Aspects as Modular Language Extensions

Eric Van Wyk

Department of Computer Science and Engineering
University of Minnesota

Abstract. Aspect-Oriented Programming provides language constructs which allow a programmer to specify cross-cutting concerns in a modular fashion. Extensible programming language frameworks, like Intentional Programming, use highly modular specifications of programming languages that allow different language feature sets to be imported into the programming environment. In this paper we show how certain aspect language features and the aspect weaving process can be specified as a *modular language extension* and imported into a base language specified in an extensible programming language framework. Our model of extensible languages is based on higher order attribute grammars and an extension called "forwarding" which mimics a simple rewriting process so that language constructs can be defined in terms of other language constructs.

keywords: aspect oriented programming, extensible languages, intentional programming, attribute grammars.

1 Introduction

The field of programming languages is a long-lived and active field that has seen the development of a plethora of language paradigms and language features. This field continues to be active because the problems faced by programmers continue to increase in complexity and extend into ever more diverse areas. Programming language researchers are continually looking for language paradigms and features that more adequately support programmer's evolving needs.

Currently, there is an active research community [3] with this end in mind investigating a set of language features which enable a programmer to modularly specify computations that do not fit neatly into a language's existent organizational framework, but instead cut across it. The *aspect* is the primary language feature for specifying these *cross cutting concerns*; programming in this style is referred to as Aspect-Oriented Programming [12, 3].

For example, consider the following AspectJ [11] *advice* declaration (modeled like the rest of our examples after those in [11]) that specifies that before any call to the `setX` method of any `Point` object a message containing the new x value and current x and y values will be displayed:

```
before (Point p, int a) : call p.setX ( a )
   { print ("new x " + a + " for point (" +
            p.x + "," + p.y + ")" ; }
```

This is a standard technique a programmer may use to trace the changing value of the x field in `Point` object. To achieve this same result without aspects the programmer would need to add the print statement before all calls to `setX` on `Point` objects throughout the program whereas above we have captured that notion in a single location. AspectJ is a set of aspect language features that have been added to the Java language. A high quality, widely available (see `www.aspectj.org`) compiler has been written for Java and the AspectJ extensions that solves the immediate problem of providing aspects in the Java language.

But this leads us to the question of what happens next time a new set of language features is developed. Must a new compiler be written which adds these features to an existing language? Would it not be better if we could import new language features, such as aspects, into an existing programming language environment? If a language is specified in a highly modular fashion, it might be possible to do just that. This is the goal of extensible languages and the Intentional Programming [16, 4, 19] project, which until recently, was under development at Microsoft.

In this paper we show how some AspectJ aspects can be added as a modular language extension to a simple object oriented language. This language is specified as an attribute grammar which makes use of several extensions to the attribute grammars originally design by Knuth [13]. These extension include higher order attributes [17], reference attributes [8], production valued attributes [18], and forwarding [18]. Higher order attributes are used to pass around code trees and are used in the weaving process to move advice code trees to the join points where they are woven into the original program. Reference attributes are useful in that we can pass an advice declaration node reference to the join points to be queried for its attributes in determining if the current join point matches the point cut designator of the advice. Forwarding enables a simple type of rewriting that is used to rewrite join points into new constructs that contain the advice code from various advice declarations.

In this paper, Section 2 describes the AspectJ aspect constructs that we will implement as language extensions. Section 3 defines the attribute grammar framework and base object oriented language to which the aspects are added. Section 4 then shows how aspect language constructs can be specified in a modular and additive fashion to be incorporated into the language and Section 5 concludes with a discussion of what was achieved and related and future work.

2 Aspect oriented programming with AspectJ

In this section we describe some of the AspectJ [11] aspect constructs that we implement as language extensions – their implementation is given in Section 4. We will only cover a few of the many aspect language constructs available in AspectJ in order to provide a basic understanding of how aspects can be added to our base object oriented language. The remaining AspectJ constructs pose

no fundamental difficulties to our model but for brevity and simplicity we omit some and present simplified versions of others.

In any incarnation of aspect oriented programming one must define the *join point* model that will be used, how the join points will be specified and how the implementation of the join points will be modified. While there are several types of join points in AspectJ, here we only concern ourselves with one: the method call. The join point model in AspectJ is dynamic, meaning that join points are points in the execution flow of a program. Thus we consider the join points to be all the method calls in an executing program. These are the points in the program whose behavior we intend to modify. We can affect the behavior of join points by executing a piece of code before or after the join point and by altering the input values to the join point. For a given method call construct in a program, we refer to the set of executions of that call in the executing program as the dynamic join points of that method call.

A *point cut* is a set of join points and a *point cut designator (pcd)* is a mechanism for specifying a point cut. Here, we are interested in the pcds *call* and *and*(&&). The pcd *call (signature)* will match method calls whose signature matches that in the call pcd. A pcd signature consists of a class name or object variable, a method name and a list of parameters specified either as type names or variables. A matching test, discussed below, determines if a method call construct matches a call pcd by examining its object, method and parameters to see if their types match those in the pcd signature.

Advice is used to affect join point behavior. It consists of a (possibly empty) list of variable declarations, a point cut designator and *advice code* to be executed either before, after or around any join points in the point cut specified by the point cut designator. *Weaving* is the process of placing the advice code before, after or around the affected join points. This process can be done either statically or dynamically. For example, consider a method call `q.setX ( 4 )` and the advice declaration given below:

```
before (Point p, int a) : call p.setX ( a )
 { print ("new x " + a + " for point (" + p.x + "," + p.y + ")";}
```

If `q` is defined as an object of class `Point` or a sub class of point, then the weaver can generate the code

```
{ print ("new x " + 4 + " for point (" + q.x + "," + q.y + ")";
  q.setX ( 4 ) ; }
```

to replace the original method call since `q` will always be an instance of `Point`. On the other hand, if `q` is declared to be a super-class of `Point` (that has a method `setX`) then we can not statically determine if `q` is an instance of class `Point`. In this case, we will weave the advice code inside an if-then statement to make this check at run-time and generate the code

```
{ if q.instanceOf (‘‘Point’’) then
    print ("new x " + 4 + " for point (" + q.x + "," + q.y + ")";
  q.setX ( 4 ) ;  }
```

If there is no sub-type relationship between `p` and `q` then this pcd does not match the method call and the weaver does nothing.

The process of matching a pcd p against a method call construct at compile time tells us if (i) none of the method call's join points match p, if (ii) all of the method call's join points match p or if (iii) neither (i) nor (ii) can be determined (by the test) at compile time and thus a run-time test is required. In this last case, the matching test also return the boolean expression code to be used in the run time test.

In case (i), the match test returns a $NoMatch$ value indicating that no weaving is required for this piece of advice. For (ii) the match test returns a value $Static\ \sigma$ where σ is a substitution which maps variables declared in the advice to constructs in the matched method call. In the example above, $\sigma = [p \mapsto q, a \mapsto 4]$. This substitution is applied to the advice code to generate the actual advice code which is woven into place at the method call. In case (iii) the test returns a value $Dynamic\ \sigma\ test_f$ where σ is the same as above and $test_f$ is the boolean expression for the dynamic test. We apply the substitution to the advice code and this test code and use it in the weaving process to generate the actual code to be woven for this method call. This application of σ to the advice code and test code can be seen in the examples above. That is, for the above σ, σ(`p.setx(a)`) is `q.setx(4)` and σ(`p.instanceOf(``Point'')`) is `q.instanceOf(``Point'')`.

Intuitively, the weaving of advice code and the object program is achieved by rewriting method calls to code fragments containing the method call and its advice and dynamic test code. If we can not statically determine if the pcd of a particular piece of advice matches all the dynamic join points for a method call construct, we will wrap the advice code in an *if-then* statement that determines if it matches at run time. In essence, the rewriting is done for each advice construct in the program. For a particular before-advice declaration adv_n with pcd pcd_n and advice code *code* the rewrite rules are as follows:

$$
\begin{array}{lcl}
o.m(p_1 \ldots p_n) & \Longrightarrow & \{\ \sigma(code); \\
 & & \quad o.m(p_1 \ldots p_n);\ \} \\
\quad \textbf{if}\ match(pcd_n, o.m(p_1 \ldots p_n)) = Static\ \sigma & &
\end{array}
$$

$$
\begin{array}{lcl}
o.m(p_1 \ldots p_n) & \Longrightarrow & \{\ if\ \sigma(test\_code)\ then\ \sigma(code); \\
 & & \quad o.m(p_1 \ldots p_n)\} \\
\quad \textbf{if}\ match(pcd_n, o.m(p_1 \ldots p_n)) = Dynamic\ \sigma\ test\_code & &
\end{array}
$$

$$
\begin{array}{lcl}
o.m(p_1 \ldots p_n) & \Longrightarrow & o.m(p_1 \ldots p_n) \\
\quad \textbf{if}\ match(pcd_n, o.m(p_1 \ldots p_n)) = NoMatch & &
\end{array}
$$

Around-advice does not precede or follow the join point, but instead encloses it and has the ability to call the join point method call with different parameters than those in the original method call. (We will examine a type of around-advice that does not use the *proceed* construct from AspectJ but simply calls methods directly. While this does limit the utility of around-advice it is rich enough to

expose the interesting problems in specifying aspects as language extensions.) Below is an example of an around-advice declaration:

```
around (Point p, int a) : p.setx (a, int)
  { print (p.x) ; p.setx ( a + 5 ) ; print (a + p.x); }
```

When matched again the method call `q.setX(4)` the advice code would be woven into the join point to create

```
{ print (q.x) ; q.setx(4 + 5); print (4 + q.x); }
```

Weaving for an around-advice declaration adv_n with pcd pcd_n and advice code *code* for a static match can be specified by the following rewrite rule:

$$o.m(p_1 \ldots p_n) \quad \Longrightarrow \quad \{\ \sigma(code);\}$$
$$\textbf{if } match(pcd_n, o.m(p_1 \ldots p_n)) = Static\ \sigma$$

3 Attribute grammar specification of the base language

In this section we present the attribute grammar based framework used for specifying modular definitions of languages. It this framework, Knuth's attribute grammars are extended with higher order attributes [17], reference attributes [8], production valued attributes [18], and forwarding [18]. Each of these is discussed as they are encountered below. We also provide some of the productions and attribute definitions which define the base object oriented language to which we will add aspects as a language extension. Most of the language definition is what one would expect and thus we discuss only those definitions which are the most important and least obvious.

The production signatures of a significant part of the language is shown in Table 1. (The use of the superscripts is described below.) The non terminals in abstract syntax grammar include $\{Expr, Dcl, Type\}$. For simplicity we do not make a syntactic distinction between expressions and statements; both are represented by the non-terminal *Expr* and statements are simply side-effecting expressions. The value of an *exprSeq* is the value of its second expression, and the value of a block is the value of its child expression. The *Dcl* non-terminal represents variable and type declarations, and *Type* represents type expressions, including type identifiers. The set of (abstract) terminal symbols contains $\{id\}$.

3.1 Abstract syntax and semantic trees

Attribute values will range over an unspecified set of primitive values, such as integers and strings, and a set of higher order values, such as tree nodes, abstract semantic trees and (semantic) productions. A *node* is just a record containing fields for inherited and synthesized attributes. The types of nodes correspond to the non-terminal and terminal symbols of the grammar. We will superscript these symbols with an n to indicate the different types of nodes. For example,

assign :	$Expr^f ::= Expr^f\ Expr^f$
ifthenelse :	$Expr^f ::= Expr^f\ Expr^f\ Expr^f$
block :	$Expr^f ::= Dcl^f\ Expr^f$
exprSeq :	$Expr^f ::= Expr^f\ Expr^f$
genVarDcl :	$Dcl^f ::= id\ Type^f$
dclSeq :	$Dcl^f ::= Dcl^f\ Dcl^f$
classDcl :	$Dcl^f ::= id\ Type^f$
classType :	$Type^f ::= Type^f\ Dcl^f$
intType :	$Type^f ::= \epsilon$
genVarRef :	$Expr^f ::= id$
intVarRef :	$Expr^f ::= Dcl^n\ id$
objectVarRef :	$Id^f ::= Dcl^n\ id$
methodCall :	$Expr^f ::= Expr^f\ id\ Exprs^f$
fieldRef :	$Expr^f ::= Expr^f\ id$
methodDcl :	$Dcl^f ::= Type^f\ Dcl^f\ Expr^f$

Table 1. Base Language Production Signatures

$Expr^n$ denotes the type of nodes which contain the inherited and synthesized attributes for expressions and $Expr^s$ ($Expr^i$) denote records that contain just the synthesized (inherited) attributes for an $Expr$ non-terminal. We will use a dot (.) notation for referencing attribute values on nodes: $n.a$ is the value of attribute a on node n. An *abstract syntax tree* [2, p. 49] is a tree of non-terminal and terminal symbols constructed according to the productions of the abstract grammar. An *attributed tree* is a tree of attributed nodes constructed according to the productions and attribute definitions rules. *Abstract semantic trees* are used in work by Johnsson [9] on treating attribute grammars as a style of programming in lazy functional languages. These are functions which map a set of inherited attributes to a set of synthesized attributes according to the syntax productions and attribute definition rules. In higher order attribute grammars [17], semantics trees are valid attribute values. Here, we will change this definition slightly so that the output of the semantic tree function is a node containing both the input inherited attributes and the computed synthesized attributes. The types of these trees are denoted by superscripting non-terminal symbols with an f. For example, semantic trees for the $Expr$ non-terminal, have the type $Expr^f$ which is just short hand for $Expr^i \rightarrow Expr^n$, ($Expr^f \equiv Expr^i \rightarrow Expr^n$). We will often drop the adjective "abstract" from these terms.

A *semantic production* is a function whose input is semantic trees and attributed nodes and output is a semantic tree. The *assign* semantic production has the type $Expr^f \times Expr^f \rightarrow Expr^f$ which is written in the more traditional

manner in Table 1 as $assign : Expr^f ::= Expr^f\ Expr^f$ When we use the term "production" we will mean semantic production.

We will occasionally find it helpful to assign values for some, but not all, of the inherited attributes for semantic trees and assign the rest later. Thus, we have the operation $\oplus$ which takes a semantic tree and a list of attribute definitions and returns the semantic tree that will use the specified inherited attribute values instead of any it may receive later when the tree is given a set of inherited attributes in order to generate the synthesized attributes. The operation $\oplus$ is defined as follows:

$$\oplus : Node^f \times [AttrDefs] \rightarrow Node^f$$
$$n^f \oplus attrdefs = \lambda n^i \rightarrow n^f\,(n^i\ 'owb'\ attrdefs)$$

where $n\ 'owb'\ ads$ is the same as n except that the attribute definitions in ads have replaced the corresponding ones in n.

It is also convenient to be able to recover the semantic tree function used to create a node from the node itself. Thus, there is an attribute $this^f$ for this purpose. It is defined on all productions, but we only show it here on the *assign* production. All others are specified in a similar fashion.

$$assign : Expr_0^f ::= Expr_1^f\ Expr_2^f$$
$$Expr_0^n.this^f = assign(Expr_1^f, Expr_2^f)$$

3.2 Attributes

Types in our base language are supported by a *type* attribute whose type is $Type^n$ – it is a reference attribute. Thus, we can query an expression *expr* for properties of its type, like its size in bytes, by $expr.type.size\_in\_bytes$.

The inherited attribute *env* has type *Env* where $Env \equiv [\ (\ String, Dcl^n\ )\ ]$ – a list of tuples of strings and declaration nodes. It is used to look up an identifier's declaration node using the function $lookup : String \times [\ (\ String, Dcl^n\ )\ ] \rightarrow Dcl^n$. Scope rules are enforced by adding nested declarations to the front of the list. This attribute is automatically copied from a node to its child nodes if no other definition is provided. The synthesized attribute *defs* is defined on *Dcl*s, has the same type as *env* and is used to gather declarations from *Dcl*s. Some productions and attribute definitions for these attributes are shown in Table 2. As is the norm, we will use numeric subscripts to distinguish between like named non-terminals.

3.3 Forwarding and production-valued attributes

Forwarding [18] is a technique for providing default attribute values for nodes that complements other default schemes such as the automatic copying of inherited attribute values to a node's children. Forwarding is the main technical contribution of the reduction engine in the Intentional Programming system. A production that contains a "forwards to" clause constructs a semantic tree from

$block : Expr_0^f ::= Dcl^f \ \ Expr_1^f$
$\quad Expr_1^n.env = Expr_0^n.env + Dcl_1^n.defs$

$genVarDcl : Dcl^f ::= id \ \ Type^f$
$\quad Dcl^n.defs = [(id.lexeme, Dcl^n)]$

$dclSeq : Dcl_0^f ::= Dcl_1^f \ \ Dcl_2^f$
$\quad Dcl_0^n.defs = Dcl_0^n.defs + Dcl_0^n.defs$
$\quad Dcl_2^n.env = Dcl_0^n.env + Dcl_1^n.defs$

$classDcl : Dcl^f ::= id \ \ Type^f$
$\quad Dcl^n.defs = [(id.lexeme, Dcl^n)]$

$intType : Type^f ::= \epsilon$
$\quad Type^n.size\_in\_bytes = 4$
$\quad Type^n.varDclForward = intVarDcl$

$classType : Type_0^f ::= Type_1^f \ \ Dcl^f$
$\quad Type_0^n.varDclForward = objectVarDcl$

Table 2. Definitions of *env* and *defs*.

various (higher order) attribute values on it parent and child nodes. This semantic tree (which is a function) is implicitly provided with the inherited attributes of the parent node to generate a node of the same type as the root node. This node is called the *forwards-to node*. When the parent node is queried for any attribute value for which it does not provide an explicit definition, the value returned to the query is the value of that attribute on the forwards to node. An example will help to clarify. Consider the following definition of an if-then instruction which has a condition and then-clause but not else-clause:

$ifthen : Expr_0^f ::= Expr_1^f \ \ Expr_2^f$
forwardsTo $ifthenelse \ Expr_1^f \ \ Expr_2^f \ \ skip$

Assuming the language has an if-then-else instruction and a skip instruction, we can model the behavior of the if-then by having it forward to an if-then-else whose else-clause is a skip instruction. There are other ways to implement this example, it is provided only to demonstrate forwarding. A more interesting example appears in [18] where we use forwarding in a modular implementation of operator overloading. The similar combinator **forwardsTo**^{n}, take a node instead of a semantic tree and forwards directly to that node instead of using the nodes inherited attributes to build a node from a semantic tree.

Production-valued attributes are attributes that, as the name implies, have semantic productions as their values. This is natural generalization of higher order attributes which have semantic trees as values. These attributes are commonly used in conjunction with forwarding. As we shall see, it is convenient

to have different productions for variable declarations and references of different types. Since type information is computed after parsing, the parser can not use type-specific declaration and reference productions. Thus, we have a generic variable declaration production *genVarDcl* and variable reference production *genVarRef*. The *genVarDcl* production forwards to a node created by the type specific productions *intVarDcl* and *objectVarDcl*. As seen in Table 3, *genVarDcl* uses the production-valued attribute *varDclForward* off of its type child and its own children to build the forwards-to node. If its type child is an integer type, it uses the production *intVarDcl*; if it is a class, the production *objectVarDcl* is used. The type-specific declaration productions define a polymorphic attribute $varRefForward : \alpha \rightarrow Expr^f$ to enable the use of type-specific variable reference productions. This attribute is a function (production) that takes a single parameter and returns an *Expr* semantic tree. In the case of *intVarDcl* and *objectVarDcl*, the parameter is an identifier. It is also convenient for variable reference productions to have references to their declaration nodes. Thus, the type specific variable reference productions take an additional child as a parameter: a reference to their declaration nodes. The key idea is the the generic variable reference production determines what it forwards to by looking at the variables declaration node. We will use this same technique in the aspect weaving process described in Section 4. These productions are shown in Table 3. (When productions have multiple definitions, the scattered attribute definitions are collected to completely define the production.)

3.4 Attribute evaluation

With the use of forwarding, we have to potential of creating very many trees and unnecessarily evaluating many attributes. Consider the *if-then* forwarding example. Evaluating all the attributes on the child nodes of the *if-then* would be wasted effort since all queries for attributes of the left hand side *Expr* node will be forwarded to the forwards-to node which has its own copies of the children whose attribute values will presumably be evaluated.

To counter this problem, we rely on lazy evaluation. Attribute values are not calculated unless they are needed. Our prototype system follows the example of Johnsson [9] and uses the lazy functional language Haskell [15] as our implementation language.

4 Defining aspect constructs as language extensions

We would like to add aspects to the above language design in a modular and additive way. By "modular" we mean that we would like to see the following collection of productions and attribute values treated as a collection or module. By "additive" we mean that we want to add aspect to the base language with-

$typeVarRef : Type^f ::= id$
  $Type^n.name = id.lexeme$
  forwardsTo^n $lookup(id.lexeme, Type^n.env)$

$genVarDcl : Dcl^f ::= id\ \ Type^f$
  forwardsTo $(Type^n.varDclForward)\ (id,\ Type^f)$

$intVarDcl : Dcl^f ::= id\ \ Type^f$
  $Dcl^n.type = Type^n$
  $Dcl^n.name = id.lexeme$
  $Dcl^n.varRefForward = \lambda i \rightarrow intVarRef\ (Dcl^n, i)$

$objectVarDcl : Dcl^f ::= id\ \ Type^f$
  $Dcl^n.type = Type^n$
  $Dcl^n.name = id.lexeme$
  $Dcl^n.varRefForward = \lambda i \rightarrow objectVarRef(Dcl^n, i)$

$genVarRef : Expr^f ::= id$
  forwardsTo $forward^f$
 where $dcl = lookup(id.lexeme, Id^n.env)$
    $forward^f = (dcl.varRefForward)\ id$

$intVarRef : Expr^f ::= Dcl^n\ \ id$
    $Expr^n.name = id.lexeme$
    $Expr^n.type = Dcl^n.type$

$objectVarRef : Expr^f ::= Dcl^n\ \ id$
    $Expr^n.name = id.lexeme$
    $Expr^n.type = Dcl^n.type$

Table 3. Variable declaration and reference productions.

out changing any of the base language definitions but simply by providing new definitions to the existing one.[1]

Table 4 show the productions defining the abstract syntax of the aspect language features, some of which make us of new *PCD* non-terminals. We need to provide semantics, that is attribute definitions, for these productions in order to add them to the language defined above.

$methodCall'$:	$Expr^f$	$::= Expr^f\ id\ Expr^f$
$aspect$:	Dcl^f	$::= id\ Dcl^f$
$beforeAdvice$:	Dcl^f	$::= Dcl^f\ PCD^f\ Expr^f$
$aroundAdvice$:	Dcl^f	$::= Dcl^f\ PCD^f\ Expr^f$
$callPCD$:	PCD^f	$::= objPCD^f\ mthPCD^f\ prmPCD^f$
$classPCD$:	$objPCD^f$	$::= id$
$objectPCD$:	$objPCD^f$	$::= id$
$methodPCD$:	$mthPCD^f$	$::= id$
$paramPCD$:	$prmPCD^f$	$::= id$
$instanceOfPCD$:	PCD^f	$::= Type^f$
$andPCD$:	PCD^f	$::= PCD^f\ PCD^f$
$substVarDcl$:	Dcl^f	$::= Expr^n$

Table 4. Aspect Language Productions

We will first discuss the definition of the aspect, advice and point cut designator constructs and then see how they are used in the weaving process in the declaration of the new join point method call production $methodCall'$.

4.1 Aspects

An *aspect* production is a collection of advice declarations and it's primary role, as far as this paper is concerned, is in making the advice declarations available to the rest of the program. At each potential join point, we want to have access to a list of all of the advice declaration nodes which contain the join point in their scope. To accomplish this, advice declarations are added to the environment attributes *adv_env* and *adv_defs* in a fashion similar to that used for *env* and *defs*. Since we do not need to look up advice declaration nodes by a name, the type of *adv_env* and *adv_defs* is just $[Dcl^n]$. The *aspect* production simply copies the *defs* and *adv_defs* attributes from its child declarations. A default value of the empty list ([ ]) is assigned to *adv_defs* is none is provided and the *adv_env*

[1] The only non-additive change is to the parser so that the new method call production $methodCall'$ is used instead of the original. But even this can be done in an additive manner.

attribute is copied to the children of a node by default if no other definition is provided.

$aspect : Dcl_0^f ::= id\ Dcl_1^f$
$\quad Dcl_0^n.defs = Dcl_1^n.defs$
$\quad Dcl_0^n.adv\_defs = Dcl_1^n.adv\_defs$

$block : Expr_0^f ::= Dcl^f\ Expr_1^f$
$\quad Expr_1^n.adv\_env = Dcl^n.adv\_defs + Expr_0^n.adv\_env$

4.2 Advice declarations

Advice declarations define the attributes which are needed in the weaving process. These include an *adviceType* attribute which takes values from the set $\{Before, After, Around\}$. The pcd^n attribute is a reference to the child PCD node. The attribute *code* is the semantic tree that is woven in at the join point. The before-advice production is shown below; the others are similarly defined:

$beforeAdvice : Dcl_0^f ::= Dcl_1^f\ PCD^f\ Expr^f$
$\quad Dcl_0^n.adviceType = Before$
$\quad Dcl_0^n.pcd^n = PCD^n$
$\quad Dcl_0^n.code = Expr^f$
$\quad Dcl_0^n.defs = [\,]$
$\quad Dcl_0^n.adv\_defs = [Dcl_0^n]$
$\quad PCD^n.env = Dcl_1^n.defs + Dcl_0^n.env$
$\quad Expr^n.env = Dcl_1^n.defs + Dcl_0^n.env$

4.3 Point Cut Designators

Point cut designator nodes have a *match* attribute which tests if it matches a join point construct. This function takes an $Expr^n$ and returns a value of the algebraic type $Match$ defined as follows:

data $Match = NoMatch \mid Static\ Env \mid Dynamic\ Env\ Expr^n$

The behavior of this function was sketched in Section 2 and its implementation in this language framework is shown in Table 5. Recall that the variables declared in the advice and used in the advice code and dynamic test code will need to be replaced by the appropriate constructs from the matched join point. This substitution σ is implemented by an Env environment. Since an Env matches strings to declaration nodes (Dcl^n), we use a new declaration production *substVarDcl* for the declarations in a $Match$ environment. Recall that the *genVarRef* production will query the environment *env* for its *id*'s declaration node and then query that declaration node for the (function that builds the) semantic tree that it should forward to. The *substVarDcl* production will build the declaration nodes which *genRefVar* queries for variables in the $Match\ Env$. It will answer these

queries with a semantic tree that has had its *env* inherited attribute value determined already. This will aid in avoiding improper name capture in the weaving process described below. This semantic tree generates the construct that the substitution variables matched. It is defined as follows:

$$substVarDcl : Dcl^n ::= Expr^n$$
$$Dcl^n.varRefForward = Expr^n.this^f \oplus [env = Expr^n.env]$$

This will effectively implement the substitution process described in Section 2.

In Table 5, the *callPCD* production is used to match *call* point cut designators against method calls by calling the *match* function on its child PCD nodes and combining their results with $\wedge_{pcd}$. This operator has type $Match \times Match \rightarrow Match$. It combines matches in the expected way, combining the substitution environments and dynamic test code of the parameter matches.

The *methodPCD*'s *match* function checks if the identifier of the PCD is the same as the method name found at the join point. If they are, it returns a static match with an empty substitution, otherwise no match is returned.

The *objectPCD* is used when an object variable is used in the PCD instead of a class name. The object at the method call is passed as the *expr* parameter to the *objectPCD* match function. If the *expr*'s type is a sub-type of the class type of the object in the pcd, then we have a static match and must pair the string *id.lexeme* and the matched object *expr* and add them to the $Match$ environment. We create the required Dcl^n by using the $substVarDcl$ production and giving it the matched object expression *expr*. In the case that the object type at the PCD is a sub-type of the matched object type *expr.type* then we will need a run-time test to ensure that the actual object referenced by *expr* is indeed of the proper class. The test code generated in this case uses the base language reflective `instanceOf` method to do this test.

The *classPCD* production behaves in the same manner, except that it will need to generate a new identifier for the $Match$ environment if a dynamic test is required. The *paramPCD* production is not shown but it's behavior is the expected one. Also, the *andPCD* behaves as expected by calling the $\wedge_{pcd}$ operator.

4.4 Weaving

Intuitively, the weaving of advice code and the object program is achieved by rewriting method calls to blocks in which *before* advice code proceeds the method call, *after* advice code follows it, and *around* advice encompasses it. If we can not statically determine if the pcd of a particular piece of advice matches all the dynamic join points for a method call construct, we will wrap the advice code in an *if* statement that determines if it matches at run time. In Section 2 we saw that the essence of this problem can be stated using rewrite rules. In this section, we show how those rewrite rules are encoded into out attribute grammar framework.

To achieve such rewriting, we will introduce a new method call production $methodCall'$ that will either forward to the standard method call if there is no

$$
\begin{array}{l}
callPCD : PCD^f ::= objPCD^f\ \ mthPCD^f\ \ prmPCD^f \\
\quad PCD^n.match = \lambda expr \rightarrow\ \ objPCD^n.match(expr.objRef) \wedge_{pcd} \\
\qquad\qquad\qquad\qquad\quad mthPCD^n.match(expr.methRef) \wedge_{pcd} \\
\qquad\qquad\qquad\qquad\quad prmPCD^n.match(expr.paramRef)
\end{array}
$$

$$
\begin{array}{l}
methodPCD : mthPCD^f ::= id \\
\quad mthPCD_0^n.match = \lambda id' \rightarrow \textbf{if}\ id.lexeme \equiv id'.lexeme \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ Static[\,]\ \textbf{else}\ NoMatch
\end{array}
$$

$$
\begin{array}{l}
objectPCD : objPCD^f ::= id \\
\quad objPCD^n.match = \lambda expr \rightarrow \textbf{if}\ expr.type.isSubTypeOf\ \ dcl_n.type \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ Static\ [(id.lexeme, substVarDcl(expr))] \\
\qquad\qquad\qquad\qquad\quad \textbf{else if}\ dcl_n.type.isSubTypeOf\ \ expr.type \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ Dynamic\ [(id.lexeme, substVarDcl(expr))] \\
\qquad\qquad\qquad\qquad\qquad\qquad methodCall(genVarRef(mk\_id(id.lexeme)), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad mk\_id(\text{“}inClass''), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad mk\_strConst(dcl_n.type.className)) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ NoMatch \\
\textbf{where}\ dcl = lookup(id.lexeme, objPDC^n.env)
\end{array}
$$

$$
\begin{array}{l}
classPCD : objPCD^f ::= Type^f \\
\quad objPCD^n.match = \lambda expr \rightarrow \textbf{if}\ expr.type.isSubTypeOf\ \ Type^n \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ Static\ [\,] \\
\qquad\qquad\qquad\qquad\quad \textbf{else if}\ Type^n.isSubTypeOf\ \ expr.type \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ Dynamic\ [(newId, substVarDcl(expr))] \\
\qquad\qquad\qquad\qquad\qquad\qquad methodCall(genVarRef(mk\_id(newId)), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad mk\_id(\text{“}inClass''), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad mk\_strConst(dcl_n.type.className)) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ NoMatch \\
\textbf{where}\ newId = gensym()\quad (*\ generate\ a\ unique\ identifier\ name\ *)
\end{array}
$$

$$
\begin{array}{l}
\wedge_{pcd} : Match \times Match \rightarrow Match \\
m_1 \wedge_{pcd} m_2 = \textbf{case}\ m_1\ \textbf{of} \\
\quad NoMatch \rightarrow NoMatch \\
\quad Static\ e_1 \rightarrow \textbf{case}\ m_2\ \textbf{of} \\
\qquad\qquad\qquad NoMatch \rightarrow NoMatch \\
\qquad\qquad\qquad Static\ e_2 \rightarrow Static(e_1 + e_2) \\
\qquad\qquad\qquad Dynamic\ e_2\ \ cd \rightarrow Dynamic(e_1 + e_2)\ \ cd \\
\quad Dynamic\ e_1\ \ cd_1 \rightarrow \textbf{case}\ m_2\ \textbf{of} \\
\qquad\qquad\qquad\qquad NoMatch \rightarrow NoMatch \\
\qquad\qquad\qquad\qquad Static\ e_2 \rightarrow Dynamic(e_1 + e_2)\ \ cd_1 \\
\qquad\qquad\qquad\qquad Dynamic\ e_2\ \ cd_2 \rightarrow Dynamic(e_1 + e_2)\ (andExpr\ \ cd_1\ \ cd_2)
\end{array}
$$

Table 5. Point cut designator productions.

more advice with a pcd that matches the method call or will forward to the code fragment block containing the appropriate advice code. The production *methodCall ′* is defined in Table 6. The three synthesized attributes defined by

$$
\begin{array}{l}
methodCall' : Expr_0^f ::= Expr_1^f\ id\ Expr_2^f \\
\quad Expr_0^n.objRef = Expr_1^n \\
\quad Expr_0^n.methRef = id \\
\quad Expr_0^n.paramsRef = Expr_2^n \\
\quad \textbf{forwardsTo}\ (forward \oplus [adv\_env = new\_adv\_env]) \\
\textbf{where} \\
\quad new\_asp\_env = \textbf{if}\ Expr_0^n.adv\_env \equiv [\,]\ \textbf{then}\ [\,]\ \textbf{else}\ tail\ Expr_0^n.adv\_env \\
\quad forward = \\
\qquad \textbf{if}\ Expr_0^n.adv\_env \equiv [\,]\ \textbf{then}\ methodCall(Expr_1^f, id, Expr_2^f) \\
\qquad \textbf{else} \\
\qquad\quad \textbf{let}\ \ adv^n = head\ Expr_0^n.adv\_env \\
\qquad\qquad res = adv^n.pcd^n.match(Expr_0^n) \\
\qquad\quad \textbf{in} \\
\qquad\qquad \textbf{if}\ adv^n.adviceType \equiv Before\ \textbf{then} \\
\qquad\qquad\quad \textbf{case}\ res\ \textbf{of} \\
\qquad\qquad\qquad NoMatch \rightarrow methodCall'(Expr_1^f, id, Expr_2^f) \\
\qquad\qquad\qquad Static\ s \rightarrow \textbf{let}\ advice\_code = adv^n.code \oplus \\
\qquad\qquad\qquad\qquad\qquad\qquad [env = s + adv^n.env] \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ block(dclSkip, exprSeq(advice\_code, \\
\qquad\qquad\qquad\qquad\qquad\qquad methodCall'(Expr_1^f, id, Expr_2^f))) \\
\qquad\qquad\qquad Dynamic\ s\ t \rightarrow \textbf{let}\ advice\_code = adv^n.code \oplus \\
\qquad\qquad\qquad\qquad\qquad\qquad [env = s + adv^n.env] \\
\qquad\qquad\qquad\qquad\qquad test\_code = t \oplus [env = s + adv^n.env] \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ block(\ dclSkip, exprSeq(\\
\qquad\qquad\qquad\qquad\qquad\qquad if\_then(test\_code, advice\_code), \\
\qquad\qquad\qquad\qquad\qquad\qquad methodCall'(Expr_1^f, id, Expr_2^f)\ )) \\
\qquad\qquad \textbf{else if}\ adv^n.adviceType \equiv After\ \textbf{then} \\
\qquad\qquad\quad ... \\
\qquad\qquad \textbf{else if}\ adv^n.adviceType \equiv Around\ \textbf{then} \\
\qquad\qquad\quad \textbf{case}\ res\ \textbf{of} \\
\qquad\qquad\qquad NoMatch \rightarrow methodCall'(Expr_1^f, id, Expr_2^f) \\
\qquad\qquad\qquad Static\ s \rightarrow adv^n.code \oplus [env = s + adv^n.evn] \\
\qquad\qquad\qquad Dynamic\ s\ t \rightarrow \ldots
\end{array}
$$

Table 6. New method call production.

methodCall′ are used by the *callPCD* production to get access to the child nodes of the method call in order to evaluate the *match* function.

The value *forward* is the semantic tree this production forwards to and is an almost mechanical encoding of the rewrite rules from Section 2 into a the given code. We define a new value for the inherited attribute *adv_env* for the rewritten method call to be advice environment for the current method call *without* the

first advice declaration. Once it has been used in the current rewrite, we don't want to use it again. We thus use the operation $\oplus$ to assign the new correct value to *adv_env* on the forwarded-to semantic tree.

In calculating the value of *forward*, if there are no advice declarations in $Expr_0^n.adv\_env$ then the rewriting weaving process has completed and we simply forward to the standard method call $methodCall$. Otherwise, the first advice declaration is stored in adv^n and the result of matching this method call expression with the advice declarations pcd is stored in res. We then check the value of $adviceType$ in order to properly weave the advice code. Despite the less than appealing abstract syntax notation, it is clear that we generate the appropriate code based on the result of the *match* function stored in res.

Of interest is the changing of the *env* attribute on the advice and dynamic test code. Since we are moving semantic trees to new locations in this weaving process, we must be careful about moving free variables; these are those whose declaration node is not in the semantic tree being moved. Our main concern is that when the moved semantic tree for the advice code or dynamic test is used in its new location it is not adversely affected by an enclosing environment *env*. This simplest way to handle this problem is to simply bring along its environment from its original location, which is very simple when using reference attributes. We augment that environment with the $Match$ environment so that the substitution variables will find the appropriate $substVarDcl$ nodes in its environment. Since the substitution variable will forward to the appropriate matched constructs we must ensure that the variables in the matched expressions also have access to the proper environment. This is handled by $substVarDcl$ described above.

5 Conclusion

In this paper we have shown that many AspectJ aspect constructs can be specified as a modular language extension which can be easily added to an existing object oriented language specified in an extensible language framework. Many other AspectJ constructs, such as control flow pcds, property based matching and different types of join points can also be added with a similar amount of work, but they are omitted here for brevity. We have also left out some details of the specified constructs and some error checking specifications as well since the goal was to give the flavor of how aspects can be added and not to provide an all-encompassing definition. The omitted specifications are rather pedestrian and of little general interest.

There is much work in the attribute grammar literature on modular attribute grammars and their use in language specification ([10, 7, 5, 1, 6] are but a few examples) and we hope to build upon that previous work. It is a goal of this paper to provide evidence for the conjecture that it is the combination of higher order attributes, reference attributes, production-valued attributes and forwarding that enables languages to be specified in a highly modular fashion. Removing any of these extensions from the attribute grammar definition above would have a significant negative impact on its modularity.

We also provide a significant example of a type of "module" that can be specified as an attribute grammar fragment and how it can be used to build more expressive languages by adding these modules into an existing language framework. z This work was partially inspired by the paper "A Semantics for Advice and Dynamic Join Point ins Aspect-Oriented Programming" by Wand, Kiczales and Dutchyn [20] in which they provide a denotational semantics for aspect constructs with fundamental properties similar to those in AspectJ. The aspects presented in that paper are part of the Aspect Sand Box Project which aims to provide an environment which allows the experimentation with different varieties of aspects. The extensible language framework provided here would also provide an attractive environment for examining different types of aspect constructs since the aspect definitions are modularized and could be easily swapped into and out off a separately specified base language. The declarative nature of attribute grammars also simplifies the specification of the syntax and semantics of aspect constructs.

There are two (at least) interesting ways in which the specifications given here could be simplified. First, we are interested in ways in which rewrite rules with side conditions over attribute values can be mapped automatically into attribute grammar specifications so that the definition of the forwards to semantic tree for *methodCall′* would not have to be written out by hand as it is in this paper. We are also interested in using object-oriented extensions to attribute grammars, exemplified in [14], in our specifications.

References

1. S. R. Adams. *Modular Grammars for Programming Language Prototyping.* PhD thesis, University of Southampton, Department of Electronics and Computer Science, UK, 1993.
2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
3. D. Crwaford, editor. *Communications of the ACM*, volume 44 number 10, October 2001.
4. K. Czarnecki and U. W. Eisenecker. *Generative Programming: methods, tools and applications.* Addison-Wesley, 2000.
5. D. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, 1990.
6. R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars. In *19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 223–234, 1992.
7. H. Ganzinger and R. Giegerich. Attribute coupled grammars. *SIGPLAN Notices*, 19:157–170, 1984.
8. G. Hedin. Reference attribute grammars. In *2nd International Workshop on Attribute Grammars and their Applications*, 1999.
9. T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, 1987.

10. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 Object–Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.
12. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 Object–Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.
13. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in 5(2):95–96, 1971.
14. M. Mernik, M. Lenic, E. Avdicausevic, and Z. Viljem. Multiple attribute grammar inheritance. In *2nd International Workshop on Attribute Grammars and their Applications*, 1999.
15. S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98. Available at URL: `http://www.haskell.org`, February 1999.
16. C. Simonyi. The future is intentional. *IEEE Computer*, May 1999.
17. D. Swierstra and H. Vogt. Higher-order attribute grammars. In H. Albas and B. Melichar, editors, *International Summer School on Attribute Grammars Applications and Systens: SAGA*, volume 545 of *Lecture Notes in Computer Science*, pages 256–296. Springer-Verlag, 1991.
18. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th International Conf. on Compiler Construction*, Lecture Notes in Computer Science. Springer-Verlag, 2002. To appear.
19. E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. Intentional Programming: a host of language features. Technical Report PRG-RR-01-21, Computing Laboratory, University of Oxford, 2001.
20. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In 9^{th} *Workshop on Foundations of Object-Oriented Languages, FOOL 9*, pages 67–87, Porland, OR, 2002.