Splice: Aspects that Analyze Programs

Sean McDirmid and Wilson C. Hsieh
School of Computing, University of Utah
50 S. Central Campus Dr. Salt Lake City UT 84112, USA
{mcdirmid,wilson}@cs.utah.edu

November 9, 2004

Abstract

This paper describes Splice, a system for writing aspects that perform static program analyses to direct program modifications. The power of an inter-procedural data-flow analysis enables an aspect to examine the flow of data around a program execution point when it determines what code to add or change at that point. For example, an aspect can change the target set of an iteration based on how elements are skipped during the iteration. Splice aspects are written in a rule-based logic programming language with features that help aspect programmers express analyses. We show how a prototype of Splice is used to write two useful aspects in the areas of domain-specific optimization and synchronization.

1 Introduction

Aspect-oriented programming (AOP) systems, such as AspectJ [10], provide mechanisms for organizing code that go beyond conventional objects and functions. An aspect can modify the execution of a program at well-defined points in its execution, which are known as *join points*. The description of join points in AOP systems has generally been limited to their syntactic structure. For example, a set of join points can be described as calls to a particular set of methods. However, current aspect systems cannot describe a join point that depends on what can happen before or after the join point. This paper shows why such join points are useful and how they are supported in Splice, a system for writing aspects that can use static program analysis to identify join points. As a simple example, consider Java code that raises salaries of employees who are programmers:

```
-Iterator i = employees/*jp-1*/.iterator();/* original    code */
+Iterator i = programmers      .iterator();/* transformed code */
 while (i.hasNext())
 { Employee e = (Employee) i.next();
   if (e.isProgrammer()) e.raiseSalary(+20)/*jp-2*/; }
```

The efficiency of this loop can be improved by maintaining and iterating over an extent set of programmers rather than employees. Although this transformation can be performed by hand, implementing it as an aspect could improve modularity by separating a performance concern from base functionality. The aspect would add the `programmers` set to the program and manipulates the program's code at multiple join points. Two of these join points, labeled `jp-1` and `jp-2`, can only be identified with the help of a program analysis. Join point `jp-1` is associated with the `employees` field access used to create an iterator. Identifying `jp-1` requires a data-flow analysis because the aspect must discover that each element of the iterator is only updated when an `isProgrammer` method call on the element is true. When `jp-1` is identified, an aspect can replace the `employees` field access with a `programmers` field access. Join point `jp-2` is used to identify `jp-1` by determining when an iterator element is updated, which occurs when a method call can update the field of an `Employee` object. In this example, a call to the `raiseSalary(int)` method can call another method, that sets a `salary` field. Identifying `jp-2` requires an inter-procedural analysis because an aspect must discover that the method being called can either update an instance field or call a method that can update an instance field.

The need for AOP systems with both data-flow and inter-procedural join point identification mechanisms has been mentioned before by Kiczales [9]. Reasoning about inter-procedural data-flow information requires the use of a static

program analysis based on abstract interpretation [2], which are commonly used for compilation. Our approach in Splice is to enable an aspect to encode an analysis that determines how a program's execution is transformed by the aspect.

An aspect in Splice is expressed as a collection of rules in a logic programming language with temporal and negation operators. Aspect rules control the propagation of flow information during an analysis; i.e., they can concisely describe flow functions. Aspect rules also operate over a program representation that helps programmers deal with the complexity of variable re-assignment, branches, and loops. The analysis specified by an aspect is automatically divided into multiple passes and stages, which transparently ensures the sound and complete evaluation of rules in the presence of negation. With these features, Splice aspects that perform useful tasks can be expressed with a manageable amount of code; e.g., a Splice aspect that implements the transformation in this section consists of nine rules and 35 logic terms.

The program analysis defined by an aspect is performed over a finite domain of facts that is computed by applying an aspect's rules to a program. A standard gen-kill iterative data-flow analysis algorithm is used to traverse all points in a program repeatedly until fix-points are reached. The analysis supports both forward and backward flows, and is also inter-procedural. Propagation of facts between procedures is flow-insensitive and context-insensitive to enhance scalability. Although flow-insensitive inter-procedural fact propagation sacrifices precision, an aspect can often use static type information to compensate for the precision loss.

We have used a prototype of Splice to write and test complete aspects in the areas of domain-specific optimizations and synchronization. The rest of this paper is organized as follows. Section 2 provides an overview of Splice. Section 3 demonstrates how the transformation in this Section can be implemented in Splice. Section 4 describes Splice's design and implementation. Section 5 presents an informal evaluation of Splice. Related work and our conclusions are presented in Sections 6 and 7.

2 Splice Overview

We introduce Splice by showing how it can be used to write an aspect that implements a synchronization policy. However, before we describe this synchronization policy and show how it is implemented in Splice, we present Splice's basic features.

2.1 Basics

Facts are used in Splice to represent knowledge about a program being processed by an aspect. For example, the fact `Locked([getBalance()],[Accnt.lock])` specifies that the lock `Accnt.lock` must be acquired around `getBalance()` method calls. Aspects are expressed as a collection of rules that manipulate a program by proving and querying facts about the program. A rule is composed out of rule variables, consequents, and antecedents in a form like `rule R0 [x] P(x): Q(x);`, which reads as "for any binding of rule variable `x`, consequent `P(x)` is a fact whenever antecedent `Q(x)` is a fact." Splice supports bottom-up logic programming, so unlike Prolog the consequents of a rule become facts as soon as the rule's antecedents are facts. By rule `R0`, `P(5)` is a fact whenever `Q(5)` is a fact, where *unification* binds `x` to `5` so that the structure of antecedent `Q(x)` is equivalent to the structure of fact `Q(5)`.

Predicates generalize facts like classes generalize objects; e.g., `Locked` is the predicate of fact `Locked([getBalance()],[Accnt.lock])`. Built-in predicates have names that begin with `@`. The built-in predicate `@now` is used to inspect the expressions of the Java program being processed. Consider the definition of a rule from an aspect that implements a synchronization policy:

```
rule S0 [var,lck,obj,args,mthd] Track(var,lck), Needs(var,lck):
                @now(var = obj.mthd(|args)), Locked(mthd,lck);
```

The antecedent `@now(var = obj.mthd(|args))` in rule `S0` identifies a call to a method (`mthd`) on an object (`obj`) with arguments (`args`) and a result (`var`). The body of a `@now` antecedent is expressed in a Java-like expression syntax that can refer to rule variables or Java identifiers. An exception to standard Java syntax is the use of the tail operator (`|`), which is used in `(|args)` to specify that `args` is the list of the call's arguments. For unification purposes, the expressions in a Java program are treated as facts, where Java constructs and variables are listed in brackets. When rule `S0` is applied to the Java expression `x = accntA.getBalance()`, rule variable `var` is bound to Java variable `[x]`, `obj` is bound to `[accntA]`, `mthd` is bound to method `[getBalance()]`, and `args` is bound to the empty list.

```
global Locked(method,variable); backward WillUse(variable);
forward Needs(variable,lock), Track(variable,lock;)
rule S0a [lck,var,mthd] Needs(var,lck), Track(var,lck):
            @now(var = *.mthd(|*)),  Locked(mthd,lck);
rule S0b [var0,var1,lck,args] Needs(var1,lck), Track(var1,lck):
       @nowi(var1,args), @contains(args,var0), Track(var0,lck);
```

Figure 1: Predicates and rules that are used to define `Needs` facts.

Although we refer to source code variables `[x]` and `[accntA]` in facts for example purposes, aspects do not directly process source code. Instead, aspects process a control-flow graph (CFG) of a procedure's expressions, where all expression results are assigned to unique temporary variables. The Java syntax bodies of `@now` antecedents are "compiled" to unify with expressions in this representation. A discussion of the program representation aspects process is presented in Section 4.1.

2.2 Temporal Reasoning

The synchronization policy implemented in this section specifies that a certain lock must be acquired before a method call whose result is derived from some protected state. To ensure that a call result remains consistent with the protected state, which could be updated in another thread, the lock must be held as long as the call result can be used. To implement this synchronization policy, an aspect must reason about how and when variables are used. Consider the following three lines of code that withdraw `[z]` amount of money from an account `accntA`:

```
A: x = accntA.getBalance();
B: y = x - z;
C: accntA.setBalance(y);
```

If lock `Accnt.lock` protects `getBalance()` method calls, `Accnt.lock` must be acquired before line `A` and released after line `C` because variable `[x]` is assigned at line `A` and variable `[y]`, which requires the same lock because it is derived from the value of variable `[x]`, is used at line `C`. Implementing this reasoning in Splice begins by identifying three kinds of facts from the problem description.

First, a `Locked(method,lock)` fact identifies a `method` whose call results must be protected by a `lock`. For the example code, we assume the fact `Locked([getBalance()],[Accnt.lock])` is true. Second, a `Needs(variable,lock)` fact identifies program execution points where a `lock` must be held to ensure a `variable` is consistent with some protected state. For the example code, the fact `Needs([x],[Accnt.lock])` should be true from after line `A` until just after line `B` to ensure that the lock is held when variable `[x]` can be used. To implement this behavior, `Needs` facts should *flow forward*; i.e., they should be propagated forward to following CFG points. Third, a `WillUse(variable)` fact identifies program execution points where a `variable` can possibly be used at future execution points. For the example code, the fact `WillUse([y])` should be true before line `C` because variable `[y]` will be used at line `C`. To implement this behavior, `WillUse` facts should *flow backward*; i.e., they should be propagated backward to preceding CFG points.

Predicate declarations involved in the definitions of `Needs` and `WillUse` facts are shown at the top of Figure 1. When a predicate is declared by an aspect, its flow and arguments are specified. The declaration `global Locked(method,variable)` specifies that facts of the `Locked` predicate have two arguments and a global flow, meaning they are true at any point in a program's execution. `WillUse` facts are declared with backward flows, and `Needs` and `Track` facts are declared with forward flows.

Rules `S0a` and `S0b` in Figure 1 define how `Needs` facts are proven. Rule `S0a` specifies that a variable is tracked (`Track(var,lck)`) and a lock needs to be held for the variable (`Needs(var,lck)`) when the variable is assigned from a call (`@now(var = *.mthd(|*))`) and the called method is associated with the lock (`Locked(mthd,lck)`). Wild card values (`*`) are placeholders for "any value." Rule `S0b` specifies that a variable `var1` is tracked and locked when a variable `var0` is tracked and locked, and `var0` is in the argument list of variables (`@contains(args,var0)`) used by the expression that assigns `var1` (`@nowi(var1,args)`). The `@contains` antecedent in rule `S0b` queries the membership of a variable (`var1`) in a list of variables (`args`). The `@nowi` antecedent in rule `S0b` unifies with all kinds

```
rule S1 [var,args] WillUse(var):
  Track(var,*), @nowi(*,args), @contains(args,var);
rule S2 [var,lck] !Needs(var,lck): !WillUse(var), Needs(var,lck);
```

Figure 2: Rules that define and use `WillUse` facts.

of expressions in the same format; e.g., the expression `y = x - z` in the example code binds `var1` to `[y]` and `args` to `([x],[z])`.

Rule `S1` in Figure 2 defines how `WillUse` facts are proven. It specifies that a variable will be used (`WillUse(var)`) when it is tracked (`Track(var,*)`) and the current expression (`@nowi(*,args)`) uses the variable in its arguments (`@contains(args,var)`). `Track` facts proven by rules `S0a` and `S0b` and queried by rule `S1` address an inefficiency in bottom-up logic programming where all facts are proven as long as they can be derived by the aspect's rules, even if those facts are not useful to the aspect. If `Track` facts were not used, rule `S1` would propagate `WillUse` facts anytime a variable was used in the program even though most of these facts would never be queried by other rules. `Track` facts also get around a restriction on how negation can be used in Splice, as will be discussed in Section 2.3.

A rule executes, or *fires*, when all of its antecedents are unified with facts or Java code. When a rule fires, it is associated with a *firing point* in a procedure's CFG. For any rule with a `@now` or `@nowi` antecedent, such as rules `S0a`, `S0b`, and `S1`, the firing point is the location of the expression that is unified with the antecedent. When applied to the example code, rule `S0a` fires at line `A`, `S0b` fires at line `B`, and `S1` fires at lines `B` and `C`. When a rule fires, the forward and backward-flow facts it proves becomes true after and a backward-flow fact becomes true before the firing point of the rule. When rules `S0a`, `S0b`, and `S1` are applied to the example code, fact `Needs([x],[Accnt.lock])` becomes true after line `A`, fact `Needs([y],[Accnt.lock])` becomes true after line `B`, fact `WillUse([x])` becomes true before line `B`, and fact `WillUse([y])` becomes true before line `C`.

2.3 Negation

Rule `S2` in Figure 2 specifies that a lock no longer needs to be held for a variable (`!Needs(var,lck)`) when the variable will no longer be used (`!WillUse(var)`) and the lock currently needs to be held for the variable (`Needs(var,lck)`). The negation of an antecedent or consequent occurs using the `!` operator. A negated antecedent is true when it does not exist as a fact. Fact `WillUse([y])` does not exist at and after line `C` in the example code at the beginning of Section 2.2, which restricts where rule `S2` can fire. A negated consequent stops a fact from being true immediately after or before the firing point if the fact respectively has a forward or backward flow.

Because rule `S2` does not have a `@now` or `@nowi` antecedent and its `Needs` consequent has a forward flow, it fires at the earliest point where its antecedents are true. The firing point of a rule depends on the flow of its consequents; e.g., the firing point of a rule with a backward-flow consequent is the latest point where its antecedents are true. In the example code, line `B` is the earliest point where `WillUse([x])` is false and `Needs([x],[Accnt.lock])` is true, and line `C` is the earliest point where `WillUse([y])` is false and `Needs([y],[Accnt.lock])` is true. Therefore, rule `S2` fires twice in the example code: at line `B` to negate the fact `Needs([x],[Accnt.lock])`, and at line `C` to negate the fact `Needs([y],[Accnt.lock])`.

The rules in Figures 1 and 2 collectively describe an analysis whose execution must occur in two stages to ensure soundness. The first stage consists of a forward analysis pass that fires rules `S0a` and `S0b` followed by a backward analysis pass that fires rule `S1`. The second stage consists of a forward analysis pass that fires rule `S2`. The analysis must execute in two stages because the rule `S2` negates a `WillUse` antecedent, so cannot fire until `WillUse` facts have been completely proven. Rules `S0b` and `S1` in Figure 1 also refer to `Track` antecedents rather than `Needs` antecedents to prevent an invalid cyclic dependency between `Needs` and `WillUse` facts that would be rejected as invalid in Splice. As we will describe in Section 4.2, Splice automatically detects invalid dependencies and divides an analysis into multiple stages.

2.4 Advice

Using facts defined by rules in Figures 1 and 2, additional rules can insert code into a program to acquire and release locks. First, we need some way of keeping track of whether or not a lock is held. Forward-flow `Has(lock)` facts can describe when a lock is held. Rules that insert lock acquisition and release calls are shown in Figure 3. Rule

```
forward Has(lock);
rule S3 [lck,mthd] Has(lck), @before(lck.[acquire]()):
        !Has(lck), Locked(mthd,lck), @now(*.mthd(|*));
rule S4 [lck] !Has(lck), @after(lck.[release]()):
                         Has(lck), !Needs(*,lck);
```

Figure 3: Two rules and a predicate that are used to insert synchronization code into a program.

`S3` adds code to acquire a lock (`@before(lck.[acquire]())`) and keeps track of the held lock (`Has(lck)`) when the lock is not already held (`!Has(lck)`) and a method associated with the lock (`Locked(mthd,lck)`) is called (`@now(*.mthd(|*))`). A `@before` consequent performs the before advice of AspectJ: it inserts code specified using a Java-like syntax at a new point that occurs immediately before the rule's firing point. Rule `S4` adds code to release a lock (`@after(lck.[release()])`) and keeps track of the release (`!Has(lck)`) when the lock is currently held (`Has(lck)`) and the lock no longer needs to be held for any variable (`!Needs(*,lck)`). An `@after` consequent performs the after advice of AspectJ: it inserts code at a new point that immediately occurs after the rule's firing point.

When the rules in Figures 1, 2, and 3 are applied to the example code at the beginning of Section 2.2, the following code results:

```
+  Accnt.lock.acquire();
A: x = accntA.getBalance();
B: y = x - z;
C: accntA.setBalance(y);
+  Accnt.lock.release();
```

Rule `S3` will fire at line `A` because lock `Accnt.lock` is not held. Rule `S4` fires after line `C` because neither fact `Needs([x],[Accnt.lock])`, negated by rule `S2` at line `B`, nor fact `Needs([y],[Accnt.lock])`, negated at line `C`, are true.

2.5 Assignments, Branches, and Loops

The example code described at the beginning of Section 2.2 does not contain branches, loops, or variable re-assignments. At first, these constructs appear to be very problematic because re-assignments create aliases, branches split and merge, and loops create an infinite number of values. However, an aspect programmer can often ignore these constructs because of the way a program is represented to the aspect (described in Section 4.1). Aspects can also specify how facts are propagated through splits and merges and inspect conditional branch tests (demonstrated in Section 3.1).

2.6 Inter-procedural Reasoning

In Section 2.2, we assumed the existence of the fact `Locked([getBalance()],[Accnt.lock])`. Although this fact can be manually asserted in an aspect, using rules to prove `Locked` facts is appealing because an aspect is more reusable when it hard-codes less information. `Locked` facts have a global flow so when a `Locked` fact is a consequent of a rule processing the code of one method, that same `Locked` fact can be used as an antecedent by another rule processing the code of another method. Proving a global-flow fact in one method and querying it in another method is how inter-procedural analysis occurs in Splice.

To prove `Locked` facts in this example, a programmer manually specifies fields they want protected by a specific lock. Any method that returns the value of these fields directly or indirectly will also be protected by the lock. Also, any method that returns a locked method's call result is transitively protected by the lock. Consider the following code for class `Account`:

```
class Account {
 int balance;
 int getBalance() { return this.balance; }
 void setBalance(int x) { this.balance = x; }
 int debit(int z) { int y = this.getBalance() - z;
                    this.setBalance(y);  return y; } }
```

```
rule S5 [lck,var,fld] Track(var), Needs(var,lck):
                @now(var=*.fld), Locked(fld,lck);
rule S6 [lck,mthd,var] Locked(mthd,lck):
 Needs(var,lck), @current(mthd,*), @now(return var);
rule S3 [lck,mthd0,mthd1] Has(lck), @before(lck.[acquire]()):
 !Has(lck), Locked(mthd0,lck), @now(*.mthd0(|*)),
 @current(mthd1,*), !Locked(mthd1,lck);
rule S7 [lck,fld] Has(lck), @before(lck.[acquire]()):
 !Has(lck), Locked(fld,lck), @now(*.fld=*);
```

Figure 4: Rules `S5` and `S6` that prove `Locked` facts and rule `S3` modified from Figure 3 to not acquire locks in locked methods.

We assume that the programmer specifies that account balance state is to be protected by manually asserting the fact `Locked([Account.balance],[Accnt.lock])` in an aspect (using the `fact` keyword). Since the method `getBalance()` returns the value of the `balance` field, a rule should prove the fact `Locked([getBalance()],[Accnt.lock])`. Since the method `debit(int)` returns a result derived from a `getBalance()` call, a rule should also prove the fact `Locked([debit(int)],[Accnt.lock])`. Rules `S5` and `S6` that implement this behavior are shown in Figure 4. Rule `S5` is like rule `S0` in Figure 1 except it identifies field access results, rather than method call results, that are protected by locks. Rule `S6` identifies the current method as being protected by a lock (`Locked(mthd,lck)`) whenever a variable that needs the lock (`Needs(var,lck)`) is returned by the method (`@now(return var)`). Antecedents of the built-in predicate `@current` describe the method currently being processed by the aspect and its arguments; a wild card in rule `S6` ignores the current method's arguments.

Without the modifications to rule `S3` in Figure 4, a lock would be acquired inside methods `getBalance()` and `debit(int)` because they access fields and call methods protected by the lock. Rule `S3` in Figure 4 is modified with two new antecedents that ensures the rule only fires in a method (`@current(mthd1,*)`) that is not already protected by the lock (`!Locked(mthd1,lck)`). Rule `S7` in Figure 4 ensures that a lock is acquired whenever a protected field is assigned. Unlike the case of reading state, the lock does not need to be held for any duration longer than the point where the field is assigned. When rule `S7` is applied to the above code, lock acquisition and release calls are made around the field store in method `setBalance(int)`, but calls to method `setBalance(int)` are not protected since they do not return protected state.

The synchronization aspect described in this section is ready to do useful work with only nine rules composed of about forty consequents and antecedents. However, many possible program behaviors are not addressed by this aspect. For example, this aspect does not address the storing of data protected by a lock into an unprotected field because the implemented synchronization policy does not mention this behavior. This aspect can serve as a starting point to implement synchronization policies with more advanced features.

3 Loop Compression

The example described in Section 1 is a transformation that we refer to as *loop compression*. A loop compression is enabled by a fact `Compress(original,compressed,filter)`, which adds and maintains a `compressed` set that contains those elements of an `original` set for which the result of a `filter` method call on the element is true. Loop compression replaces any iteration over the `original` set with an iteration over the `compressed` set if elements are updated only when the result of a `filter` method call on the element is true. Loop compression can be divided into three subtasks: identify contexts where iteration replacement should occur (Section 3.1), identify operations that can update state (Section 3.2), and add and maintain the compressed set (Section 3.3).

3.1 Replacement

To understand the organization of the replacement task, consider an elaboration of the example code in Section 1:

```
-L1: Iterator i = this.employees  .iterator();
+L2: Iterator i = this.programmers.iterator();
     while (i.hasNext())
```

```
global Updating(method), Compress(original,compressed,filter);
universal forward Filter(object,filter);
rule CT [obj,fltr]    Filter(obj,fltr):
 @true(obj.fltr()), Compress(*,*,fltr);
backward NonUpdated(iterator,filter);
rule NU [elm,it,fltr] NonUpdated(it,fltr):
 Compress(*,*,fltr), !Filter(elm,fltr),
 @past(elm=(*) it.[next]()), Updating(mthd), @now(elm.mthd(|*));
rule RP [it,obj,orig,cmps,fltr]
  @replace(it = obj.cmps.[iterator]()): !NonUpdated(it,fltr),
      @now(it = obj.orig.[iterator]()), Compress(orig,cmps,fltr);
```

Figure 5: Splice code that implements field replacement in a loop compression aspect.

```
 L3: { Employee e = (Employee) i.next();
 L4:   if (e.isProgrammer())
 L5:     e.raiseSalary(+20);             }
```

Loop compression rules should be able to replace line `L1` with `L2` by realizing that an element `[e]` of the iterator `[i]` is only updated when an `isProgrammer()` method call on it is true. Before being implemented in Splice, this description must be re-worded so that it is directly expressible in the temporal logic that Splice supports, where negation must be used to translate terms like "must," "only," and "unless." To do this, we can invert the problem description: if there is any situation where the `isProgrammer()` method is not called or its result is not conservatively known as true when an element of the iterator is updated, then replacement does not occur.

We first identify the kinds of facts involved in replacement. `Compress` and `Updating(method)` facts will be defined by rules in Section 3.2 and Section 3.3, but are needed to express rule antecedents in this section. `Filter(object,filter)` facts can describe the case where a `filter` method has been called on an `object` and the result of this method call is definitely true. `Filter` facts should have a forward flow because they indicate that the true path of a conditional has been taken. `Filter` facts should also be merged universally, so they do not remain true when the false path and true path of the conditional merge. Universal facts implement "meet over all paths" propagation semantics, meaning they are true at a point in a procedure's CFG only if they are true on all paths through the point. By default, facts implement "meet over any path" propagation semantics, meaning they are true at a point in a procedure's CFG as long as they are true on one path through the point. `NonUpdated(iterator,filter)` facts can describe the case where an element of an `iterator` can be updated when a `filter` method is not known to be true on that element. `NonUpdated` facts should have a backward flow because they convey what can happen in the future. Rules `CT` and `NU` in Figure 5 respectively prove `Filter` and `NonUpdated` facts. Rule `RP` in Figure 5 then uses these facts to implement field replacement.

Rule `CT` identifies a filter method whose result is true for an object (`Filter(obj,fltr)`) when the true path of a conditional branch is taken that tests the result of a filter method call on the object (`@true(obj.fltr())`). To prevent unneeded `Filter` facts from being proven, the filter method must also be involved in a compression (`Compress(*,*,fltr)`). The `@true` antecedent used in rule `U0` unifies with the first point of a true path for a conditional branch that tests its enclosed expression. `Filter` facts have a forward flow and are declared universal, so they are true at points along the true path of the conditional but becomes false as soon as the conditional's true and false paths merge.

Rule `NU` identifies an iterator whose elements are not updated when a filter method is true (`NonUpdated(it,fltr)`). This occurs when the next element of the iterator (`@past(elm = (*) it.[next]())`) is updated (`Updating(mthd)`), and the call result of a filter method on the iterator element is not known to be true (`!Filter(elm,fltr)`). The `Compress(*,*,fltr)` antecedent is also needed in rule `NU` because unification cannot bind rule variable `fltr` through the negated `Filter` antecedent. Antecedents of the `@past` predicate inspect expressions like `@now` antecedents, except an `@past` antecedent unifies with a statement located at a point that always executes before the rule's firing point.

Rule `RP` implements the field access replacement portion of the loop compression aspect. The `@replace` consequent in rule `RP` performs AspectJ-style around[1] advice by replacing code that occurs at the rule's firing point. Re-

[1] "Proceed" is subsumed in Splice by unification.

```
rule V0 [mthd,fld]  Updating(mthd):
                   @current(mthd,*), @now(this.fld = *);
rule V1 [mth0,mth1] Updating(mth1):
    @current(mth1,*),Updating(mth0),@now(this.mth0(|*));
```

Figure 6: Rules that generically prove `Updating` facts.

placement occurs at a field access used to create an iterator when a compression of the field exists (`Compress(orig,cmps,fltr)`) and the iterator's elements is not updated when the filter method is not true (`!NonUpdated(it,fltr)`). Assuming the existence of facts `Updating([raiseSalary(int)])` and `Compress([employees],[programmers],[isProgrammer()])`, rule `RP` will replace line `L1` with line `L2` in the example code because rule `NU` cannot fire at any point in this code.

3.2 Detecting Updates

To understand the organization of the update identification task, consider the following code that implements a part of class `Employee`:

```
class Employee
{ private int salary; ...
  void setSalary(int n)  { salary = n; }
  void raiseSalary(int n) { setSalary((salary*(100+n))/100); }}
```

Loop compression rules should be able identify method `setSalary(int)` as an updating method because it sets the `salary` field of class `Employee`. The method `raiseSalary(int)` should also be identified as an updating method because it indirectly sets the `salary` field by calling the `setSalary(int)` method. The task of identifying updating methods is similar to the task of identifying methods protected by a lock in Figure 4 of Section 2.6. Rules `V0` and `V1` in Figure 6 identify updating methods simply by identifying methods whose code can update instance fields (rule `V0`) or identifying methods whose code can call updating methods (rule `V1`). Applying rules `V0` and `V1` to the code for class `Employee` will prove the facts `Updating([setSalary(int)])` and `Updating([raiseSalary(int)])`.

3.3 Activation

To understand the organization of the task that adds and maintains a compressed set, consider the following code that implements a part of class `Company`:

```
class Company
{     Set employees   = new HashSet();
+ L0: Set programmers = new HashSet();
  void add(Employee e)
  {   employees.add(e);
+ L3: if (e.isProgrammer()) programmers.add(e); }
  void doit() { ... }                           }
```

The rules of a loop compression aspect add a new `programmers` field to class `Company`, initialize it to a set (line `L0`), and maintain it as a filtered subset of the set referred to by the `employees` field (line `L3`). Programmers activate loop compression by asserting `DoCompress` facts. When a programmer manually asserts a `DoCompress(original,filter)` fact, the loop compression will compress an `original` field according to a `filter` method; e.g., the fact `DoCompress([employees],[isProgrammer()])` directs the loop compression aspect to compress employees into a programmers set. When a `DoCompress` fact is asserted, the loop compression aspect must add the compressed field, initialize it, and maintain it. Rules that implement this behavior are shown in Figure 7.

Rule `C0` introduces a new compressed field into a class with the same attributes of an original field when a programmer asserts a `DoCompress` fact. The `@field` antecedent in rule `C0` queries the attributes of the original field. The `@field` consequent in rule `C0` introduces a new compressed field into the same class and with the same attributes as the original field. For our example, the compressed field will not actually be named `programmers`, but some unique

```
global DoCompress(original,filter);
rule C0 [orig,cmps,fltr,T,name0,name1,flags] @uniqueid(name1),
 @field(cls,flags,T,name1,cmps), Compress(orig,cmps,fltr):
 @field(cls,flags,T,name0,orig), @isa(T, [Set]),
 DoCompress(orig,fltr);
rule C1 [orig,cmps,fltr,T,obj]   @after(obj.cmps = new T()):
         Compress(orig,cmps,fltr), @now(obj.orig = new T());
rule C2 [orig,cmps,fltr,elm]
        @after({ if (elm.fltr()) this.cmps.[add](elm); }):
  Compress(orig,cmps,fltr), @now(this.orig.[add](elm));
rule C3 [orig,cmps,fltr,elm] @after(this.cmps.[remove](elm)):
     Compress(orig,cmps,fltr), @now(this.orig.[remove](elm));
```

Figure 7: Rules that prove and setup `Compress` facts.

name (`@uniqueid(name)`). The `@isa` antecedent in rule `C0` ensures that the field type is compatible with `java.util.Set`. Rule `C1` initializes the new field to an object in the same way an original object is initialized. Rules `C2` and `C3` ensure the compressed field is a subset of the original field that only contains elements that are true according to a specified filter method.

With nine rules composed of 35 consequents and antecedents, the loop compression aspect described in this section can be used in many situations. However, many details have not been addressed by the aspect's rules that might be significant in specific contexts. For example, this version of the loop compression aspect does not deal with removal through an iterator, alternative set add and remove mechanisms, pointer aliasing, and so on. Therefore, this loop compression aspect is not universally applicable and it is not meant to be a general compiler optimization. Instead, when to use the aspect is under programmer control, and the aspect's rules can be enhanced to handle special cases that may arise. The loop compression aspect can also be modified to accommodate more advanced kinds of filtering; e.g., arithmetic expressions or mapping relationships.

4 Technology

Splice can be described as a bottom-up (forward chaining) logic programming system with Java program manipulation capabilities. In bottom-up logic programming, a rule fires as soon as facts can unify with its antecedents, and facts proven by the firing rule are immediately used to fire other rules. With the addition of temporal logic and negation operators, bottom-up logic programming can express program analyses. Each point in the program's execution is associated with some code, which can be inspected by a rule to determine what facts about the program should propagate to future or past points, or even to points in other methods. Negation allows these rules to reason about what is not true at a point, and also to explicitly prevent the propagation of some facts.

The rest of this section describes Splice's major design and implementation issues. Section 4.1 describes how programs are represented to aspects, which influences how an aspect reasons about assignment, branching, and looping. Section 4.2 describes the role of negation in Splice's design, which is complicated by Splice's support for data-flow analysis. Section 4.3 describes the analysis performed by an aspect. Finally, Section 4.4 discusses the implementation of our Splice prototype.

4.1 Program Representation

Splice aspects analyze programs in a "flat" non-syntax tree representation where the result of every expression is assigned to a unique temporary variable. Because of ambiguity relating to multiple variable assignments, Splice transforms a program into single static assignment (SSA) form, where an SSA variable identifies the result of evaluating a single Java expression [3]. Instead of source code variables, SSA variables are bound to rule variables during unification with `@now`, `@nowi`, and `@past` antecedents. There are two issues with SSA that are addressed in Splice. First, SSA variables assigned in loops are actually reassigned dynamically, so the same SSA variable can refer to different values during an analysis. Second, SSA variables that merge multiple SSA variable together, known as ϕ *variables*, create aliases, so the set of facts true for the ϕ variable and the variable it merges together are related.

```
i = accounts.iterator(); int a;
while (true) {
+A:    Accnt.lock.acquire();
 B:    a = i.next().getBalance();
 C:    if (a > 0) break;
+D:    Accnt.lock.release();    }
 E: accntB.setBalance(a);
+F: Accnt.lock.release();
```

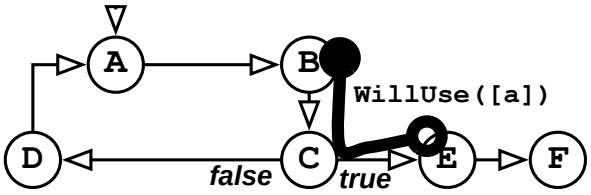

Figure 8: Code that iterates through a list of accounts (left), and an illustration (right) of when the fact `WillUse([a])` is true in the code's CFG (right).

The first issue is dealt with by automatically "expiring" (negating) a forward or backward-flow fact at a point where an SSA variable it refers to is assigned. Expiration of a forward-flow fact occurs immediately at the point of assignment, and expiration of a backward-flow fact occurs by not propagating it to points that precede the assignment. As an example of how expiration works, consider how the synchronization aspect in Section 2 is applied to the code in Figure 8. Variable `[a]` is assigned at line `B` in a loop and used at line `E` outside of the loop. By rule `S1` in Figure 2, fact `WillUse([a])` will become true before line `E`, which is then propagated backwards so it is true at lines `C` and `B` (illustrated in Figure 8). However, since variable `[a]` is assigned at line `B`, the `WillUse` fact will not be true before line `B`. This results in fact `WillUse([a])` not being true along the false path from line `C`, so the lock can correctly be released in the loop at line `D`. Because facts can expire when they refer to values created in loops, aspect programmers must carefully ensure that forward and backward-flow facts are still true when needed by only associating them with values known to still be in use.

The second issue is dealt with by "transferring" forward or backward-flow facts between a ϕ variable and variables it merges together. To explain how fact transferring works, consider a ϕ variable `m_1` that merges variables `i_0` and `i_2` together. When the forward or backward-flow fact `P(i_0)` is proven in a scope where `m_1` exists, fact `P(m_1)` is automatically proven if `P` is not universal. If `P` is universal, then fact `P(m_1)` is proven true if fact `P(i_2)` is also proven true. When fact `P(m_1)` is proven true, facts `P(i_0)` and `P(i_2)` are automatically proven true only if `P` is not universal. The explicit negation of a forward or backward-flow fact is handled in an analogous way. Fact expiration and transferring have an important interaction: fact transferring between a ϕ variable and one of its merged variables only occurs as long as both variables are not reassigned.

The way forward and backward-flow facts expire and are transfered via ϕ variable relationships enables inductive reasoning about loop behavior. Consider the following loop in SSA form that sets elements of an array to 0:

```
i_0 = 0; while (true)
{ m_1 = φ(i_0,i_2);
  if (m_1 >= a.length) break;
  a[m_1] = 0; i_2 = m_1 + 1; }
```

At the point where the above loop breaks, an aspect can recognize that every element of the array has been set to zero using the following three rules:

```
universal forward Zeroed(array,index), Zeroed(array);
rule Z0 [i]     Zeroed(*,i): @now(i=0);
rule Z1 [a,i,j] Zeroed(a,j):
     Zeroed(a,i), @past(a[[i]]=0), @now(j=i+1);
rule Z2 [a,i] Zeroed(a): @false(i>=a.length), Zeroed(a,i);
```

Rule `Z0` proves the zeroed base case, where the array index is zero, and rule `Z1` proves the zeroed inductive case, where the index is incremented by one. Rule `Z2` detects the end of the above loop where the array has been completely zeroed out. When variable `m_1` is assigned, the facts `Zeroed(*,[i_0])` and `Zeroed([a],[i_2])` are true, so fact `Zeroed([a],[m_1])` is true. Rule `Z1` associates zeroed information with the next index of the loop because any fact that refers to the current index stored in variable `m_1` will expire at the beginning of the next loop iteration where `m_1` is reassigned.

4.2 Negation

Support for negation is necessary in Splice because it enables reasoning about negative information; i.e., events that cannot occur in a program. Without negation, the aspects described in Sections 2 and 3 and many other useful aspects could not be expressed in Splice. Negation is implemented in Splice with the following constraint: the proof of a fact must not depend on its negation, which is referred to as the *stratified negation* [1] in a logic program. Because forward and backward-flow facts become true after or before, but not at, rule firing points, the proof of these facts can depend on their negation in certain circumstances. However, temporal paradoxes must not occur. Consider the following two rules:

```
backward WillRise(stock); forward Buy(stock);
rule G0 [stck] Buy(stck): WillRise(stck);
rule G1 [stck] !WillRise(stck): Buy(stck);
```

Rules `G0` and `G1` form a temporal paradox that has no well-defined meaning in Splice. Splice will reject aspect executions when a fact depends on its negation. If the fact is a forward or backward-flow fact, then it can depend on its negation if the negation only occurs through facts of the same flow. These criteria are checked conservatively, and its possible for Splice to reject an aspect execution that actually has a well-defined meaning. However, stratification in Splice is local [13], where facts are stratified dynamically rather than statically, which accepts more aspect executions than more restrictive syntactic stratification.

When no contradictions exist, an aspect's execution has a unique perfect model that ensures soundness [1]: i.e., no facts are proven that are not true. The perfect model ensures that a rule does not fire until it is definitely known where or not that a fact can unify with a non-negated antecedent or no fact can unify with a negated antecedent. For an executing aspect, contradictions are detected and the perfect model is found through a dependency graph that associates facts with the proven and negated facts their proofs or negations can depend on. When the proof or negation of fact `f` depends on the proof or negation of fact `g`, the "stage" where the proof or negation of fact `f` is definitely known is computed according the two following constraints:

1. The proof or negation of fact `f` is not known at least until proof or negation of fact `g` is known;
2. If fact `f` and `g` are not both forward-flow or are not both backward-flow, then the proof or negation of fact `f` is not known until after the proof or negation of fact `g`;

As an example, consider the synchronization aspect in Section 2. When the aspect is applied to a program, the negation of its forward-flow `Needs` and `Has` facts are not known until the second stage of the analysis because their proofs depend on negated backward-flow `WillUse` facts. A rule with negated antecedents cannot fire until the latest stage where the negation of all facts that can unify with its negated antecedents are known. Therefore, rule `S2` in Figure 2 and rule `S4` in Figure 3 will not fire until the second stage of the aspect's execution.

The fact dependency graph is computed in a pre-processing step of the implementation that applies the rules of an aspect to a program in a flow-insensitive way that ignores negated antecedents. Computing the fact dependency graph is only a secondary by-product of pre-processing, which is essential in simplifying the analysis described in Section 4.3 by narrowing the set of rules that can fire at each point and computing a finite domain of facts that can be true during analysis.

4.3 Analysis

As just described, an analysis in Splice is divided into multiple stages to support the sound execution of any negation used in an aspect, which ensures that analysis execution is monotonic. Each stage is itself divided into multiple forward, backward, and flow-insensitive passes. Rules with forward and backward-flow consequents are only fired respectively during forward and backward analysis passes. Rules with global-flow consequents are fired during a flow-insensitive pass. The direction of an analysis pass determines the order that CFG points are traversed, which will determine the firing point of a rule that can otherwise fire at multiple adjacent points. For a forward analysis pass, a rule will fire at a point closest to the entrance of a procedure. For a backward analysis pass, a rule will fire at a point closest to the exits of a procedure.

For each stage of an analysis, analysis passes are continually performed until no new facts can be proven; i.e., the stage reaches a fix-point. The execution of individual forward and backward analysis passes are very standard

finite-domain iterative data-flow analyses. A set of dirty CFG nodes, initially the entry node (forward) or exit nodes (backward) of a method, tracks what CFG nodes need to be traversed. Traversal of a node manages fact expiration and transferring and fires rules according to the code at the node and facts that are true at the node. A set of resulting facts is propagated to the nodes that follow (forward) or precede (backward) the node, adding any node to the dirty set whose set of true facts changes. A fix-point for a forward or backward analysis pass is reached when the dirty set is empty.

4.4 Implementation

Aspect execution in our current prototype of Splice consists of five steps:

1. The Java bytecode of the program is transformed into a CFG-based SSA form.
2. As described in Section 4.2, the rules of an aspect are applied to the program in a flow-insensitive way. Pre-processing computes a fact dependency graph and for each program point, a set of pre-unified rules that can fire at the point. This simplifies the firing of rules in Step four.
3. A fact-dependency analysis described in Section 4.2 detects invalid proof dependencies and computes analysis stages where facts are known.
4. Analysis is performed as described in Section 4.3.
5. Proven code advice facts (e.g., `@after`) are type checked, and compiled into the resulting Java bytecode of the program. Any detected type errors or conflicts will result in an aspect execution error.

5 Discussion

In this section we informally evaluate Splice's design, implementation, and usefulness.

5.1 Usability

Splice is designed to provide aspects with accessible analysis capabilities. As a result, many of Splice's features reduce the amount of effort needed to express an analysis. Splice's internal use of an SSA representation eases reasoning about variable re-assignment, branches, and loops. Stratified negation eliminates concerns about the sound use of negation and the repetition of analysis passes ensures completeness. Soundness and completeness, however, only apply to the execution of an aspect's rules. Splice itself cannot guarantee that the programmer's rules are correct!

Even with its usability features, the task of expressing an analysis in Splice is non-trivial. An aspect programmer must design an analysis using temporal logic, which is not a core programming skill and can be very tricky to use correctly. In the course of developing the two aspects in Section 2 and Section 3, we found that most of our bugs were related to mistakes in understanding the temporal properties of the aspects. Part of this problem arises from the use of negation and multiple rules to encode natural English terms such as "only," "must," and "unless" (Section 3.1), where the encoding often introduces bugs into the aspect.

It is easy to make mistakes in the encoding of an aspect. Therefore, the ability to debug a Splice aspect is very important. The most straightforward way for a programmer to debug an aspect is to view how it transforms a program, and then debug the aspect along with the program. Debugging will also require an inspection of the aspect's reasoning of why it transforms a program in some way. Unfortunately, our current Splice prototype does not provide any debugging support.

5.2 Precision

Aspect programmers can only expect so much precision from an analysis in Splice. Although analysis precision is not significantly limited by branching, looping, and variable re-assignments, an analysis cannot correlate when multiple conditional branches are resolved the same way; e.g., according to whether or not a debug flag is set. The most significant source of imprecision occurs in analyzing how methods interact with each other. By making global facts visible between method implementations in a flow-insensitive way, analysis sacrifices precision so its execution time

can scale linearly with the size of a program. Among other things, an inter-procedural analysis is not accurate enough to reason about dynamic dispatch and the effects of execution on memory, including pointer aliasing. Sometimes, this imprecision is acceptable because static information about the program, such as static types, can be used by an aspect to fill in the gaps. However, commonly used programming styles that avoid the use of static information can defeat these assumptions. Examples of these styles in Java include the use of reflection or calling a generic `run()` method in the `Runnable` interface.

One solution to the precision problem is to make analysis in Splice more powerful; e.g., flow-sensitive inter-procedural, context-sensitive, path-sensitive, and capable of performing shape analysis. However, these mechanisms will not scale to analyzing large programs. The other solution relies on the aspect programmer to make up for the lost precision through the encoding of additional rules and manually asserted facts. For example, alias annotations could be encoded in an aspect and verified by its rules in a scalable way. However, this requires more effort from programmers to encode what otherwise could be automatically derived.

5.3 Performance

Automatically ensuring the soundness (Section 4.2) and completeness (Section 4.3) of an aspect requires that a Splice aspect execute about half as fast as a comparable traditional analysis. For example, the analysis expressed by the synchronization aspect in Section 2 requires at least six passes to execute when only three passes are really needed to execute the analysis correctly. The extra passes in our approach are only used to recognize that completeness has occurred.

We believe that the performance of our prototype implementation is adequate. When using our prototype implementation to apply the loop compression aspect in Section 3 to a Java program of 1000 lines, execution takes about 40 seconds.[2] Though not very fast, execution time rises linearly as the size of the program grows, which is not surprising since the inter-procedural analysis used in Splice is flow-insensitive. The bottleneck in our current prototype is not analysis, rather, it is the pre-processing step used to perform the rule unification that simplifies the analysis step. The pre-processing step can become very slow as the number of possibly true global-flow facts increases because they require re-processing of the whole program.

5.4 Applicability

This paper has shown how Splice can be used to write fairly complete synchronization and loop compression aspects. Whereas AspectJ effectively enables the separation of simple concerns related to tracing, logging, security, and instrumentation, Splice expands this domain to include concerns that require data-flow information. Unfortunately, Splice is not dramatically better than AspectJ in addressing complex functionality concerns in real programs where join points must be identified manually. This restriction may negate any advantages Splice has over AspectJ in implementing these concerns.

The synchronization and loop compression aspects shown in this paper provide some evidence that using Splice is worthwhile for implementing concerns that are related to characteristics of a program execution like performance and robustness. Similar to loop compression, Splice can be used to automate domain-specific optimizations that currently need to be performed by hand. Similar to synchronization, Splice can be used to automate the addition of "policy" code into a program, although the programmer will probably need to manually accept these code additions because of constraints on analysis precision. Although not explored in this paper, Splice could also be used to implement better "static-advice aspects" that only check program properties, such as enforcing Law of Demeter rules that ensure loose object coupling [11].

6 Related Work

Aspect-oriented programming based on code transformations at join points is enabled by AspectJ [10]. AspectJ pointcut designators can identify execution events such as a method call and field accesses as they occur in isolation. The `cflow` pointcut designator provides some inter-procedural reasoning capabilities by allowing an aspect to dynamically inspect program control-flow through the call stack. Splice does not currently provide `cflow`-like inspection capabilities.

[2] We ran on a Powerbook G4 @ 867 MHz.

Currently, AspectJ provides no support for static data-flow or inter-procedural analysis. However, proposals have been made to enhance AspectJ with a data-flow pointcut designator `dflow` and a prediction-based pointcut designator `pcflow` [9]. The `dflow` designator is proposed to reason about how a Java value is computed in the data-flow of a method. The `pcflow` designator is proposed to reason about what join points can occur in the future of a program's control-flow, including what join points can occur when a method is called. Although the `pcflow` designator has only been mentioned, a version of the `dflow` designator has actually been prototyped and shown to be useful in security applications [12]. Unlike Splice, this `dflow` designator is not implemented using static analysis; instead, special data-flow tags are propagated and inspected at run-time.

Using logic programming features in a meta-programming language is referred to as logic meta-programming (LMP). TyRuBa is a LMP system for manipulating Java code [5]. The Smalltalk Open Unification Language (SOUL) [15] enables Prolog programs to manipulate the execution of Smalltalk programs. Neither system provides meta-programs with data-flow analysis capabilities. The case for expressing program analysis problems in a general logic programming system is made in [4]. Splice does not rely on a general logic programming system—it is designed specifically to support static analysis. Path logic programming extends Prolog with program transformation and static analysis capabilities [6, 14]. Splice differs from path logic programming by concentrating on higher-level transformations needed to do AOP.

Splice can be viewed as an extensible compilation system; e.g., the loop compression aspect in Section 3 is an example of how an aspect can implement domain-specific optimizations. Magik [7] allows programmers to incorporate application-specific extensions into the compilation process. The Broadway compiler [8] can utilize programmer-supplied annotations to optimize library operations. In contrast to these systems, Splice aspects performs anayses and transformations through higher-level LMP abstractions.

7 Conclusions and Future Work

Splice is the first AOP system that provides aspects with static analysis mechanisms to identify join points. Splice aspects are written in a rule-based language with logic programming features that provide convenient access to static analysis mechanisms. Although Splice aspects are not omnipotent, e.g., they cannot reason about pointer aliasing, they can still be useful in improving programmer productivity.

Looking forward, we are currently exploring how join points can be used more expressively once they have been identified. Current mechanisms for expressing advice in Splice and AspectJ are not very declarative and could benefit from more powerful reasoning facilities like those used to identify join points in Splice. Solving this problem of "join point specification" will make AOP systems better platforms for generative programming.

Splice has been implemented and used to write the loop compression and synchronization aspects presented in this paper. We expect to make the prototype publicly available in the near future on our web site at `www.cs.utah.edu/plt/splice`.

ACKNOWLEDGEMENTS

We thank Eric Eide, Matthew Flatt, Gary Lindstrom, and the anonymous reviewers for comments on drafts of this paper. Sean McDirmid and Wilson Hsieh were supported in full by NSF CAREER award CCR–9876117.

References

[1] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–147, 1988.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis by construction and approximation of fixpoints. In *Proc. of POPL*, pages 238–252, Jan. 1977.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficient computing static single assignment form and the control dependence graph. In *ACM Transactions on Programming Languages and Systems, 13(4)*, pages 84–97, Oct. 1991.

[4] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general logic programming systems - a case study. In *Proc. of PLDI*, pages 117–126, June 1996.

[5] K. De Volder. Type-oriented logic meta programming. PhD Thesis, 2001.

[6] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .NET intermediate language using path logic programming. In *Proc. of PPDP*, Oct. 2002.

[7] D. R. Engler. Incorporating application semantics and control into compilation. In *Proc. of DSL*, pages 103–118, Oct. 1997.

[8] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proc. of DSL*, Oct. 1999.

[9] G. Kiczales. The fun has just begun, Mar. 2003. Presentation at AOSD.

[10] G. Kiczales, E. Hilsdale, J. Hungunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. of ECOOP*, June 2001.

[11] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the Law of Demeter with AspectJ. In *Proc. of AOSD*, Mar. 2003.

[12] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proc. of APLAS*, Nov. 2003.

[13] T. Przymusinski. On the declarative semantics of deductive database and logic programs. In *Foundations of Deductive Databases and Logic Programming*, pages 193–216, 1988.

[14] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. In *Proc. of POPL*, Jan. 2004.

[15] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proc. of TOOLS USA*, pages 112–124, 1998.