our previous examples, the advice proceeds and there is a subclass, SCl, which is used for the argument to the method call. Evaluation of this program reaches a stage where the result of the BIND rule is:

$$\begin{aligned}
&\langle \texttt{under chain}\\
&\qquad [\![\lfloor\langle -,\texttt{t},\texttt{s}\rangle,\ \texttt{L2},\ \texttt{s.proceed(t)},\ \tau',\ \tau'\rfloor]\!],\\
&\qquad\quad (\!|\texttt{call},-,\texttt{m},-,\tau'|\!)\ \texttt{(L0, L1)}, J, S\rangle
\end{aligned}$$

where, as before, L2 is the location of A's instance and L0 is the location of a Cl instance, but now L1 is the location of a SCl instance. This triple evolves by the ADVISE rule, which calculates

$$\langle\!\langle \texttt{s.proceed(t)} \rangle\!\rangle_{\bullet,j} = \texttt{chain}\ \bullet, j\ \texttt{(s, t)}$$

where $j = (\!|\texttt{call},-,\texttt{m},-,\tau'|\!)$. The rule then substitutes into this expression according to the binding term $\langle -,\texttt{t},\texttt{s}\rangle$ to form its result, with the order of the two locations swapped as compared to the original, advice-free example above:

[2]Technically the store must be different than before, due to the aspect instance in the initial store. However, because S is underspecified, we use the same meta-variable here to facilitate comparisons.

```
aspect A {
    Object around(Cl t, Cl s) :
        call(Object m(..))
            && target(Cl t) && args(Cl s)
    {
        s.proceed(t) // swaps target, argument
    }
}

class Cl extends Object {
    Object m(Cl a) { this; a }
}

class SCl extends Cl {
    Object m(Cl a) { new Object() }
}

new Cl().m(new SCl);
```

Figure 8: A Sample Program Demonstrating Proceed

$\hookrightarrow \langle$`under under chain` $\bullet, j$ `(L1, L0)`$, J'', S\rangle$ (ADVISE)

$\hookrightarrow \langle$`under`
`(fun m`$\langle$`Cl this, Cl a`$\rangle$`.(new Object()):`τ `(L1,L0)),`
$J'', S\rangle$

(CALL$_B$)

The method body found by the CALL$_B$ rule is declared in `SCl`, instead of in `Cl`.

We invite the reader to consider the same example, but replace the advice's `call` pointcut descriptor with a similar `execution` one. This will demonstrate that changing the target object when proceeding at an `exec` join point does not affect method selection.

3.3 Static Semantics of MiniMAO$_1$

We next sketch some of the static semantics of MiniMAO$_1$. We focus on the typing of pointcuts and advice, since they are the most interesting deviations from past work.

The rules for typing pointcut descriptors make use of a simple algebra over $\mathscr{T} \cup \{\bot\}$, whose only operator, $\sqcup$, is used to combine type information when pointcuts are intersected:

$$t \sqcup \bot = t \qquad \bot \sqcup t = t \qquad \bot \sqcup \bot = \bot$$

The operation is undefined for $t \sqcup s$, because in the type judgment for pointcuts such a combination would indicate a collision and is disallowed. This operation is also lifted to type sequences.

The type of a pointcut descriptor, *pcd*, has six parts, $\hat{u}\,.\,\hat{u}'\,.\,U\,.\,\hat{u}''\,.\,V_1\,.\,V_2$, where:

— $\hat{u}$ is the `this` type matched by *pcd*;

— $\hat{u}'$ is the target type;

— U is the tuple of argument types;

— $\hat{u}''$ is the return type;

— V_1 is the set of variables that would definitely be bound by *pcd* at a matched join point; and

— V_2 is the set of variables that might be bound there.

Each of the type parts may also be $\bot$ to indicate that the information cannot be determined from the pointcut descriptor. The two sets of variables, V_1 and V_2, represent "must-bind" and "may-bind" sets respectively, which are useful in reasoning about variable bindings in pointcut unions and intersections. Well-typed advice requires that the must-bind and may-bind sets are identical (see the first hypothesis of T-ADV below).

The pointcut descriptor typing rules are mostly straightforward. We discuss a couple of them here. The T-TARGPCD rule gives the type for a `target` pointcut descriptor:

T-TARGPCD
$$\frac{\Gamma(var) = t}{\Gamma \vdash \texttt{target(}\ t\ var\ \texttt{)} : \bot\,.\,t\,.\,\bot\,.\,\bot\,.\,\{var\}\,.\,\{var\}}$$

The hypothesis of the above rule looks up the type of *var* in the type environment Γ. (Γ is a partial map from $\mathscr{V} \cup \{\texttt{this}, \texttt{proceed}\}$ to $\mathscr{T}$.) The conclusion of the rule records the target type, t, of the pointcut descriptor and records that the must- and may-bind sets are both $\{var\}$. The rules for the other base cases (`call`, `execution`, `this`, and `args`) are similar.

The most interesting of the typing rules for recursive pointcut descriptors is the one for intersection:

T-INTPCD
$$\frac{\begin{array}{c}\Gamma \vdash pcd_1 : \hat{u}_1\,.\,\hat{u}'_1\,.\,U_1\,.\,\hat{u}''_1\,.\,V_1\,.\,V'_1 \\ \Gamma \vdash pcd_2 : \hat{u}_2\,.\,\hat{u}'_2\,.\,U_2\,.\,\hat{u}''_2\,.\,V_2\,.\,V'_2 \\ \hat{u} = \hat{u}_1 \sqcup \hat{u}_2 \quad \hat{u}' = \hat{u}'_1 \sqcup \hat{u}'_2 \quad U = U_1 \sqcup U_2 \quad \hat{u}'' = \hat{u}''_1 \sqcup \hat{u}''_2 \\ V'_1 \cap V'_2 = \emptyset \qquad V = V_1 \cup V_2 \qquad V' = V'_1 \cup V'_2\end{array}}{\Gamma \vdash pcd_1 \texttt{ \&\& } pcd_2 : \hat{u}\,.\,\hat{u}'\,.\,U\,.\,\hat{u}''\,.\,V\,.\,V'}$$

This rule allows for the combination of the various binding forms in pointcut descriptors like `target(T t) && args(S s)`. The first two hypotheses obtain the types of pcd_1 and pcd_2. The next four hypotheses combine these types using the $\sqcup$ operator described above. These hypotheses select the non-$\bot$ entries from the types and prevent duplicate bindings. For example, if both pcd_1 and pcd_2 have a non-$\bot$ target type, $\hat{u}'_1 \sqcup \hat{u}'_2$ is undefined and pcd_1 `&&` pcd_2 has no type. Finally the last three hypotheses deal with the must- and may-bind sets. $V'_1 \cap V'_2 = \emptyset$ requires no overlap in the sets variables that may be bound by the two pointcut descriptors. The last two hypotheses calculate the combined must- and may-bind sets.

Advice is well typed if its pointcut descriptor matches a join point where the code under the join point has target type u_0, argument types $u_1, \ldots, u_p$ and return type u.

T-ADV
$$\frac{\begin{array}{c} var_1 : t_1, \ldots, var_n : t_n \vdash pcd : \text{␣}\,.\,u_0\,.\,\langle u_1, \ldots, u_p\rangle\,.\,u\,.\,V\,.\,V \\ V = \{var_1, \ldots, var_n\} \\ var_1 : t_1, \ldots, var_n : t_n, \texttt{this} : a, \texttt{proceed} : (u_0 \times \ldots \times u_p \to u) \vdash e : s \\ s \preccurlyeq t \preccurlyeq u \end{array}}{\vdash t\ \texttt{around(}\ t_1\ var_1, \ldots, t_n\ var_n\ \texttt{)}\ \ :\ \ pcd\ \ \{\ \ e\ \ \}\ \ \text{OK in } a}$$

The "␣" in the first hypothesis indicates that the type bound by a `this` pointcut descriptor does not affect the advice type. The pointcut descriptor must also specify bindings for all of the formal parameters of the advice; the use of $\{var_1, \ldots, var_n\}$ for both the must- and may-bind sets ensures this. Finally, the body of the advice is typed in an environment that gives each formal its declared type; gives `this` the aspect type, a; and gives `proceed` the type derived from *pcd*. In this environment, the advice body must have a type that is a subtype of the declared return type of the advice. In turn, this declared return type must be a subtype of the return type of the original code under the join point. This allows the result of the advice to be substituted for the result of the original code.

Rule T-ADV permits advice to declare a return type that is a subtype of that of the advised method. This means that advice like:

```
A around(C t) :
    call(B m(..)) && target(C t) && args()
{ t.proceed() }
```

is not well typed if A is a proper subtype of B: the proceed expression has type B, which is not a subtype of the declared return type of the advice. Wand et al. [17, §5.3] argue that this advice should be typable, but we disagree. This case is really no different than a super call in a language with covariant return-type specialization. In such a language, an overriding method that specializes the return type cannot merely return the result of a super call as its result. The overriding method must ensure that the result is appropriately specialized.

3.4 Meta-theory of MiniMAO$_1$

The key property of MiniMAO$_1$ is that it is type sound: a well-typed program either converges to a value or exception, or else it diverges. We prove this using the usual subject reduction and progress theorems. For MiniMAO$_0$, the proofs closely follow those of Flatt et al. [8]. The soundness proof for MiniMAO$_1$ relies on a pair of key lemmas that we sketch here. The companion technical report [4] gives the full details.

The first key lemma is used in the BIND case of the subject reduction proof. The lemma relates advice binding to advice typing. It is used to argue that the list of advice that matches at a joinpt expression can be used by the BIND rule to generate a well typed chain expression. We prove the lemma using a structural induction on the type derivation for the pointcut of the matching advice.

The second key lemma states that advice chaining, replacing proceed expressions with chain expressions, does not affect typing judgments given the appropriate assumptions. This lemma is used for the ADVISE case in the subject reduction proof.

The subject reduction and progress theorems are standard and are elided. Finally, we have the soundness theorem.

THEOREM 1 (SOUNDNESS). *Given a program*

$$P = decl_1 \ldots decl_n\ e, \text{ with } \vdash P\ OK,$$

and a valid store S_0, *then either the evaluation of e diverges or else* $\langle e, \bullet, S_0 \rangle \overset{*}{\hookrightarrow} \langle v, J, S \rangle$ *and one of the following hold for v:*

— $v = loc$ *and* $loc \in dom(S)$,

— $v = \texttt{null}$, *or*

— $v \in \{\texttt{NullPointerException}, \texttt{ClassCastException}\}$

4. RELATED WORK

No previous work deals with the actual AspectJ semantics of argument binding for proceed expressions and an object-oriented base language. Wand et al. [17] present a denotational semantics for an aspect-oriented language that includes temporal pointcut descriptors. Our use of an algebra of binding terms for advice matching is derived from their work. Their semantics binds all advice parameters at the join point instead of at each subsequent proceed expression. Their calculus is not object-oriented and so does not deal with the effects on method selection of changing the target object. Douence et al. [6] present a system for reasoning about temporal pointcut matching. They do not formalize advice parameter binding and do not include proceed in their language.

Jagadeesan et al. [10] present a calculus for a multithreaded, class-based aspect-oriented language. They omit methods, using advice for all code abstraction. The lack of separate methods simplifies their semantics, but makes their calculus a poor fit for our planned studies of a verification logic for AspectJ-like languages. Also, their calculus does not include the ability of advice to change the target object of an invocation. In an unpublished paper [11] they add a sound type system to their calculus. Our type system is motivated by that work, but extends it to handle the separate this, target, and args binding forms and the ability of advice to change the target object.

Masuhara and Kiczales [14] give a Scheme-based model for an AspectJ-like language. They do not include around advice in their model. They do sketch how this could be added, but do not address the effect on method selection of changing the target object.

Aldrich [2] presents a system called "open modules" that includes advice and dynamic join points with a module system that can restrict the set of control flow points to which advice may be attached. The system is not object-oriented, so it does not address the issue of changing the target of a method call, and it does not include state. Dantas and Walker [5] present a simple object-based calculus for "harmless advice". They use a type system with "protection levels" to keep aspects from altering the data of the base program. However, in keeping with this non-interference property, they do not allow advice to change values when proceeding to the base program.

Bruns et al. [3] describe μABC, a name-based calculus in which aspects are the primitive computational entity. Their calculus does not include state directly, but can model it via the dynamic creation of advice. However, it is not obvious how such a model of state could be used for our planned study of aspect-oriented reasoning when aspects may interfere with the base program via the heap. Also, while their calculus does allow modeling of a form of proceed, It is difficult to see how it could be used to study the effects of advice on method selection. Finally, their calculus is untyped and is not class-based.

Walker et al. [16] use an innovative technique of translating an aspect-oriented language into a labeled core language, where the labels serve as both advice binding sites and targets for goto expressions, where they are used to translate around advice that does not proceed. While their work does consider around advice and proceed in an object-oriented setting—the object calculus of Abadi and Cardelli [1]—it does not consider changing any arguments to the advised code, let alone the effects on method selection of changing the target object of an invocation.

5. CONCLUSION

In many respects MiniMAO$_1$ faithfully explains the semantics of AspectJ's around advice on method call and execution join points. In particular, MiniMAO$_1$ faithfully models the binding of arguments and the ability of proceed to change the target object in a call join point. The semantics supports this ability by breaking the processing of method calls into several steps: (i) creating the join point for the call, (ii) finding matching advice, (iii) evaluating each piece of advice, and (iv) finally creating an application form. Since the target object is only used to determine the method called in step (iv) (the CALL$_B$ rule), the advice can change the target by using a different target in the proceed expression. Such a change affects the application form created, which affects the join point created for the method's execution.

In addition to the necessary simplifications, MiniMAO$_1$, also has a few interesting differences from AspectJ. In particular the typing of proceed and the various pointcut descriptions has a different philosophy from AspectJ. Its typing in MiniMAO$_1$ corresponds to the type of the method being advised, instead of being related to the type of the advice's formal parameters. This contributes to a simpler and more understandable semantics for proceed.

Future work involves using MiniMAO$_1$ to study the reasoning problems indicated in the introduction.

6. ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their helpful comments.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[2] Jonathan Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL 2004 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, pages 7–18, Lancaster, UK, 2004. URL `http://www.cs.iastate.edu/~leavens/FOAL/papers-2004/proceedings.pdf`.

[3] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μabc: A minimal aspect calculus. In *Proceedings of the 2004 International Conference on Concurrency Theory*, pages 209–224. Springer-Verlag, 2004.

[4] Curtis Clifton and Gary T. Leavens. MiniMAO: Investigating the semantics of proceed. Technical Report TR05-01, Iowa State University, 2005. Available from `ftp://ftp.cs.iastate.edu/pub/techreports/TR05-01/TR.ps.gz`.

[5] Daniel S. Dantas and David Walker. Harmless advice. In *The 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*, Long Beach, California, 2005. ACM.

[6] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Reflection 2001*, number 2192 in LNCS. Spring-Verlag, November 2001.

[7] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[8] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, chapter 7, pages 241–269. Springer-Verlag, 1999. URL `http://citeseer.ist.psu.edu/flatt99programmers.html`.

[9] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[10] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In Luca Cardelli, editor, *ECOOP 2003, European Conference on Object-Oriented Programming, Darmstadt, Germany*, volume 2743, pages 54–73. Springer-Verlag, 2003.

[11] Radha Jagadeesan, Alan Jeffrey, and James Riely. A typed calculus for aspect oriented programs. Available from `ftp://fpl.cs.depaul.edu/pub/rjagadeesan/typedABL.pdf`, Feb 2004.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072, pages 327–353. Springer-Verlag, Berlin, 2001.

[13] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[14] Hidehiko Masuhara and Gregar Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP 2003 - Object-Oriented Programming European Conference*, pages 2–28. Springer-Verlag, 2003.

[15] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[16] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, Uppsala, Sweden, 2003. ACM Press.

[17] Mitchell Wand, Gregor Kiczales, and Chris Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Trans. on Prog. Lang. and Sys.*, 26(5):890–910, 2004.

[18] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115 (1):38–94, 1994.

Expressiveness and Complexity of Crosscut Languages

Karl J. Lieberherr
Northeastern University,
Boston, MA
lieber@ccs.neu.edu

Jeffrey Palm
Northeastern University,
Boston, MA
jpalm@ccs.neu.edu

Ravi Sundaram
Northeastern University,
Boston, MA
koods@ccs.neu.edu

ABSTRACT

Selector languages, or crosscut languages, play an important role in aspect-oriented programming (AOP). Examples of prominent selector languages include the pointcut language in AspectJ, traversal specifications in Demeter, XPath, and regular expressions. A selector language expression, also referred to as a *selector*, selects nodes on an instance graph (an execution tree or an object tree) that satisfies a meta graph (a call graph or a class graph). The implementation of selector languages requires practically efficient algorithms for problems such as: Does a selector always (or never) select certain nodes **Select-Always** (**Select-Never**), does a selector ever select a node **Select-Sat**, does one selector imply another selector **Select-Impl** or may an edge in an instance graph lead to a node selected by the selector **Select-Completion**.

We study these problems from the viewpoints of two important selector languages called SAJ, inspired by AspectJ, and SD, inspired by Demeter, and several of their sublanguages. We show a polynomial-time two-way reduction between SD and SAJ revealing interesting connections promoting transfer of algorithmic techniques from AspectJ to Demeter and vice-versa. We provide several practically useful polynomial-time algorithms for some of the problems, and we show others to be NP- or co-NP-complete. We present a fixed parameter tractable (FPT) algorithm for one of the NP-complete problems. This early result indicates a line of attack for dealing with the intractability inherent in these problems.

The paper provides a list of algorithmic results that are of interest to developers of scalable AOP tools. We discuss the consequences of this paper for our DAJ implementation.

General Terms

AspectJ, Demeter, pointcut designators, traversal strategies

1. INTRODUCTION

Aspect-oriented programs consist of two building blocks: WhereToInfluence and WhatToDo. The WhereToInfluence part defines the points in an executing program where we want to influence the program. The WhatToDo part defines how to influence the program. In this paper we analyze declarative, non Turing-complete selector (or crosscut) languages to formulate the WhereToInfluence part.

In a pioneering paper, Masuhara and Kiczales [16] compare crosscutting in four aspect-oriented mechanisms, including AspectJ and Demeter. We extend their work to include both algorithmic upper bounds as well as hardness results on several computational problems underlying AspectJ and Demeter. For example, motivated by another influential paper by Masuhara and Kiczales [17], we show that general elimination of run-time tests in AspectJ programs, even without negation in the pointcuts, is NP-complete in the general case.

Our analysis is at a high level of abstraction, yet detailed enough to provide useful practical input for the implementation of selector languages. The analysis is useful to current tools, e.g., AspectJ and Demeter (DemeterJ, DJ, DAJ[2]), and for many more aspect-oriented languages to come. Our model is a three level model [11] where at the top level we have selectors (e.g., pointcut designators or traversal strategies), at the second level meta graphs (e.g., static call graphs or class graphs) and at the third level instance trees (e.g., dynamic call trees or object trees) conforming to the meta graphs. The purpose of the selectors is to choose a set of nodes in the instance trees, or equivalently to choose a set of paths from the root of the trees to those nodes. For an example, the meta graph for the AspectJ program in Figure 2 is sketched in Figure 1.

We study several algorithmic problems for two kinds of selector languages and their sublanguages. The first language, called SAJ, is an abstraction of the AspectJ pointcut language. We lump all primitive pointcuts together into a term $n(l)$, selecting all the nodes with label l. We use `flow`(S), selecting all nodes reachable from the root through a node in S. And we add the set-theoretic operators `|`, `&` and `!`.

The second language, SD, is an abstraction and generalization of the Demeter traversal strategies. We use the version described in Palsberg *et al.* [21] but extended with the set-theoretic operators `&` and `!`. SD is more flow oriented, and we reuse the semantics from [21] in terms of path sets.

We consider two kinds of applications of selector languages.

AspectJ-style applications: The selector language is used to select nodes in the execution trees and their corresponding shadows in the program. The virtual machine decides, based on the input data, which execution tree to construct and the tree is traversed in full but only a subset of the nodes satisfies the selector expression. The term pointcut language is used instead of selector language.

Demeter-style applications: The selector language is used to select nodes in the object trees and their corresponding shadows in the meta graph. The object tree is given as input, and the tree is partially traversed reaching all the nodes satisfying the selector expression. The term traversal language is used instead of selector language.

One point of this paper is to also consider SAJ for Demeter-style applications and SD for AspectJ-style applications. The paper points out the close relationship between those two languages. We consider the following algorithmic problems for SAJ and SD and their sublanguages. For all of those problems we consider the version where the meta graph is given and for Select-Sat-Static we consider the case where only the selector is given as input and we ask for the existence of a suitable meta graph. **Select-Always:** Does a selector always select nodes with label A in all instances? This problem is useful for AspectJ-style applications of selector languages: it frees us from having to do any checking at run-time. See papers by Masuhara/Kiczales [17], Oege deMoor [23], and Wu/Lieberherr [27]. **Select-Always** is also useful for Demeter-style applications of selector languages: We are not required to do any run-time checking to ensure that the traversal is at the right place.

Select-Never: This is similar to the previous item. Does a selector select no nodes with label A in any instance?

Select-Sat-Static: Does a selector ever select a node? Here, we check whether a given selector has an effect on at least one instance graph by selecting at least one node. Selectors that never select a node are useless and should be corrected.

Select-Sat: Like **Select-Sat-Static**, except that in addition to the selector a meta graph is also given as input.

Select-Impl: Does one selector imply another selector? **Select-Impl** is useful in predicate dispatch languages, such as Fred [19] and Socrates [20], where inheritance is replaced by predicate implication. We cover here the special case where the predicates are declarative.

Select-First: Does an edge in an instance graph lead to a node selected by the selector? This is useful for guiding traversals [11] and for deciding whether a selector influences a particular branch of the execution of a program [17]. Our results in this paper should be considered in the context of the General Pointcut Satisfiability Problem:

> Given an AspectJ pointcut p and a Java program G,
> is there an execution of G in which p will select
> at least one join point?

This problem is undecidable even for a very simple pointcut language because the undecidability comes from the conditional statements in G.

Therefore we consider a conservative approximation of the program in the form of a call graph. We assume that all calls inside a procedure could happen. Because of the simple structure of meta graphs, we treat dynamic dispatch in a very simple way: zero or more of the calls could happen.

In this paper we show two kinds of results: lower-bound results, like NP-hardness and co-NP-completeness results and upper-bound results, like that certain checking problems can be solved in polynomial time. For the lower-bound results it is sufficient to consider only very limited programs, e.g., programs that only contain calls (without conditional statements). For the usefulness of our upper-bound results the conservative approximation mentioned above is an issue that needs to be explored further. The conservative approximation allows for many more possible program executions than can happen in practice. But still the upper-bound results are interesting because universally quantified statements (over all executions/instances) for the approximation are correct statements for the real program.

The following properties are preserved by the conservative approximation: not Select-Sat, Select-Always, Select-Never, Select-Impl, not Select-First. Note that Select-Sat is not preserved by the approximation because the meta graph might have an instance in which a join point is selected but that instance might never happen as an execution in the real program.

We show a polynomial-time two-way reduction from SD to SAJ revealing interesting connections and promoting the transfer of algorithmic techniques from AspectJ to Demeter and vice-versa. We provide several practically useful polynomial-time algorithms for some of the problems ,and we show others to be NP-complete or co-NP-complete. We present a fixed parameter tractable (FPT) algorithm for one of the NP-complete problems. This early result indicates a line of attack for dealing with the intractability inherent in these problems.

Our NP-completeness proofs are simple but not trivial. For example, we show that satisfiability and other problems for AspectJ pointcuts without complement are already NP-complete. The point of our reduction is that when we translate a boolean formula to a pointcut satisfiability problem, we can use the graph to simulate negation although the pointcut language does not itself contain negation.

In this paper we often refer to the traversal graph defined in [13, 11]. For the purpose of this paper we view the traversal graph as the Cartesian product of two graphs, where one graph is the meta graph and the other is the graph version of the SD selector expression. The Cartesian graph product $G = G_1 \times G_2$ of graphs G_1 and G_2 with disjoint point sets V_1 and V_2 and edge sets E_1 and E_2 is the graph with point set $u = (u_1, u_2)$ and $v = (v_1, v_2)$ adjacent with whenever $[u_1 = v_1$ and u_2 adj $v_2]$ or $[u_2 = v_2$ and u_1 adj $v_1]$ [8]. We note that the meta graph structure and selector language in [11] are more expressive and hence required a more elaborate construction of traversal graphs.

Our paper uncovers novel aspects of the interplay between predicates and graphs. We believe that there is potential for further connections between this paper and the seminal work of Courcelle relating logic and graphs [1].

In summary the paper provides a novel framework for the study of the expressiveness of selector languages and their related algorithmic problems. We discuss the consequences of this paper for our DAJ implementation.

The rest of the paper is organized as follows: In section 2 we introduce our framework by defining meta graphs and instance graphs and our selector languages, SAJ and SD, including translations between them. In section 3 we introduce the problems, including the practical motivation behind them. Section 4 discusses a Fixed Parameter Tractable algorithm for Satisfiability with an application to Select-Sat. Section 5 contains related work and section 6 conclusions and future work.

2. GRAPH STRUCTURE AND SELECTOR LANGUAGE

For a particular graph there are a possibly infinite number of *instances* conforming to the graph structure, each of which, later, will be mapped to an AspectJ program execution call trace, or a Demeter object graph traversal. To select interesting points in an execution call trace or an object graph traversal, we have a general selector language which, later, will be mapped to AspectJ's pointcut designator language or Demeter's traversal specification.

2.1 Directed Graph and Instances

DEFINITION 1 (DIRECTED GRAPH). *A directed graph G is a pair $< V, E >$, where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges. There is a distinguished vertex $r \in V$, which is the starting vertex in G.* $\mathsf{Start}(G)$ *is defined on a graph G that returns its distinguished starting vertex for G from which all other nodes are reachable.*

We assume a labeling from nodes and edges to a finite alphabet, so that $\mathsf{Label}(x)$ is the label for a node or edge x.

DEFINITION 2 (INSTANCES OF GRAPH). *A directed graph I is called an instance of G, if I is a tree,* $\mathsf{Root}(I) = \mathsf{Start}(G)$ *and for each edge $e = (u, v) \in E(I)$, there is an edge $e' = (u', v') \in G$ so that* $\mathsf{Label}(u) = \mathsf{Label}(u')$ *and* $\mathsf{Label}(v) = \mathsf{Label}(v')$.

2.2 Paths

A *path* in a graph is a sequence $v_1 \ldots v_n$ where $v_1, \ldots, v_n$ are nodes of the graph; and $v_i \rightarrow v_{i+1}$ is an edge of the graph for all $i \in 1..n-1$. We call v_1 and v_n the source and the target of the path, respectively. If $p_1 = v_1 \cdots v_i$ and $p_2 = v_i \cdots v_n$, then we define the concatenation $p_1 p_2 = v_1 \cdots v_i \cdots v_n$.[1]

Suppose P_1 and P_2 are sets of paths where all paths in P_1 have the target v and where all paths of P_2 have the source v. Then we define[2]

$$P_1 \cdot P_2 = \{p \mid p = p_1 p_2 \text{ where } p_1 \in P_1 \text{ and } p_2 \in P_2\}.$$

$\mathsf{Paths}_\Phi(A, B)$ is defined as all paths from A to B in Φ where A and B are nodes of the meta graph Φ.

[1]The v_i in a path don't have to be distinct. v_1 is a path from source v_1 to target v_1 where $n = 1$.

[2]$P_1 \cup P_2$ is the set union of the paths in P_1 and P_2.

2.3 General Selector Language

We use two selector languages, SAJ and SD, based roughly on the selector languages of AspectJ and Demeter, respectively. SAJ has the form

$$S ::= l \mid \texttt{flow}(S) \mid S \texttt{ | } S \mid S \texttt{ \& } S \mid !S \qquad (1)$$

where l is a node label. The following are the evaluation rules for SAJ. We state them as a reduction, S_I:

$$\begin{aligned} S_I(l) &= \{v \mid v \in I \wedge \text{Label}(v) = l\} \\ S_I(\texttt{flow}(S)) &= \{v \mid \text{some } n \in S_I(S) \text{ reaches } v \in I\} \\ S_I(S_1 \texttt{ | } S_2) &= S_I(S_1) \cup S_I(S_2) \\ S_I(S_1 \texttt{ \& } S_2) &= S_I(S_1) \cap S_I(S_2) \\ S_I(!S) &= \backslash S_I(S) \end{aligned}$$

A *traversal specification* in SD has the form

$$D ::= [A, B] \mid D \cdot D \mid D \texttt{ | } D \mid D \texttt{ \& } D \mid !D \qquad (2)$$

where A and B are nodes of a meta graph. Such a specification denotes a set of paths in a given meta graph Φ, intuitively as follows:

Selector	**Set of paths**
$[A, B]$	The set of paths from A to B in Φ
$D_1 \cdot D_2$	Concatenation of sets of paths
$D_1 \texttt{ \| } D_2$	Union of sets of paths
$D_1 \texttt{ \& } D_2$	Intersection of sets of paths
$!D$	All paths from $\mathsf{Source}(D)$ to $\mathsf{Target}(D)$ not satisfying D

For a traversal specification to be meaningful, it has to be well-formed. Formally, well-formedeness is defined in terms of two functions, Source and Target, which both map a specification to a node. The following chart shows the denitions for Source and Target where $\mathsf{Source}(D)$ is the source node determined by D, and $\mathsf{Target}(D)$ is the target node determined by D:

Selector: D	$\mathsf{Source}(D)$	$\mathsf{Target}(D)$
$[A, B]$	A	B
$D_1 \cdot D_2$	$\mathsf{Source}(D_1)$	$\mathsf{Target}(D_2)$
$D_1 \texttt{ \| } D_2$	$\mathsf{Source}(D_1)$	$\mathsf{Target}(D_1)$
$D_1 \texttt{ \& } D_2$	$\mathsf{Source}(D_1)$	$\mathsf{Target}(D_1)$
$!D$	$\mathsf{Source}(D)$	$\mathsf{Target}(D)$

A traversal specication is well-formed if it determines a source node and a target node, if each concatenation has a meeting point, and if each union of a set of paths preserves the source and the target. This is expressed by the predicate WF:

$$\begin{aligned} \mathsf{WF}([A, B]) &= \text{true} \\ \mathsf{WF}(D_1 \cdot D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\ &\quad\ \mathsf{Target}(D_1) =_{\mathsf{nodes}} \mathsf{Source}(D_2) \\ \mathsf{WF}(D_1 \texttt{ | } D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\ &\quad\ \mathsf{Source}(D_1) =_{\mathsf{nodes}} \mathsf{Source}(D_2) \wedge \\ &\quad\ \mathsf{Target}(D_1) =_{\mathsf{nodes}} \mathsf{Target}(D_2) \\ \mathsf{WF}(D_1 \texttt{ \& } D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\ &\quad\ \mathsf{Source}(D_1) =_{\mathsf{nodes}} \mathsf{Source}(D_2) \wedge \\ &\quad\ \mathsf{Target}(D_1) =_{\mathsf{nodes}} \mathsf{Target}(D_2) \\ \mathsf{WF}(!D) &= \mathsf{WF}(D) \end{aligned}$$

If D is well-formed and compatible with Φ, then $\mathsf{PathSet}_\Phi(D)$ is a set of paths in Φ from the source of D to the target of D, defined as follows:

$$\begin{aligned}
\mathsf{PathSet}_\Phi([A,B]) &= \mathsf{Paths}_\Phi(A,B)\\
\mathsf{PathSet}_\Phi(D_1 \cdot D_2) &= \mathsf{PathSet}_\Phi(D_1)\cdot\mathsf{PathSet}_\Phi(D_2)\\
\mathsf{PathSet}_\Phi(D_1 \mid D_2) &= \mathsf{PathSet}_\Phi(D_1)\cup\mathsf{PathSet}_\Phi(D_2)\\
\mathsf{PathSet}_\Phi(D_1 \mathbin{\&} D_2) &= \mathsf{PathSet}_\Phi(D_1)\cap\mathsf{PathSet}_\Phi(D_2)\\
\mathsf{PathSet}_\Phi(!D) &= \mathsf{Paths}_\Phi(\mathsf{Source}(D),\mathsf{Target}(D))\\
&\quad -\mathsf{PathSet}_\Phi(D)
\end{aligned}$$

We show a reduction from SD to SAJ. In the following, SD expressions are on the left-hand side and SAJ expressions are on the right:

$$\begin{aligned}
T([A,B]) &\to \texttt{flow}(A)\ \texttt{\&}\ B\\
T(D_1\cdot D_2) &\to \texttt{flow}(T(D_1))\ \texttt{\&}\ T(D_2)\\
T(D_1 \mid D_2) &\to T(D_1) \mid T(D_2)\\
T(D_1 \mathbin{\&} D_2) &\to T(D_1)\ \texttt{\&}\ T(D_2)\\
T(!D) &\to\ !T(D)
\end{aligned}$$

Here is an example reduction of $[A,B]\cdot[B,C]$:

$$\begin{aligned}
T([A,B]\cdot[B,C]) &\to \texttt{flow(flow}(A)\ \texttt{\&}\ B)\ \texttt{\&}\ \texttt{flow}(B)\ \texttt{\&}\ C\\
&= \texttt{flow(flow}(A)\ \texttt{\&}\ B)\ \texttt{\&}\ C\\
&= \texttt{flow}(A)\ \texttt{\&}\ \texttt{flow}(B)\ \texttt{\&}\ C
\end{aligned}$$

We also show an informal[3] reduction from SAJ to an SD expression D. In the following, SAJ expressions are on the left-hand side, and SD expressions are on the right:

$$\begin{aligned}
T(n(l)) &\to [\mathsf{Source}(D),l]\\
T(\texttt{flow}(l)) &\to [\mathsf{Source}(D),l]\cdot[l,\mathsf{Target}(D)]\\
T(S_1 \mid S_2) &\to T(S_1) \mid T(S_2)\\
T(S_1 \mathbin{\&} S_2) &\to T(S_1)\ \texttt{\&}\ T(S_2)\\
T(!S) &\to\ !T(S)
\end{aligned}$$

3. PROBLEMS

In the following section we present various problems related to selector expressions and reason about their complexity. Theorems are presented in tables of the form:

	SD	SAJ
-	R_1	R_2
&	R_3	R_4
!	R_5	R_6

Each R_i is a complexity result. The first row represents complexity results for the languages shown in grammars (1) and (2) without intersection or negation, called the base language; the second row shows results for these languages without negation; and the third row shows results for these languages without intersection. A Y in a result represents a problem that is trivially true. All proofs are in [12].

[3]This is informal because a resultant in SD could have multiple targets.

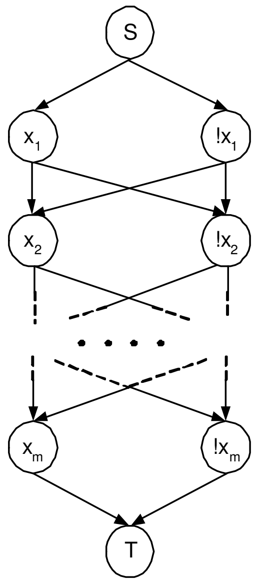


Figure 1: Ladder graph.

Problem	SD	SAJ
-	P	P
&	NP-complete	NP-complete
!	NP-complete	NP-complete

Table 1: Complexity results for many problems.

We split this section according to general problems – e.g. Select-Sat. We refer to particular instances of these problems for certain languages by the form $A/B/C$ where A is a general problem or $*$ for all problems, C is the language SD or SAJ, B is one of $-$, &, or ! representing the version of language C. For example, Select-Sat/&/SAJ represents the Select-Sat problem over the SAJ language with intersection, and */-/SD represents any problem on the base language, -, over the SD language.

We use a *ladder graph*, as shown in Figure 1, as our main tool to represent boolean forumulas. This graph consists of a root s, target t, and nodes x_i and $!x_i$ for $i = 1$ to m. A path from s to t must go through only one x_i or $!x_i$ for all i to reflect the fact that each literal in a boolean formula may be assigned either true or false; but not both. In addition, we use the following generic constructions.

Many of the problems have similar complexity results, which are given in Table 1.

3.0.1 SD Generic Construction

For the */-/SD case, we turn the selector into a graph p' ($[A,B]$ becomes an edge from A to B.) Then we construct the cross product traversal graph $T(G,p')$ [13, 11].

The motivation for the cross product $T(G,p')$ is as follows: Implementing the strategy $S=[A,B]$ on a class graph G [14] is straight-forward (called the FROM-TO computation): In G we do a forward depth-first traversal from A and a backward depth-first traversal from B and take the intersection of the two. The resulting graph succinctly represents the desired path set. For a general strategy we want to reduce the problem of succinctly representing the path set to the FROM-TO problem and this reduction is achieved by replacing the class graph with a much larger graph and doing the FROM-TO computation in that graph. This much larger graph is precisely the cross product of the strategy and the class graph.

3.0.2 SAJ Generic Construction

We need a generic construction for the */-/SAJ case. We use the */-/SD case as a guide. In the */-/SD case we flag each edge selected by a primitive $\texttt{flow}(A \cdot B)$ with $A \cdot B$. This is basically the idea behind the traversal graph construction. We need this labeling to avoid information loss (i.e. the short-cuts and zigzags of Palsberg *et al.*, [21]). We use a similar approach for */-/SAJ. The edges selected by each primitive $\texttt{flow}(A)$ are labeled by $\texttt{flow}(A)$. We can reduce the SAJ expression to the form $s_1 \mid \cdots \mid s_k$ for $1 \leq k$, where each s_i is in the form of either $n(l)$ or $\texttt{flow}(s')$ because

$$\texttt{flow}(n(A_1) \mid \texttt{flow}(n(A_2))) = \texttt{flow}(n(A_1)) \mid \texttt{flow}(n(A2)).$$

Therefore we can build in polynomial time a structure, called the flow graph, that plays the same role as the traversal graph. The size of the flow graph is bounded by the size of the meta graph times the number of flow expressions in the selector (after removal of nested flows).

We use this construction for */-/SAJ where * in Select-Never (is the node ever in the flow graph?), Select-Sat (is the flow graph empty?), Select-Impl (is one flow graph a subgraph of another flow graph?) and Select-First (which edges are in the flow graph?).

In our NP-completness proofs we leave out the part that shows that a problem is in NP and we focus on the harder NP-hard part. We leave the NP membership part as an exercise to the reader.

3.1 Select-Sat

We are presenting a proof sketch of one of our complexity-theoretic results as an example of the kinds of gadgets we use in our reductions.

DEFINITION 3 (SELECT-SAT). *Given a selector p and a meta graph G, is there an instance tree for G for which p selects a non-empty set of nodes.*

Table 1 shows the complexity results for Select-Sat. The Select-Sat/-/SD problem has been implemented for a special case in Demeter/C++ and for the general case in DemeterJ, DJ and DAJ. Our users demanded such a test because knowing that a traversal specification (selector) will never select a node indicates, usually, a false assumption about the class graph (meta graph). Select-Sat/*/SAJ is not currently implemented in AspectJ, and this can make it harder to debug pointcut designators. A small typo in one of the pointcuts may empty the set of selected join points. It would be helpful to get a warning for the pointcuts that select an empty set of join points. We hope that our FPT algorithms in Section 4 will lead to interesting algorithms for the NP-complete cases for AspectJ and for Demeter.

SAJ Expression			Pointcut
p_1	$=$	$x1 \mid !x2 \mid x3$	`p1()`
p_2	$=$	$!x1 \mid x2$	`p2()`
p_3	$=$	$x1$	`p3()`
p_4	$=$	$!x3$	`p4()`
p_{all}	$=$	p_1 & p_2 & p_3 & p_4	`all()`

Table 2: SAJ expressions and AspectJ pointcuts.

PROOF *Select-Sat/&/SD.* The proof is by reduction from 3-SAT. Consider a 3-SAT formula ϕ. Let $v_1, v_2, \ldots, v_n$ be the variables. Create a meta graph that is a dag as follows: a source s with arcs going to x_1 and $!x_1$, arcs from x_i and $!x_i$ to x_{i+1} and $!x_{i+1}$ and finally from x_n and $!x_n$ to a sink t. This is $G(\phi)$, called a ladder graph, as shown in Figure 1. Now create an atomic selector for each literal and create the total selector $S(\phi)$ by taking the union and intersection over literals for each clause. For a literal $li = v_i/!v_i$ create the selector "from s to t via li" – i.e. "$[s, v_i] \cdot [v_i, t]$". Clearly, $(S(\phi), G(\phi))$ is satisfiable iff ϕ is satisfiable. □

Our reduction constructs a meta graph and a selector from the Boolean formula. But our meta graph is really an abstraction of a Java program and the selector an abstraction of an AspectJ pointcut designator. An important point of our paper is that the meta graph/selector abstraction is good enough to reason about the computational complexity at the AspectJ level. To demonstrate this point, we translate an example boolean formula shown in Table 2 directly to an AspectJ pointcut in Figure 2. Here, `x1`, `x2`, `x3`, `nx1`, `nx2`, and `nx3` in Figure 2 correspond to x_1, x_2, x_3, $!x_1$, $!x_2$, and $!x_3$, respectively.

3.2 Select-Sat-Static

DEFINITION 4 (SELECT-SAT-STATIC). *Given a selector p, is there a meta graph G and an instance tree for G for which p selects a non-empty set of nodes.*

A Select-Sat-Static test is a must for a "perfect" aspect-oriented system, because a selector that fails for all meta graphs is clearly useless. Yet, both AspectJ and the Demeter Tools don't implement such a test, maybe, because it is perceived to be unlikely that a user writes such pointcuts or traversal strategies. Again, we hope that our FPT ideas in Section 4 will help to develop practically useful algorithms.

The following are the complexities for Select-Sat-Static:

Select-Sat-Static	SD	SAJ
-	Y	Y
&	Y	Y
!	NP-complete	NP-complete

We mention also that the following problem is NP-complete for both SAJ and SD (even without complement) if we allow that an instance may be a directed acyclic graph (dag), not

```
public class Example {
 public static void main(String[] s) {x1(); nx1();}
 static void  x1() { x2(); nx2(); }
 static void  x2() { x3(); nx3(); }
 static void  x3() {    target(); }
 static void nx1() { x2(); nx2(); }
 static void nx2() { x3(); nx3(); }
 static void nx3() {    target(); }
 static void target() {}
}
aspect Aspect {
 pointcut p1():  cflow(call (void  x1()))
   ||            cflow(call (void nx2()))
   ||            cflow(call (void  x3()));
 pointcut p2() : cflow(call (void nx1()))
   ||            cflow(call (void  x2()));
 pointcut p3() : cflow(call (void  x1()));
 pointcut p4() : cflow(call (void nx3()));
 pointcut all(): p1() && p2() && p3() && p4();
 before(): all() && !within(Aspect) {
   System.out.println(thisJoinPoint);
 }
}
```

Figure 2: AspectJ example.

just a tree. Since a tree is a dag, we restrict our definition of the problem to trees.

DEFINITION 5 (SELECT-SAT-DYNAMIC). *Given a selector p, a meta graph G, and an instance tree I for G, does p select a non-empty set of nodes in I?*

3.3 Select-Impl

DEFINITION 6 (SEL). SEL(s, G, I) *is the set of nodes selected by s in I (which conforms to G).*

DEFINITION 7 (SELECT-IMPL). *Given two selector expressions s_1 and s_2 and a graph G, for all instances I of G: SEL(s_1,G,I) is a subset of SEL(s_2,G,I).*

Predicate-dispatch-based aspect languages such as Socrates [20] use selector implication as a primitive to generalize inheritance. Selector implication is also useful in other applications. For example, a security policy might state that a set of nodes accessible by one role (e.g., worker) must always be a subset of the set of nodes accessible by another role (e.g., manager). Table 1 shows the complexity results for !Select-Impl– hence in this table all NP-complete results are co-NP-complete results for Select-Impl.

3.4 Select-First

DEFINITION 8 (SELECT-FIRST). *Given a selector p, a meta graph G, and an instance I, compute the set of outgoing edges from a node of I satisfying G that might lead to a target node selected by p.*

In the Demeter case the Select-First predicate is the fundamental tool to implement traversals efficiently. The approach is to combine the selector and meta graph into a new graph that for each node tells which outgoing edges are worthwhile traversing. Worthwhile means that it may lead to a target node satisfying p in an appropriate subobject. See [15] for the generalization of this predicate to class graphs with is-a and has-a edges. [21] contains an efficient implementation for a special case that was used in Demeter/C++. The D*J tools use the AP Library [13] that implements Select-First/-/SD using the ideas in [11].

The NP-completeness result for Select-Sat/&/SD has interesting implications for the semantics of traversals as we make the selector language more expressive. The DAJ tool [2] is an extension of AspectJ with traversals and strategies. Using the AspectJ declare construct we could write:

```
declare strategy: sname: "{A -> B}" ;
declare traversal: void foo(): sname(Visitor);
```

In this DAJ example the expression `"A -> B"` is analogous to the SD expression $[A, B]$ This selector expression uses SD without negation but with intersection. This traversal defines an adaptive method called `foo` using the strategy named `sname` and the `Visitor`, which is a normal Java class. In DAJ intersection is used frequently because it also plays the role of cleaning the class graph from unwanted information.

The semantics of a traversal is defined in terms of Select-First [11, 15]. This works well for SD without intersection and complement because we have an efficient algorithm. In the presence of intersection, we currently implement the following solution: We assume that intersection only appears at the outermost level. This is a reasonable assumption. To implement (s_1 & s_2), compute the traversal graph t_1 for s_1 and G and the traversal graph t_2 for s_2 and G. Then we simulate both t_1 and t_2 on an instance graph. But unfortunately this gives the wrong semantics because we might go down an edge in the instance graph although it never leads to a target. Instead we need to construct the cross product of t_1 and t_2, leading to an explosion in the number of nodes if we do this multiple times. We know now that there is no way around this because of the NP-completeness of the underlying problem.

For the AspectJ case the predicate is useful to implement `cflow`. It tells us along which execution paths we are in the scope of a pointcut designator where we have to execute advice. Table 1 shows the complexity results for Select-First.

Consider a selector expression p and a meta graph G in Select-Sat/&/SAJ. Let's assume that we can compile p and G into a function $\mathsf{Super}(r)$ that given a node r of an instance conforming to G, computes the set of outgoing edges from r that may lead to a selected node. The function Super encodes the information about p and G into a form that is useful for deciding which edges are worthwhile to traverse to reach a target node.

Let's assume that we can construct Super in polynomial-time and that Super runs in polynomial-time. This would create a polynomial algorithm for Select-Sat/&/SAJ. Namely, we compile the pair (p, G) into $\mathsf{Super}(r)$ and run $\mathsf{Super}(r)$ on an instance I of G that has the root and an edge to each of the successors of the root. Note that for each meta graph G we can generically construct such an instance. Clearly, the size of r is bounded by the size of G. The input (p, G) is satisfiable iff $\mathsf{Super}(r)$ returns a non-empty set on I; i.e., there is an instance graph in which at least one node is selected.

Note that, the same argument holds for: Select-Sat/&/SD. In order to prove that (p, G) is unsatisfiable (co-NP-complete problem) we need only run Super on a generically constructed instance.

As soon as the selector language becomes too powerful, selecting nodes in instances becomes expensive. We can use this to prove that Select-First/&/SAJ and Select-First/&/SD are NP-complete.

3.5 Select-Always

DEFINITION 9 (SELECT-ALWAYS). *Given a selector p and a meta graph G and a node n in G, for all instance graphs I of G all of the instances of n in I are selected by p.*

If an AspectJ or Demeter compiler could answer this question efficiently we could drastically speed up compilation time. Table 1 shows the complexity results for !Select-Always.

3.6 Select-Never

DEFINITION 10 (SELECT-NEVER). *Given a selector p and a meta graph G and a node n in G, for all instance graphs I of G none of the instances of n in I are selected by p.*

In addition to the benefits found from Select-Always, efficient solutions to this problem could provide useful feedback to users when writing pointcuts or traversals. Often one writes a pointcut and then refactors a system. The user would want to know when her pointcuts were possibly no longer valid after this refactoring. This is just one example of why this is an important problem. Table 1 shows the complexity results for !Select-Never.

4. FPT ALGORITHMS

We have shown that Select-Sat is NP-complete. As noted in [6] the fact that a problem has been shown to be NP-hard is not a cause for despair. All it really means is that the initial hope for an exact general algorithm is in vain. There are a few different avenues of attack at this point - the use of randomness, the search for good approximate solutions and use of parametrization. Here we focus on this last approach. This is work in progress and it is not yet practically useful because our algorithm works only on a very special kind of graph structure.

We look more closely at the structure of the input. Select-SAT consists of a meta graph and a selector. We have shown this problem to be NP-hard even when the meta graph is the ladder graph and the selector is a 3-SAT formula. In practice though, it is often the case that the selector rarely has too many clauses. In particular we consider situations where our meta graph is a generalization of the ladder graph and the conjunctive selector formula has only k clauses. We ask the question - what is the behavior for a fixed k? Observe that the naive approach of trying every possible setting of the variables in the selector leads to an exponential-time (2^n) algorithm. We now demonstrate that in fact for fixed k, this problem, which we call the k-generalized-ladder-Select-Sat, is solvable in time that is linear in the size of the formula and the graph.

The approach of parametrization has been developed by Downey and Fellows in a seminal series of papers [3]. They show that the usual combinatorial explosion involved in NP-hard problems can often be handled if one can get one's hands on the right parametrization. In cases where such a parametrization exists, the problem is said to be *Fixed Parameter Tractable.* More precisely, a parametrized problem $< x, k >$, where x is the input and k the parameter, is said to be in FPT if there exists an algorithm and a constant c (independent of k), and a function f such that the algorithm accepts valid inputs in time $f(k)|x|^c$. Note for example that Vertex Cover is in FPT where k, the size of the cover, is fixed. On the other hand Independent Set with k representing the size of the independent set continues to be intractable even when k is fixed.

We now define the problem k-generalized-ladder-Select-Sat and present a fast kernelization scheme to solve it.

DEFINITION 11. *k-generalized-ladder-Select-Sat consists of a generalized ladder graph and a selector formula in conjunctive normal form. The generalized ladder graph is a directed acyclic leveled graph that has a unique source s and unique sink t. The graph contains all edges between adjacent levels. At each level the graph has no more than $f_i(k)$ vertices, where i represents the level. See Figure 3. The selector formula is in CNF and has at most k clauses.*

Note that our earlier NP-hardness proof goes through for k-generalized-ladder-Select-SAT when k is considered to vary with n, instead of being fixed.

THEOREM 1. *k-generalized-ladder-Select-Sat is in FPT.*

PROOF. At a high level our strategy is to find in time polynomial in n, a kernel or the hard core of the problem which only depends on k and not on n; and then we employ a search tree strategy to try all possible cases in the kernel. Let $f_{\max} = \max_i f_i(k)$ denote the maximum number of vertices over all rows of the generalized ladder graph.

Kernelization. Consider the selector formula. Each literal is of the form v where v is a vertex in the associated generalized ladder graph and selects the set of paths from s to t going through that vertex v. If the formula has any single literal clauses then since all paths from s to t satisfying the formula must pass through that vertex we can prune the metagraph by removing all vertices other than v from its level. Note that in this manner we account for all single literal clauses or the metagraph gets pruned into the empty graph in which case we know that the selector formula is unsatisfiable. We

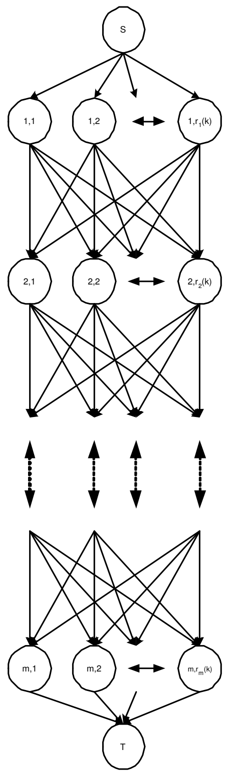


Figure 3: General ladder graph.

are now left to consider the case where we have taken care of all single literal clauses, i.e. we can assume that the formula only consists of clauses with 2 or more literals. Consider any clause with more than $k * f_{\max}$ literals. Observe, that to satisfy each of the remaining (upto) k clauses we need to only satisfy 1 literal in each clause. Since the clause in consideration has more than $k * f_{\max}$ literals that means this clause contains a literal that is on a level of the meta graph different from that of any other vertex needed for satisfying any of the other clauses. Hence such a clause can be trivially satisfied. Thus we can eliminate all clauses with more than $k * f_{\max}$ literals. Thus we are left with a formula with at most k clauses where each clause has between 2 and $k * f_{\max}$ literals.

Search tree. Now try setting to true all possible choices of literals, one from each clause, there are at most $k^{k*f_{\max}}$ possible choices and for each possible choice compute the subgraph of the meta graph that satisfies that choice. If all subgraphs are empty then we know that the selector is unsatisfiable. If some subgraph is nonempty then consider the clauses that were pruned for having more than $k * f_{\max}$ literals and pick a literal in each of these clauses on a level different from all the previously chosen literals and prune this subgraph so as to satisfy these clauses.

It is easy to see that the above scheme has running time $O(n) + O(k^{k*f_{\max}})$ and hence k-generalized-ladder-Select-Sat is in FPT. □

5. RELATED WORK

[16] is an interesting study of crosscutting mechanisms. They discuss both the WhereToInfluence-part and the WhatToDo-part while we focus on the WhereToInfluence-part only. But in their Table 1 they also put pointcuts and traversal specifications at the same level as we do in this paper. (Demeter actually uses another incarnation of AOP which is not discussed in either paper: The visitor signatures are pointcuts and the visitor method bodies are the advice.) The crosscut definition in [16] can be applied to selector languages: Two selectors p_1 and p_2 crosscut if the set of selected nodes intersect at the instance level or meta graph level but none is a subset of the other. Crosscutting of selector expressions is very typical especially if we consider the nodes along the paths as well (not just the target nodes).

The two papers differ in that we focus on algorithms and complexity results of selector languages.

In [17], the issue of unnecessary run-time checks in AspectJ is discussed. The meta graph is considered to be included in the program text. They use partial evaluation to remove unnecessary pointcut tests. They don't analyze the complexity of the underlying task but instead use a powerful, but potentially expensive tool, to attack the problem. We show that general elimination of run-time tests (Select-Never and Select-Always) is NP-complete in the general case.

In Eichberg et al. [5] they use functional queries as their selector language. This is an interesting generalization of the kind of selector languages discussed in this paper. It would be useful to analyze the combinatorial problems discussed in this paper for a simple functional query language as selector language. Eichberg et al. use XQuery (based on XPath) as the query language which supports the descendent axis (denoted by "//") that can express traversal like $[A, B]$ (from A to B) in our SD selector language.

The study of selector languages is an active topic in the database community over the past few years. Schwentick [22] does an extensive study of the equivalent of the Select-Impl problem for XPath and show it to be co-NP-complete for a particular subset of XPath. In a paper by Neven and Schwentick it is shown: Theorem 7. Containment of XP(DTD, /, //, *)-expressions is in P. This problem matches with our Select-Impl/-/SD which we also have shown to be in P [13]. DTD's correspond to our meta graphs. The difference with our work is that XPath slices the selector language world in a way that is different from AspectJ pointcuts (SAJ) or Demeter traversals (SD). Our paper also differs in that we provide a unifying model to study key properties of a wide variety of selector languages.

Sereni and de Moor [23] study the static determination of `cflow` pointcuts in AspectJ. They reason also in terms of sets of paths, but they use a regular expression style selector language. They model pointcut designators as automata which is similar to our translation of selectors into graphs.

They do whole program analysis on the program's call graph and try to determine whether a potential join point fits into one of the following three cases: (1) it *always* matches a `cflow` pointcut; (2) it *never* matches a `cflow` pointcut; (3) it *maybe* matches a `cflow` pointcut. In case (3), there is still a need to have dynamic matching code. They didn't analyze the computational complexity of (1, Select-Always) and (2, Select-Never). Our NP-completeness results for Select-Always and Select-Never complement their practical analysis.

In [4] an AspectJ compiler, called abc, is discussed and they found several improvements to implementing `cflow` over the AspectJ compiler ajc. Our work assumes a whole program analysis but should provide useful input to compiler writers. Using traversal graphs for compiling certain AspectJ programs should lead to even more speed-ups.

Mendelzon and Wood [18] analyzed the complexity of finding regular paths in graphs, which is similiar to our Select-First and Select-Sat problems with subtle differences. They showed that finding simple regular paths in a graph is NP-complete problem while finding regular paths is a polynomial-time problem (if the regular expression language is not too rich). Their selector language is a regular expression language that could be studied in a similar way we have sudied SAJ and SD. Mendelzon and Wood don't consider instance graphs: they operate at the level of selectors (regular expressions) and meta graphs only.

The work on JAsCo [24, 25] is using a pointcut-style notation and Demeter-style traversal specifications in the same system. The selector language approach described in this paper might lead to a tighter integration of the two languages.

Gybels and Brichau [7] present a number of language features that could be useful for expressing more expressive pattern-based crosscuts. The language presented is pattern-based, similar to that found in AspectJ [9], uses Prolog, and is implemented on SmallTalk. It first the adds unification as a feature, which allows variable binding. Another feature are object reifying predicates that (1) provide access to the "context object" property of the matched join point, (2) provide direct access to the state of objects, and (3) can express the way a certain object should respond to messages.

Lastly, join point shadows are used to access static properties of the program, and recursion is allowed in defintions. The latter makes this language Turing complete.

Walker presents the concept of *Implicit Context* in his dissertation [26]. Implicit context consists of three concepts: boundaries between conflicting world views, contextual dispatch which is used to alter communications, and communication history which is used to retrieve previous state when performing contextual dispatch. This allows a programmer to express the essential structure of our software modules, through the use of implicit context, to make those modules easier to reuse and the systems containing those modules easier to evolve. Expressing these context requires expressive languages which could benefit from our work.

Several papers use regular expressions as selector language [23] and [10]. Several of our results should carry over to regular expressions but the details need to be worked out in future work.

6. CONCLUSIONS AND FUTURE WORK

We have studied graph-theoretic decision problems fundamental to aspect-oriented software development. We have simplified our model by considering only meta graphs and instance graphs with has-a edges. But it is not hard to generalize our algorithms and proofs to more general meta graphs as has been done in [11] for Select-First/-/SD.

The simplified model promotes a succinct description of both upper and lower bounds for a variety of relevant problems. In doing so we have made contributions to complexity theory – a new FPT algorithm for a subset of Select-Sat and new NP-completeness proofs – and PL theory – two models of selector languages and a collection of related algorithms useful in AOSD tools (compilers, IDEs) that assume the whole world assumption.

The NP-completeness results are useful for three reasons: (1) The NP-completeness of the monotone version of the Satisfiability problem for the AspectJ pointcut language (Select-Sat/&/SAJ) is surprising because Satisfiability for monotone boolean formulas can be solved in polynomial-time.(2). They help us to steer around language features that might be expensive to implement. (3) In case we need the NP-complete language features, we can think carefully about what kind of algorithms degrade gracefully if certain features of the input are bounded. This is the topic of FPT.

Many of the efficient algorithms we describe are practically useful, and have not been described in the literature so far. We have implemented algorithms for several of the */-/SD problems in D*J and they are distributed separately through the AP Library. Select-First/-/SD is used heavily in the D*J tools whenever an object is traversed. An empirical study of traversals is in [28].

This is just the beginning in reasoning about the relationship between different pointcut languages and learning how to utilize different languages' features in an efficient manner. For example, a common AspectJ idiom is to capture a call only in certain contexts; say a call to `f()` but not underneath a call to `g()`. This is written in AspectJ as

```
call(void f()) & !cflow(void g())
```

We can use our results from this paper to see that reasoning about this statement uses an NP-complete sublanguage. However, we can write an equivalent Demeter traversal as

```
from main() bypassing g() to f()
```

that uses a polynomial-time sublanguage. So, we will use this framework to unify multiple pointcut languages in an intelligent manner.

In future work we want to study incremental versions of the problems which are important for incremental compilation. We also want to focus on studying *shy* selector languages. Both SAJ and SD are shy selector languages but they can be improved and maybe integrated. A "control-flow-shy" selector language is discussed in [5]. In addition to minimizing information from the class graph, we want to minimize information from the control-flow graph in the selectors.

7. REFERENCES

[1] B. Courcelle. Graph rewriting: An algebraic and logical approach. In J. van Leeuwen et al, editor, *Handbook of Theoretical computer Science, Vol B*. North Holland, 1990.

[2] Doug Orleans and Karl J. Lieberherr. DAJ: Demeter in AspectJ home page. http://www.ccs.neu.edu/research/demeter/DAJ/.

[3] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[4] B. Dufour, C. Goard, L. Hendren, C. V. erbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of aspectj programs. In D. Schmidt, editor, *OOPSLA*, Vancouver, CA, 2004. ACM Press.

[5] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *The Second ASIAN Symposium on Programming Languages and Systems ASPLAS*, 2004.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[7] 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.

[8] F. Harary. *Graph Theory*. Addison Wesley, 1994.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In J. Knudsen, editor, *ECOOP*, Budapest, 2001. Springer Verlag.

[10] S. Krisnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *FSE*, 2004.

[11] K. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *TOPLAS*, 26(2):370–412, 2004.

[12] K. J. Lieberherr, J. Palm, and R. Sundaram. Expressiveness and complexity of crosscut languages. Technical Report NU-CCIS-04-10, Northeastern University, September 2004.

[13] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997.

[14] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997.

[15] K. J. Lieberherr and M. Wand. Traversal semantics in object graphs. Technical Report NU-CCS-2001-05, Northeastern University, May 2001.

[16] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *ECOOP*, June 2003.

[17] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In R. Cytron and G. Leavens, editors, *FOAL*, Enschede, Netherlands, 2002.

[18] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. In *VLDB*, 1989.

[19] D. Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, The Netherlands, April 2002.

[20] D. Orleans. The Socrates Programming Language, September 2004. http://socrates-lang.sf.net/.

[21] J. Palsberg, C. Xiao, and K. J. Lieberherr. Efficient implementation of adaptive software. *TOPLAS*, 17(2):264–292, Mar. 1995.

[22] T. Schwentick. Xpath query containment. *SIGMOD Rec.*, 33(1):101–109, 2004.

[23] D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 30–39. ACM Press, 2003.

[24] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.

[25] W. Vanderperren. *Combining Aspect-Oriented and Component-Based Software Engineering*. PhD thesis, Vrije Universiteit Brussel, 2004.

[26] R. Walker. Essential software structure through implicit context. *Ph.D. dissertation, The University of British Columbia*, March 2003.

[27] P. Wu and K. J. Lieberherr. Compilation of Pointcut Designators using Traversals. Technical Report NU-CCIS-03-16, Northeastern University, December 2003.

[28] P. Wu and M. Wand. An Empirical Study of the Demeter System. In *Proceedings of the SPLAT workshop of the 3rd international conference on Aspect-Oriented Software Development*, 2004.