in FOAL 2002 Proceedings: Foundations Of Aspect-Oriented Languages, Workshop at AOSD 2002, pp.17-25, 2002

Compilation Semantics of Aspect-Oriented Programs

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

Gregor Kiczales
Department of Computer Science
University of British Columbia
gregor@cs.ubc.ca

Chris Dutchyn
Department of Computer Science
University of British Columbia
cdutchyn@cs.ubc.ca

ABSTRACT

This paper presents a semantics-based compilation framework for an aspect-oriented programming language based on its operational semantics model. Using partial evaluation, the framework can explain several issues in compilation processes, including how to find places in program text to insert aspect code and how to remove unnecessary run-time checks. It also illustrates optimization of calling-context sensitive pointcuts (`cflow`), implemented in real compilers.

Keywords

Aspect SandBox, dynamic join point model, partial evaluation, Futamura projection, compile-time weaving, context-sensitive pointcut designators (`cflow`)

1. INTRODUCTION

This work is part of a larger project, the Aspect SandBox (ASB), that aims to provide concise models of aspect-oriented programming (AOP) for theoretical studies and to provide a tool for prototyping alternative AOP semantics and implementation techniques. To avoid difficulties to develop formal semantics directly from artifacts as complex as AspectJ and Hyper/J, ASB provides a suite of interpreters of simplified languages. Those languages have sufficient features to characterize existing AOP languages. In this paper we report one result from the ASB project—a semantics-based explanation of the compilation strategy for advice dispatch in AspectJ like languages[6, 7, 11, 12].

The idea is to use partial evaluation to perform as many tests as possible at compile-time, and to insert applicable advice bodies directly into the program. Our semantic framework also explains the optimization used by the AspectJ compiler for context-sensitive pointcuts (`cflow` and `cflowbelow`).

Some of the issues our semantic framework clarifies include:

- The mapping between dynamic join points and the points in the program text, or *join point shadows*, where the compiler actually operates.
- What dispatch can be 'compiled-out' and what must be done at runtime.
- The performance impact different kinds of advice and pointcuts can have on a program.
- How the compiler must handle recursive application of advice.

1.1 Join Point Models

Aspect-oriented programming (AOP) is a paradigm to modularize crosscutting concerns[13]. An AOP program is effectively written in multiple modularities—concerns that are local in one are diffuse in another and vice-versa. Thus far, several AOP languages are proposed from practical to experimental levels[3, 11, 12, 16, 17].

The ability of an AOP language to support crosscutting lies in its *join point model* (JPM). A JPM consists of three elements:

- The *join points* are the points of reference that aspect programs can use to refer to the computation of the whole program. *Lexical* join points are locations in the program text (*e.g.,* "the body of a method"). *Dynamic* join points are run-time entities, such as events that take place during execution of the program (*e.g.,* "an invocation of a method").
- A *means of identifying* join points. (*e.g.,* "the bodies of methods in a particular class," or "all invocations of a particular method")
- A *means of specifying semantics* at join points. (*e.g.,* "run this code beforehand")

As an example, in AspectJ:

*This work is carried out during his visit to University of British Columbia.

- the join points are nodes in the runtime control flow graph of the program, such as when a method is called (and returns), and when a field is read (and the value is returned). (*e.g.,* "a call to method `set(int)` of class `Point`"[1])

- the means of identifying join points is the *pointcut* mechanism, which can pick out join points based on things like the name of the method, the package, the caller, and so forth. (*e.g.,* "`call(void Point.set(int))`")

- the means of specifying semantics is the *advice* mechanism, which makes it possible to specify additional code that should run at join points.
(*e.g.,* "`before : call(void Point.set(int))`
`{ Log.add("set"); }`")

In this paper, we will be working with a simplified JPM similar to the one from AspectJ. (See Section 2.1 for details.)

The rest of the paper is organized as follows. Section 2 introduces our AOP language, AJD, and shows its interpreter. Section 3 presents a compilation framework for AJD excluding context-sensitive pointcuts, which are deferred to Section 4. Section 5 relates our study to other formal studies in AOP and other compilation frameworks. Section 6 concludes the paper with future directions.

2. AJD: DYNAMIC JOIN POINT MODEL AOP LANGUAGE

This section introduces our small AOP language, AJD, which implements core features of AspectJ's dynamic join point model. The language consists of a simple object-oriented language and its AOP extension. Its operational semantics is given as an interpreter written in Scheme. A formalization of a procedural subset of AJD is presented by Wand and the second and the third authors[20].

2.1 Informal Semantics

We first informally present the semantics of AJD. In short, AJD is an AOP language based on a simple object-oriented language with classes, objects, instance variables, and methods. Its AOP features covers essential part of AspectJ (version 1.0).

2.1.1 Object Semantics

Figure 1 is an example program. For readability, we use a Java-like syntax in the paper[2]. It defines a `Point` class with one integer instance variable `x`, a unary constructor, and three methods `set`, `move` and `main`.

When method `main` of a `Point` object is executed, line 7 creates another `Point` object and runs the constructor defined at line 3. Line 8 invokes method `move` on the created object. Finally, line 9 reads and displays the value of variable `x` of the object.

[1]For simplicity later in the paper, we are using one-dimensional points as an example.

[2]Our implementation actually uses an S-expression based syntax.

```
1  class Point {
2    int x;
3    Point(int ix)      { this.set(ix); }
4    void set(int newx) { this.x = newx; }
5    void move(int dx)  { this.set(this.x + dx); }
6    void main() {
7      Point p = new Point(1);
8      p.move(5);
9      write(p.x); newline();
10   }
11 }
```

Figure 1: An Example Program. (`write` and `newline` are primitive operators.)

$p \in \{\text{pointcuts}\}$, $m \in \{\text{method signatures}\}$,
$n \in \{\text{constructor signatures}\}$, $v \in \{\text{identifiers with types}\}$

$$
\begin{array}{rcl}
p & ::= & \texttt{call}(m) \mid \texttt{execution}(m) \mid \texttt{new}(n) \\
 & \mid & \texttt{target}(v) \mid \texttt{args}(v, \ldots) \mid p\texttt{\&\&}p \mid p\texttt{||}p \mid \texttt{!}p \\
 & \mid & \texttt{cflow}(p) \mid \texttt{cflowbelow}(p)
\end{array}
$$

Figure 2: Syntax of Pointcuts.

2.1.2 Aspect Semantics

To explain the semantics of AOP features in AJD, we first define its JPM.

2.1.2.1 Join Point

The *join point* is an action during program execution, including method calls, method execution, object creation, and advice execution. (Note that a method invocation is treated as a call join point at the caller's side and an execution join point at the receiver's side.) The *kind* of the join point is the kind of action (*e.g.,* call and execution).

2.1.2.2 Means of Identifying Join Points

The means of identifying join points is the pointcut mechanism. A *pointcut* is a predicate on join points, which is used to specify the join points that a piece of advice applies to. The syntax of pointcuts is shown in Figure 2. Since pointcuts can have parameters, the evaluation of a pointcut with respect to a join point results in either bindings that satisfy the pointcut (meaning true), or false.

The first three pointcuts (`call`, `execution`, and `new`) match join points that have the same kind and signature as the pointcut. The next two pointcuts (`target` and `args`) match any join point that has values of specified types. The next three operators (`&&`, `||` and `!`) logically combine or negate pointcuts. The last two pointcuts match join points that have a join point matching their sub-pointcuts in the call-stack. These are discussed in Section 4 in more detail. Interpretation of pointcuts is formally presented in other literature[20].

2.1.2.3 Means of Specifying Semantics

The means of specifying semantics is the advice mechanism. A piece of *advice* contains a pointcut and a body expression. When a join point is created, and it matches the pointcut of a piece of advice, the body of the advice is executed. There are two types of advice, namely `before` and `after`. A `before`

```
1  before : call(void Point.set(int)) && args(int z) {
2    write("set:"); write(z); newline();
3  }
```

Figure 3: Example Advice.

```
1   (define eval
2     (lambda (exp env jp)
3       (cond
4        ((const-exp? exp) (const-value exp))
5        ((var-exp? exp) (lookup env (var-name exp)))
6        ((call-exp? exp)
7         (call (call-signature exp)
8               (eval (call-target exp) env jp)
9               (eval-rands (call-rands exp) env jp)
10              jp))
11       ...)))
12  (define call
13    (lambda (sig obj args jp)
14      (execute (lookup-method (object-class obj) sig)
15               obj args jp)))
16  (define execute
17    (lambda (method this args jp)
18      (let ((class  (method-class method))
19            (params (method-params method)))
20        (eval (method-body method)
21              (new-env (list* 'this '%host params)
22                       (list* this  class  args))
23              jp))))
```

Figure 4: Expression Interpreter.

advice is executed before the original action is taken place. Similarly, the `after` is executed after the original action is completed.

Figure 3 shows an example of advice that lets the example program to print a message before every call to method `set`. The keyword `before` specifies the type of the advice. pointcut is written after the colon. The pointcut matches join points that call method `set` of class `Point`, and the `args` sub-pointcut binds variable `z` to the argument to method `set`. Line 2 is the body, which prints messages and the value of the argument.

When the program in Figure 1 is executed together with the advice in Figure 3, the advice matches to the call to `set` twice (in the constructor and in method `set`), it thus will print "`set:1`", "`set:6`" and "`6`".

2.2 AJD Interpreter

The interpreter of AJD consists of an expression interpreter and several definitions for AOP features including the data structure for a join point, wrappers for creating join points, a weaver, and a pointcut interpreter.

2.2.1 Expression Interpreter

Figure 4 shows the core of the expression interpreter excluding support for AOP features. The main function `eval` takes an expression, an environment, and a join point as its parameters. The join point is an execution join point at the enclosing method or constructor.

An expression is a parsed abstract syntax tree. There are predicates (*e.g.,* `const-exp?` and `call-exp?`) and selectors (*e.g.,* `const-value` and `call-signature`) for the syntax trees. An environment binds variables to mutable cells; *i.e.,* an assignment to a variable is implemented as side-effect in Scheme. An object is a Scheme data structure that has a class information and mutable fields for instance variables. Likewise, an assignment to an instance variable is implemented as side-effect.

Each action that creates join points is defined as a separate sub-function, so that we can add AOP support later.

For example, the interpreter evaluates a method call expression in the following manner. First, sub-expressions for the target object and operand values are recursively evaluated (ll.8–9). Next, in function `call`, a method is looked-up in the class of the target object (l.14). Then, in function `execute`, an environment that binds the formal parameters to the operand values is created (ll.25–26)[3]. Finally, the body of the method is evaluated with newly created environment (ll.24–27).

field	available information
`kind`	`call`, `execution`, etc.
`name`	name of method or class
`target`	target of method invocation
`args`	arguments to a method
`stack`	(explained in Section 4)

Table 1: Fields in Join Points

```
1  (define call
2    (lambda (sig obj args jp)
3      (weave (make-jp 'call  sig obj args jp)
4             (lambda (args jp)
5               ;; body of the original call goes here
6               )
7             args)))
```

Figure 5: A Wrapper.

2.2.2 Join Point

A join point is a data structure that is created upon an action in the expression interpreter. A piece of advice obtains all information about advised action from join points. In our implementation, a join point is a record of `kind`, `name`, `target`, `args`, and `stack`. Table 1 summarizes values in those fields. There are selectors, such as `jp-kind`, and a constructor, `make-jp`, for accessing and creating join points.

2.2.3 Wrapper

In order to advice actions performed in the expression interpreter, we wrap the interpreter functions so that they (conceptually) create dynamic join points. Figure 5 shows how `call`—one of such a function—is wrapped. When a method is to be called, the function first creates a join point that represents the call action (l.3) and applies it to `weave`, which executes advice applicable to the join point (explained below). The lambda-closure passed to `weave` (ll.4–6) defines the action of `call`, which is executed during the weaving process.

Likewise, the other functions including method execution,

[3]The pseudo-variable `%host` is used for looking-up methods for super classes.

```
1  (define weave
2    (lambda (jp action args)
3      (call-befores/afters *befores* args jp)
4      (let ((result (action args jp)))
5        (call-befores/afters *afters* args jp)
6        result)))
7  (define call-befores/afters
8    (lambda (advs args jp)
9      (for-each (call-before/after args jp) advs)))
10 (define call-before/after
11   (lambda (args jp)
12     (lambda (adv)
13       (let ((env (pointcut-match? (advice-pointcut adv)
14                                   jp)))
15         (if env
16             (execute-before/after adv env jp))))))
17 (define execute-before/after
18   (lambda (adv env jp)
19     (weave (make-jp 'aexecution adv #f #f '() jp)
20            (lambda (args jp)
21              (eval (advice-body adv) env jp))
22            '())))
```

Figure 6: Weaver.

```
1  (define pointcut-match?
2    (lambda (pc jp)
3      (cond
4       ((and (call-pointcut? pc) (call-jp? jp)
5             (sig-match? (pointcut-sig pc) (jp-name jp)))
6        (make-env '() '()))
7       ((and (args-pointcut? pc)
8             (types-match? (jp-args jp)
9                           (pointcut-arg-types pc)))
10       (make-env (pointcut-arg-names pc) (jp-args jp)))
11      ...
12      (else #f))))
```

Figure 7: Pointcut Interpreter.

object creation, and advice execution (defined later) are wrapped.

2.2.4 Weaver

Figure 6 shows the definition of the weaver. Function `weave` takes a join point (`jp`), a lambda-closure for continuing the original action (`action`), and a list of arguments to `action` (`args`). It also uses advice definitions in global variables (`*befores*` and `*afters*`). It defines the order of advice execution; it executes `before`s first, then the original action, followed by `after`s last.

Function `call-befores/afters` processes a list of advice. It matches the pointcut of each piece of advice against the current join point (ll.13–14), and executes the body of the advice if they match (ll.15–16). In order to (potentially) advise execution of advice, the function `execute-before/after` is also wrapped. Line 21 actually executes the advice body in an environment that provides the bindings expressed by the pointcut.

Calling `around` advice has basically the same structure for the `before` and `after`. It is, however, more complicated due to its interleaved execution for the `proceed` mechanism.

2.2.5 Pointcut interpreter

The pointcut interpreter `pointcut-match?`, shown in Figure 7, matches a pointcut to a join point. Due to space limitations, we only show rules for two types of pointcuts. The first rule (ll.4–6) defines that a `call`(m) pointcut matches to a `call` join point that whose `name` field matches to m. It returns an empty environment that represent 'true' (l.6). An `args`($t\ x, \ldots$) pointcut (where t and x are a type and a variable, respectively) matches to any join point whose arguments have the same type to $t, \ldots$ (ll.7–10). It returns an environment that binds variable $x, \ldots$ in the pointcut to the value of the argument in the join point (l.10). False is returned when matching fails (l.12).

3. COMPILING AJD PROGRAMS BY PARTIAL EVALUATION

3.1 Outline

Our compilation framework is based on partial evaluation of an interpreter, which is known as *the first Futamura projection*[9]. Given an interpreter of a language and a program to be interpreted, partially evaluating the interpreter with respect to the subject program generates a compiled program (called a *residual* program). Following this scheme, we can expect that partial evaluation of an AOP interpreter with respect to a subject program *and advice definitions* would generate a compiled, or *statically woven* program.

While the AJD interpreter is written as to 'test-and-execute' *all* pieces of advice at each *dynamic* join point, our compilation framework successfully inserts *only applicable* advice to each shadow of join points. This is achieved in the following way:

1. Our compilation framework runs partial evaluation with AJD interpreter and each method definition.
2. The partial evaluator processes the expression interpreter, which virtually walks over the expressions in the method. All shadows of join points are thus instantiated.
3. At each shadow of join points, the partial evaluator further processes the weaver. Using statically given advice definitions, it (conceptually) inserts test-and-execute sequence of all advice.
4. For each piece of advice, the partial evaluator reduces the test-and-execute code into a conditional branch whose condition is either constant or dynamic value, and then-clause executes the advice body. Depending on the condition, the entire code or the test code may be removed.
5. The partial evaluator may process the execution code of the advice body. It thus instantiates shadows of join points in the advice body. By recursively following the steps from 3, 'advised advice execution' is also compiled.

As is mentioned in the above step 1, we run partial evaluation with respect to each method definition. This is because the applicable method for a method call can not be determined at compile-time in object-oriented languages. Therefore, we start partially evaluation the `execute` function with its `method` parameter. The rest of the parameters (`env` and `jp`) are set to unknown at partial evaluation time. The residual program serves as a compiled (or statically woven) code of the method written in Scheme. The function is stored in a dispatch table so that it will be directly called at run-time.

For partial evaluation, we used PGG, an offline partial evaluator for Scheme[19].

3.2 How AJD is Partially Evaluated

An offline partial evaluator processes a program in the following way. It first annotates subexpressions in the program as either *static* or *dynamic*, based on their dependency on the statically known parameters. Those annotations are often called *binding-times.* It then processes the program from the beginning by actually evaluating static expressions and by returning symbolic expressions for dynamic expressions. The resulted program, which is called *residual program*, consists of a dynamic expressions in which statically computed values are embedded.

This subsection explains how the AJD interpreter is partially evaluated with respect to a subject program, by emphasizing what operations can be performed at partial evaluation time. Although the partial evaluation is an automatic process, we believe understanding this process is crucially important for identifying information available at compile-time and also for developing better insights into design of hand-written compilers.

3.2.1 Compilation of Expressions

The essence of the Futamura projection is to perform computation involving `exp` at partial evaluation time. Specialization of `execute` with each static `method` makes `eval` of `exp` static, and subsequent execution keeps this static property of `exp`. In contrast, `call` applies the `method` parameter as a dynamic value to `execute` due to the nature of dynamic dispatching in object-oriented languages. We therefore configure[4] the partial evaluator not to process `execute` from `call` so that it will not 'downgrade' the binding-time of `exp` to dynamic.

The environment (`env`) is treated as dynamic. With more careful interpreter design, we could make it *partially-static* data, in which variables are static and values are dynamic. However, this is not the focus of this paper.

3.2.2 Compilation of Advice

As is mentioned in Section 3.1, our compilation framework inserts advice bodies into their applicable shadows of join points with appropriate guards. Below, we explain how this is done by the partial evaluator.

1. A wrapper (*e.g.,* Figure 5) creates a join point upon calling `weave`. The first three fields of the join point, namely `kind`, `within` and `name`, are static because they merely depend on the program text. The rest fields have values computed at run-time. Those static fields could be passed to the weaver either by using partially-static data structure[4] or by rewriting the program to keep those three values in a split data structure. We took the latter approach for the ease of debugging and also for other technical reasons.

2. Function `weave` (Figure 6) is executed with a partially static join point, an action, and dynamic arguments. Since the advice definitions are statically available, the partial evaluator unrolls loops that test each advice definition (*i.e.,* `for-each` in `eval-befores/afters`).

3. As explained in Section 3.2.3, matching a static pointcut to a partially static join point may result in either a static or dynamic value. Therefore, the test-and-execute sequence (in `eval-before/after`) become one of the following three:

   Statically false: No action is taken; i.e., no code is inserted into compiled code.

   Statically true: The body of the advice is partially evaluated; *i.e.,* the body is inserted in compiled code without guards.

   Dynamic: In this case, partial evaluation of `pointcut-match?` generates an `if` expression whose then-clause is the above 'statically true' case and the else-clause is 'statically false' case. Essentially, the advice body is inserted with a guard.

4. In the statically true or dynamic cases at the above step, the partial evaluator processes the evaluation of the advice body. Since the wrapper of the advice execution calls `weave`, application of advice to the execution of advice body is also compiled.

5. When the original action is evaluated (l.4 in Figure 6), the residual code of the original action is inserted. This residual code from `weave` will thus have the original computation surrounded by applicable advice bodies.

3.2.3 Compilation of Pointcut

In step 3 above, pointcut interpreter (Figure 7) is partially evaluated with a static pointcut and a partially static join point. The partial evaluation process depends on the type of the pointcut. For pointcuts that depend on only static fields of a join point (namely `call`, `execution` and `new`), the condition is statically computed to either an environment (as true) or false. For pointcuts that test values in the join point (namely `target` and `args`), the partial evaluator returns residual code that dynamically tests the types of the values in the join point. For example, when `pointcut-match?` is partially evaluated with respect to `args(int x)`, the following expression is returned as residual code.

```
(if (types-match? (jp-args jp) '(int))
    (make-env '(x) (jp-args jp))
    #f)
```

Logical operators (namely `&&`, `||` and `!`) are partially evaluated into an expression that combines the residual expressions of its sub-pointcuts. The remaining two pointcuts (`cflow` and `cflowbelow`) are discussed in the next section.

[4]To do so, we rewrite `call` to call `execute*` instead of `execute`, and manually give dynamic binding-time to `execute*`.

```
1   (define point-move
2     (lambda (this1 args2 jp3)
3       (let* ((args7 (list (+ (get-field this1 'x)
4                               (car args2))))
5              (jp8 (make-jp this1 args7 (jp-state jp3))))
6         (if (types-match? args7 '(int))
7             (begin (write "set:")
8                    (write (car args7)) (newline)))
9         (execute* (lookup-method (object-class this1)
10                                 'set)
11                  args7 jp8)))))
```

Figure 8: Compiled code of `move` method of `Point` class.

The actual `pointcut-match?` is written in a continuation-passing style so that partially evaluator can reduce a conditional branch in the weaver (ll.15–16 in Figure 6) for the static cases. This is a standard technique in partial evaluation.

3.3 Compiled Code

Figure 8 shows the compiled code for the `move` method defined in Figure 1 combined with the advice given in Figure 3. For readability, we post-processed the residual code by eliminating dead code, propagating constants, renaming variable names, resolving environment accesses, and so forth.[5] Note that the compiled code manipulates only the dynamic portion of join points, as we split them into static and dynamic parts.

It first creates a parameter list (ll.3–4) and a join point (l.5) for method call. Lines 6 to 8 are advice body with a guard. The guard checks the residual condition for `args` pointcut. (Note that no run-time checks are performed for `call` pointcut.) If matched, the body of the advice is executed(ll.7–8). Finally, the original action (*i.e.,* method call) is performed (ll.9–11).

As we see, advice execution is successfully compiled. Even though there is a shadow of `execution` join points at the beginning of the method, no advice bodies are inserted in the compiled function as it does not match any advice.

4. COMPILING CALLING-CONTEXT SENSITIVE POINTCUTS

As briefly mentioned before, `cflow` and `cflowbelow` pointcuts can investigate join points in the call-stack; *i.e.,* their truth value is sensitive to calling context. Here, we first show a straightforward implementation that is based on a stack of join points. It is inefficient, however, and can not be compiled properly.

We then show a more optimized implementation that can be found in AspectJ compiler. The implementation exploits incremental natures of those pointcuts, and shown as a modified version of AJD. We can also see those pointcuts can be properly compiled by using our compilation framework.

[5]The shown code is compiled with optimized `cflow` evaluation mechanism presented in Section 4.3. Therefore, the last field of the join point is used in a manner different from Figure 5.

```
1   after : call(void Point.set(int))
2           && cflow(call(void Point.move(int))
3                    && args(int w)) {
4     write("under move:"); write(w); newline();
5   }
```

Figure 9: Advice with `cflow` Pointcut.

```
(define pointcut-match?
  (lambda (pc jp)
    (cond ...
         ((cflow-pointcut? pc)
          (or (pointcut-match? (pointcut-body pc) jp)
              (pointcut-match? pc (jp-stack jp))))
         ...)))
```

Figure 10: Naive Algorithm for Evaluating `cflow`.

To keep discussion simple, we only explain `cflow` in this section. Extending our idea to `cflowbelow` is easy and actually done in our experimental system.

4.1 Calling-Context Sensitive Pointcut: cflow

A pointcut `cflow`(p) matches to any join points if there is a join point that matches to p in its call-stack. Figure 9 is an example. The `cflow` pointcut in lines 2–3 specifies join points that are created during the method call to `move`. When this pointcut matches a join point, the `args(int w)` sub-pointcut gets the parameter to `move` from the stack.

As a result, execution of the program in Figure 1 with pieces of advice in Figures 3 and 9 prints "`set:1`" first, "`set:6`" next, and then "`under move:5`" followed by "`6`" last. The call to `set` from the constructor is not advised by the advice using `cflow`.

4.2 Stack-Based Implementation

A straightforward implementation is to keep a stack of join points and to examine each join point in the stack from the top when `cflow` is evaluated.

We use the `stack` field in a join point to maintain the stack. Whenever a new join point is created, we record previous join point in the `stack` field (as is done as the last argument to `make-jp` in Figure 5). Since join points are passed along method calls, the join points chained by the `stack` field from the current one form a stack of join points.

The algorithm to evaluate `cflow`(p) simply runs down the stack until it finds a join point that matches to p, as shown in Figure 10. If it reaches the bottom of the stack, the result is false.

The problem with this implementation is run-time overhead. In order to manage the stack, we have to push[6] a join point each time a new join point is created. Evaluation of `cflow` takes linear time in the stack depth at worse. When `cflow` pointcuts in a program match only specific join points, keeping the other join points in the stack and testing them is waste of time and space.

[6]By having a pointer to 'current' join point in parameters to each function, pop can be automatically done by returning from the function.

Our compilation does not help those problems. Rather, it highlights the problems. Since relationship between caller and receiver is unknown to the partial evaluator, the `stack` field of a join point becomes dynamic. Consequently, a stack of join points becomes partially-static in which only some fields of the topmost element are static, while the other elements are totally dynamic. When partial evaluator processes `pointcut-match?` with a static `cflow` pointcut and a partially static join point, the second recursive call (l.6 in Figure 10) supplies a dynamic (not partially static) join point. This makes the residual code a loop that dynamically tests each join point in the stack except for the top element[7]; *i.e.,* all the tests involving with `cflow` are performed at runtime.

4.3 State-Based Implementation

A more optimized implementation of `cflow` in AspectJ compiler is to exploit its incremental nature. This idea can be explained by an example. Assume (as in Figure 9) that there is pointcut "`cflow(call(void Point.move(int)))`" in a program. The pointcut becomes true once `move` is called. Then, until the control returns from `move` (or another call to `move` is taken place), the truth of the pointcut is unchanged. This means that the system needs only manage the state of each `cflow`(p) and update that state at the beginning and the end of join points that make p true. Note that the state should be managed by a stack because it may be rewound to its previous state upon returning from actions.

This state-based optimization can be explained in the following regards:

- The state-based implementation avoids repeatedly matching `cflow` bodies to the same join point in the stack, which can happen in the stack-based implementation. This is achieved by evaluating bodies of `cflow` at each join point in advance, and records the result as its state for later use.
- The state-based implementation makes static evaluation (*i.e.,* compilation) of `cflow` bodies possible, which can not in the stack-based implementation. This is because bodies are evaluated at each shadow of join points.
- The state-based implementation usually performs a smaller number of stack operations because the state of a `cflow` pointcut needs not be updated at the join points not matching to its body. On the other hand, the stack-based implementation has to push all join points on the stack.
- The state-based implementation evaluate `cflow` pointcut in constant time in by having a stack of states for each `cflow` pointcut.

It is not difficult to implement this idea in AJD. Figure 11 outlines the algorithm. Before running a subject

```
1  (define weave
2    (lambda (jp action args)
3      (let ((new-jp (update-states *cflow-pointcuts*
4                                   jp jp)))
5        ...the body of original weave...
6        )))
7  (define update-states
8   (lambda (pcs jp njp)
9    (if (null? pcs)
10     njp
11     (update-states (cdr pcs) jp
12       (let ((env (pointcut-match?
13                   (pointcut-body (car pcs)) jp)))
14        (if env
15          (update-state njp (pointcut-id (car pcs))
16                        env)
17          njp))))))
18 (define pointcut-match?
19   (lambda (pc jp)
20     (cond ...
21          ((cflow-pointcut? pc)
22           (lookup-state jp (pointcut-id pc)))
23          ...)))
```

Figure 11: State-based Implementation of `cflow`. (`update-state` *`jp id new-state`*`)` returns a copy of *`jp`* in which *`id`*'s state is changed to *`new-state`*. `(lookup-state` *`jp id`*`)` returns the state of *`id`* in *`jp`*.

```
(let* ((val7 ...create a point object ...)
       (args9 '(5))
       (jp8 (make-jp this1 args9 (jp-state jp3))))
  (if (types-match? args9 '(int))
      (begin
        (execute* (lookup-method (object-class val7)
                     'move)
                  val7 args9
                  (state-update jp8 '_g1
                                (new-env '(w) args9)))
        ... write and newline ...)
      ... omitted ...))
```

Figure 12: Compiled code of "`p.move(5)`" with `cflow` advice. (Definitions of variables `env6`, `this1` and `jp` are omitted.)

program, the system collects all `cflow` pointcuts in the program, including those appear inside of other `cflow` pointcuts. The collected pointcuts are stored in a global variable `*cflow-pointcuts*`. The system also gives unique identifiers to them, which are accessible via `pointcut-id`. We rename the last field of a join point from `stack` to `state`, so that it stores the current states of all `cflow` pointcuts.

When the interpreter creates a join point, it also updates the states of all `cflow` pointcuts by wrapping `weave`. The wrapper creates a copy of the new join point with updated `cflow` states (ll.3–4), and performs the original action. Function `update-states` evaluates the sub-pointcut of each `cflow` pointcut (ll.12–13), and updates the state if the result is true (ll.15–16).

Interpretation of `cflow` pointcut is merely looks up the state in the current join point (l.22).

4.4 Compilation Result

Figures 12 and 13 show excerpts of compiled code for the program in Figure 1 with the two pieces of advice in Figures

[7] If the partial evaluator supported polyvariant specialization[5]. Otherwise, the test for the topmost element is also coerced to dynamic.

```
1   (define point-move
2    (lambda (this1 args2 jp3)
3     (let* ((args5 (list (+ (get-field this1 'x)
4                            (car args2))))
5            (jp6 (make-jp this1 args5 (jp-state jp3))))
6      (if (types-match? args5 '(int))
7        (begin
8          (write "set:") (write (car args5)) (newline)
9          (let* ((val7
10                 (execute* (lookup-method
11                            (object-class this1) 'set)
12                           this1 args5 jp6))
13                (env8 (state-lookup jp6 '_g1)))
14           (if env8
15             (begin (write "under move:")
16                    (write (lookup env8 'w)) (newline)))
17           val7))
18       ...omitted...))))
```

Figure 13: Compiled code of method `move` with `cflow` advice.

3 and 9. The compiler gave `_g1` to the `cflow` pointcut as its identifier.

Figure 12 corresponds to "`p.move(5);`" (l.8 in Figure 1). Since the method call to `move` makes the `cflow` to true, the compiled code updates the state of `_g1` to an environment created by `args` pointcut in the join point (ll.9–10), and passes the updated join point to the method.

Figure 13 shows the compiled `move` method. We can see the additional code for the advice using `cflow` at lines 13–16. It dynamically evaluates the `cflow` pointcut by merely looking its state up, and runs the body of advice if the pointcut is true. The value of variable `w`, which is bound by `args` pointcut in `cflow`, is taken from the recorded state of `cflow` pointcut. Since the state is updated when `move` is to be called, it gives the argument value to `move` method.

To summarize, our framework compiles a program with `cflow` pointcuts into one with state update operations at each join point that matches the sub-pointcut of each `cflow` pointcut, and state look-ups in the guard of advice bodies. In terms of run-time checks for pointcuts, the code is basically identical to the one generated by AspectJ compiler.

5. RELATED WORK

In reflective languages, some crosscutting concerns can be controlled through meta-programming[10, 18]. Several researchers including the first author have successfully compiled reflective programs by using partial evaluation[2, 14, 15]. It is more difficult, however in reflective languages, to ensure successful compilation because the programmer can easily write a meta-program that confuses the partial evaluator.

Wand and the second and the third authors presented a formal model of the procedural version of AJD[20]. Our model is based on this, and used it for compilation and optimizing `cflow` pointcuts.

Douence et al. showed an operational semantics of an AOP system[8]. Their system is based on a 'monitor' that observes the behavior of subject program, and the weaving is triggered by means of pattern matching to a stream of events. They also gave a program transformation system that inserts code to trigger the monitor into subject program. Our framework automatically performs this insertion by using partial evaluation.

Andrews proposed process algebras as a formal basis of AOP languages[1]. In his system, advice execution is represented by synchronized processes, and compilation (static weaving) is transformation of processes that removes synchronization. Our experience suggests that powerful transformation techniques like partial evaluator would be needed to effectively remove run-time checks in dynamic join point models.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a compilation framework to an aspect-oriented programming (AOP) language, AJD, based on operational semantics and partial evaluation techniques. The framework explains issues in AOP compilers including identifying shadows of join points, compiling-out pointcuts and recursively applying advice. The optimized `cflow` implementation in AspectJ compiler can also be explained in this framework.

The use of partial evaluation allows us to keep simple operational semantics in which everything is processed at run-time, and to relate the semantics to compilation. Partial evaluation also allows us to understand the data dependency in our interpreter by means of its binding-time analysis. We believe this approach would be also useful to prototyping new AOP features with effective compilation in mind.

Although our language supports only core features of practical AOP languages, we believe that this work could bridge between formal studies and practical design and implementation of AOP languages.

Future directions of this study could include the following topics. Optimization algorithms could be studied for AOP programs based on our model, for example, elimination of more run-time checks with the aid of static analysis. Our model could be refined into more formal systems so that we could relate between semantics and compilation with correctness proofs. Our system could also be applied to design and test new AOP features.

7. REFERENCES

[1] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In Yonezawa and Matsuoka [21], pages 187–209.

[2] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation —for a better understanding of reflective languages—. *Lisp and Symbolic Computation*, 9:203–241, 1996.

[3] Lodewijk Bergmans and Mehmet Aksits. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, October 2001.

[4] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *ACM Conferenceon Lisp and Functional Programming*, pages 1–10, 1992.

[5] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[6] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring operating system aspects: using AOP to improve OS structure modularity. *Communications of the ACM*, 44(10):79–82, October 2001.

[7] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering*, pages 88–98, Vienna, Austria, 2001.

[8] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [21], pages 170–186.

[9] Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from *Systems, Computers, Controls*, 2(5):45–50, 1971.

[10] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353, 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*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[14] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In Mary E. S. Loomis, editor, *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, volume 30(10) of *ACM SIGPLAN Notices*, pages 300–315, Austin, TX, October 1995. ACM.

[15] Hidehiko Masuhara and Akinori Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In Eric Jul, editor, *European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 418–439, Brussels, Belgium, July 1998. Springer-Verlag.

[16] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In Yonezawa and Matsuoka [21], pages 73–80.

[17] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns using hyperspaces. Technical report, IBM, 1999.

[18] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Conference record of Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[19] Peter J. Thiemann. Cogen in six lines. In *International Conference on Functional Programming (ICFP'96)*, 1996.

[20] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic joint points in aspect-oriented programming. In *Proceedings of The Ninth International Workshop on Foundations of Object-Oriented Languages (FOOL9)*, January 2002.

[21] Akinori Yonezawa and Satoshi Matsuoka, editors. *Third International Conference Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, Koyoto, Japan, September 2001. Springer-Verlag.