9-22-2006

Modular Compilation Strategies for Aspect-Oriented Constructs

Robert Dyer
Iowa State University, rdyer@iastate.edu

Hridesh Rajan
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Programming Languages and Compilers Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/251

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
Modular Compilation Strategies for Aspect-Oriented Constructs

Robert Dyer  
Iowa State University  
rdyer@iastate.edu

Hridesh Rajan  
Iowa State University  
hridesh@iastate.edu

Abstract
In our previous work, we presented an aspect-oriented intermediate language, named Nu, to preserve design modularity in object code. Nu is based on two primitives: bind and remove. We showed that maintaining modularity in object code significantly improved the incremental compilation time of aspect-oriented programs. The key contribution of this work is a set of compilation strategies to Nu for a number of AspectJ constructs such as control flow (cflow and cflowbelow), instantiation (perthis, pertarget, perflow, perflowbelow) and dynamic checks (if, this, target, args), as well as composition operators (&& and |). The motivation was to determine if these high-level language constructs need to be supported in the intermediate language. Our compilation strategies are modular and textually local. To compile a construct in a module, only the information about that module’s implementation and the specification of other modules referenced in that module are needed. The generated intermediate code for a construct in a source module is confined to a single module in the object code. We show that our compilation strategies improve incremental compilation time of aspect-oriented programs. We also analyze our intermediate language with respect to constructs that are not directly supported.

1. Introduction
Emerging aspect-oriented (AO) techniques [9, 16] provide software engineers with new possibilities for keeping conceptual concerns separate at the source code level [6, 29]. However, to remain compatible with existing virtual machines current AO compilers sacrifice this separation in transforming source to object code by loosing textual locality and intermingling concerns. This in turn affects the efficiency and complexity of other development tools such as incremental compiler, debuggers, profilers, etc, potentially decreasing their scalability. Our previous work provides some evidence to support the hypothesis that simple and elegant enhancements in intermediate language designs will enable aspect-oriented compilers to maintain design modularity in the object code, make development processes scalable, and in doing so afford the benefits of aspect-oriented techniques to large-scale software systems [7, 26].

In previous work, we proposed an AO intermediate language model called Nu to preserve design modularity in object code [26]. Nu adds two primitives to the OO intermediate language model. In a following work, we showed that preserving AO design modularity in the object code significantly improves the incremental compilation time of AO programs [7]. Improvements of up to ten times were shown in most cases. It remained unclear, however, whether our intermediate language model was able to support most constructs in a high-level language as expressive as AspectJ.

In this context, we describe a language design experiment. We selected AspectJ-like languages [1, 15] as the subject. We then developed compilation strategies for constructs in these languages to Nu. Our compilation strategies composed the Nu primitives in simple but interesting ways to express high-level language constructs at the intermediate language level. Our compilation strategies guided our intermediate language design process. If we were able to find a textually local translation to the low-level language, we concluded that the construct in question is syntactic sugar and therefore the intermediate language does not need to support it. The intuition is similar to Felleisen’s notion of macro-eliminable programming language extensions [11], however, we did not attempt to prove that a textually local translation does not exist if we could not find one.

Our compilation strategies are modular, in the sense that to compile a construct in a module, only the information about that module’s implementation and the specification of other modules referenced in that module are needed. Our compilation strategies are also textually local. By textually local we mean that the generated intermediate code for a construct in a source module is confined to a single module in the object code. Modularity and textual locality of generated code in our compilation strategies resulted in a more efficient incremental compilation of AO programs.

To evaluate our approach we compare it with conventional compilation techniques. Two measures are used. We compare the incremental compilation time of both techniques. We compare the files modified by both techniques as a result of a small change. Our results show significant decreases in the incremental compilation times of each construct and decrease in number of files modified by the incremental compiler. We analyzed our intermediate language design, with respect to constructs for which modular and textually local compilation strategies did not appear feasible.

The rest of this paper is organized as follows. Section 2 motivates our approach. Section 3 briefly describes the Nu language model. Section 4 discusses our compilation strategies for high-level language constructs. Section 5 describes our evaluation. Section 6 discusses related work. Section 7 discusses future work and concludes.

2. Motivation
In order to generate object code that is compliant with the existing virtual machines (VM) such as Java Virtual Machine (JVM) [19] for AspectJ [15] and .NET Framework [23] for Eos [28] aspect weavers sacrifice the design modularity in transforming source to object code (and thus the very term weaving) [26]. The aspect-oriented modularity that was present in the source code is thus lost after compilation. Sacrificing design modularity affects the performance of incremental compilation, which is demonstrated in in-
increased incremental compilation time. Incremental compilation is defined as the property of a compiler such that a small change in syntax or semantic structure requires only a small amount of reprocessing to reflect the change [2]. In a recent report, Lesiecki [17] observed that on an average incremental compilation of 700 classes and around 70 aspects using the AspectJ compiler usually takes at least 2-3 seconds longer than the near instant compilation using a pure Java compiler.

To illustrate the issue, let us consider the compilation of the cflow construct. In current implementations of aspect-oriented compilers, cflow is implemented by generating additional code to perform runtime checks and determine if the program is currently in the control flow of a join point (such as the execution of a method). For example, the pointcut expression in Figure 1(a) will result in the generation of a set of instructions to dynamically check at all program points where a method `read(..)` is called to determine if the program is currently in the control flow of any method in java.io.InputStream.

Now, a change in the pointcut expression, say to the pointcut in Figure 1(b), will have an impact on all program points where the dynamic check was generated and instead of checking for methods belonging to type java.io.InputStream the generated code will now check for methods belonging to type java.io.FileInputStream.

A simple change in the source code will thus lead to non-trivial processing.

Our previous work [7] examined this problem in detail by measuring the incremental compilation time of AspectJ [1] programs after making minor modifications to the source code. Figure 2 shows the incremental compilation times of the Azureus peer-to-peer application, a medium-scale system with around 3500 classes, 2000 source files, and 200 KLOC. In most cases, a small change in the aspect resulted in an increase in incremental compilation time of 10 times when compared to the incremental compilation of a Java class in the system. This problem becomes even more apparent with larger systems. Eclipse, a Java integrated development environment, is a large-scale system. Figure 3 shows the incremental compilation time of a subset of Eclipse\(^1\) (over 12000 classes, 7000 source files, and almost 800 KLOC). In most cases, the incremental compilation times in this system are over 30 seconds for very simple changes. The increased incremental compilation time can potentially affect the build-test-debug cycle common in many software development processes.

### 3. Nu Language Model

In this section, we will describe various elements of the Nu language model.

#### 3.1 Join point model

Similar to AspectJ [15], Nu adds a new concept to the underlying language semantics – join points. Instead of AspectJ's join point model, Nu uses a model proposed by Endoh et al [10]. Their model is called point-in-time model. In this model, a join point represents a specific point in the execution of the program, such as the beginning of a method execution or the termination of a method. This differs from AspectJ's region-in-time model, which defines a join point as a duration of an event such as the entire execution of a method from beginning to its termination, but is sufficiently expressive to represent common advising scenarios and offers a finer grained representation of join points.

These three join points eliminate the need for three different types of advice: namely before, after, and after throwing advice. The before call, after call, and after throwing call become equivalent to call, reception, and failure. These three join points eliminate both the need for three different types of advice: namely before, after, and after throwing advice. The before call, after call, and after throwing call become equivalent to call, reception, and failure, respectively.

Similarly, corresponding to AspectJ's execution join point, there are three join points execution, return, and throw.

#### 3.2 Methods model advice in Nu

Similar to AspectWerkz [5], CaesarJ [22], classpects in Eos [28], etc, Nu models an advice as an object-oriented method. Figure 4(b) shows an example aspect in AspectJ (lines 1–6, Figure 4(a)) translated to Nu. The aspect in the previous example is modeled by an object-oriented class (lines 1–9, Figure 4(b)). The advice in the previous example (lines 3–5, Figure 4(a)) is modeled as a normal object-oriented method `trace()` (lines 6–8). Currently, this method can have either no argument or one argument of type Nu.Runtime.JoinPoint. This argument is similar to the implicit argument thisJoinpoint in AspectJ. In future, we plan to address this limitation so that variable binding of advice arguments with the context information at the join point becomes possible.

\(^1\) A subset was selected due to memory constraints with Java and the large amount of memory required by the AspectJ compiler.
aspect TraceSet {
    pointcut tracedCall(): execution(* *.Set(...));
    before(): tracedCall() {
        /* Trace the methods */
        id = Dispatcher.Bind(p, "TraceSet.trace");
    }
}

static void trace() {
    /* Trace the methods */
}

class TraceSet {
    static IPattern p = new Execution(new Method("*.Set"));
    static {
        Dispatcher.Bind(p, "TraceSet.trace");
    }
    static void trace() {
        /* Trace the methods */
    }
    static void Off() {
    }
    static void On() {
    }
}

static IPattern p = new Execution(new Method("*.Set"));

Figure 4. An example of a simple aspect

3.3 Patterns model pointcuts in Nu

Pointcuts in Nu are modeled as first-class objects of type IPattern (patterns). For example, the pointcut (line 2, Figure 4(a)) is now modeled as a first class entity of type Execution (line 2), which is a subtype of IPattern in Nu. The supporting framework for Nu provides this type and it is used to select method-execution join points. The constructor of the type Execution takes an argument of type Method, which is another type provided by the supporting framework for Nu. This type is used to select all method related join points. The Method type provides several constructors, one of which takes the name of the method (here a wildcard expression).

Each pattern selects a set of join points. If the pattern instance is used as an inner pattern for another pattern instance, the outer instance filters the join points selected by the inner instance according to some criteria. In the example above, Method("*.Set") is used as an inner instance to the Execution pattern. The outer instance selects all method related join points where the method name is Set. The outer instance selects the method-execution join points from among those join points.

3.4 Bind/Remove in Nu

The Nu language model [24, 26] adds two new primitives to object-oriented intermediate languages: bind and remove. These primitives expect a pattern and reference to a method as arguments. In the .NET framework version of Nu, this reference is modeled as a delegate. In the JVM version of Nu, we are currently just using the name of the method. The runtime infrastructure creates a reference to the method based on this name. This is an engineering limitation of the current version that will be fixed in later versions. From hereon, we will refer to this reference as the delegate to the method or simply delegate.

The bind primitive associates the supplied delegate with the join points matched by the pattern. As a result, the delegate is invoked when the program execution reaches the join point. Remove eliminates this association. The example in Figure 4(b) uses the bind primitive provided by Nu to associate the delegate to the method trace() (line 4) to execute at the join points selected by the pattern (line 2). After the bind call is complete, this association is active until it is explicitly deactivated by calling remove, which is not shown in this example. The example creates the association in the static initializer of the class TraceSet.

The example in Figure 4(c) demonstrates runtime advising in Nu. The class TraceSet allows turning tracing of join points on and off at runtime using the On and Off methods, respectively. When On is called, the trace method is associated with all method execution join points selected by the pattern.

Inter-type declarations are currently not supported in Nu. The rationale behind this decision is discussed in detail in Section 4.2.

4. Modularity-Preserving Compilation Strategies

In this section, we describe modularity-preserving compilation strategies for a significant subset of AspectJ constructs into our base intermediate language. The base intermediate language consists of the two new constructs bind and remove as well as core patterns for selecting join points. The bind and remove constructs are represented as Java native methods implemented inside a modified Java Hotspot virtual machine. The patterns are represented in the form of Java classes provided as a runtime library.

For the ease of readability, all examples presented of our intermediate language are shown as high-level Java source. This source code could be generated from their AspectJ versions by a compiler implementing our strategies. The actual intermediate representation would be the compiled Java byte code containing calls to the native bind and remove methods. Similarly, the examples being transformed are given as high-level AspectJ source instead of their compiled Java byte code. Please note that the comparisons being made are actually of the Java byte code versions of each example.

4.1 Aspect and Advice Strategies

The compilation strategy for aspect and advice for the example in Figure 4(a) was previously demonstrated in Figure 4(b). The aspect TraceSet is transformed into an object-oriented class. This is similar to how AspectJ currently compiles aspects by creating a class for each aspect that contains all advice defined in the aspect as methods. The generated class TraceSet contains the advice (lines 3–5, Figure 4(a)) as the object-oriented method trace() (lines 6–8, Figure 4(b)). For languages such as CaeserJ [21, 22], AspectWerkz [5], and Eos [27], this transformation will not be necessary since these languages already model aspects as classes at the source level.

4.2 Inter-Type Declaration Strategies

The AspectJ language provides constructs to allow aspects to declare new methods or fields in another type, declare a type extends a new class, or declare a type implements new interfaces. These are inter-type declarations. An example from the AspectJ
public void m2() {...}

The compilation strategy for inter-type declarations involves directly adding the declarations to the class that it crosscuts. In cases where the declaration affects more than one class, this will require modifying multiple files. Clearly, this strategy is not modular since where the declaration affects more than one class, this will require directly adding the declarations to the class that it crosscuts. In cases where the declaration affects only one class, so the effect on typical uses of inter-type declarations, such as those in Figure 5, afflicts very few classes (and often only one class), so the effect on incremental compilation will be minimal.

Inter-type declarations could also be emulated using partial classes in C# 2.0 [8]. Figure 6 shows an example of using partial classes. The class Class1 is defined in File1.cs (Figure 6(a)) and contains method m1. The method m2 can be introduced into Class1 (Figure 6(b)) by using a partial class declaration in another file, say File2.cs. The partial class also declares that Class1 now implements interface IExample. When compiled, the class Class1 will contain both methods m1 and m2 and implement the interface IExample. Partial classes however require both class declarations to have the partial keyword and do not allow modification of a class’s inherited class.

We realize this solution is not completely satisfactory, however, to keep our intermediate language as simple as possible, we did not consider further extensions to support inter-type declarations.

4.3 Control Flow Construct Strategies

In this sub-section, we describe compilation strategies for control flow constructs.

4.3.1 Cflow Construct

The AspectJ language provides a construct called cflow to separate crosscutting concerns based on the control flow of the program. The AspectJ programming guide [1] informally defines a cflow pointcut as follows: The cflow pointcut picks out all join points that occur between entry and exit of each join point P picked out by Pointcut, including P itself. Hence, it picks out the join points in the control flow of the join points picked out by Pointcut. One current compilation technique for cflow constructs [1] works as follows. First, generate a stack in the aspect-like constructs. Second, insert instructions to push and pop a unique control flow identifier into this stack at the entry and exit of the method Word.Set(). Third, insert instructions to check whether that unique control flow identifier is present on the stack at every point in the program where a call to the method Bit.Set() is possible [20].

An example usage of this pointcut expression is shown in Figure 7. In this example, the aspect Counting uses the cflow construct to count the number of calls to the method Bit.Set() in the control flow of the method Word.Set(). The pointcut expression will select all calls to the method Bit.Set() that occur between entry and exit of the method Word.Set().

The compilation strategy for the cflow construct uses a simple pattern. First, generate two new methods, say cflowBind and cflowRemove, making sure that the names are unique in the class (since the class may already contain other methods). Second, bind these two methods to execute at the entry and exit of the method Word.Set(), respectively. Third, generate code in cflowBind and cflowRemove to bind and remove the code to the actual advice to execute whenever Bit.Set() is called. A stack is used to track multiple bind calls to Word.Set(), allowing the code to remove the proper association. Figure 7(b) shows the results of the transformation of the example program in Figure 7(a).

For this transformation to work correctly for multi-threaded programs, a per-thread semantics of bind and remove needs to be developed. The reason follows from the following scenario. Imagine a system with two threads running concurrently. If the first thread is executing Word.Set, then any call to Bit.Set that occurs prior to the return of Word.Set is under the control-flow of Word.Set. If the second thread makes a call to Bit.Set before executing Word.Set at that time, a bind exists for Bit.Set. If the semantics were not defined on a per-thread basis, this bind would be active for the second thread as well and the counting advice would execute.

To simplify the semantics, the following restriction may be imposed on it: A bind should only modify the join points in the context of the calling thread and only the thread that called bind is able to remove the associations that it created. The termination of a thread causes all associations created by that thread to be automatically removed, since reaching a join point in the context of that thread is now impossible. In the future, we will fully explicate these semantics.

4.3.2 Cflowbelow Construct

AspectJ also provides a cflowbelow construct which is similar to the cflow construct, except it does not pick out the join points...
class Counting {
    static Stack counter;
    private static Stack ids;
    private static int initialDepth;
    static {
        Dispatcher.Bind(new Execution(new Method("Word.Set")), "Counting.cflowBind");
        Dispatcher.Bind(new Return(new Method("Word.Set")), "Counting.cflowRemove");
        Dispatcher.Bind(new Failure(new Method("Word.Set")), "Counting.cflowRemove");
    }
    public static void countCalls() {
        Dispatcher.Remove(((Integer)ids.pop()).intValue());
    }
    private static void cflowRemove() {
        initialDepth = Thread.currentThread().countStackFrames();
    }
    public static void cflowBind() {
        DISPATCHER.BIND.getCounting.calls();
        count++; 
    }
}

public static void logSavingsWithdrawal(JoinPoint thisJP) {
    if (!(thisJP.getTarget() != null && thisJP.getTarget().class.equals(SavingsAccount.class)))
        return;
    Dispatcher.Remove(((Integer)ids.pop()).intValue());
    count++;
}

Figure 8. The transformed program without cflowbelow

4.4 Dynamic Checking Construct Strategies

AspectJ also provides constructs called this, target, and args that can be used to match join points based on dynamic types or values. The AspectJ programming guide [1] informally defines the this, target, and args constructs as picking "each join point where the this object (the object bound to this), target object (the object on which a method is called or a field is accessed), and arguments are [an] instance of a particular type." The if construct is defined as picking "join points based on a dynamic property. Its syntax takes an expression, which must evaluate to a boolean true or false." This sub-section presents the compilation strategies for these constructs.

4.4.1 Target Construct

An example usage of the target construct is shown in Figure 9(a). The aspect uses a call construct to select calls to all methods named withdraw. It then uses the target construct to select only those join points where the target object (the object withdraw is being invoked on) is of type Account and logs the call.

aspect WithdrawLogger {
    before(): call(* *.withdraw(..)) && target(Account+) {
        /* log the withdrawal */
    }
    before(): call(* *.withdraw(..)) && target(SavingsAccount) {
        /* log the withdrawal from savings */
    }
}

(a) An example usage of the target construct

Figure 9. The target construct

The compilation strategy for the target construct is to generate code in the advice body that uses reflective join point information to check that the target object is of the appropriate type. An example of this transformation applied to the program in Figure 9(a) is shown in Figure 9(b). Since target allows use of the sub-type operator, there are actually two possible transformations. The first transformation is when the sub-type operator is present and is shown on line 7. The generated check uses instanceof to attempt to match the join point target's type to the type specified. The instanceof operator automatically handles sub-type matching. The second transformation is when the sub-type operator is not present and is shown on lines 12–13. The generated check gets the Java Class of the join point target and compares it to the Java Class of the type specified. Since this transformation is not using instanceof it must take care to handle the case when the target is actually null (for instance, when the method is static). The instanceof operator safely returns false when the left operand is null so this additional check is not needed.

4.4.2 This Construct

The compilation strategy for the this construct is identical to the target strategy: generate code in the advice body that uses reflective join point information to check that the currently executing object is of type Account. An example is similar to Figure 9(a) (replacing target on line 2 with this). The resulting compiled version is similar to Figure 9(b) (replacing getTarget() on lines 7,12,13 with getThis()).

4.4.3 Args Construct

Programs using the args construct can be transformed by generating code in the advice body that uses reflective join point information to check if the currently executing object has arguments matching the parameters of the construct.

2 A Java Class is an object-oriented representation of the class’s type. Every type in Java has a Class object associated with it.
8 aspect WithdrawLogger {
9 before(Double amount): call(* Account.withdraw(..))
10 /* log the withdrawal */;
11 }
12 }
13 }  
(a) An example usage of the \texttt{args} construct

11 class WithdrawLogger {
12 static {
13 Dispatcher.Bind(new Call(new Method("Account.withdraw")),
14 "WithdrawLogger.logWithdrawal");
15 }
16 public static void logWithdrawal(JoinPoint thisJP) {
17 if (((thisJP.getArgs()[0].length > 0 &&
18 (thisJP.getArgs()[0]) instanceof Double)))
19 return;
20 Double amount = (Double)(thisJP.getArgs()[0]);
21 /* log the withdrawal */
22 }
23 }
24 }  
(b) The transformed program without \texttt{args}


Figure 10. The \texttt{args} construct

The example in Figure 10(a) shows an aspect that uses the \texttt{args} construct (line 3). The generated class shown in Figure 10(b) inserted a test in the advice body that uses reflective information to verify the first argument of the join point is of type \texttt{Double} (lines 7–9). If a join point calls the advice and the first argument of the join point is not of type \texttt{Double}, the original advice body will not execute.

A problem arises when deploying this strategy for Java versions of \texttt{Nu} since Java lacks a unified type system. If the argument was actually of primitive type \texttt{double}, it would automatically be boxed by the framework to the type \texttt{Double} when stored in the argument array. This makes it difficult to target arguments that are of primitive types. A simple solution to this problem would be to have the framework box each primitive type to a custom class instead of the built-in Java class for that primitive. Note that since \texttt{Nu} has a unified type system, this problem does not occur in \texttt{Nu} implementations of \texttt{Nu}.

One of the more useful benefits of using the \texttt{args} construct is the ability to bind context information. This allows access to the arguments through locally named variables and saves the user the effort of retrieving the argument from the arguments array and casting it to the appropriate type. An example of binding context information is given in Figure 10. The aspect binds the first argument to \texttt{amount} (lines 2–3). This allows access to the method's first argument in the advice body through the variable named \texttt{amount}. The compilation strategy is to generate code that performs this binding inside the advice body. The transformed class binds the argument to the variable \texttt{amount} (line 10), taking care to cast the argument appropriately.

4.4.4 If Construct

An example usage of the \texttt{if} construct is shown in Figure 11(a). The aspect \texttt{WithdrawLogger} will log the call to the method \texttt{withdraw} if the boolean value of \texttt{logging} is true and if the amount being withdrawn is larger than zero. This example also demonstrates our previous strategy for compiling \texttt{args} with bound context.

Programs using the \texttt{if} construct can be transformed by generating code in the advice body that performs the check(s). An example of this transformation is shown in Figure 11(b). The \texttt{if} construct generates a conditional test (lines 11–12) in the advice body of the generated class that returns if the test evaluates to false. The example also makes use of the strategy for \texttt{args} (lines 8–10) presented earlier, allowing the conditional test to use the bound context variable amount (which was bound on line 11).

4.5 Instantiation Construct Strategies

By default, AspectJ creates one instance of each aspect in the system. Creation of multiple instances of an aspect is achieved through constructs like \texttt{perthis}, \texttt{pertarget}, \texttt{perflow}, and \texttt{perflowbelow}. The transformation of these constructs is presented in this section.

4.5.1 Perthis Construct

The AspectJ programming guide [1] informally defines \texttt{perthis} as associating an instance “with each object that is the currently executing object at any join point in Pointcut.”

An example usage of the \texttt{perthis} construct is shown in Figure 12(a). In this example, the aspect \texttt{LogWithdrawals} creates a
new instance for each unique object at the `Account.withdraw()` execution join point (lines 1–2). Then before the execution of `Account.withdraw()`, logs the withdrawal using the aspect instance matching `thisJP.getThis()`. The compilation strategy for `perthis` constructs follows. First, create a map that takes an object and returns an instance of the aspect (an IdentityHashMap is used for reference equality). Second, generate a uniquely named method, such as `makeInstance()`, that creates an instance of the aspect and stores it in the map using `thisJP.getThis()` as the key. Third, `bind` the creation of `Account` objects to `makeInstance()` in the static initializer. Fourth, for each advice body, generate a uniquely named instance method that performs the advice. Finally, for each advice body, generate a uniquely named static method that retrieves from the map the correct instance of the aspect and calls the appropriate generated instance method. An example of this transformation applied to Figure 12(a) is shown in Figure 12(b).

While it is not possible to create multiple instances of an aspect in AspectJ without using a construct such as `perthis`, a similar transformation is possible but requires creating a second class that contains methods for all of the aspect’s advice. Then the advice body will simply use the map to obtain an instance of this helper class and call the advice methods. The advice bodies become proxies to the generated methods in the helper class. This approach, however, is less modular than the transformation in Nu.

### 4.5.2 Pertarget Construct

The `pertarget` construct, which is informally defined by the AspectJ programming guide [1] as associating an instance “with each object that is the target object at any join point in Pointcut”, may be transformed similarly to `perthis`. Instead of calling `getThis()` in the instance delegate and delegate advice wrapper (lines 10 and 13 respectively, Figure 12(b)), however, we call `getTarget()`.

### 4.5.3 Percflow Construct

AspectJ also offers per control flow instantiation of aspects using the `percflow` construct. The `percflow` construct is informally defined in the AspectJ programming guide [1] as creating an instance of the aspect “for each entrance to the control flow of the join points defined by Pointcut.”

An example program using `percflow` is shown in Figure 13(a). Each time the control flow of the program enters `Account.withdraw()`, an instance of the aspect will be created and when the control flow exits `Account.withdraw()` that instance will become available for garbage collection. Instances of the aspect will create a unique file and during the lifetime of the aspect write a trace to that file of any call to a method in `Account`.

The compilation strategy for programs with the `percflow` construct is similar to the `cflow` strategy previously described in Section 4.3.1. An example of this transformation applied to the program in Figure 13(a) is shown in Figure 13(b). Since a single thread is or is not in the control flow of a join point, there will be at most one instance of the aspect (per-thread) at any given moment. To transform the aspect, first generate two new uniquely methods, say `cflowBind()` and `cflowRemove()` (lines 13–24). Second, `bind` these two methods to execute at the entry and exit of the method `Account.withdraw()`, respectively (lines 6–11). Third, generate code in `cflowBind()` to create an instance of the aspect if the stack is empty (lines 14–15). Fifth, generate a constructor for the aspect that `binds` the current aspect instance’s advice to the join point. Sixth, generate a uniquely named method, say `percflowRemove()`, to `remove` the association between the advice and the join point. Finally, generate code at the end of the aspect `LogWithdrawalTrace percflow(withdrawal) {` to call `percflowRemove()` on the instance and then set the instance to `null` if the stack is empty (line 20–23).

As was the case with the `cflow` and `cflowbelow` strategies shown in Section 4.3, to work correctly for multi-threaded programs a per-thread semantics of `bind` and `remove` needs to be specified.

### 4.5.4 Percflowbelow Construct

The `percflowbelow` compilation strategy is similar to the `percflow` strategy. The difference is the generation of extra code to track the depth of the stack and is shown in Figure 14 (lines 5, 17, 36–37). This is the same technique used by the `cflowbelow` strategy in Section 4.3.2. Inside the advice body, a check is generated to determine if the stack depth is greater (lines 36–37). If the stack depth is greater, we are `below` the control flow of `Account.withdraw()` and may continue executing the advice body.

As was the case with the `cflow`, `cflowbelow`, and `percflow` strategies, to work correctly for multi-threaded programs a per-thread semantics of `bind` and `remove` needs to be specified.

### 4.6 Composition Operator Strategies

The AspectJ language provides operators such as `&&` (conjunction) and `||` (disjunction) for complex pointcut expressions. The informal meaning of the conjunction operator is to pick out each join point that is matched by both pointcuts. The informal meaning of the disjunction operator is to pick out each join point matched by either pointcut.
1. aspect LogWithdrawalTrace percflowbelow(withdrawal()) {
2.   pointcut withdrawal(): execution(* Account.withdraw(...));
3.   before(): call(* Account.*(...)) {
4.     /* Store a trace for this withdrawal in its own file */
5.   } }
6. }

(a) An example usage of the percflowbelow construct

1. class LogWithdrawalTrace {
2.   private static LogWithdrawalTrace instance = null;
3.   private static Stack counter = new Stack();
4.   private int id;
5.   private static int initialDepth;
6.   static {
7.     Dispatcher.Bind(new Execution(new Method("Account.withdraw")),
8.       "LogWithdrawalTrace.cflowBind");
9.     Dispatcher.Bind(new Return(new Method("Account.withdraw")),
10.    "LogWithdrawalTrace.cflowRemove");
11.    Dispatcher.Bind(new Failure(new Method("Account.withdraw")),
12.    "LogWithdrawalTrace.cflowRemove");
13.   }
14.   private static void cflowBind() {
15.     if (counter.size() == 0) {
16.       instance = new LogWithdrawalTrace();
17.       initialDepth = Thread.currentThread().countStackFrames();
18.     }
19.     counter.push(instance);
20.   }
21.   private static void cflowRemove() {
22.     counter.pop();
23.     instance = null;
24.   } }
25.   public void percflowRemove() {
26.     Dispatcher.Remove(id);
27.   }
28.   public void percflowbelow(withdrawal()) {
29.     Dispatcher.Bind(new Call(new Method("Account.*")),
30.       "LogWithdrawalTrace.trace", this);
31.     id = Dispatcher.Bind(new Call(new Method("Account.*")),
32.       "LogWithdrawalTrace.cflowBind");
33.     Dispatcher.Bind(new Execution(new Method("*.remove")),
34.       "LogModifications.log");
35.     Dispatcher.Bind(new Execution(new Method("*.remove")),
36.       "LogModifications.log");
37.   }
38.   public void percflowbelow(withdrawal()) {
39.     Dispatcher.Remove(id);
40.   }

(b) The transformed program without percflowbelow

Figure 14. The percflowbelow construct

1. aspect LogModifications {
2.   after(): execution(*.add(...) || execution(*.remove(...) { 3.     /* Log the modification */
4.   })
5. }
6. }

(a) An example usage of the || operator

1. class LogModifications {
2.   static {
3.     Dispatcher.Bind(new Execution(new Method("*.add")),
4.       "LogModifications.log");
5.     Dispatcher.Bind(new Execution(new Method("*.remove")),
6.       "LogModifications.log");
7.   }
8.   public static void log() {
9.     /* Log the modification */
10.   }
11. }

(b) The transformed program without ||

Figure 15. The || operator

An example usage of the disjunction operator is given in Figure 15(a). The LogModifications aspect uses a disjunction operator to select the execution join points of all methods named either add or remove and logs their execution.

Programs using the disjunction operator, such as the one in Figure 15(a) can be transformed into programs not using the disjunction operator. Two Bind calls are created in the static initializer of the class containing pointcut. Both Bind calls use the log() method as a delegate. An example of this transformation applied to the program in Figure 15(a) is shown in Figure 15(b).

Elimination of the disjunction operator is possible in Nu since multiple patterns may be bound to the same delegate. The delegate will only execute once if any of the patterns bound to it matches the join point. Transforming the disjunction operator in AspectJ is still possible, however, the code of the advice would either have to be duplicated or moved to a helper method which is called from the advice bodies. A problem arises in AspectJ after this transformation when both operands match the join point, as the advice would then be called twice. This problem does not occur in Nu since a delegate will only be invoked at most one time for each matching join point instead of once for each matching pointcut.

We have already shown how to transform the conjunction operator when one of the operands is this, target, args, or if in Section 4.4. We have also shown how to transform the conjunction operator when one of the operands is cflow or cflowbelow in Section 4.3. At this time however, we do not have a formal strategy to handle every possible use of conjunction. We believe that for most cases, the compilation strategy for conjunction simply becomes applying previously defined strategies repeatedly, but have not verified this. Due to a lack of a precise strategy for conjunction, we also were unable to verify a strategy for complex aggregations of conjunction and disjunction. It is still possible that conjunction will need to be introduced as a pattern in our intermediate language and if so, disjunction may need introduced into the intermediate language as well.

4.7 Putting it all together

A more realistic example containing multiple high-level constructs is shown in Figure 16(a). The aspect has a perthis instantiation model for each instance of Account (line 1). It defines a pointcut to log that a transfer was required in order to make a withdraw from an account (line 4), but only if the static variable logAll is true.

Compilation of this aspect requires applying several previously defined compilation strategies. The result is given in Figure 16(b). First, a method for the advice is generated with the name beforeCallTransfer(). Second, the strategy for cflow is applied which creates two new methods cflowBind() and cflowRemove() to bind and remove the delegate to the call of Account.transfer(...). Third, the if compilation strategy is applied. This generates a conditional check inside the delegate method beforeCallTransfer() (line 21–22). Fourth, the strategy for args is applied. This generates another conditional check inside the delegate method. As an optimization, this check was combined with the one created in the previous step. Code was also generated for the bound context information (line 23). Finally, the strategy for perthis is applied, creating static proxy methods for each previously generated method in the class (which are now instance methods).

4.8 Summary

In this section, we described compilation strategies for 10 AspectJ constructs and 2 operators into our intermediate language. We provided transformations for control flow, dynamic checking, and instantiation constructs. Correctness of the control flow transformations for multi-threaded programs require the development of a per-thread semantics of Nu.
5. Evaluation

This section evaluates our proposed compilation strategies. We evaluate the modularity of the high-level constructs after applying our transformations, the decrease in incremental compilation times due to the maintained modularity, and the runtime efficiency of the transformations.

5.1 Maintained Modularity

Our main claim is that the compilation strategies we propose maintain the modularity of the original source code in object code. To illustrate, let us consider the compilation of the AspectJ aspect shown in Figure 16(a). The compiled object code is presented in Figure 17(a). The aspect has contained the concerns to one module at the source code level, however, after compilation those concerns are now scattered and tangled with other modules. The modularity of the source code is thus lost in the object code.

The object code for the Nu version of Figure 16(a), which was generated using our compilation strategies from Section 4, is shown in Figure 17(b). Notice how the modularity maintained by the source code is still present in the object code.

We applied the examples given in Section 4 to the Azureus software package. A small change was made in each file and an incremental compilation performed with the AspectJ incremental compiler. We considered a small change to be a textually local change in the source file, such as changing the pointcut `this(Account)` to `!this(Account)`. Figure 18 shows the number of files modified for each construct by the incremental compiler as a result of a small change in only one file. In each case, the Nu versions of

![Figure 16. A larger example showing perthis, cflow, if, args, and conjunction](image-url)
5.2 Improved Incremental Compilation

To evaluate our claim of lower incremental compilation times for AO programs using our compilation strategies, we used the AspectJ compiler ajc to compile AspectJ aspects and their transformed Nu counterparts. Again, we used the Azureus software package and ajc version 1.5.0 (at the time of this paper, version 1.5.2 of the AspectJ compiler was available, however, we did not use the newer version due to problems when performing incremental compilations). All tests were performed on a 3.8 GHz Pentium 4 with 3.5GB of memory.

Using modified versions of the examples given in Section 4 (modified so the pointcuts target classes and packages in the Azureus system), we took the average of 100 incremental compilation times of a minor change in each file. Figure 19 shows the results of our tests. Compile times for the AspectJ versions took ten to thirty times longer than their transformed counterparts. These results are consistent with our previous work [7].

Note that the times shown were the incremental compilation times of AspectJ source code to Java byte code for the AspectJ versions and of Java source code to Java byte code for the transformed counterparts. A fairer comparison would also include the time to read and transform the AspectJ source code in the Nu compilation times.

5.3 Runtime Efficiency

We believe our compilation strategy for cflow offers improvement over current strategies. All the generated code fragments are contained in one module, thereby improving traceability of the construct. The overhead of cflow occurs at two locations in a program: at the entry and exit of the join point we are tracking the control flow of and at all join points that need to know about that control flow. Current strategies use counters or stacks to track the entry and exit of the join point and then check those counters/stacks at join points to determine if they are in the control flow. Our approach also uses a stack to track entry/exit of the join point. There is additional overhead in the form of one bind and one remove call. Our strategy, however, generates code that requires no checks at other join points. In cases where there are a large number of join points under the control flow in question, our strategy will clearly be superior. In the remaining cases, the overhead associated with bind and remove comes into play. Current performance of Nu JVM w.r.t. stock JVM is shown in Figure 20. The X-axis shows various parts of the Java grande benchmark. The Y-axis shows the method calls per second. The performance is comparable in some cases; however, in others there is a significant overhead. We are currently working on reducing these overheads.

There is a performance tradeoff in some cases. For example, in the case of dynamic checking constructs (this, target, args, if), instead of performing the check before calling the advice, the application will now call the advice and then perform the check. As a result, a few more calls to the advice will be made than necessary, introducing an overhead. This overhead also exists in the version of ajc we used. However, an adaptive optimization technique [13] may be developed that can optimize the performance critical sections of the program, leaving the rest amenable to separate and incremental compilation.
6. Related Work

Gray and Roychoudhury [12] demonstrated aspect weaving using a program transformation engine. They were able to perform aspect weaving in languages that do not support aspects by modeling aspect constructs as transformations. Our approach is similar in that both use transformations to model AO constructs, however, our approach is not a weaving technique. Instead, our approach models the constructs using bind and remove primitives supported by the Nu virtual machine.

Ligatti et al [18] describe the semantics of MinAML by providing a core language with join points and advice. This is similar to our goal of stripping down the language in Nu, however, MinAML extends their semantics to allow for dynamic constructs such as cflow. We have shown that these constructs are not needed and believe we will be able to model the semantics of constructs such as cflow without modifying the semantics of our core language.

Steamloom [3] advocates moving weaving inside the virtual machine (VM), which also allows preservation of the modularity of the source code at the intermediate language level. This is similar to our approach in that both advocate support for aspect-orientation at the VM level; however, the focus of our approach is on intermediate language design to support compilation of high-level AO languages whereas the focus of Steamloom project is on improving the efficiency of aspect-oriented virtual machines.

In a recent work, Bockisch et al describe a more efficient implementation of the cflow construct [4]. Their approach advocates having “direct access to the internal structures of the VM running an application, such as the call stack, as well as the integration of these techniques into the just-in-time compiler.” The counter and guards techniques they implemented in Steamloom showed significant improvements in the execution speed of control-flow constructs. Their approaches, however, still incur overhead at dependent join point shadows to check if the current join point is in the control-flow of the constituent join point. Our approach incurs no overhead at dependent join points that are not in the control-flow of the constituent join point and only the overhead of invoking the delegate when in the control-flow. Our approach does, however, incur some overhead at constituent join points in the form of invoking two delegates (one on entry and one on exit of the join point), one bind call, one remove call, and pushing/popping an identifier onto/to a stack. Depending on the exact implementation of Nu, this overhead could be significant. We are currently investigating methods to efficiently support Nu.

7. Conclusion and Future Work

In this work, we described modular and textually local compilation strategies for high-level AO languages to our intermediate language Nu and showed that these techniques improved the efficiency of aspect-oriented incremental compilation. Some of these compilation strategies such as that for this, target, etc, can also be applied in existing aspect language compilers heuristically in cases where the cost of updating a large number of join points is significant. Other compilation strategies such as that for cflow, cflowbelow show new possibilities enabled by simple combinations of the Nu primitives. Proving semantic equivalence of theAspect construct and the transformed code in Nu is the next logical step of our work. To do so, we will develop a formal semantics of Nu, including the per-thread semantics and the semantics of our compilation strategies. Techniques similar to those used for validating compiler optimizations [14, 25] can also be used. A declarative style of specifying program transformations can also be used to represent these transformations [30]. Giving simple transformations of code and eliminating certain constructs can potentially make modeling the semantics of AO languages easier. For example, let us consider the semantics of Pitλ [10]. Endoh et al [10] first developed the semantics for what they considered the core part of Pitλ, which was only the call and reception join points. They then build on the core semantics and added constructs such as cflow. By demonstrating cflow as a composition of Bind and Remove calls using Execution and Return pointcuts, there is no need to modify the base semantics to accommodate the cflow construct.

8. Acknowledgements

We would like to thank the anonymous reviewers of the POPL 2007 conference for their helpful comments. We would also like to thank Juri Memmert for helpful discussions and comments on the draft.

References


