Electronic Notes in Theoretical Computer Science 82 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 20 pages

Aspects as Modular Language Extensions

Eric Van Wyk [1]

Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota, U.S.A.

Abstract

Extensible programming languages and their compilers use highly modular specifications of languages and language extensions that allow a variety of different language feature sets to be easily imported into the programming environment by the programmer. Our model of extensible languages is based on higher-order attribute grammars and an extension called "forwarding" that mimics a simple rewriting process. It is designed so that no additional attribute definitions need to be written when combining a language with language extensions. Thus, programmers can remain unaware of the underlying attribute grammars when building customized languages by importing various extensions. In this paper we show how aspects and the aspect weaving process from Aspect-Oriented Programming can be specified as a *modular language extension* and imported into a base language specified in an extensible programming language framework.

1 Introduction

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

Currently, there is an active research community [1] with this end in mind investigating a set of language features that enable 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 feature for specifying these *cross cutting concerns*; programming in this style is referred to as Aspect-Oriented Programming [19,11,3].

[1] Email: evw@cs.umn.edu

©2003 Published by Elsevier Science B. V.

For example, consider the following AspectJ [18] *advice* declaration that specifies that before any call to the `setX` method of any `Point` object a message containing the new `x` value and current `x` and `y` values will be displayed:

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

This is a standard technique a programmer may use to trace the changing value of the x field in `Point` objects when the source code for the `Point` class cannot be modified. To achieve this same result without aspects the programmer would need to add the print statement before all calls to `setX` on `Point` objects throughout the program whereas with aspects this notion can be captured in a single location. AspectJ is a particular set of aspect language features that have been added to the Java language. A high quality compiler (see `www.aspectj.org`) has been written for Java and the AspectJ extensions that solves the immediate problem of providing aspects in Java.

But this leads us to the question of what happens when another set of interesting language features is proposed. Must a new compiler be written that adds these features to an existing language? Would it not be better if we could import new language features, such as aspects, into an existing programming language environment? This would enable us to combine aspects with other interesting extensions such as generics [8], those found in Explicit Programming [10] or programmer-defined extensions that are specific to the programmer's domain. If a language is specified in a highly modular and extensible fashion, it is possible to do just that.

In this paper we show how some AspectJ aspects can be added as a modular language extension to a simple object-oriented language. The key point in this exercise is that by specifying the base language and extensions appropriately as higher order attribute grammars [24] that use forwarding [28], the language composition process is simply the union of the productions and attribute definitions in the base language and extension. No "glue" attribute definitions or productions need to be written. This makes it possible for programmers to combine languages and extensions without being aware of the underlying attribute grammars. The attribute grammar in our framework makes use of several extensions to the attribute grammars originally design by Knuth [20] including higher order attributes [24], reference attributes [15] and forwarding [28]. Higher order attributes are used to pass around code trees and are used in the weaving process to move advice code trees to the join points where they are woven into the original program. Reference attributes are useful in that variable references can maintain a link to their declaration nodes. Forwarding enables a simple type of rewriting that is used to rewrite join points into new constructs that contain the advice code from various advice declarations and thus avoid writing "glue" attribute definitions.

Section 2 describes the AspectJ aspect constructs that we will implement here as language extensions. Section 3 defines the attribute grammar frame-

work and base 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 language and Section 5 concludes with a discussion of what was achieved and related and future work.

2 Aspect oriented programming with AspectJ

In this section we describe some of the AspectJ [18] aspect constructs that we implement here as language extensions. We will only cover a few of the many aspect language constructs in AspectJ in order to provide a basic understanding of how aspects can be added to our base object-oriented language. The remaining AspectJ constructs pose no fundamental difficulties to our model.

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

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

Advice is used to affect join point behavior. It consists of a (possibly empty) list of variable declarations, a point cut designator and *advice code* to be executed either before, after or around any join points in the point cut specified by the point cut designator. *Weaving* is the process of placing the advice code before, after or around the affected join points. This process can be done either statically or dynamically. For example, consider a method call `q.setX ( 4 )` and the advice declaration in Section 1. 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 we can not statically determine if `q` is an instance of class `Point`. In this case, we will weave the advice code inside an *if-then* statement to make this check at run-time and generate the code

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

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

The process of matching a pcd *pcd* against a method call construct at compile time tells us if (*i*) none of the method call's dynamic join points match *pcd*, if (*ii*) all of the method call's dynamic join points match *pcd* or if (*iii*) neither (*i*) nor (*ii*) can be determined at compile-time and thus a run-time test is required. In this last case, the matching test also returns the boolean expression code to be used in the run-time test. In case (*i*), the match test returns a *NoMatch* value indicating that no weaving should be done for this piece of advice. For (*ii*), the match test returns a value *Static* σ where σ is a substitution that maps variables declared in the advice to constructs in the matched method call. In the example above, $\sigma = [p \mapsto q, a \mapsto 4]$. This substitution is applied to the advice code to generate the actual advice code that is woven into place at the method call. In case (*iii*) where we can not statically determine if the pcd of a particular piece of advice matches all the dynamic join points for a method call construct, the test returns a value *Dynamic* σ *test* where σ is the same as above and *test* is the boolean expression for the dynamic test. We apply the substitution to the advice code and this test code and use it in the weaving process to generate the actual code to be woven for this method call. This application of σ to the advice code and test code can be seen in the examples above.

Intuitively, the weaving of advice code and the object program is achieved by rewriting method calls to code fragments containing the method call and its advice and dynamic test code. For a particular *beforeAdvice* declaration with pcd *pcd* and advice code *code* the rewrite rules are as follows:

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

and

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

A goal of this paper is to show how simple rewrite rules like these can be used in an attribute grammar system extended with forwarding that does not destructively change the structure of the abstract syntax tree when applying such rewrite rules.

3 Language extension in attribute grammars

As stated above, we want to specify programming languages and language extensions in such a way that programmers can import extensions into their base programming language as easily as a class or library is imported. To achieve this goal, languages and extensions are specified as (fragments of) attribute grammars in such a way that the union of the productions and attribute definitions in the language and extensions form the complete specification of the abstract syntax and semantics of the extended language – the programmer does not need to write any attribute grammar fragments to "glue" the extension to the base language. This ability relies *critically* on forwarding [28]. In this section we discuss the two types of language extension that are required, how these kinds of extensions are implemented in attribute grammars through the use of the forwarding extension, and sketch the attribute grammar specification of the base language that we will extend with AOP constructs.

3.1 Two types of language extension

We identify two types of language extensions that are needed to add aspect declarations and aspect weaving to an object-oriented base language. Although we are adding aspects here, these types of extension are general-purpose and are all that is required for many other language extensions. Recall that our goal is not just to add AOP features to a base language but to create a framework in which many such extensions can be modularly added.

3.1.1 Extension by adding new constructs

The first type of extension is, as expected, *the addition of new language constructs.* This is done by adding new productions and their associated attribute definitions to the base language, thus extending the abstract syntax and providing a semantics for the new language constructs. It is often convenient to define new language constructs in terms of existing language constructs in much the same way that macros define new constructs by expanding into existing language constructs that in essence provide the semantics of the macro. In our case, however, the new language constructs may perform a significant amount of semantic analysis through the attributes that they explicitly define and will only rely on an expansion or rewriting to existing language constructs to provide definitions to the attributes that they do not explicitly define.

This can be clarified by an example. Consider the simple and rather contrived example of adding an *if-then* statement to a language that has an *if-then-else* as the only (non-looping) branching statement. We may want the *if-then* to define its own pretty-print attribute for displaying the programmer-written source code and to generate an error if the condition of the *if-then* is not a boolean-typed expression. However, we would like to rewrite an *if* ⟨*cond*⟩ *then* ⟨*body*⟩ to *if* ⟨*cond*⟩ *then* ⟨*body*⟩ *else skip* when we need attribute values that are not explicitly defined by *if-then*, for example, a *jbc* attribute

defining the construct's translation to Java byte code. We will see below that *forwarding* simulates this capability.

3.1.2 Extension by modifying existing constructs

The second type of language extension is to provide a form of 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 constructs for new attributes introduced by the language extension. For example, adding 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 the construct to one that implements the desired modification. These rewrites may wrap additional statements or expressions around the construct. This type of extension is used to implement simple rewrites like those shown in Section 2 that define the aspect weaving processes. These can also be implemented by forwarding. The type of rewriting that is used here is very simple. We do a minimal amount of pattern matching in determining where to apply a rewrite by only checking that the same production was used to construct the pattern and the potential tree to be rewritten. The main determinant in deciding to perform a rewrite is a side condition that tests attribute values of the candidate attributed trees.

3.2 Forwarding in attribute grammars

An advantage of defining languages in attribute grammars is that the evaluation order of attributes is not explicitly specified by the language implementer but is automatically determined from the dependencies between the attributes [20]. We would like to have a similar property for rewrites in that we don't have to explicitly specify when they are to be performed; as we will see, forwarding provides this property.

Forwarding [28] is a technique for providing default attribute values for nodes that complements other default schemes such as the automatic copying of inherited attribute values to a node's children. A production that contains a *forwards-to* clause constructs a semantic tree from productions, its child semantic 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. When the forwarding construct is queried for an attribute value that it does not explicitly define, the value returned for the query is the value of that attribute on the forwards-to construct. The specification of the *if-then* example from above is shown below and will help to clarify.

if-then : $Stmt_0$ ::= *Expr* $Stmt_1$
    $Stmt_0.pp$ = "*if*" + $Expr.pp$ + ...
    $Stmt_0.errors$ = **if not** $Expr.type.isType(boolean)$ **then**...
    forwardsTo $if - then - else\ Expr\ Stmt_1\ skip$

Assuming the language has an *if-then-else* instruction and a skip instruction, we can model the behavior of the *if-then* by having it forward to an *if-then-else* whose else-clause is a skip instruction. When the *if-then* is queried for its pretty-print attribute *pp* it returns the value it defines, but when queried for its Java byte-code attribute, *jbc*, it forwards this query to the *if-then-else* that returns its semantically-equivalent *jbc* attribute value. 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. Pictorially, the abstract syntax tree for this construct can be seen in Figure 1.

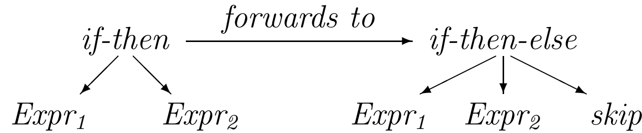

Fig. 1. Abstract syntax tree with forwarding

Our first attempt at implementing the aspect weaving rewrite rules may resemble the *if-then* production above. We would add a new "method call weaver" production that looks like the following:

$methodCallWeaver : Expr_0 ::= Expr_1 \; Id \; Expr_2$
$\quad Expr_0.pp = Expr_1.pp + \text{"."} + Id.pp + \text{"("} + Expr_2.pp + \text{")"}$
$\quad$ **forwardsTo if** ⟨ there is an applicable rewrite rule ⟩ **then**
$\qquad$ ⟨ advise code ⟩ ;
$\qquad methodCallWeaver \; Expr_1 \; Id \; Expr_2$
$\quad$ **else**
$\qquad methodCall \; Expr_1 \; Id \; Expr_2$

This production defines a *pp* attribute much like the *if-then* production. It also must determine if there is an applicable rewrite rule that can be applied to this method call. 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 the match is not successful, this production simply forwards to the standard non-weaving method call. If the match is successful this method call forwards to the sequential composition of the advice code (from 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 *environment* attribute in much the same way that variable references look up their declarations in an environment attribute. The rewrite rules are created at compile time from the aspect declarations in the object program and a standard inherited environment attribute provides a convenient mechanism to move in-scope rewrite rules to the method calls where they may be used. As we will see, aspect declarations add rewrite rules to the environment and static join point retrieve them from the environment.

The above approach has a flaw however. It requires that the *methodCall-Weaver* production be used (by the parser) to construct the original abstract syntax tree (AST). This prevents us from adding other language extensions that also rewrite method calls in this way since each assumes that it be put into the original AST.

To allow simple rewrites like those required for aspect weaving, extensible languages are defined in our framework such that each production has an associated "wrapper" or "place-holder" production where both 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 to apply, it forwards to the tree built by the corresponding non-wrapper production that does define the attributes specifying the semantics of the construct. It is a generalization of what we have above.

The method-call wrapper-production *methodCall_W*, that is used by the parser to build the initial AST is shown below.

$$
\begin{array}{l}
\textit{methodCall_W}\colon \textit{Expr}_0 ::= \textit{Expr}_1\ \textit{Id}\ \textit{Expr}_2 \\
\qquad \textbf{forwardsTo}\ \textit{forward} \\
\quad \textbf{where} \\
\qquad \textit{matchTree} = \textit{methodCall}\ \textit{Expr}_1\ \textit{Id}\ \textit{Expr}_2 \\
\qquad \textit{forward} = \textbf{case}\ \textit{getRWT}\ \textit{Expr}_0.\textit{env}\ \textit{matchTree}\ \textbf{of} \\
\qquad\qquad\qquad \textit{Nothing} \rightarrow \textit{matchTree} \\
\qquad\qquad\qquad \textit{Just}(\textit{env}', \textit{rwt_func}) \rightarrow (\textit{rwt_func}\ \textit{Expr}_0.\textit{env}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{`}\textit{w_inh}\text{`}\ (\textit{env} = \textit{env}')
\end{array}
$$

It calls the function *getRWT* with the current environment and the tree (built with the non-weaving production) to find an applicable rewrite. If there are no matching rewrite rules, the *getRWT* function returns a *Nothing* value and thus the *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 the ordered pair of a new environment that does not contain the matching rewrite rule and a function generating the matching rewrite rule's instantiated right hand side. We create the construct (to forward to) by providing this function with the current environment ($\textit{Expr}_0.\textit{env}$). The production then forwards to this construct that has its *env* attribute defined here by the 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. As we'll see below, we pass the rewrite function *rwt_func* the current environment, containing the matched rewrite, so that this matched rewrite can be applied to components of the matched tree, but not re-applied to the matched tree itself.

Since the wrapper-production defines very few attributes, the requests for attributes, such as *jbc*, are forwarded to the constructed right hand side tree effectively simulating the replacement normally done in term rewriting systems.

A similar wrapper-production is defined for each attribute-defining production though the only other one we will see in this paper is the one for variable references. These wrapper-productions can be automatically generated from the attribute-defining ones.

But 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.

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 how these rewrite rules and their associated match-functions are computed and added to the environment.

3.3 Attribute grammar specification of the base language

In the remainder of this section we present the attribute grammar based framework used for specifying modular definitions of languages. It this framework, Knuth's attribute grammars are extended with higher order attributes [24], reference attributes [15] and forwarding [28]. We also specify some of the productions and attribute definitions that define the base language, but most are what one would expect and thus we discuss only the most important definitions.

The production signatures of a significant part of the base language are shown in Table 1. 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* and statements are simply side-effecting expressions. The *Dcl* non-terminal represents variable and type declarations, and *Type* represents type expressions, including type identifiers.

3.3.1 Abstract syntax and semantic trees

Attribute values will range over an unspecified set of primitive values, such as integers and strings, and a set of higher order values, such as tree nodes and abstract semantic trees. A *node* is just 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 the node's type. For example, $Expr^n$ denotes the type of nodes that contain the inherited and synthesized attributes for expressions and $Expr^s$ ($Expr^i$) denote records that contain just the synthesized (inherited) attributes for an *Expr* non-terminal. We will use the dot (.) notation for referencing attribute values on such nodes: $n.a$ is the value of attribute a on

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*
dclSeq :	*Dcl*	::=	*Dcl Dcl*
classType :	*Type*	::=	*Type Dcl*
intType :	*Type*	::=	ϵ

Table 1
A selection of base language production signatures

node n.

(*Abstract*) *semantic trees* are used in Johnsson [16] in treating attribute grammars as a style of programming in lazy functional languages. These trees are functions that map a set of inherited attributes to a set of synthesized attributes according to the syntax productions and attribute definition rules. In higher order attribute grammars [24], semantic trees are valid attribute values. Here, we will change this definition slightly so that the output of the semantic tree function is a node containing both the input inherited attributes and the computed synthesized attributes. The types of these trees are denoted by superscripting non-terminal symbols with an f in order to distinguish them from nodes of the same non-terminal.[2] For example, semantic trees for the *Expr* non-terminal, have the type $Expr^f$ that is just shorthand for $Expr^i \rightarrow Expr^n$.

We will also use this type notation to refer to values of these types. As is the norm, we will use numeric subscripts to distinguish between like-named non-terminals. Since the non-terminals correspond to both nodes and semantic trees, we use the distinguishing superscripts n and f in the attribute definitions.

3.3.2 Attributes of interest

A significant attribute is the environment attribute *env*. This inherited attribute is used to make variable and type declarations available to variable and type references such that the scope rules of the base language are enforced. It is also used to make AOP advice declarations available to the static join points that they may affect. The type of *env* is named *Env* and is a list of tagged elements; the tag determines the purpose of the entry and the types of values stored in the element. In the case of variable declarations, the tag is *VarDcl*

[2] The use of superscripts was omitted from the previous productions but they can be inferred.

and the element component is an ordered pair with type ($String, Dcl^n$) containing the name of the variable and (a reference to) the variable's declaration node. As expected, scope rules are enforced by adding nested declarations to the front of the list. This attribute is automatically copied from a node to its child nodes if no other definition is provided. The synthesized attribute *defs* is defined on *Dcl*s, has the same type as *env* and is used to gather *env* declaration entries from *Dcl*s. Some productions and attribute definitions for these attributes are shown below.

$dclSeq : Dcl_0 ::= Dcl_1 \; Dcl_2$
  $Dcl_0^n.defs = Dcl_1^n.defs + Dcl_2^n.defs$
  $Dcl_2^n.env = Dcl_1^n.defs + Dcl_0^n.env$
$block : Expr_0 ::= Dcl \; Expr_1$
  $Expr_1^n.env = Dcl^n.defs + Expr_0^n.env$
$varDcl : Dcl ::= Id \; Type$
  $Dcl^n.defs = [\; VarDcl \; (Id^n.lexeme, Dcl^n) \;]$
  $Dcl^n.type = Type^n$

In order to avoid inappropriate name capture of variable references when moving semantic trees around for the rewriting process, we have three productions for variable references: *bindingVarRef*, *varRef_W* and *varRef* as seen in Table 2. The *bindingVarRef* production looks up variable declarations in the environment using the *dcl_lookup* function that returns the variable's declaration node. It then forwards to the variable reference wrapper-production that builds its tree from the declaration *node* and the identifier semantic tree. It does not 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. The *varRef_W* wrapper-production is similar to the *methodCall_W* wrapper-production we saw above. We have shown its definition of its *this_f* attribute. 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 gets it from its forward-to construct. The value of this attribute is used as a semantic tree that we may want 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.[3] 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.

Types in our base language are supported by a *type* attribute whose type is $Type^n$, a reference attribute, that references the variables type by fol-

[3] Note that it is still possible to incorrectly move a variable outside of its scope.

$bindingVarRef : Expr ::= Id$
  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**
     $Nothing \rightarrow matchTree$
     $Just(rwt, env') \rightarrow rwt$ ‘w_inh‘ $(env = env')$

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

Table 2
Variable reference productions.

lowing the similarly named attribute on the variable's declaration. We can query an expression *expr* for properties of its type, like its size in bytes, by $expr.type.size_in_bytes$. 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 type is a subclass of the class being defined. It's definition is elided but straightforward. Examples of these productions can be seen below:

$intType : Type ::= \epsilon$
  $Type^n.size_in_bytes = 4$

$classType : Type_0 ::= Type_1\ Dcl$
  $Type^n_0.isSubTypeOf = \lambda type_n \rightarrow ...$

3.3.3 Attribute evaluation

With the use of forwarding, we have the potential of creating very many trees and unnecessarily evaluating many attributes. Consider the *if-then* forwarding example. Evaluating all the attributes on the child nodes of the *if-then* would be wasted effort since most queries for attributes of the forwarding *if-then Expr* node will be forwarded to the forwards-to node that has its own copies of the children whose attribute values will presumably be evaluated. 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 [16] and uses the lazy functional language Haskell [22] as our implementation language. This evaluation scheme provides us with a reasonably efficient implementation since only the (portions of) trees that are needed are generated and evaluated.

4 Defining aspect constructs as language extensions

In this section we provide the specification of the *beforeAdvice* declaration and show it how creates the rewrite rules that implement aspect weaving. Below are 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.

beforeAdvice :	*Dcl*	::=	*Dcl PCD Expr*
callPCD :	*PCD*	::=	*objPCD mthPCD prmPCD*

classPCD :	*objPCD*	::=	*Id*	*objectPCD* :	*objPCD*	::=	*Id*
methodPCD :	*mthPCD*	::=	*Id*	*varPCD* :	*prmPCD*	::=	*Id*
wildCardPCD :	*prmPCD*	::=	ϵ				

We will discuss the definition of the advice and point cut designator constructs and then see how they are used to generate a rewrite rule that is put into the environment *env*. We've seen above how the weaving process is carried out by the application of these rewrite rules in the production *methodCall_W*.

4.1 Advice declarations

The *beforeAdvice* declaration production, shown in Table 3, defines the rewrite rule and its associated matching functions and adds it to the environment. The advice production generates a declaration from a (compound) declaration Dcl_1, a point cut designator *PCD* and the advice code *Expr*. Since the declaration *Dcl* declares the (pattern) variables that are used in the pcd and the advice code, its declarations ($Dcl_1^n.defs$) are added to the environment of the pcd 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* in *beforeAdvice* is added to the environment in an element tagged by *RWT* to distinguish it (and other rewrites) from other kinds of declarations in the environment. 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* in the program) and tests if it matches the point cut designator by calling the pcd's *match_sjp* function (defined below) on *sjp*. If *match_sjp* doesn't match and returns a *NoMatch* value, then the rewrite rule returns a *Nothing* value. Otherwise there is a *Static* or *Dynamic* match and it returns *Just* the function that generates the semantic tree that is to be forwarded to at the 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 Table 3) that has not had this rewrite rule removed. This is used in our example here to ensure that this rewrite can be applied to the static join point's object expression *sjp.object_n* and argument expression $sjp.param_n$ if needed. The method-call

$$
\begin{array}{l}
\mathit{beforeAdvice} : \mathit{Dcl_0} ::= \mathit{Dcl_1}\ \mathit{PCD}\ \mathit{Expr} \\
\quad \mathit{PCD}^n.\mathit{env} = \mathit{Dcl}_1^n.\mathit{defs} + \mathit{Dcl}_0^n.\mathit{env} \\
\quad \mathit{Expr}^n.\mathit{env} = \mathit{Dcl}_1^n.\mathit{defs} + \mathit{Dcl}_0^n.\mathit{env} \\
\quad \mathit{Dcl}_0^n.\mathit{defs} = [\ \mathit{RWT}\ \mathit{rwt_rule}] \\
\quad \textbf{forwardsTo}\ \mathit{dclSkip} \\
\textbf{where}\ \mathit{rwt_rule} = \lambda\ \mathit{sjp} \rightarrow \textbf{case}\ (\mathit{PCD}^n.\mathit{match_sjp})\ \mathit{sjp}\ \textbf{of} \\
\qquad \mathit{NoMatch} \rightarrow \mathit{Nothing} \\
\qquad \mathit{Static}\ s \rightarrow \mathit{Just}\ (e \rightarrow \mathit{exprSeq} \\
\qquad\quad (\mathit{Expr}^n.\mathit{this_f}\ \text{`}\mathit{w_inh}\text{`} \\
\qquad\qquad (\mathit{env} = s + \mathit{Expr}^n.\mathit{env})) \\
\qquad\quad (\mathit{meth_call}\ \mathit{sjp}\ e)) \\
\qquad \mathit{Dynamic}\ s\ \mathit{test_f} \rightarrow \mathit{Just}\ (e \rightarrow \mathit{exprSeq} \\
\qquad\quad ((\mathit{ifthen}\ \mathit{test_f}\ \mathit{Expr}^f)\ \text{`}\mathit{w_inh}\text{`} \\
\qquad\qquad (\mathit{env} = s + \mathit{Expr}^n.\mathit{env})) \\
\qquad\quad (\mathit{meth_call}\ \mathit{sjp}\ e)) \\
\quad \mathit{meth_call}\ =\ \lambda\ \mathit{sjp}\ e \rightarrow \mathit{methodCall_W} \\
\qquad (\mathit{sjp}.\mathit{object_n}.\mathit{this_f}\ \text{`}\mathit{w_inh}\text{`}\ (\mathit{env} = e)) \\
\qquad (\mathit{sjp}.\mathit{meth_n}.\mathit{this_f}) \\
\qquad (\mathit{sjp}.\mathit{param_n}.\mathit{this_f}\ \text{`}\mathit{w_inh}\text{`}\ (\mathit{env} = e))
\end{array}
$$

Table 3
beforeAdvice production.

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 s and its original environment. The *match_sjp* test returns the rewrite rules that are used to rewrite pattern variables to expressions at the static join point. 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. In the following subsection, we describe 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 nodes have a *match_sjp* function-valued attribute that tests if the pcd matches a static join point. This function takes an $Expr^n$ and returns a value of the algebraic type $Match$ defined as follows:

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

The behavior of this function was sketched in Section 2 and its implementation in this language framework is shown in Table 4. The *methodCallPCD* production is used to match *call* point cut designators against method calls by calling the *match_sjp* function on its child PCD nodes and combining their results with $\wedge_{pcd}$. This operator has type $Match \times Match \rightarrow Match$ and is shown in Table 5. It combines matches in the expected way, combining the substitution environments and dynamic test code of the parameter matches.

$$
\begin{array}{l}
methodCallPCD : PCD ::= objPCD\ mthPCD\ prmPCD \\
\quad PCD^n.match_sjp = \lambda\ sjp \rightarrow\ objPCD^n.match_sjp\ (sjp.objRef) \wedge_{pcd} \\
\qquad\qquad\qquad\qquad\qquad mthPCD^n.match_sjp\ (sjp.methRef) \wedge_{pcd} \\
\qquad\qquad\qquad\qquad\qquad prmPCD^n.match_sjp\ (sjp.paramRef) \\
objectPCD : objPCD ::= Id \\
\quad objPCD^n.match_sjp \\
\qquad = \lambda\ sjp \rightarrow \textbf{if}\ sjp.type.isSubTypeOf\ \ objPDC^n.type \\
\qquad\qquad \textbf{then}\ Static\ [RWT\ (varRefRWT\ Id^n\ sjp)] \\
\qquad\qquad \textbf{else if}\ objPDC^n.type.isSubTypeOf\ \ sjp.type \\
\qquad\qquad\quad \textbf{then}\ Dynamic\ [RWT\ (varRefRWT\ Id^n\ sjp)] \\
\qquad\qquad\qquad (methodCall\ sjp.this_f\ mkId(\text{``}instanceOf\text{''}) \\
\qquad\qquad\qquad\quad mkStrConst(objPDC^n.type.className)) \\
\qquad\qquad\quad \textbf{else}\ NoMatch \\
methodPCD : mthPCD ::= Id \\
\quad mthPCD^n.match_sjp = \lambda Id' \rightarrow \textbf{if}\ Id.lexeme = Id'.lexeme \\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ Static[\,]\ \textbf{else}\ NoMatch \\
varPCD : prmPCD ::= Id \\
\quad varPCD^n.match_sjp = \lambda sjp \rightarrow\ Static\ [RWT\ (varRefRWT\ Id^n\ sjp)] \\
wildCardPCD : prmPCD ::= \epsilon \\
\quad varPCD^n.match_sjp = \lambda sjp \rightarrow\ Static\ [\,]
\end{array}
$$

Table 4
Point cut designator productions.

The *objectPCD* production is used when an object variable is used in the PCD instead of a class name as in the examples in Section 2. The object at the method call is passed to *match_sjp* as the *sjp* parameter. If the *sjp*'s type is a sub-type of the class type of the object in the pcd, then we have a static match and we create a rewrite rule that maps Id to the matched object sjp and it 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 PCD is a sub-type of the matched object type *sjb.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 base 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 in the PCD. Since the variable 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, however, shown below. The *wildCardPCD* also always provides a static match, but generates no rewrites.

$$
\begin{array}{l}
\wedge_{pcd} : Match \times Match \rightarrow Match \\
m_1 \wedge_{pcd} m_2 = \textbf{case}\ m_1\ \textbf{of} \\
\quad NoMatch \rightarrow NoMatch \\
\quad Static\ s_1 \rightarrow \textbf{case}\ m_2\ \textbf{of}\ NoMatch \rightarrow NoMatch \\
\qquad\qquad\qquad\qquad\qquad Static\ s_2 \rightarrow Static(s_1 + s_2) \\
\qquad\qquad\qquad\qquad\qquad Dynamic\ s_2\ t \rightarrow Dynamic(s_1 + s_2)\ t \\
\quad Dynamic\ s_1\ t_1 \rightarrow \textbf{case}\ m_2\ \textbf{of} \\
\qquad\qquad\qquad NoMatch \rightarrow NoMatch \\
\qquad\qquad\qquad Static\ s_2 \rightarrow Dynamic(s_1 + s_2)\ t_1 \\
\qquad\qquad\qquad Dynamic\ s_2\ t_2 \rightarrow Dynamic(s_1 + s_2)\ (andExpr\ t_1\ t_2)
\end{array}
$$

Table 5
Point cut designator *and* operator.

4.2.1 Generating rewrite rules

Recall that the advice code and possibly 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. This substitution σ is implemented by a set of rewrite rules, similar to those for rewriting method calls, and are returned as an 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. This 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.)

$$
\begin{array}{l}
varRefRWT :: Id \rightarrow Expr \rightarrow Expr \rightarrow Maybe(Env \rightarrow Expr^f) \\
varRefRWT\ advice_var_n\ sjp\ n \\
\quad = \textbf{if}\ \ n.isVarRef \wedge n.varDcl = advice_var_n.varDcl \\
\qquad \textbf{then}\ Just(\ e \rightarrow sjp.this_f)\ \textbf{else}\ Nothing
\end{array}
$$

If this test succeeds, then we want to rewrite the advice variable to the semantic tree extracted from the matched static join point via the *this_f* attribute. [4]

In this section we have shown how advice declarations can be specified as modular language extensions and describe how they specify the rewrite rules, for method calls and advice variables, that implement aspect weaving.

5 Discussion, future work and related work

5.1 Discussion

In this paper we have shown that many AspectJ aspect constructs can be specified as a modular language extension that can be added in a 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.* Many other AspectJ constructs, such as control flow pcds, property based matching and different types of join points can also be added with a similar amount of work, but they are omitted here for brevity. More complex pcds such as *cflow* which are satisfied by checking if a certain pattern of method calls can be found on the run-time call stack can be specified using the techniques above. We need only create and maintain a run-time data structure that keeps track of the methods have been called. New method calls update this data structure and the *cflow* pcd 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.

5.2 Future work

We have said above that we would like the process of language extension to be as easy for the programmer as importing a class. This is a long term and, admittedly, rather ambitious 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. 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 [28] how operator overloading can be handled with forwarding, so many of the types of syntactic extensions one would make can be handled. When this fails the use of the disambiguation filters of generalized LR parsers in [26] may be useful.

Hand-coding the rewrite functions as we've done using forwarding is rather

[4] Note that $Expr^n$s that are variable references define *varDcl*, but others do not. Allowing them do so 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.

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 “complied” into the attribute grammar that uses forwarding that we saw above.

5.3 Related Work

There is much work in the attribute grammar literature on modular attribute grammars and their use in language specification ([17,14,12,2,13] are but a few examples). It is a goal of this paper to show that it is the combination of higher order attributes, reference attributes and forwarding that enables languages to be specified in a highly modular fashion. Removing any of these extensions from the attribute grammar definition above would have a significant negative impact on its modularity [28]. Forwarding was, in fact, used in a previous version of Microsoft’s Intentional Programming system [23].

There are other language extension systems and we describe those most closely related to our work here. The ASF+SDF [27] system is based on modular algebraic specifications and term rewriting, although primitive recursvie schemes, a subclass of algebraic specifications are comparable to stongly non-circular attribute grammars [27, page 48]. Both ASF+SDF and our system allow for modular specification of languages and language extensions but we begin with attribute grammars (as oppsed 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 attributes values that are not directly available in terms.

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 [4], JTK [7], `<bigwig>` [9] and Maya [5]. But by using forwarding, we can specify extensions that are difficult or impossible to express in these systems since the forwards-to construct can depend on any semantic value (attribute) of the langague constructs (though Maya does also base macro dispatch on the static type of macro parameters). In this sense we have much in common with meta-object protocol systems like OpenJava [25]. We also allow extension writers to define their own semantics by introducing new attributes. The only macro system, to our knowledge, that also allows this is the Scheme macro system McMacMic [21]. 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.

It is worth noting that Maya has been used to implement aspects as a language extension [6], but it implements dynamic aspect weaving instead of the more difficult static weaving presented here.

References

[1] Aspect Oriented Software Development web pages. See `http://www.aosd.net`.

[2] S. R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, Department of Electronics and Computer Science, UK, 1993.

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

[4] 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.

[5] 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)*. ACM, June 2002.

[6] 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.

[7] 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.

[8] 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.

[9] C. Bradrand and M. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proc. Partial Evaluation and Semantics-Based Program Manipulation, Workshop*. Association of Computing Machinery, 2002.

[10] 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.

[11] D. Crawford, editor. *Communications of the ACM*, volume 44, October 2001.

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

[13] 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.

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

[15] G. Hedin. Reference attribute grammars. In *2nd International Workshop on Attribute Grammars and their Applications*, 1999.

[16] T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173. Springer-Verlag, 1987.

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

[18] 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 *LNCS*, pages 327–353, 2001.

[19] 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 *LNCS*, pages 220–242, 1997.

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

[21] 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 *LNCS*, pages 105–120, 1999.

[22] 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.

[23] C. Simonyi. The future is intentional. *IEEE Computer*, May 1999.

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

[25] 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 *LNCS*, pages 117–133. Springer-Verlag, 2000.

[26] 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 *LNCS*, pages 143–158, 2002.

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

[28] 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 *LNCS*, pages 128–142. Springer-Verlag, 2002.