A Compilation and Optimization Model for Aspect-Oriented Programs*

H. Masuhara, G. Kiczales, and C. Dutchyn

[1] Graduate School of Arts and Sciences, University of Tokyo
[2] Intentional Software Corporation
[3] Department of Computer Science, University of British Columbia

Abstract. This paper presents a semantics-based compilation model for an aspect-oriented programming language based on its operational semantics. Using partial evaluation, the model 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.

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[1].

In this paper we report one result from the ASB project—an operational-semantics based explanation of the compilation and optimization strategy for AspectJ-like languages[5,9]. To avoid difficulties to develop formal semantics directly from artifacts as complex as AspectJ, we used a simplified language. It yet has sufficient features to discuss compilation and optimization of real languages.

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 model also explains the optimization used by the AspectJ compiler for calling-context sensitive pointcuts (`cflow` and `cflowbelow`).

Some of the issues our semantic model 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.

* An early version of the paper was presented at FOAL 2002, Workshop on Foundations of Aspect-Oriented Languages at AOSD 2002.

[1] `http://www.cs.ubc.ca/labs/spl/projects/asb.html`

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 46–60, 2003.

1.1 Join Point Models

Aspect-oriented programming (AOP) is a paradigm to modularize crosscutting concerns[10]. An AO 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[3,9,13,14].

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 programs including aspects can affect. *Lexical* join points are locations in the program text (*e.g.,* "the body of a method"). *Dynamic* join points are run-time actions, 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 effecting* at join points. (*e.g.,* "run this code beforehand")

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 simplified JPM, namely Pointcut and Advice (PA), and shows its interpreter. Section 3 presents a compilation scheme for PA excluding context-sensitive pointcuts, which are deferred to Section 4. Section 5 relates our study to other formal studies in AOP and other compilation schemes. Section 6 concludes the paper with future directions.

2 PA: Dynamic Join Point Model AOP Language

This section introduces our small join point model, namely Pointcut and Advice (PA), which implements core features of the AspectJ's dynamic join point model. PA is modeled as an AOP extension to a simple object-oriented language. Its operational semantics is given as an interpreter written in Scheme. A formalization of a procedural subset of PA is presented by Wand, Kiczales and Dutchyn[19].

2.1 Informal Semantics

We first informally present the semantics of PA. In short, PA is a dynamic join point model that covers core features of AspectJ on top of a simple object-oriented language with classes, objects, instance variables, and methods.

Object Semantics. Figure 1 is an example program[2]. For readability, we use a Java-like syntax in the paper[3]. It defines a `Point` class with one integer instance variable `x`, a unary constructor, and three methods `set`, `move` and `main`.

[2] For simplicity later in the paper, we are using one-dimensional points as an example.
[3] Our implementation actually uses an S-expression based syntax.

```
class Point {
  int x;
  Point(int ix)      { this.set(ix); }
  void set(int newx) { this.x = newx; }
  void move(int dx)  { this.set(this.x + dx); }
  void main()        { Point p = new Point(1);
                       p.move(5); write(p.x); newline(); } }
```

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

When method `main` of a `Point` object is executed, it creates another `Point` object, and runs the constructor body. The `main` method then invokes method `move` on the created object, reads the value of variable `x` of the object and displays it.

Aspect Semantics. To explain the semantics of AOP features, we first define the PA join point model.

Join Point. The *join point* is an action during program execution, including method calls, method executions, object creations, and advice executions. (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).

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. There are five kinds of primitive pointcuts, namely `call(`m`)`, `execution(`m`)`, `new(`m`)`, `target(`t v`)`, and `args(`t v`,...)`, three operators (`&&`, `||` and `!`), and two higher-order pointcuts, namely `cflow(`p`)` and `cflowbelow(`p`)`.

The first three primitive pointcuts (`call`, `execution`, and `new`) match join points that have the same kind and signature as the pointcut. The next two primitive pointcuts (`target` and `args`) match any join point that has values of specified types. The three operators logically combine or negate pointcuts. The last two higher-order 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[19].

Means of Effecting at Join Points. The means of effecting at join points 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 the advice, the advice body is executed. There are two types of advice, namely `before` and `after`[4]. A

[4] For simplicity we omit `around` advice and `after returning` advice which can inspect return values. However, our experimental implementation actually supports those types of advice.

```
(define eval
 (lambda (exp env jp)
  (cond ((const-exp? exp) (const-value exp))
        ((var-exp? exp)   (lookup env (var-name exp)))
        ((call-exp? exp)  (call (call-signature exp)
                                (eval (call-target exp) env jp)
                                (eval-rands (call-rands exp) env jp) jp))
        ...)))
(define call
  (lambda (sig obj args jp)
    (execute (lookup-method (object-class obj) sig) obj args jp)))
(define execute
  (lambda (method this args jp)
    (eval (method-body method)
          (new-env (append '(this %host) (method-params method))
                   (append (list this (method-class method)) args))
          jp)))
```

Fig. 2. Expression Interpreter.

`before` advice runs before the original action is taken place. Similarly, the `after` runs after the completion of the original action.

The following example advice definition lets the example program to print a message before every call to method `set`:

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

It consists of a keyword for the kind of the advice (`before`), a pointcut, and a body in braces. 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`. The body of the advice prints messages and the value of the argument.

When the `Point` program is executed together with the above advice, 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 Interpreter

The interpreter 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.

Expression Interpreter. Figure 2 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, which can be tested with `const-exp?`, etc., and can be accessed with `const-value`, etc. An environment

```
(define call
  (lambda (sig obj args jp)
    (weave (make-jp 'call  sig obj args jp)
           (lambda (args jp) ...body of the original call...)
           args)))
```

Fig. 3. A Wrapped Interpreter Function.

binds variables to mutable cells. An object is a Scheme data structure that has a class information and mutable fields for instance variables.

The body of `eval` is a simple case-based test on expression types. Some operations are defined as separated functions for the later extension of AOP features.

Join Point. A join point is a data structure that is created upon an action in the expression interpreter:[5]

```
(define-struct jp (kind name target args stack))
```

The `kind` field specifies the kind of the join point as a symbol (*e.g.,* `'call`). The `name` field has the name of the method being called. The `target` and `args` fields have the target object and the arguments of the method invocation, respectively. The `stack` field will be explained in Section 4.

Wrapper. In order to advice actions performed in the expression interpreter, we wrap the interpreter functions so that they create dynamic join points. Figure 3 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 and applies it to `weave`, which executes advice applicable to the join point (explained below). The lambda-closure passed to `weave` defines the action of the original `call`, which is executed during the weaving process.

Likewise, the other functions including method execution, object creation, and advice execution (defined later) are wrapped.

Weaver. Figure 4 shows the definition of the weaver. Function `weave` takes a join point, a lambda-closure for continuing the original action, and a list of arguments to the closure. 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, and executes the body of the advice if they match. In order to advise execution of advice, the

[5] This non-standard Scheme construct defines a structure named `jp` with five fields named `kind`, `name`, `target`, `args`, and `stack`.

```
(define weave
  (lambda (jp action args)
    (call-befores/afters *befores* args jp)
    (let ((result (action args jp)))
      (call-befores/afters *afters* args jp)
      result)))
(define call-befores/afters
  (lambda (advs args jp)
    (for-each (call-before/after args jp) advs)))
(define call-before/after
  (lambda (args jp)
    (lambda (adv)
      (let ((env (pointcut-match? (advice-pointcut adv) jp)))
        (if env (execute-before/after adv env jp))))))
(define execute-before/after
  (lambda (adv env jp)
    (weave (make-jp 'aexecution adv #f #f '() jp)
           (lambda (args jp) (eval (advice-body adv) env jp))
           '())))
```

Fig. 4. Weaver.

```
(define pointcut-match?
  (lambda (pc jp)
    (cond ((and (call-pointcut? pc) (call-jp? jp)
                (sig-match? (pointcut-sig pc) (jp-name jp)))
           (make-env '() '()))
          ((and (args-pointcut? pc)
                (types-match? (jp-args jp) (pointcut-arg-types pc)))
           (make-env (pointcut-arg-names pc) (jp-args jp)))
          ...
          (else #f))))
```

Fig. 5. Pointcut Interpreter.

function `execute-before/after` is also wrapped. The lambda-closure in the function actually executes the advice body.

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.

Pointcut interpreter. The function `pointcut-match?` in Figure 5 matches a pointcut to a join point. Due to space limitations, we only show rules for two types of pointcuts. The first clause of the `cond` matches a `call`(m) pointcut to a `call` join point that has a matching `name` field matches to m. It returns an empty environment that represent 'true'. The second clause matches an `args`($t\ x, \ldots$) pointcut to any join point when `args` filed has values of types $t, \ldots$. The result in this case is an environment that binds variables $x, \ldots$ to the values in the `args` field. The last clause returns false for unmatched cases.

3 Compiling Programs by Partial Evaluation

Our compilation scheme is to partially evaluate an interpreter, which is known as *the first Futamura projection*[7]. Given an interpreter of a language and a program to be interpreted, partial evaluation of the interpreter with respect to the subject program generates a compiled program (called a *residual* program). By following this scheme, partial evaluation of an AOP interpreter with respect to a subject program *and* advice definitions would generate a compiled, or *statically woven* program.

The effect of partial evaluation is removal of unnecessary pointcut tests. While the interpreter tests-and-executes *all* pieces of advice at each *dynamic* join point, our compilation scheme successfully inserts *only* applicable advice to each shadow of join points. This is achieved in the following way:

1. Our compilation scheme partially evaluates the interpreter with respect to each method definition.
2. The partial evaluator (PE) 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 PE 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 PE reduces the test-and-execute code into a conditional branch that has either a constant or dynamic value as its condition, and the advice body as its then-clause. Depending on the condition, the entire code or the test code may be removed.
5. The PE processes the code that executes the advice body. It thus instantiates shadows of join points in the advice body. The steps from 3 recursively compiles 'advised advice execution.'

We used PGG, an offline partial evaluator for Scheme[17], for partial evaluation.

3.1 How the Interpreter Is Partially Evaluated

An offline partial evaluator processes a program in the following way. It first annotates expressions 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 by actually evaluating static expressions and by returning symbolic expressions for dynamic expressions. The resulted program, which is called *residual program*, consists of dynamic expressions in which statically computed values are embedded.

This subsection explains how the 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 compile-time information and also for developing better insights into design of hand-written compilers.

Compilation of Expressions. The essence of the first Futamura projection is to evaluate computation involving `exp` away. In fact, occurrences of `exp` in the interpreter are annotated as static except for the first argument to `execute` in function `call`. The argument is dynamic due to the nature of dynamic dispatching in object-oriented languages. We therefore invoke the partial evaluator for each method definition, and replaced the function `execute` with the one that dynamically dispatches on a receiver's type. This standard partial evaluation technique is known as 'The Trick.'

The environment (`env`) is regarded as a partially-static data structure; *i.e.,* the variable names are static and the values are dynamic. As a result, the partial evaluator compiles variable accesses in the subject program into accesses to elements of the argument list in the residual code.

Compilation of Advice. As is mentioned at the beginning of the section, our compilation scheme 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 3) creates a join point upon calling `weave`. The first two fields of the join point, namely `kind` and `name`, are static because they only depend on the program text. The rest fields have values computed at run-time. We actually split the join point into two data structures so that static and dynamic fields are stored separately. With partial evaluators that support partially static data structures[4], we would get the same result without splitting the join point structure.
2. Function `weave` (Figure 4) is executed with the static join point, an action, and dynamic arguments. Since the advice definitions are statically available, the partial evaluator unrolls the `for-each` in in function `eval-befores/afters`.
3. The result of `pointcut-match?` can be either static or dynamic depending on the type of a pointcut. Therefore, the test-and-execute sequence (in `eval-before/after`) becomes one of the following three:
   Statically false: 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.
   Dynamic: Partial evaluation of `pointcut-match?` generates an `if` expression with the body of advice in the then-clause and an empty else-clause. 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. If the advice is applicable to more than one join point shadows in a method, the compiled body is shared as a Scheme function thanks to a mechanism in the partial evaluator. Since the wrapper of the `execute-before/after` calls `weave`, application of advice to the advice body is also compiled.
5. When the original action is evaluated, 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.

```
(define point-move
  (lambda (this1 args2 jp3)
    (let* ((jp4   (make-jp 'execution 'move this1 args2 jp3))
           (args5 (list (+ (get-field this1 'x) (car args2))))
           (jp6   (make-jp 'call 'set this1 args5 jp4)))
      (if (type-match? args5 '(int))
          (begin (write "set:") (write (car args5)) (newline)))
      (execute* (lookup-method (object-class this1) 'set)
                this1 args5 jp6))))
```

Fig. 6. Compiled code of `move` method of `Point` class.

Compilation of Pointcut. In step 3 above, `pointcut-match?` is partially evaluated with a static pointcut and static fields in a 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 (*e.g.,* `call`), the condition is statically computed to either an environment or false. For pointcuts that test values in the join point (*e.g.,* `target`), 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 the 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.

The actual `pointcut-match?` is written in a continuation-passing style so that partially evaluator can reduce a conditional branch in `call-before/after` for the static cases. This is a standard technique in partial evaluation, but is crucially important to get right results.

3.2 Compiled Code

Figure 6 shows the compiled code for `Point.move` combined with the advice given in Section 2.1. For readability, we post-processed the residual code by eliminating dead code, propagating constants, renaming variable names, combining split join point structures, and so forth. The post-process was done automatically except for renaming and combining.

The compiled function first creates a join point `jp4` for the method execution, a parameter list and a join point `jp6` for the method call. The `if` expression is the 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. Finally, the original action is performed.

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 an optimized implementation that can be found in AspectJ compiler. The implementation exploits incremental natures of those pointcuts, and is presented as a modified version of PA interpreter. We can also see those pointcuts can be properly compiled by using our compilation scheme.

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. The following definition is an example advice that uses a `cflow` pointcut. The `cflow` pointcut matches join points that are created during method calls to `move`. When this pointcut matches a join point, the `args(int w)` sub-pointcut gets the parameter to `move` from the stack.

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

As a result, execution of the `Point` program with two pieces of advice presented in Section 2.1 and above 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 3). 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. Restoring old join points is implicitly achieved by merely using the original join point in the caller's continuation.

The following definition shows the algorithm to interpret `cflow`(p) that simply runs down the stack until it finds a join point that matches to p. If it reaches the bottom of the stack, the result is false.

```
(define pointcut-match?
  (lambda (pc jp)
    (cond ((cflow-pointcut? pc)
           (let loop ((jp jp))
             (and (not (bottom? jp))
                  (or (pointcut-match? (pointcut-body pc) jp)
                      (loop (jp-stack jp))))))
          ...)))
```

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 worst. 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.

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 shown previously) 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 value of the pointcut is unchanged. This means that the system only needs managing the state of each `cflow`(p) and updating 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 has to be rewound to its previous state upon returning from actions.

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

- It avoids repeatedly matching `cflow` bodies to the same join point in the stack by evaluating bodies of `cflow` upon creation of each join point, and recording the result.
- It makes static evaluation (*i.e.,* compilation) of `cflow` bodies possible because they are evaluated at each shadow of join points. As a result, management of a `cflow` state is only taken place at shadows of join points matching to the body of `cflow`.
- It evaluates `cflow` pointcut in constant time by merely peeking the top of a stack of states for each `cflow` pointcut.

It is straightforward to implement this idea in the PA interpreter. Figure 7 outlines the algorithm. Before running a subject program, the system collects all `cflow` pointcuts in the program, including those appear inside of other `cflow` pointcuts, and stores 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.

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

```
(define weave
  (lambda (jp action args)
    (let ((new-jp (update-states *cflow-pointcuts* jp)))
      ...the body of original weave...)))
(define update-states
  (lambda (pcs jp)
    (fold (lambda (pc njp) ;; fold: ('a*'b->'a)*'a*'b list->'a
            (let ((env (pointcut-match? (pointcut-body pc jp))))
              (if env (update-state njp (pointcut-id pc) env)
                      njp)))
          jp pcs)))
(define pointcut-match?
  (lambda (pc jp)
    (cond ((cflow-pointcut? pc) (lookup-state jp (pointcut-id pc)))
          ...)))
```

Fig. 7. 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*.

When evaluation of an expression creates a join point, it first updates the states of all `cflow` pointcuts by wrapping `weave` by calling function `update-states`. The function `update-states` evaluates the body of each `cflow` pointcut, and updates the state only if the result is true. Otherwise, the state is unchanged. Therefore, after partial evaluation, the code for updating state is also eliminated when the body of the `cflow` is statically determined as false. The conditional case for `cflow` pointcuts in `pointcut-match?` merely looks up the state in the current join point.

Support for `cflowbelow` pointcuts is to extend the state to a pair of states. We omit details due to space limitation.

Those two stack- and state-based implementations can also be understood as initial- and final-algebra representations[18, etc.] of join points. The stack-based implementation defines a join point as the following data structure:

```
(define-struct jp (kind name target args stack))
```

By noticing that the `stack` field of join points is accessed only for matching the join points to the `cflow` pointcuts in the program, the structure can take a final-algebra representation:

```
(define-struct jp (kind name target args r1 r2 ... rn))
```

where `r`*i* is the result of `pointcut-match?` on the *i*'th `cflow` pointcut in the program. This is exactly what we have done for the state-based implementation.

4.4 Compilation Result

Figures 8 shows excerpts of compiled code for the `Point` program with the two advice definitions shown before. The compiler gave `_g1` to the `cflow` pointcut as its identifier.

```
(let* ((val7 ...create a point object ...);----compiled code of p.move(5)
       (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 ...))
(define point-move ;-------------------------compiled code of Point.move
 (lambda (this1 args2 jp3)
  (let* ((args5 (list (+ (get-field this1 'x) (car args2))))
         (jp6 (make-jp this1 args5 (jp-state jp3))))
   (if (types-match? args5 '(int))
     (begin (write "set:") (write (car args5)) (newline)
            (let* ((val7 (execute* (lookup-method (object-class this1)
                                                  'set)
                                   this1 args5 jp6))
                   (env8 (state-lookup jp6 '_g1)))
              (if env8 (begin (write "under move:")
                              (write (lookup env8 'w)) (newline)))
         val7))
     ...omitted...))))
```

Fig. 8. Compiled code of `p.move(5)` and `Point.main` with `cflow` advice.

The first expression corresponds to `p.move(5);` in `Point.main`. Since the method call to `move` makes the state of the `cflow` to true, the compiled code updates the state of `_g1` to an environment created by `args` pointcut in the join point, and passes the updated join point to the method.

The next function shows the compiled `move` method. The second `if` expression and the preceding `state-lookup` are for the advice using `cflow`. It 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 scheme 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. By comparing the compiled code with the one generated by AspectJ compiler, we observe that those two compilation frameworks insert update operations for the `cflow` states into the same places.

5 Related Work

In reflective languages, some crosscutting concerns can be controlled through meta-programming[8,16]. Several studies successfully compiled reflective programs by using partial evaluation[2,11,12]. It is more difficult to ensure successful

compilation in reflective languages because the programmer can write arbitrary meta-programs.

Wand, Kiczales and Dutchyn presented a formal model of the procedural version of PA[19]. 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[6]. In their system, a 'monitor' pattern matches a stream of events from a program execution, and invokes advice code when matches. A program transformation system inserts code into the monitored program so that it triggers the monitor. In our scheme, partial evaluator automatically performs this insertion.

Andrews proposed process algebras as a formal basis of AOP languages[1], in which advice execution is represented as synchronized processes. 'Compilation' can be understood as removal of the synchronization. However, our experience suggests that transformation techniques as powerful as partial evaluation would be necessary to properly remove run-time checks.

6 Conclusion and Future Work

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

The use of partial evaluation allows us to keep simple operational semantics and to relate the semantics to compilation. It also helped 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[15]. 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.

Acknowledgments. The authors are grateful to Kenichi Asai, Oege de Moor, Kris de Volder, Mitchell Wand and participants of FOAL2002 workshop for their comments on the previous version the paper. The discussion on the initial- and final-algebra representations was first pointed out by Mitchell Wand. We would also like to thank the anonymous reviewers for their comments. Most of the work is carried out during the first author's visit to University of British Columbia.

References

1. James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In Yonezawa and Matsuoka [20], 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 Aksit. 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. Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *FSE-9*, pages 88–98, 2001.
6. Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In Yonezawa and Matsuoka [20], pages 170–186.
7. 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.
8. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
9. 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.
10. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97*, pages 220–242, 1997.
11. Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *OOPSLA'95*, pages 300–315, 1995.
12. Hidehiko Masuhara and Akinori Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In *ECOOP'98*, pages 418–439, 1998.
13. Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In Yonezawa and Matsuoka [20], pages 73–80.
14. Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns using hyperspaces. Research Report 21452, IBM, April 1999.
15. Damien Sereni and Oege de Moor. Static analysis of aspects. In *AOSD2003*, 2003.
16. Brian Cantwell Smith. Reflection and semantics in Lisp. In *Conference record of Symposium on Principles of Programming Languages*, pages 23–35, 1984.
17. Peter J. Thiemann. Cogen in six lines. In *ICFP'96*, 1996.
18. Mitchell Wand. Final algebra semantics and data type extension. *Journal of Computer and System Sciences*, 19:27–44, 1979.
19. Mitchell Wand, Gregor Kiczales, and Chris Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Proceedings of FOAL2002*, pages 1–8, 2002.
20. Akinori Yonezawa and Satoshi Matsuoka, editors. *Third International Conference Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, 2001.