Implementing Aspect-Oriented Programming Constructs as Modular Language Extensions

Eric Van Wyk

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minnesota, USA

Abstract

Extensible programming languages and their compilers are experimental systems that use highly modular specifications of languages and language extensions in order to allow a variety of language features to be easily imported, by the programmer, into his or her programming environment. Our framework for extensible languages is based on higher-order attribute grammars extended with a mechanism called "forwarding" that mimics a simple rewriting process. Forwarding is designed such that no additional attribute definitions need to be written when combining a "host" language with language extensions (specified as attribute grammars) thus allowing for the modular composition of language features. This means that programmers can remain unaware of the underlying attribute grammars when building customized languages by importing language extensions. This paper shows how aspects and the aspect weaving process from Aspect-Oriented Programming can be specified as a *modular language extension* and imported into an extensible host language. This paper also illustrates how an extensible compiler framework exposes its underlying semantic analyses and how this can provide a convenient arena for researchers to explore new aspect-oriented language features.

Key words: extensible languages, extensible compilers, attribute grammars, forwarding, aspect-oriented programming

Email address: `evw@cs.umn.ed` (Eric Van Wyk).

1 Introduction

1.1 Motivation

The active field of programming languages is continually investigating new language features to help reduce the semantic gap between the programmer's high-level understanding of the problem and the relatively low-level language in which the problem solutions are encoded. Many language features such as generics in object-oriented programming, higher-order functions, and aspects from aspect-oriented programming are helpful in specifying abstractions. Also, domain-specific languages help reduce this gap by raising the level of abstraction of the language in many problem domains. However, a fundamental issue remains: problems often cross multiple domains and no language contains all of the general-purpose and domain-specific features needed to adequately address all aspects of the problem. Thus, programmers cannot "say what they mean" but must encode their ideas as programming idioms at a lower level of abstraction.

Extensible languages provide a promising way to reduce this semantic gap. An extensible language can easily be extended with the unique combination of general-purpose and domain-specific language features that raise the level of abstraction to that of a particular problem, a view supported by Steele in "Growing a language" [52]. In "Impact of economics on compiler optimization" [49], Robison complements this view by showing that it is not economically feasible to include many important (domain-specific) optimizations in traditional compilers. Instead, these should also be language extensions that programmers can add to their compiler's optimization repertoire. Evidence of programmer's need to extend their language can be seen in the many popular C++ template libraries: Loki's design pattern implementations [3], CGAL's computational geometry optimizations [21], and the higher-order functions of FC++ [42]. Despite the limitations of template meta-programming these do provide a mechanism for extending C++.

We are currently developing an experimental extensible compiler framework for implementing extensible languages that allows programmers to import, into their *host language*, new language constructs, new semantic analyses, and new program transformations which may be used to optimize programs. Such language features are specified by *modular language extensions*. Before describing our framework two motivating examples are presented.

SQL as a domain-specific language extension: As a first example, consider a programmer who needs to write a Java program that accesses a relational database in which the queries to the database are written in the SQL query language. Traditionally this is done by using the JDBC library[66] to send SQL commands at run-time as character strings to the database server over a "connector". On the server the commands are processed and performed. For example, to select the customer name fields from database records whose quantity field is greater than a Java variable `value` one may write:

```
Statement stmt = connector.createStatement();
stmt.execute("select CUST_NAME from CUSTOMERS where QUANTITY > "
              + value );
```

A problem with this approach is that any syntax errors or type errors are not detected until runtime. A better solution would be to implement SQL as a language extension that the programmer can import into his or her host language and use as follows:

```
on connector execute { select CUST_NAME from CUSTOMERS
                              where QUANTITY > value }
```

The `on ... execute ...` command takes a database connection and an SQL command. Because this and the SQL constructs have been imported into the host language, they can be parsed and type-checked by the extended compiler and any errors can be reported to the programmer at compile time.

Aspects as a general-purpose language extension: A second example comes from aspect-oriented programming (AOP) [36,59,2,22]. Of interest here are language features that allow a programmer to modularly specify computations that do not fit neatly into a language's organizational framework, but instead cut across it. The *aspect* is the primary language construct for specifying such *cross cutting concerns* and programming in this style is referred to as aspect-oriented programming. Consider an example from AspectJ [35], a popular aspect-oriented programming language that extends Java with aspect-oriented language features (a broader introduction AOP and AspectJ can be found in Section 2). This is an *advice* declaration that specifies that before any call to the *setX* method on a *Point* object a message containing the new value for x and the current values of x and y 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 that is used to trace the changing value of the x field in *Point* objects when the source code for the *Point* class cannot be modified. Without the use of aspects, modularity is lost as the print statement must be added before all calls to *Point setX* methods, thus scattering it throughout the program. With aspects, however, this notion can be specified in a single location.

A high quality compiler has been written for AspectJ[1] that solves the immediate problem of providing aspects in Java. But what happens when the programmer wants to write a program that uses aspects as well as constructs that are part of some other extension to Java such as the SQL constructs from above? This paper describes an extensible compiler framework and shows how some of the core language constructs in AspectJ can be implemented as a modular language extension.

1.2 Using Extensible Languages and Language Extensions

To understand the extensibility we seek, a *critical distinction* is made between two activities. The first is the *implementation of a language extension*, which is performed by a domain-expert feature designer. The second is the *selection of the language extensions* that will be imported into an extensible language in order to create an extended language. This is performed by a programmer. This paper consistently uses the terms "feature designer" and "programmer" to highlight these different roles. The manner in which extensible languages and language extensions are used in our framework is diagrammed in Figure 1.

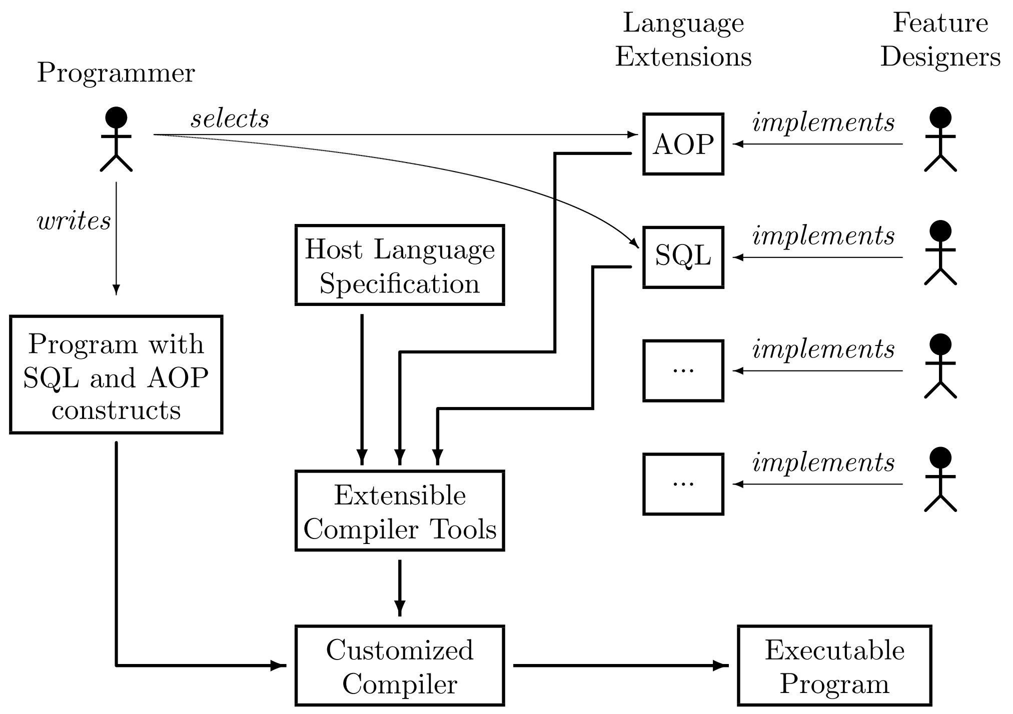

Fig. 1. Using Extensible Languages and Language Extensions.

From the programmer's perspective, importing new language features should

[1] This compiler is freely available from `http://www.eclipse.org/aspectj`.

be as easy as importing a library in a traditional programming language in order to make it available to the program. They need only *select* the language extensions they want to use and write their program to use the language constructs defined in the extensions and the host language. Following the examples above, the programmer could select the AOP and SQL language extensions. They need not know anything about the implementation of the host language or the language extensions. As the programmer sees it, the specifications for the selected language extensions and the host language are provided to the extensible compiler tools that generate a customized compiler. This compiler implements the unique combination of language features that the programmer needs to address the particular task at hand. This compiler takes as input their program written with constructs from the host language and the language extensions and generates an executable program. Thus, there is an initial "compiler generation" step that the tools, at the direction of the programmer, must perform. Language extensions are not loaded into the compiler during compilation.

The perspective of the feature designer is somewhat different. Feature designers are sophisticated domain experts. Besides being knowledgeable about their domain, they must also be somewhat knowledgeable about programming language design. In our framework, they will need to understand the implementation of the host language to some degree. This depends on how sophisticated the language extension is. In the case of the AOP and SQL language extensions, this is non-trivial, but in our opinion it is reasonable. Critically, language feature designers do not however need to know anything about the implementations of other language extensions. If such knowledge was required, then the modularity that we seek in language extensions will be lost. The programmer should be able to select the set of language extensions that they need – the feature designers of these language extensions will not be aware of the other language extensions being imported by the programmer.

It is worth noting that some language extensions may be relatively simple and introduce constructs that are not much more complicated than macros. These may be written by knowledgeable programmers. But our emphasis here is to make a clear distinction between the feature designers which must understand, to some degree, the implementation of the host language and the programmers which need not understand the implementation of the host language or any language extensions.

1.3 Limits of language extensibility

It is important to note that we do not claim that all modifications and enhancements to a language can be implemented as a language extension. As will become clear in Section 3, our framework allows for three types of language features to be added to a language as a language extension: (*i*) new language constructs, (*ii*) new semantic analyses, and (*iii*) new program transformations. Part of the specification of a new language construct is how it is translated into constructs in the host language. Thus, language constructs that can not be implemented by constructs in the host language can not easily be specified as modular language extensions in our framework. For example, some implementations of generics in Java [47,14] can not be translated into correct Java programs but must be mapped directly into Java byte-codes. [2]

Another potential limitation on extensibility in our framework comes from the fact that there are a few techniques for increasing the extensibility of a host language that must be written into the specification of the host language from the beginning. We claim that using the attribute grammars with forwarding formalism of our framework for specifying languages automatically yields a highly extensible host language. But there is an additional consideration that must be made in the host language specification that further increases the extensibility of the host language. As is shown in Sections 3.2.3.2 and 3.2.3.3, a specific type of transformation can be added to a host language as a language extension only if the appropriate extension points are explicitly specified in the host language specification. These extension points enable a type of rewriting (that goes beyond the rewriting enabled by forwarding alone) that is otherwise not available. These extension points are not designed to enable a specific language extension but to enable a general technique that can be used to implement a wide variety of language extensions. They do, however, highlight the fact that some thought must be given regarding the tools that will be needed by language extension feature designers when the host language is specified.

Despite these limitations, we believe that a great many expressive and powerful language extensions can be expressed in our framework. The example in this paper of implementing constructs from aspect-oriented programming in our framework as a modular language extension is meant to demonstrate that point. AOP provides powerful and expressive constructs to programmers and these require considerable semantic analysis and program transformation capabilities. Our framework provides these capabilities to feature designers and thus we believe that many domain-specific and general-purpose language extensions can be implemented in our system.

The remainder of this papers is organized as follows. Section 2 describes the AspectJ aspect constructs that are implemented as language extensions. Sec-

[2] This is because in method overloading in Java byte codes the return type of the method is also used, along with the input parameter types, in determining which method will be used. In Java, only the input parameter types are used.

tion 3 defines the attribute grammar framework and host object-oriented language to which the aspects are added. Section 4 shows how aspect language constructs can be specified in a modular and additive fashion to be incorporated into the host language. Section 5 illustrates how new point cut designators based on the underlying semantics of the object program can be explored and specified. Section 6 describes related work and Section 7 concludes with a discussion of what was achieved and future work.

2 Aspect oriented programming with AspectJ

Aspect-Oriented Programming provides language constructs that support the modularization of "cross-cutting concerns" such as error checking, logging, and monitoring. Without AOP, these concerns are typically implemented by code that is scattered across the entire program since it does not fit into the primary organizational structure of the language, but instead cuts across it. In object-oriented languages this organizational structure is implemented by a class hierarchy. AOP allows these cross-cutting concerns to be stated in a single location - a module called an "aspect". This section describes some of the aspect-oriented language constructs in AspectJ [35]. These are the constructs that are implemented as a modular language extension in the following sections. Although only a few of the language constructs in AspectJ are covered, these will be sufficient to provide an understanding of how aspects can be added to a host object-oriented language. We believe that the remaining AspectJ constructs pose no fundamental difficulties to our model.

Any incarnation of aspect-oriented programming must specify three things: (*i*) what the "join points" are, (*ii*) how the join points will be identified, (*iii*) and how the computations at the join points may be modified. How AspectJ specifies these is described below.

In AspectJ, a dynamic join point model is used and thus the *join points* are events that occur during the execution of the program, for example method calls and returns. Our examples are only concerned with method calls, thus this paper considers the join points to be all the method calls that occur during the execution of a program.

A *point cut* is a set of join points and a *point cut designator (pcd)* is a mechanism for specifying a point cut. Of interest is the *call* point cut designator. The point cut designator *call* (*signature*) will match method calls with type signatures that match the one in the call point cut designator. Such a signature consists of a class name or object variable, a method name and a list of parameters specified either as a type name, a variable, or a wild-card (written as "..." or "*"). A matching test, discussed below, determines if a method call matches a call point cut designator by examining the method call's object, method and parameters to determine if their types match those in the point cut designator signature. 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. We will refer to a method call construct in the program a *static join point* since it represents possibly many dynamic join points.

In AspectJ, the behavior of these join points can be modified by executing a piece of code either before or after the join point event and by altering the values at the join point. *Advice* is used to modify the behavior of join points. An advice construct consists of a possibly empty list of variable declarations, a point cut designator and the *advice code* that is to be executed either before, after or around the join points that match the point cut designator. The variable declarations identify the data at the join point that the advice code will access. The *weaving* process in essence[3] inserts the advice code before, after, or around the affected join points. Although the weaving process discussed here is done statically, the decision to execute the advice code may have to be made dynamically. Consider, for example, a method call `q.setX ( 4 )` and the advice declaration:

```
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 will 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 it can not always be statically determined if `q` is an instance of class `Point` and must therefore delay the decision to execute the advice code until run-time. In this case, the advice code is woven inside an *if-then* statement to perform this check at run-time. Thus, the weaver will generate the code

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

[3] AspectJ performs weaving at the byte-code level but the effect of weaving can be seen by the source-level weaving described in this paper.

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

The process of matching, at compile time, a point cut designator *pcd* against a method call construct will determine if

(1) none of the method call's dynamic join points match *pcd*,
(2) if all of the method call's dynamic join points match *pcd* or,
(3) if neither (1.) nor (2.) can be determined at compile-time and thus a run-time test must be performed.

In this final case, the matching test will also return the boolean expression that will be used in the run-time test. In case (1.), the match test returns a *NoMatch* value indicating that weaving should not be done for this advice. For (2.), the match test will return a *Static* σ value in which σ is a substitution that maps variables declared in the advice to constructs in the matched method call. In the example from above, $\sigma = [p \mapsto q, a \mapsto 4]$. This substitution is applied to the advice code to generate the actual advice code that is woven into place at the method call. In case (3.) it can not be statically determined if the point cut designator of a particular piece of advice matches all the dynamic join points for a method call construct. Thus, the test returns a *Dynamic* σ *test* value in which σ is the same type of substitution as above and *test* is the boolean expression code to be used in the dynamic test. The substitution is again applied to the advice code and this test code. The resulting code is used 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 possibly dynamic test code. For a particular *before* advice declaration with point cut designator *pcd* and advice code *code* the rewrite rules are as follows:

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

and

$$o.m(p_1 \ldots p_n) \quad \Longrightarrow \quad \{\ if\ \sigma(test)\ then\ \{\sigma(code)\}; o.m(p_1 \ldots p_n);\}$$
$$\textbf{if}\ match(pcd, o.m(p_1 \ldots p_n)) = Dynamic\ \sigma\ test$$

3 Language extension in attribute grammars

As is stated above in Section 1 and illustrated in Figure 1, the goal is to specify programming languages and language extensions in such a manner that programmers are able to easily import extensions into their host programming language. To achieve this, host languages and language extensions are specified as (fragments of) attribute grammars [37] in such a way that the simple "union" of the productions and attribute definitions in the language and extension specifications form a complete specification of the abstract syntax and semantics of the new extended language.

Section 3.1 discusses the two ways in which language extensions are commonly implemented in attribute grammars through the use of *forwarding*. Forwarding [62] is a mechanism that allows one to mimic a simple term rewriting process in the attribute grammar framework; it is defined below in Section 3.2.

Section 3.3 sketches the attribute grammar specification of the host language that will be extended with the aspect-oriented programming constructs shown in Section 2. This attribute grammar is the "Host Language Specification" shown in Figure 1. Section 4 describes the attribute grammar specification that defines these aspect-oriented programming constructs. This attribute grammar specification is the implementation of the "AOP" language extension shown in Figure 1. The "Extensible Compiler Tools" in that figure simply merge all these specifications into one attribute grammar specification that defines the "Customized Compiler". This resulting attribute grammar contains all of the productions defined in the host language and extension specifications with their associated attribute definition rules.

3.1 Two types of language extension

We identify two types of language extensions that are used to add aspect declarations and aspect weaving to an object-oriented host language. Although it is aspects that are added here, these techniques for language extension are general-purpose and are all that is required for many other kinds of language extensions. Recall that our goal is not just to add aspect-oriented programming features to a host language, but to build a framework in which many such extensions can be modularly added.

As one might expect, the first type of extension is the addition of new language constructs. This is done by simply adding new productions and their associated attribute definitions to the host language. This extends the abstract syntax and provides a semantics for the new language constructs. For example, a *for-each* loop construct, like the following, that iterates over all elements in a

Java Collection type is a language construct that might be added to Java:

`foreach` *Point x* `in` *Constellation* `do` *x.draw*()

This will draw each *Point x* in the *Constellation* collection. Another example is the *before* advice construct described in Section 2.

The second type of language extension mechanism provides a modification of existing language constructs. This can be done in two ways. The first simply allows for the addition of new attribute definitions to existing language construct productions for any new attributes that are introduced by a language extension. We may, for example, add a new attribute to specify the translation of the language to a new target language. The second is by specifying simple rewrite rules that rewrite a construct in an object program to one that implements the desired modification. Such rewrites typically wrap additional statements or expressions around the construct. This kind of extension implements a simple rewrite like the aspect weaving rewrites shown in Section 2. The type of rewriting that is used here is quite simple. In determining where to apply a rewrite a minimal amount of pattern matching is done to only check that the same production was used to construct the pattern and the potential tree to be rewritten. The main determinant in deciding where to perform a rewrite is a side condition that examines attribute values of the candidate attributed trees.

3.2 Forwarding in attribute grammars

3.2.1 Forwarding - motivation and definition

When defining new language constructs in language extensions it can be quite convenient to define them in terms of existing language constructs from the host language. This is done in much the same way that macros define new constructs by expanding into existing language constructs. The "expanded" macro code in essence provides the semantics of the macro. In our case, however, the new language constructs may also perform a significant amount of semantic analysis on their own through the attributes that are explicitly defined for the new construct's productions. These productions only rely on an expansion or rewriting to existing language constructs to provide definitions to the attributes that they do not explicitly define.

The idea of forwarding can be clarified by an example. Consider the `foreach` construct from Section 3.1. The *for-each* production that defines this language construct may define its own pretty-print attribute for displaying the programmer-written source code and it may check for programmer errors (*e.g.*, that *Constellation* does implement the Collection interface). To implement this, the *for-each* production would, for example, provide attribute definitions for a pretty-print attribute named *pp* and an error attribute named *errors*. However, it is desirable that the above *for-each* loop to, in essence, rewrite to the following *for loop* when we require values for attributes that are not explicitly defined by the *for-each* production. These are attributes for which there is not an attribute definition rule associated with the production.

```
{ Point x ;
  for ( Iterator iter = Constellation.iterator() ; iter.hasNext() ; )
       { x = (Point) iter.next() ;
         x.draw() ;  }
}
```

Consider the case when a *for-each* node in the abstract syntax tree is queried for one of the attributes, for example an attribute defining the construct's Java byte-code called *jbc*. In this case, the the query for *jbc* will be *forwarded* from the *for-each* node to the node representing the semantically equivalent (block containing the) for-loop. This node can then provide the value of this attribute. Since the block/for-loop construct is semantically equivalent to the for-each loop, this is appropriate. This process is diagrammed in Figure 2.

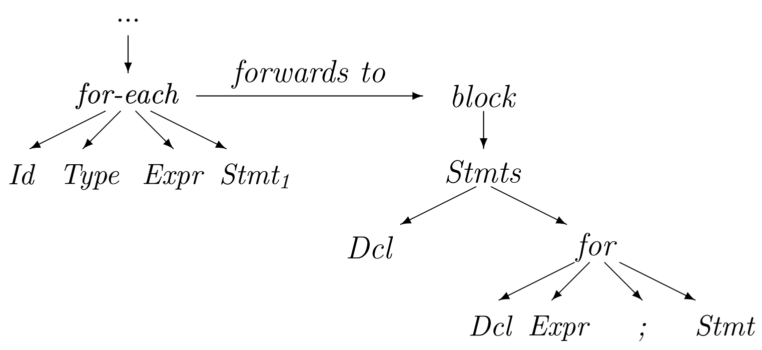

Fig. 2. Abstract syntax tree with forwarding

Forwarding [62] is a technique that provides default attribute values for productions and thus the attributed abstract syntax tree nodes that they construct. It complements other default schemes such as the `INCLUDING` construct in LIDO (the attribute grammar specification language for the Eli [27] system) that has the effect of automatically copying inherited attribute values to a node's descendants. A production that contains a *forwards-to* clause constructs an attributed tree from productions, its child trees and various (higher order) attribute values on it and its child nodes. This forwards-to construct is implicitly provided with the inherited attributes of the *forwarding* construct. Forwarding plays a role when the forwarding construct is queried for the value of an attribute that it does not explicitly define, that is, an attribute for which it does not provide a definition. When this happens, the value returned for the query is the value of that attribute on the forwards-to construct.

Forwarding can be used in implementing the language extension techniques mentioned above in Section 3.1 as the examples below will show.

3.2.2 Forwarding in adding new language constructs

To see how forwarding can be used in defining new language constructs consider the *foreach* production in Figure 3 that implements the for-each example from above. When a *for-each* construct is queried for its pretty-print attribute

```
foreach : $Stmt_0$ ::= Id Type Expr $Stmt_1$
      $Stmt_0$.pp = "foreach " + Type.pp + " " + Id.lexeme + " in " +
                    Expr.pp + " do " $Stmt_1$.pp
      $Stmt_0$.errors = if Type.typen.implements (Collection) then no-error
                        else mkError ... $Stmt_0$.pp ...
      forwardsTo parse " { 'Type' 'Id' ;
                           for ( Iterator 'iter' = 'Expr'.iterator() ;
                                   'iter'.hasNext() ; )
                             { 'Id' = ('Type')'iter'.next() ;
                               '$Stmt_1$' }
                         } "

      where iter = generate_new_unique_Id ()
```

Fig. 3. The production specifying the *foreach* loop extension.

pp it returns the value it explicitly defines, but when queried for its Java byte-code attribute, *jbc*, it forwards this query to the block containing the *for-loop* that implements the iteration. This construct then returns its semantically-equivalent *jbc* attribute value. This is illustrated in Figure 2. On the left is the original for-each tree generated by the parser and on the right is the block/for-loop tree generated by the *foreach* production when it is first queried for an attribute that it does not explicitly define. This *for-loop* construct is constructed by parsing the string in the forwardsTo clause. The unquote operator (written by wrapping the operand in single quotes, '_') allows the insertion of the child *Id*, *Type*, *Expr*, and $Stmt_1$ trees into the *for-loop* construct. By using forwarding in this way, we do not need to concern ourselves with *when* the "rewrite" takes place since both trees exist simultaneously to provide the attribute values as they are queried. Forwarding causes the abstract syntax tree to grow during the evaluation of attributes. A standard parser will generate an original abstract syntax tree as expected, but the productions that are used to build this tree may use forwarding to construct additional trees during the attribute evaluation processes.

Forwarding is similar to macro expansion in that both reuse the semantics of existing language constructs, but unlike macros, forwarding productions also define semantics, as attributes, that generate proper error messages, something macro systems cannot do. The reuse of existing language constructs means that language feature designers do not have to know all the details (that is, attributes) of the forwards-to constructs.

3.2.3 Forwarding for modifying existing constructs

3.2.3.1 Specifying new attributes Many extensions implement a new semantic analysis by providing definitions of new attributes to all productions in the host language attribute grammar and the productions defined in the language extension. Consider an extension that defines a translation of the host language to the .NET [57] intermediate language by defining an *il* attribute for all of the host language productions. If a programmer selects this extension as well as an extension containing the for-each loop defined above, can we be certain that the for-each loop will be able to provide its translation into the .NET intermediate language? The answer is yes. Since the .NET extension will have defined the *il* attribute for the host language for-loop, the for-each loop will get its value for the *il* attribute using forwarding just as it did for its *jbc* attribute. This is possible even though the for-each feature designer had no knowledge of the .NET language extension. This is not possible with standard higher order attributes; they require a *modularity-destroying attribute definition* on the *for-each* production to explicitly copy the *il* attribute from the for-loop node to the for-each node [62]. We avoid such attribute definitions since they would need to be written by the programmer who combines the for-each extension with the .NET translation extension. This would violate the distinction between the programmer and feature designer that we seek.

3.2.3.2 Forwarding for simple rewriting Our first attempt at implementing the aspect weaving rewrite rules may resemble the *for-each* production above. We could add the new "method call weaver" production:

```
methodCallWeaver : $Expr_0$ ::= $Expr_1$ Id $Expr_2$
        $Expr_0.pp = Expr_1.pp +$ "." $+ Id.pp +$ "(" $+ Expr_2.pp +$ ")"
        forwardsTo if ⟨ there is an applicable rewrite rule ⟩ then
                            ⟨ advice code ⟩ ;
                            methodCallWeaver $Expr_1$ Id $Expr_2$
                        else
                            methodCall $Expr_1$ Id $Expr_2$
```

This production defines a pretty-print attribute *pp* much like the *for-each* production. It also must determine if an applicable rewrite rule can be applied to method calls. It does this by checking that the pattern on the right hand side of a rewrite rule matches this method call and that the side condition of the rule is satisfied by the attributes on this method call. If this match is not successful then this production simply forwards to the standard non-weaving method call. Or more precisely, it forwards to the abstract syntax tree node constructed by the non-weaving method call production. However, if the match is successful, then this method call forwards to the sequential composition of the advice code (from the left hand side of the rule) and a weaving version of the method call that will repeat the process with any remaining rewrite rules.

This production gets its potential rewrite rules from an inherited *environment* attribute in much the same way that variable references look up their declarations in an environment attribute. These rewrites are generated at compile time from the aspect advice declarations found in the object program. The standard inherited environment attribute, called *env*, provides a convenient mechanism to move in-scope rewrite rules to the method calls where they may be applied. As is shown below, aspect advice constructs add rewrite rules to the environment and static join points, in our case, method calls, retrieve them from the environment.

This approach has a serious flaw, however. It requires that the parser use the *methodCallWeaver* production to construct the original abstract syntax tree. This prevents us from adding other language extensions that might also rewrite method calls in this way since each assumes that it be put into the original abstract syntax tree.

To avoid this problem and to allow simple rewrites like those required for aspect weaving, our framework defines extensible languages so that each production has an associated "wrapper" or "rewriting" production. The original and the wrapper productions have the same signature, but the wrapper-production defines only a very few specialized attributes. This wrapper production instead extracts a matching rewrite rule from the environment and forwards to its instantiated right hand side. If there are no matching rewrites, the wrapper-production forwards to the tree built by the corresponding non-wrapper production that does define all of the attributes that specify the semantics of the construct. This is a generalization of the *methodCallWeaver* production above.

```
$methodCall\_W : Expr_0 ::= Expr_1\ Id\ Expr_2$
        $\textbf{forwardsTo}\ forward$
  $\textbf{where}$
        $matchTree = methodCall\ Expr_1\ Id\ Expr_2$
        $forward = \textbf{case}\ getRWT\ Expr_0.env\ matchTree\ \textbf{of}$
                        $Nothing \rightarrow matchTree$
                        $Just(env', rwt\_func) \rightarrow (rwt\_func\ Expr_0.env)$
                                                  $`w\_inh`\ (env = env')$
```

Fig. 4. The *methodCall_W* production.

The method-call wrapper-production *methodCall_W*, that is used by the parser to build the initial AST is shown in Figure 4. This production calls the function *getRWT* with the current environment ($Expr_0.env$) and the tree (built with the non-weaving production) to determine if there are any applicable rewrites in the environment. If there are no such rewrite rules, *getRWT* returns the value *Nothing* and thus *methodCall_W* forwards to the tree *matchTree* that is built by the standard method-call production *methodCall* that does define attributes. If there is a match, *getRWT* returns a *Just* value containing an ordered pair. The first element is a new environment that does not contain the matching rewrite rule. The second is a function that generates the matching rewrite rule's instantiated right hand side. We create this right hand side construct (to forward to) by providing this function with the current environment ($Expr_0.env$). This environment contains the matched rewrite rule and it may be used on children of this construct. The production then forwards to this construct that has its *env* attribute defined here by an infix "with-inherited-attribute" operator '*w_inh*' to be the environment without the matched rewrite. This ensures that this rewrite is only applied once in this location.

Since the wrapper-production defines very few attributes, requests for attributes, such as *jbc*, are forwarded to the constructed tree that is the right hand side of a rewrite rule. This effectively simulates the destructive replacement normally done in term rewriting. A similar wrapper-production is defined for each attribute-defining production, though the only other one shown in this paper is the one for variable references. Although these wrapper-productions can be automatically generated from the attribute-defining ones they are written explicitly here to illustrate how forwarding can mimic this simple rewriting.

One may ask what happens if more than one rewrite can be applied. In this case, the other matching rewrites are still in the environment *env'* that is seen by the forwarded-to tree. This tree is built using the method-call wrapper-production as well, and thus the same process is repeated and the additional rewrites are applied. Thus, the order in which the rewrite rules are applied depends on the order in which they appear in the environment. Section 7.2 briefly discusses how the dependency system used in Explicit Programming [15] could be used to determine this ordering.

Assuming that the advice declarations add the appropriate rewrite rules to the environment, this production will effectively implement the aspect weaving process. Thus, the remainder of the paper is devoted to showing how these rewrite rules and their associated match-functions are computed and added to the environment.

The rewriting process described in this paper is just a generalization of the rewriting process seen in the *for-each* example. In that case, however the "rewrite rule" is known by the production and it simply constructs the *for-loop* directly as opposed to extracting a rewrite rule from its environment whose right hand side is a for-loop implementing the for-each construct.

3.2.3.3 Wrapper productions and attributes in the host-language

Wrapper productions, like *methodCall_W* in Figure 4, exist for all of the semantics-defining productions in the attribute grammar. These wrapper productions, as well as the different variable reference productions defined in Section 3.3.2, are part of the host language infrastructure of which feature designers need to be aware. In essence they provide extension points on which feature designers can build. Feature designers also need to be aware of some of the attributes that are used in the host language specification. For example, the *env* attribute is used for making variable and type declarations available to the parts of the program where they may be used. This attribute is also used by the wrapper productions who query the environment attribute for any rewrite rules which may apply to it. These productions and attributes are part of the infrastructure of the host language and the feature designer must be aware of them. However, most of the attributes used in the host language attribute grammar do not need to be understood by the feature designers to design language extensions.

3.3 Attribute grammar specification of the host language

In the remainder of this section we present our attribute grammar based framework used for specifying modular definitions of languages and the specification of the host language. In this framework, Knuth's attribute grammars are extended with higher order attributes [64,53], reference attributes [29] and forwarding [62]. We also specify some of the productions and attribute definitions that define the host language. But, because most are what one would expect, we discuss only the most important definitions.

Many of the significant production signatures of the host language are shown in Figure 5. The non-terminals in the abstract syntax grammar include {*Expr*, *Dcl*, *Type*, *Id*}. For simplicity we do not make a syntactic distinction between expressions and statements; both are represented by the non-terminal *Expr*. Statements are simply side-effecting expressions. The *Dcl* non-terminal represents variable and type declarations, and *Type* represents type expressions, including type identifiers.

assign :	*Expr*	::=	*Expr Expr*
block :	*Expr*	::=	*Dcl Expr*
exprSeq :	*Expr*	::=	*Expr Expr*
ifthenelse :	*Expr*	::=	*Expr Expr Expr*
bindingVarRef :	*Expr*	::=	*Id*
varRef :	*Expr*	::=	Dcl^n *Id*
methodCall :	*Expr*	::=	*Expr Id Expr*
varDcl :	*Dcl*	::=	*Id Type*
classDcl :	*Dcl*	::=	*Id Type*
methodDcl :	*Dcl*	::=	*Id Type Dcl Expr*
dclSeq :	*Dcl*	::=	*Dcl Dcl*
classType :	*Type*	::=	*Type Dcl*
intType :	*Type*	::=	ϵ

Fig. 5. A selection of host language production signatures

3.3.1 Abstract syntax trees and abstract semantic trees

Attribute values can range over an unspecified set of primitive values, such as integers and strings, and a set of higher order values, such as (references to) tree nodes and tree building functions. A *node* can be seen as a record containing fields for inherited and synthesized attributes. The types of nodes correspond to the non-terminal symbols of the grammar. We will superscript these symbols with an n to indicate a node's type. For example, $Expr^n$ denotes the type of nodes that contain the inherited and synthesized attributes for expressions. $Expr^s$ ($Expr^i$) denote records that contain just the synthesized (inherited) attributes for an *Expr* non-terminal. The dot (.) notation is used for referencing attribute values on such nodes; thus $n.a$ is the value of the attribute a on node n.

Johnsson [30] and Swierstra [39] use (*abstract*) *semantic trees* in treating attribute grammars as a *style* of lazy functional programming. These trees are defined as functions that map a set of inherited attributes to a set of synthesized attributes according to the productions and attribute definition rules of an attribute grammar. In higher order attribute grammars, such semantic trees are valid attribute values. This paper slightly modifies this definition 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 in order to distinguish them from nodes of the same non-terminal. For example, semantic trees for the *Expr* non-terminal, have the type $Expr^f$. This is just shorthand for $Expr^i \rightarrow Expr^n$. The adjective "abstract" is often dropped from these terms since this paper is primarily concerned with abstract, not concrete, syntax. The productions, and their attribute definitions, can also be interpreted as functions. For example, the method declaration production:

$$methodDcl : Dcl ::= Id\ \ Type\ \ Dcl\ \ Expr$$

can be seen as a function of type

$$Id^f \times Type^f \times Dcl^f \times Expr^f \rightarrow Dcl^f$$

The non-terminals from the right hand side of the production (*Id*, *Type*, *Dcl*, *Expr*) become the semantic trees that are inputs for the production-function (Id^f, $Type^f$, Dcl^f, $Expr^f$).

This paper will also use the superscript type notation to refer to values of these types. As is the norm, we will use numeric subscripts to distinguish between non-terminal symbols of the same type. Since the non-terminals correspond to both nodes and semantic trees, distinguishing superscripts n and f are used in the attribute definitions.

Note that these superscripts were not used on the productions in Section 3.2. They can, however, be inferred from their context. For example, in Figure 3 in the definition of the *pp* attribute, it is easy to see that the non-terminals in attribute references (*Type.pp* for example) would be superscripted with an n. Whereas the non-terminals in the forwards-to construct are used as trees and would thus be superscripted with an f.

3.3.2 Host language attribute definitions

An important attribute in the host language is the environment attribute *env*. This is an inherited attribute that is used to make variable and type declarations available to variable and type references. It is defined so that the scope rules of the host language are enforced. It is also used to make the rewrite rules generated by the advice declarations available to the static join points (the method calls) that they may affect. The type of *env* is named *Env* and is a list of tagged elements. The tag determines the purpose of each entry and the types of values stored in that element. For variable declarations, the tag is *VarDcl* and the element component is an ordered pair of type ($String, Dcl^n$). This pair contains the name of the variable being declared and a reference to the variable's attributed declaration node in the abstract syntax tree. With this reference, variable references can query attribute values directly from their declaration and thus we do not need to create, and pass to variable references, a complex symbol table structure that contains all of the information that they may need. As expected, scope rules are enforced by adding nested declarations to the front of the list and 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, also has the type *Env*, and is used to gather *env* declaration entries from declarations. Some productions and attribute definitions for these attributes are shown in Figure 6.

$$
\begin{array}{l}
\textit{dclSeq} : \textit{Dcl}_{\mathit{0}} ::= \textit{Dcl}_{\mathit{1}}\ \textit{Dcl}_{\mathit{2}} \\
\quad \textit{Dcl}^n_{\mathit{0}}.\textit{defs} = \textit{Dcl}^n_{\mathit{1}}.\textit{defs} + \textit{Dcl}^n_{\mathit{2}}.\textit{defs} \\
\quad \textit{Dcl}^n_{\mathit{2}}.\textit{env} = \textit{Dcl}^n_{\mathit{1}}.\textit{defs} + \textit{Dcl}^n_{\mathit{0}}.\textit{env} \\
\\
\textit{block} : \textit{Expr}_{\mathit{0}} ::= \textit{Dcl}\ \textit{Expr}_{\mathit{1}} \\
\quad \textit{Expr}^n_{\mathit{1}}.\textit{env} = \textit{Dcl}^n.\textit{defs} + \textit{Expr}^n_{\mathit{0}}.\textit{env} \\
\\
\textit{varDcl} : \textit{Dcl} ::= \textit{Id}\ \textit{Type} \\
\quad \textit{Dcl}^n.\textit{defs} = [\ \textit{VarDcl}\ (\textit{Id}^n.\textit{lexeme}, \textit{Dcl}^n)\ ] \\
\quad \textit{Dcl}^n.\textit{type} = \textit{Type}^n
\end{array}
$$

Fig. 6. Definitions of *env* and *defs*.

In order to avoid inappropriate name capture of variable references when moving semantic trees around for the rewriting process, the host language specifications define three productions for variable references: *bindingVarRef*, *varRef_W* and *varRef*. These are defined in Figure 7. The production *bindingVarRef* looks up variable declarations in the environment *env* using the *dcl_lookup* function. It returns the variable's declaration node. The production then forwards to the variable reference wrapper-production *varRef_W* that builds its tree from the declaration *node* of type Dcl^n and the identifier semantic tree. It does not need to look up the identifier in the environment since it has it already as a parameter. This production is thus slightly different from others in that one of its arguments is not a semantic tree but a node of type Dcl^n. When writing productions in the traditional BNF form seen in Figure 7, among others, the f superscript is assumed unless indicated otherwise as in the case with Dcl^n. The *varRef_W* wrapper-production is similar to the *methodCall_W* wrapper-production from Figure 4; in addition, the definition of the *this_f* attribute is presented. This attribute is used to extract, from any node, the semantic tree that was used to create it. It is defined in a similar fashion on all productions except for *bindingVarRef*, which receives a semantic tree from its forward-to construct. The value of this attribute is used as a semantic tree that we may want to use in a different part of the program. This causes semantic trees that are passed to new locations in the program to already have their variables bound to their declarations since this tree is built without using *bindingVarRef*. We can thus guarantee that name binding only occurs in the original abstract syntax tree and that moving trees into new locations that may have new environments does not cause any inappropriate name capture. Note, however, that it is still possible to incorrectly move a variable outside of its scope. The *varRef* production is used after all variable reference rewrites have been done and it defines the appropriate attributes such as *type* and *varDcl*, a link to its declaration node.

```
$bindingVarRef : Expr ::= Id$
    forwardsTo $varRef\_W\ dcl^n\ Id^f$
  where $dcl^n = dcl\_lookup(Expr^n.env, Id^n.lexeme)$

$varRef\_W : Expr ::= Dcl^n Id$
    $Expr^n.varDcl = Dcl^n$
    $Expr^n.this\_f = varRef\_W\ Dcl^n\ Id^f$
    forwardsTo $forward_f$
  where $matchTree = varRef\ Dcl^n\ Id^f$
        $forward_f =$ case $getRWT\ Expr^n.env\ matchTree$ of
                    $Nothing \to matchTree$
                    $Just(rwt, env') \to rwt$ `$w\_inh$` $(env = env')$

$varRef : Expr ::= Dcl^n Id$
        $Expr^n.varDcl = Dcl^n$
        $Expr^n.type = Dcl^n.type$
        $Expr^n.isVarRef = True$
```

Fig. 7. Variable reference productions.

Types in our host language are supported by a *type* attribute whose type is $Type^n$, a reference attribute, that references the variables type by following the similarly named attribute on the variable's declaration. The *classType* production defines an *isSubTypeOf* attribute whose value is a function that takes a $Type^n$ node and returns a boolean value specifying if that parameter type is a subclass of the class being defined. It's definition is elided but straightforward. Examples of these productions can be seen in Figure 8.

```
$intType : Type ::= \epsilon$
    $Type^n.size\_in\_bytes = 4$

$classType : Type_0 ::= Type_1\ Dcl$
    $Type_0^n.isSubTypeOf = \lambda type_n \to ...$
```

Fig. 8. Type productions.

3.3.3 Attribute evaluation

With the use of forwarding, potentially many trees may be created and many attribute values may be unnecessarily computed. Consider the *for-each* forwarding example. Evaluating all of the attributes on the child nodes of the *for-each* would be wasted effort. For example, there is no need to compute the *jbc* attribute on the child nodes of a *for-each*. This attribute will need to be computed on the children of the *for-loop* construct however. To counter this potential problem, we rely on lazy evaluation. Attribute values are not calculated unless they are needed. Our prototype system follows the example of Johnsson [30] and uses the lazy functional language Haskell [48] as our implementation language. Thus, forwarding does not pose any fundamental efficiency or scalability problems since only the (portions of) trees that are needed are generated and evaluated.

It is also worth noting that Augusteijn [5] reports that attribute grammar evaluators that rely on lazy evaluation have similar performance characteristics as attribute grammar systems that use an analysis of the data dependencies between attributes to compute a strict evaluation scheme.

4 Defining aspect constructs as language extensions

This section provides the specification of the before advice declaration and shows how it creates the rewrite rules that implement aspect weaving. Figure 9 shows some of the productions defining the abstract syntax of the aspect language features, some of which make use of a new point cut designator *PCD* non-terminal. We need to provide semantics, that is, attribute definitions, for these productions in order to add them to the language defined above. We will discuss the definition of the advice and point cut designator constructs and then show how they are used to generate a rewrite rule that is put into the environment *env*. We have already seen above how the weaving process is carried out by the application of these rewrite rules in the production *methodCall_W*.

$$
\begin{array}{llll}
\mathit{beforeAdvice} : & \mathit{Dcl} & ::= & \mathit{Dcl}\ \mathit{PCD}\ \mathit{Expr} \\
\mathit{callPCD} : & \mathit{PCD} & ::= & \mathit{objPCD}\ \mathit{mthPCD}\ \mathit{prmPCD}
\end{array}
$$

$$
\begin{array}{llllllll}
\mathit{classPCD} : & \mathit{objPCD} & ::= & \mathit{Id} & \mathit{objectPCD} : & \mathit{objPCD} & ::= & \mathit{Id} \\
\mathit{methodPCD} : & \mathit{mthPCD} & ::= & \mathit{Id} & \mathit{varPCD} : & \mathit{prmPCD} & ::= & \mathit{Id} \\
\mathit{wildCardPCD} : & \mathit{prmPCD} & ::= & \epsilon & & & &
\end{array}
$$

Fig. 9. Aspect production signatures.

4.1 Advice declarations

The *beforeAdvice* declaration production in Figure 10 defines the rewrite rule that will implement aspect weaving and its associated matching function and adds it to the environment. The advice production generates a declaration from a (possibly compound) declaration Dcl_1, a point cut designator *PCD* and the advice code *Expr*. Since the declaration Dcl_1 declares the (pattern) variables that are used in the point cut designator and the advice code, its declarations ($\mathit{Dcl}_1^n.\mathit{defs}$) are added to the environment of the point cut designator and advice code. Since this declaration is not needed after the weaving process, it forwards to the empty declaration production *dclSkip*.

The rewrite rule *rwt_rule* defined by *beforeAdvice* is added to the environment

```
beforeAdvice : $Dcl_0$ ::= $Dcl_1$ PCD Expr
    $PCD^n.env = Dcl_1^n.defs + Dcl_0^n.env$
    $Expr^n.env = Dcl_1^n.defs + Dcl_0^n.env$
    $Dcl_0^n.defs = [\ RWT\ rwt\_rule]$
    **forwardsTo** dclSkip
 **where** rwt_rule sjp = **case** ($PCD^n$.match_sjp) sjp **of**
                              NoMatch → Nothing
                              Static s → Just (e → exprSeq
                                   ($Expr^f$ ‘w_inh‘
                                             (env = s + $Expr^n$.env))
                                   (meth_call sjp e))
                              Dynamic s test_f → Just (e → exprSeq
                                   ((ifthen test_f $Expr^f$) ‘w_inh‘
                                             (env = s + $Expr^n$.env))
                                   (meth_call sjp e))
        meth_call sjp e  =  methodCall_W
                                (sjp.object_n.this_f ‘w_inh‘ (env = e))
                                (sjp.meth_n.this_f)
                                (sjp.param_n.this_f ‘w_inh‘ (env = e))
```

Fig. 10. *beforeAdvice* production.

in an element tagged by RWT to distinguish it and other rewrites from other kinds of declarations in the environment, such as the variable bindings shown above. The function *rwt_rule* has the type $Expr^n \rightarrow Maybe(Env \rightarrow Expr^f)$. This function takes an expression node (the potential static join point *sjp*[4] in the program) and tests if it matches the point cut designator by calling PCD's *match_sjp* function (defined below) on *sjp*. The function *match_sjp* will return a value of type $Match$ where $Match$ is defined in Figure 11. If

$$\textbf{data}\ Match = NoMatch\ \mid\ Static\ Env\ \mid\ Dynamic\ Env\ Expr^f$$

Fig. 11. *Match* type.

match_sjp doesn't match and returns a *NoMatch* value, then the rewrite rule returns a *Nothing* value indicating that this rewrite does not apply. Otherwise there is a *Static* or *Dynamic* match and *rwt_rule* returns a *Just* value containing the function that generates the semantic tree that is to be forwarded to the static join point. This is the rewrite function *rwt_func* that is returned from the function *getRWT* seen above in *methodCall_W*. This function takes as a parameter the environment (*e* in Figure 10) that has not had this rewrite rule removed. This is used to ensure that this rewrite can be applied to the static join point's object expression *sjp.object_n* and argument

[4] The term "static" here is used to emphasize that nodes passed to the *rwt_rule* function are *static* entities that represent are potential *dynamic* (execution-time) join points. The resulting *Match* value may be a *Static* or *Dynamic* value.

expression $sjp.param_n$ if need be. The method-call productions define the attributes *object_n*, *method_n* and *param_n* to make its children accessible for this test and so that they can be used to construct the rewritten method call built by *meth_call*.

In the case of a static match, *match_sjp* returns *Static s* where s is the list of rewrites to map the pattern variables in the advice code to their instantiations from the join point. In our example from Section 2, this s represents $\sigma = [p \mapsto q, a \mapsto 4]$. The environment for the advice code is thus these rewrites defined in s and in addition to the original environment. The *match_sjp* test returns the rewrite rules that are used to rewrite pattern variables to expressions at the static join point. The code that will replace the matched method call is a sequence of two expressions created by the production *exprSeq*. The first expression is the advice code $Expr^f$ from this advice declaration. The phrase '*w_inh*' ($env = s + Expr^n.env$)) ensures that the substitutions s are added to its environment. The second expression is the method call defined by *meth_call*.

Similarly, in the case of a dynamic match, *match_sjp* returns *Dynamic s test_f* where s is as before and *test_f* is the test code that must be executed at run-time to check if the run-time join point matches the PCD. The *if-then* statement that conditionally executes the advice code has the same environment as the advice code in the static match case. This statement is created using the *ifthen* production in the phrase *ifthen test_f* $Expr^f$ as seen in the *Dynamic* clause. The following subsection describes how the *match_sjp* function works to generate the necessary pattern variable rewrite rules and test code.

4.2 Point Cut Designators

Point cut designator productions define a *match_sjp* function-valued attribute that tests if the point cut designator matches a static join point that is provided to this function as a parameter. This function takes this $Expr^n$ parameter and returns a value of type $Match$ defined in Figure 11. The behavior of this function was sketched in Section 2 and its implementation in this language framework is shown in Figure 12. The *methodCallPCD* production is used to match *call* point cut designators, such as "`call p.setX ( a )`" shown above in Section 2, against method calls by calling the *match_sjp* function on its child PCD nodes and then combining their results with $\wedge_{pcd}$. This operator has type $Match \times Match \rightarrow Match$ and is shown in Figure 13. It combines matches in the expected way, combining the substitution environments and dynamic test code of the parameter matches.

The *objectPCD* production is used when an object variable is used in a point

```
$methodCallPCD : PCD ::= objPCD\ mthPCD\ prmPCD$
    $PCD^n.match\_sjp = \lambda\ sjp \rightarrow\ objPCD^n.match\_sjp\ (sjp.objRef) \wedge_{pcd}$
                                      $mthPCD^n.match\_sjp\ (sjp.methRef) \wedge_{pcd}$
                                      $prmPCD^n.match\_sjp\ (sjp.paramRef)$
$objectPCD : objPCD ::= Id$
    $objPCD^n.match\_sjp$
        $= \lambda\ sjp \rightarrow$ **if** $sjp.type.isSubTypeOf\ \ objPDC^n.type$
                    **then** $Static\ [RWT\ (varRefRWT\ Id^n\ sjp)]$
                    **else if** $objPDC^n.type.isSubTypeOf\ \ sjp.type$
                        **then** $Dynamic\ [RWT\ (varRefRWT\ Id^n\ sjp)]$
                          $(methodCall\ sjp.this\_f\ mkId(\text{“}instanceOf\text{”})$
                              $mkStrConst(objPDC^n.type.className))$
                        **else** $NoMatch$
$methodPCD : mthPCD ::= Id$
    $mthPCD^n.match\_sjp = \lambda Id' \rightarrow$ **if** $Id.lexeme = Id'.lexeme$
                                      **then** $Static[\,]$ **else** $NoMatch$
$varPCD : prmPCD ::= Id$
    $varPCD^n.match\_sjp = \lambda sjp \rightarrow\ Static\ [RWT\ (varRefRWT\ Id^n\ sjp)]$
$wildCardPCD : prmPCD ::= \epsilon$
    $varPCD^n.match\_sjp = \lambda sjp \rightarrow\ Static\ [\,]$
```

Fig. 12. Point cut designator productions.

cut designator as in the examples in Section 2. The object at the static join point method call is passed to *match_sjp* as the *sjp* parameter. If the *sjp*'s type (*sjp.type*) is a sub-type of the class type of the object in the point cut designator ($objPCD^n.type$), then we have a static match and we create a rewrite rule that maps *Id* to the matched object *sjp*. This rewrite rule becomes the environment passed back in the *Static* match. In our example in Section 2, this is the rewrite mapping *p* to *q*. The function *varRefRWT* that builds this rewrite rule is discussed below. In the case that the object type at the point cut designator ($objPCD^n.type$) is a sub-type of the matched object type (*sjp.type*) then we will need a run-time test to ensure that the actual *sjp* object is indeed of the proper class. The test code generated in this case uses the host language reflective `instanceOf` method to do this test. It is written using the abstract syntax here but it corresponds to the test condition `q.instanceOf("Point")` in Section 2.

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 environment, otherwise no match is returned. The *varPCD* is used for variables that are used in the point cut designator. Since the variables will match anything, we always generate a Static match with the required rewrite rule. The $\wedge_{pcd}$ function does need to check that when we combine two static or dynamic matches, that the environment rewrite rules do not rewrite the same variable to different expressions. For brevity, this check is not shown in our $\wedge_{pcd}$ function defined in Figure 13. The *wildCardPCD* also always provides a static match, but generates no rewrites. Also, the *andPCD* behaves as expected by calling the $\wedge_{pcd}$ operator defined in Figure 13.

```
$\wedge_{pcd}$ : $Match \times Match \rightarrow Match$
$m_1 \wedge_{pcd} m_2$ = case $m_1$ of
    $NoMatch \rightarrow NoMatch$
    $Static\ s_1 \rightarrow$ case $m_2$ of $NoMatch \rightarrow NoMatch$
                              $Static\ s_2 \rightarrow Static(s_1 + s_2)$
                              $Dynamic\ s_2\ t \rightarrow Dynamic(s_1 + s_2)\ t$
    $Dynamic\ s_1\ t_1 \rightarrow$ case $m_2$ of
                         $NoMatch \rightarrow NoMatch$
                         $Static\ s_2 \rightarrow Dynamic(s_1 + s_2)\ t_1$
                         $Dynamic\ s_2\ t_2 \rightarrow Dynamic(s_1 + s_2)\ (andExpr\ t_1\ t_2)$
```

Fig. 13. Point cut designator *and* operator.

4.2.1 Generating rewrite rules for advice variables

Recall that the advice code and the generated dynamic test code are copied to the join point where the variables that were declared in the advice declarations will need to be replaced by the appropriate constructs from the matched join point. The substitution σ is implemented by a set of rewrite rules, similar to those for rewriting method calls, and are returned as an *Env* environment. For one of these variable rewrite rules, the condition that tests if it applies to a construct in the advice code is shown in the utility function *varRefRWT* below. The *if* condition tests if the construct n is in fact a variable reference using the *isVarRef* boolean attribute that is true on variable reference expressions but false everywhere else. If it is, it tests if it has the same declaration as the advice variable *advice_var_n*. Nodes have a simple reference equality test.

```
$varRefRWT :: Id \rightarrow Expr \rightarrow Expr \rightarrow Maybe(Env \rightarrow Expr^f)$
$varRefRWT\ advice\_var\_n\ sjp\ n$
    = if $n.isVarRef \wedge n.varDcl = advice\_var\_n.varDcl$
      then $Just(\ e \rightarrow sjp.this\_f)$ else $Nothing$
```

If this test succeeds, then we want to rewrite the advice variable to the semantic tree extracted from the matched static join point using the *this_f* attribute.[5]

[5] Note that $Expr^n$s that are variable references define *varDcl*, but others do not. Allowing this simplifies our presentation here at the expense of breaking the attribute grammar rule that all nodes of the same non-terminal type define the same attributes.

This section has shown how advice declarations can be specified as a modular language extension and described how they specify the rewrite rules, for method calls and advice variables, that implement aspect weaving.

The critical feature here is that the attribute grammar fragments that implement these language features can be added to the attribute grammar specifying the host language without needing to modify the host language specification. This has important implications - the most important being that no knowledge of the underlying attribute grammar is required of the programmer to extend his or her host language with these aspect-oriented language features. Thus, a programmer can easily import these features into his or her programming environment without needing to be aware of the implementation of the host language or the language extension. The feature designer, on the other hand, does of course need to know some aspects of the implementation in order to implement the language extension.

5 Exploring extensions to aspect-oriented programming

An open extensible compiler framework, like the one proposed here, does provide an environment which supports the exploration of new language features. These features may be new language constructs or new analyses that either ensure the correct usage of the introduced language constructs or trigger optimizing transformations. This paper was partially inspired by the Aspect Sand Box Project [65] that provides a set of tools that enable programming language researchers to experiment with different incarnations of aspect-oriented programming. Our extensible compiler framework also provides an environment for language experimentation but, when the host language is a complete modern programming language, does not restrict one to a minimal language calculus in which to explore new ideas. Instead, one can experiment in the context of a complete language.

An extensible compiler based on attribute grammars can expose various semantic analyses provided by the (specification of) the host language in the form of attributes that can be referenced by a language extension. Figure 12 has shown how the point cut designators make use of the static typing performed by the host language attribute grammar. The *objectPCD* production uses the *isSubTypeOf* attribute, defined by the host language specification, to define the *match_sjp* attribute function. This function tests if there is no match, a static match, or a dynamic match between the point cut designator and the static join point parameter, *sjp*.

There are other semantic analyses performed by the host language compiler that may also be of use to a point cut designator construct. Below we describe a new point cut designator, that one might experiment with in the context of an extensible compiler framework. Our goal here is not to propose the following point cut designator as one that will solve many outstanding issues with aspect-oriented programming, but to simply illustrate the type of exploration one can do with an open, extensible compiler framework. Thus, we do not provide its complete definition but show only the components of interest.

Another type of semantic analysis that might be of use is an analysis of defined variables that can help determine which methods in a class change the state of the object. It is a common idiom in aspect-oriented programming to use such methods to trigger some sort of update event. Consider the following advice declaration:

```
after (Point p) : call p.set*(...)   {  Display.update(); }
```

The goal here is to capture all calls to the methods that change the state of a *Point* object so that after this change has occurred the *Display* can be updated. The assumption that is made is that the person who wrote the *Point* class has consistently named the methods so that all methods that change the object's state have names beginning with the string "set". Below, we show a point cut designator that can capture this intention of the above aspect directly by referring to a use-def analysis computed for each method. By doing so, we may no longer need to rely on a specific naming convention that may not always be followed by the programmer.

To capture the above intention, we might define a new point cut designator called `call_cos` so that we may write

```
after (Point p) : call_cos p.*(...)  { Display.update() ; }
```

Here, `call_cos` is a point cut designator, much like `call`, except that it only matches on methods that *change the object's state.*

To implement this point cut designator, we will assume that the underlying host language has defined two attributes *willDefineDcls* and *mayDefineDcls* that are defined on method declarations. Both of these attributes are lists of declaration nodes, that is [*Dcl*]. The attribute *willDefineDcls* is defined as the list of declaration nodes of variables or fields that the analyses can be certain will be defined by the method. Since we can not statically compute precisely the declarations that are defined by a method, the host language also performs a "may define" analysis. The attribute *mayDefineDcls* is the list of declarations of variables or fields that may be defined by the method.

It is not our concern here how these attributes are computed, we only care that they can be referenced by the productions defining the point cut designator that checks for a change in the object's state. This production is named *methodCall_COS_PCD* and can be seen in Figure 14. This production has essentially the same form as *methodCall_PCD* in Figure 12 except for the addition of the call to the function *changeObjectState* which ensures that this point cut designator will match only on those method calls that change the object's state.

```
$methodCall\_COS\_PCD : PCD ::= objPCD\ mthPCD\ prmPCD$
    $PCD^n.match\_sjp = \lambda\ sjp \rightarrow objPCD^n.match\_sjp\ (sjp.objRef) \wedge_{pcd}$
                                      $mthPCD^n.match\_sjp\ (sjp.methRef) \wedge_{pcd}$
                                      $prmPCD^n.match\_sjp\ (sjp.paramRef) \wedge_{pcd}$
                                      $changeObjectState\ sjp$
  where
   $changeObjectState\ sjp =$
     if $sjp.methRef.dcln.mayDefineDcls \cap sjp.objRef.dcln.typen.fieldDcls = \emptyset$
     then $NoMatch$
     else
     if $sjp.methRef.dcln.willDefineDcls \cap sjp.objRef.dcln.typen.fieldDcls = \emptyset$
     then $Dynamic\ [\,]$  (...dynamic test code...)
     else $Static\ [\,]$
```

Fig. 14. Change object state point cut designator

This function is passed the static join point *sjp*, the method call node in the tree that is being checked to see if it matches the change of state point cut designator in some advice declaration. If the set of items that may be defined by the method (*sjp.methRef.dcln.mayDefineDcls*) has no declarations in common with the set of field declarations in the class defining the matched object (*sjp.objRef.dcln.typen.fieldDcls*) then this method can not change the state of the object and thus *NoMatch* is returned. (Note that we assume that *fieldDcls* contains field declarations inherited from super-classes as well.) If this is not the case, we then examine the set of declarations that will certainly be defined by the method (*sjp.methRef.dcln.willDefineDcls*). If any of these are fields on the object, that is the second intersection is not the empty set, then we can be certain that the method being called by *sjp* will change the state of the object and we thus return a static match – *Static* [ ]. If this intersection is empty, then we can not statically determine if this method will change the state of the object and must thus return a dynamic match. The dynamic test code that will guard the advice code is elided in Figure 14. The code that must be generated for this test is not trivial. It amounts to wrapping all assignments to fields with a code fragment that will record which field in which object was changed. This information can then be retrieved by the dynamic test code to determine if the method has changed the state of the object. This data structure is similar to what can be used to implement the dynamic tests required for the AspectJ `cflow` point cut designator. This point cut designator specifies the join points that occur within the control flow of a parameter point cut designator. For example, `cflow ( call ( Point.setX(...)))` matches join points that occur between the initiation and termination of the execution of a *Point*'s *setX* method body, that is, within its control flow.

While this point cut designator does seem useful, it does have some drawbacks. First, the "may define" and "will define" analysis can be expensive to perform and may thus slow the compilation process. Also, this point cut designator works fine with *after* advice but not *before* advice. This is because we can *record* the changes to an object's state, but of course we can not precisely predict them. This point cut designator does illustrate, however, how one may investigate different aspect-oriented constructs in the context of an extensible compiler.

6 Related Work

6.1 Language description and implementation techniques

There are other language extension systems and this section describes those most closely related to our work.

Intentional Programming: The goals of the extensible language framework described here are closely aligned with those of the Intentional Programming (IP) [51,17,63] project, which until recently was under development at Microsoft. That system also proposed to allow programmers to write their programs using language constructs from different domains and thus raise the level of abstraction of the programming language to that of the problem. "Intentions" were high-level programming constructs that were intended to allow a programmer to directly state his or her intent. The implementation of IP was based on abstract syntax trees associated with "questions" and "question handlers". Questions were similar to attributes and question handlers were similar to attribute definition rules. Forwarding was also used in IP and was one of its primary technical contributions. In IP, if a node of the abstract syntax tree did not have a question handler to a question it was asked, it would forward that question to another language construct that may have the required question handler. However, these notions are more precisely defined in attribute grammars and we have previously shown how forwarding can be used in that framework [62].

Attribute Grammars: The problem of modular language definition has received much attention from the attribute grammar community and there are many papers addressing the study of modularity and extensibility in this framework, *e.g.* [1,19,20,24,25,28,29,33,44,50,56,64]. Some of these proposed systems are guided by ideas from functional programming and use higher order functions and attributes in their quest for modular specification, while others are inspired by the object-oriented paradigm and employ inheritance and references to achieve a separation of concerns. In our framework, besides forwarding, the attribute grammars do use several of these extensions to the attribute grammars originally designed by Knuth [37]; these include higher order attributes [64] and reference attributes [29]. The primary difference between our work and others is how modularity is approached. The other cited works generally assume that the person composing a new language from modular attribute grammar language specifications is the language designer and is thus willing to write attribute grammar specifications in order to "glue" them together. Thus, these attribute grammar features are useful to the feature designers who use our system in specifying their language extensions. But they do not support the type of modularity that is required to allow the compiler generation tools in Figure 1 to create the customized compiler by simply combining the attribute grammar specifications of the host language and the selected language extensions. Forwarding [62] is required to support the modular language extensions described in this paper. A primary difference between forwarding and object-oriented extensions of attribute grammars is that the forwarded-to construct is computed dynamically, that is at attribute evaluation time, instead of determined statically via inheritance [44] when the attribute grammar is defined.

ASF+SDF The ASF+SDF [61] system is based on modular algebraic specifications and term rewriting, although primitive recursive schemes, a subclass of algebraic specifications are comparable to strongly non-circular attribute grammars [61, page 48]. Both ASF+SDF and our system allow for modular specification of languages and language extensions but it is not clear that ASF+SDF allows for the type of modularity that we seek. Also, we begin with attribute grammars (as opposed to general term rewriting) and add a very simple form of rewriting. We do this because the rewrites we are interested in have very simple patterns but complex side conditions that depend on attribute values that are not directly available in terms.

Macro Processing: Forwarding is similar to macro expansion in that the forwards-to construct is similar to an expanded macro body. Thus, this work is similar to macro systems like JSE [8], JTS [11], `<bigwig>` [13] and Maya [9]. But by using forwarding in attribute grammars, we can specify extensions that are difficult or impossible to express in these systems since the forwards-to construct can depend on semantic values of the forwarding language constructs. Such semantic values are defined via attributes that are part of the host language specification or added by the language extension. Some macro systems do make limited use of semantics, however. The semantic macros of Maddox [41] allow the macro expansion process to access, and be driven by, semantic information provided by the underlying compiler. It does not allow new semantic analyses to be added however. Also, Maya can base macro dispatch on the static type of macro parameters and thus is not purely syntactic. The only macro system, to our knowledge, that also allows attributes is the Scheme macro system McMacMic [38]. Some of these macro systems use special parsing techniques. We separate parsing from the evaluation of ASTs; the parser builds the initial AST with forwarding placeholder productions, thus removing the need for many of these system's parsing extensions.

Meta-object protocol systems: These systems [34,16,54,55] are in some respects similar to macro processing[6] in that some allow a limited amount of syntactic extension to the base language. However, they allow for rather powerful semantic analysis. Compile-time "meta-classes" define language extensions; an instance of such an extension in a object program is represented at compile-time by a "meta-object". The fields and methods specify the properties and compile-time behavior of the language extension and this allows one to specify powerful compile-time semantic analysis. There is a close relationship between attribute grammars and these systems; the meta classes, objects, fields and methods roughly correspond, respectively, to productions, attributed abstract syntax tree nodes, attributes and function-valued attributes in attribute grammars.

Other techniques: There has been interesting work in language composition based on Action Semantics [18] and in Denotational Semantics [40]. The techniques presented in these papers do not allow the same degree of flexibility that one has with attribute grammars and forwarding, however. For example, the *for-each* extension in Section 3.2 defines its own error checking attributes to ensure that the programmer is given error messages relating to the code he or she wrote, not what it translates to. Such capabilities allow the language extension constructs to be as fully developed as language constructs in the host language.

6.2 Implementation techniques for Aspect-Oriented Programming

It is worth noting that other macro and attribute grammar systems have been used to implement aspect-oriented programming constructs. Maya has been used to implement aspects as a language extension [10], but it implements dynamic aspect weaving instead of the static weaving presented here. Aspect-Cool [6,7] is an experimental language that has been implemented using the Lisa attribute grammar system [45,44]. While this work also demonstrates that aspects can be implemented using attribute grammars, its intention is to explore different types language constructs for writing aspects. This work differs from ours in that it does not aim to support the type of modularity of language extensions that we require and was highlighted in Figure 1.

[6] OpenJava [54,55] is a hybrid macro systems that relies heavily on meta-objects.

Other techniques, besides the Aspect Sand Box Project [65] mentioned above, have been proposed for investigating different approaches to AOP. Fradet and Südholt [23] propose a generic aspect weaver that is based on the repeated application of program transformations to the object program until a fix-point has been reached. Their transformations are specified as tree-rewrite rules. Assman and Ludwig [4] propose a system for aspect weaving based on graph rewriting. Gray and Roychoudhury [26] show how a mature industrial program transformation system, the Design Maintenance System [12], can be used to build a generic weaver. Here, rewrite rules implementing weaving are written in PARLANCE, the language of DMS. Our work is similar to these in that we use a form of rewriting to implement the aspect weaving – in our case, forwarding is the technique that we use for rewriting. An extensible language framework, like ours, goes beyond these systems in that it also provides the means for specifying the language constructs that are used by the programmer to write aspects and advice code. Gray and Roychoudhury, for example, propose a defining a new language in DMS that has constructs resembling those in AspectJ that programmers can use to specify their aspects.

7 Conclusion

7.1 Discussion

This paper has shown how before-advice, one of the core aspect constructs from AspectJ, can be specified as a modular language extension. This is important because this language feature can be added to an existing object-oriented language specified in an extensible language framework by simply combining their defining attribute grammar fragments. No "glue" code was needed. Although there are some limitations to the types of language extensions that our framework can support, as described in Section 1, we feel confident that many powerful and expressive language features can be specified and implemented as modular language extensions. A contribution of this paper is to demonstrate that point by specifying some AOP constructs as language extensions.

We believe that many other constructs from AOP can be added with a similar amount of effort. More complex point cut designators such as *cflow* do require a substantial amount of work. But we hypothesize that this effort is no more than the effort required to produce a stand alone compiler using traditional techniques. For example, the cflow point cut designator is satisfied by checking if a certain pattern of method calls can be found on the run-time call stack. This can be specified using the techniques above. We will need to create and maintain a run-time data structure that keeps track of the methods that have been called. New method calls update this data structure and the *cflow* point cut designator will generate a dynamic test, not unlike the ones shown above, that checks at run-time if the pattern it specifies is satisfied by the run-time data structure.

Besides benefiting programmers, extensible compiler frameworks can provide a catalytic infrastructure for programming language research. For example, research in program annotations for information-flow aspects of security [46] could be carried out in a full, language, as opposed to a toy language, without requiring the implementation of the complete compiler. Here, we illustrate how an extensible compiler framework exposes its underlying semantic analyses and how this can provide a convenient arena in which to explore a new aspect-oriented language feature.

7.2 Future work

It was stated earlier that it is desirable for the process of language extension to be as easy for the programmer as importing a class. This is our long term goal. There are several hurdles we must clear before this can become a reality.

We've said nothing about how the concrete syntax of our language will be specified. Most parsing algorithms accept only specific classes of grammars, such as LR(k) and LL(k), and adding new concrete productions to a grammar can easily remove it from the desired class, thus causing the parser-generator to fail. For many extensions, however, a unique leading keyword, such as "before" in our aspect extensions, can make the extended concrete language parse-able. We have also shown in [62] how operator overloading can be handled with forwarding, so many of the types of syntactic extensions one would make can be handled. For more general syntactic extensions, generalized parsers [58] are useful. They can parse any context free grammar and recently more efficient implementations of these parsers have been developed [31,32,43]. For ambiguous grammars the use of the disambiguation filters of generalized LR parsers in [60] may be useful.

Hand-coding the rewrite functions as we've done using forwarding is rather straight-forward but tedious and makes the specifications hard to read. A better approach that we are investigating is ways in which the rewrite rules can be written as they are in Section 2 and automatically "compiled" into the attribute grammar that uses forwarding that was shown above.

To date we have mainly explored the expressive capabilities of our framework to ensure that powerful language extensions, such as the aspect constructs specified here, can be packaged as modular language extensions. A question that we are actively investigating now is "Can we ensure that language extensions, written by different feature developers, will always work together and not interfere with one another?" This is an important issue and we have identified a few questions whose answers may be helpful in finding a solution.

First, for a new language construct, how can we be certain that when it is queried for an attribute defined by another language extension it will be able to compute the value of that attribute? As we described in Section 3.2, if we (*i*) require all new language constructs to forward (directly or indirectly) to constructs in the host language and (*ii*) require attributes introduced in an extension to provide attribute definitions to all appropriate productions in the host language, then we can guarantee that all attributes will at least be defined.

Second, since changing the order in which rewrite rules, like the weaving rules described above, can change the behavior of the compiler, can feature developers provide some information to determine the order in which the rules should be applied? Here, we could follow the example set by Explicit Programming [15], a system for specifying language extensions primarily in the form of new modifiers for classes. These are syntactically similar to the *public* and *private* modifiers found in Java and other object-oriented languages. In this framework, a language extension may define a class modifier, perhaps *JavaBean*, that automatically generates the "getter" and "setter" methods for class fields that are used to package the class as a JavaBean component. Another language extension may add a field to a class. If both extensions are used together then the order in which the transformations that implement these extensions are made will determine whether or not the added field has getter and setter methods generated for it. Explicit Programming provides a simple dependency system in which each transformation states which dependencies it *satisfies* and *requires*. A transformation that requires a specific dependency will not be applied until the transformations that satisfy it are first applied. We are exploring how effective a similar system would be in our framework.

Clearly, answering these two questions does not solve the larger problem of ensuring language extension compatibility but only starts to frame the issue.

8 Acknowledgements

We would like to thank the anonymous reviewers for their suggestions that have helped to improve this paper.

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] M. Aksit, editor. *Proceedings of the 2nd international conference on Aspect-oriented software development.* ACM Press, 2003.

[3] Andrei Alexandrescu. *Modern C++ Design, Generic programming and design patterns applied.* Addison Wesley, 2001.

[4] Uwe Assman and Andreas Ludwig. Aspect weaving with graph rewriting. In *Proc. 1st Sym., Generative and Component-Based Software Engineering*, volume 1799 of *Lecture Notes in Computer Science*, pages 24–36. Springer-Verlag, 1999.

[5] A. Augusteijn. *Functional Programming, Program Transformations and Compiler Construction.* PhD thesis, Eindhoven Technical University, 1993.

[6] E. Avdičaušević, M. Mernik, M. Lenič, and V. Žumer. Aspectcool: an experiment in design and implementation of aspect-oriented language. *SIGPLAN Not.*, 36(12):84–94, 2001.

[7] E. Avdičaušević, M. Mernik, M. Lenič, and V. Žumer. Experimental aspect-oriented language - aspectcool. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 943–947. ACM Press, 2002.

[8] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 31–42. ACM Press, 2001.

[9] J. Baker and W. Hsieh. Maya: Multiple-dispatch syntax extension in java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281. ACM, June 2002.

[10] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 86–95. ACM Press, 2002.

[11] D. Batory, D. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–53. IEEE, 2–5 1998.

[12] Ira Baxter, Christopher Pidgeon, and Micheal Mehlich. DMS: Program transformation for practical scalable software evolution. In *Fifth International Conference on Software Engineering (ICSE)*, pages 625–634, May 2004.

[13] C. Brabrand and M. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. Partial Evaluation and Semantics-Based Program Manipulation, Workshop*, pages 31–40. Association of Computing Machinery, 2002.

[14] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications (OOPSLA98)*, pages 183–200. ACM Press, 1998.

[15] A. Bryant, A. Catton, K. De Volder, and G. Murphy. Explicit programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 10–18. ACM Press, 2002.

[16] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299. ACM Press, 1995.

[17] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[18] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Program.*, 47(1):3–36, 2003.

[19] D. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, 1990.

[20] 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.

[21] A. Fgabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schoenherr. The cgal kernel: A basis for geometric computation. In *Proc. Workshop on Applied Computational Geometry*, May 1996.

[22] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison Wesley, 2004.

[23] Pascal Fradet and Mario Südholt. Towards a generic framework for aspect-oriented programming. In *Proc. Third Aspect Oriented Programming Workshop*, volume 1543 of *Lecture Notes in Computer Science*, pages 394–397. Springer-Verlag, 1998.

[24] H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programing*, 3(3):223–278, 1983.

[25] H. Ganzinger and R. Giegerich. Attribute coupled grammars. *SIGPLAN Notices*, 19:157–170, 1984.

[26] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45. ACM Press, 2004.

[27] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35:121–131, 1992.

[28] G. Hedin. An object-oriented notation for attribute grammars. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'89*. Cambridge University Press, 1989.

[29] G. Hedin. Reference attribute grammars. *Informatica*, 24(3):301–317, 2000.

[30] 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.

[31] A. Johnstone and E. Scott. Generalised regular parsers. In *Proc. International Conf. on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 232–246. Springer-Verlag, 2003.

[32] A. Johnstone, E. Scott, and G. Economopoulos. Generalised parsing: Some costs. In *Proc. International Conf. on Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 2004.

[33] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[34] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the MetaObject Protocol*. MIT Press, 1991.

[35] 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.

[36] 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.

[37] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in 5(2):95–96, 1971.

[38] S. Krishnamurthi, M. Felleisen, and B. F. Duba. From macros to reusable generatvie programming. In K. Czarnacki and U. Eisenecker, editors, *First International Symposium on Generative and Component-Based Software Engineering*, volume 1799 of *Lecture Notes in Computer Science*, pages 105–120, 1999.

[39] M. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN '87*, 1987.

[40] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *Proc. Sixth European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996.

[41] W Maddox. Semantically-sensitive macroprocessing. Master's thesis, The University of California at Berkeley, Computer Science Division (EECS), Berkeley, CA 94720, December 1989.

[42] Brian McNamara and Yannis Smaragdakis. Functional programming in C++. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 118–129. ACM Press, 2000.

[43] S. McPeak and G. C. Necula. Elkhound: a fast, practical glr parser generator. In *Proc. International Conf. on Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer-Verlag, 2004.

[44] M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer. Multiple attribute grammar inheritance. *Informatica*, 24(3):319–328, 2000.

[45] Marjan Mernik, Nikolaj Korbar, and Viljem &#381;umer. Lisa: a tool for automatic language implementation. *SIGPLAN Not.*, 30(4):71–79, 1995.

[46] Andrew Meyers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[47] Martin Odersky and Philip Wadler. Pizza into java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159. ACM Press, 1997.

[48] 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.

[49] Arch D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10. ACM Press, 2001.

[50] Joao Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[51] C. Simonyi. The future is intentional. *IEEE Computer*, 32(5):56–57, May 1999.

[52] G. L. Jr Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999. Invited talk at OOPLSA'98.

[53] D. Swierstra and H. Vogt. Higher-order attribute grammars. In *International Summer School on Attribute Grammars Applications and Systens: SAGA*, volume 545 of *Lecture Notes in Computer Science*, pages 256–296. Springer-Verlag, 1991.

[54] M. Tatsubori. An extension mechanism for the java language. Master's thesis, University of Tsukuba, Graduate School of Engineering, Ibaraki, Japan, 1999.

[55] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. Openjava: A class-based macro system for java. In W. Cazzola, R. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 117–133. Springer-Verlag, 2000.

[56] T. Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *ACM Sigplan '90 Conference on Programming Languages Design and Implementation*, pages 197–208, 1990.

[57] T. Thai and H. Q. Lam. *.NET Framework Essentials*. O'Reilly, 2nd edition, 2002.

[58] M. Tomita. *Efficient Parsing for Natural Language.* Int. Series in Engineering and Computer Science. Kluwer Academic Publishers, 1986.

[59] E. Tzilla, R. E. filman, and A. Bader, editors. *Communications of the ACM*, volume 44, October 2001. Special issue on Aspect-oriented programming.

[60] M. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized lr parsers. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, 2002.

[61] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping, An Algebraic Specification Approach.* World Scientific, 1996.

[62] 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*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.

[63] 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.

[64] H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *Conference on Programming Languages Design and Implementation*, pages 131–145, 1990. Published as *ACM SIGPLAN Notices*, 24(7).

[65] 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.

[66] S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC API Tutorial and Reference: Universal Data Access for the Java 2 Platform.* Addison-Wesley Professional, 2nd edition, 1999.