Rewritable Reference Attributed Grammars

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
(torbjorn|gorel)@cs.lth.se

Abstract. This paper presents an object-oriented technique for rewriting abstract syntax trees in order to simplify compilation. The technique, Rewritable Reference Attributed Grammars (ReRAGs), is completely declarative and supports both rewrites and computations by means of attributes. We have implemented ReRAGs in our aspect-oriented compiler compiler tool JastAdd II. Our largest application is a complete static-semantic analyzer for Java 1.4. ReRAGs uses three synergistic mechanisms for supporting separation of concerns: inheritance for model modularization, aspects for cross-cutting concerns, and rewrites that allow computations to be expressed on the most suitable model. This allows compilers to be written in a high-level declarative and modular fashion, supporting language extensibility as well as reuse of modules for different compiler-related tools. We present the ReRAG formalism, its evaluation algorithm, and examples of its use. Initial measurements using a subset of the Java class library as our benchmarks indicate that our generated compiler is only a few times slower than the standard compiler, javac, in J2SE 1.4.2 SDK. This shows that ReRAGs are already useful for large-scale practical applications, despite that optimization has not been our primary concern so far.

1 Introduction

Reference Attributed Grammars (RAGs) have proven useful in describing and implementing static-semantic checking of object-oriented languages [1]. These grammars make use of *reference attributes* to capture non-local tree dependences like variable decl-use, superclass-subclass, etc., in a natural, yet declarative, way.

The RAG formalism is itself object-oriented, viewing the grammar as a class hierarchy and the abstract syntax tree (AST) nodes as instances of these classes. Behavior common to a group of language constructs can be specified in superclasses, and can be further specialized or overridden for specific constructs in the corresponding subclasses.

In plain RAGs, the complete AST is built prior to attribute evaluation. While this works well for most language constructs, there are several cases where the most appropriate tree structure can be decided only *after* evaluation of some of the attributes. I.e., the context-free syntax is not sufficient for building the desired tree, but contextual information is needed as well. By providing means for rewriting the AST based on a partial attribution, the specification of the remaining attribution can be expressed in a simpler and more natural way.

This paper presents ReRAGs, Rewritable Reference Attributed Grammars, which extend RAGs with the capability to rewrite the AST dynamically, during attribute evaluation, yet specified in a declarative way. ReRAGs form a conditional rewrite system

where conditions and rewrite rules may use contextual information through the use of attributes. We have implemented a static-semantics analyzer for Java using this technique. Based on this experience we exemplify typical cases where rewriting the AST is useful in practice.

ReRAGs are closely related to Higher-ordered Attribute Grammars (HAGs) [2], [3] and to the technique of forwarding in HAGs [4]. A major difference lies in the object-oriented basis of ReRAGs, where reference attributes are kept as explicit links in the tree and subtrees are rewritten in place. HAGs, in contrast, have a functional programming basis, viewing the AST as well as its attributes as structured values without identity.

ReRAGs also have similarities to tree transformation systems like Stratego [5], ASF+SDF [6], and TXL[7], but improves data acquisition support through the use of RAGs instead of embedding contextual data in rewrite rules. Rewrite application strategies differ in that ReRAGs only support the above described declarative approach while the above mentioned systems support user defined strategies. In Stratego and AST+SDF the rewrite application strategy is specified through explicit traversal strategies and in TXL the rewrite application order is implicitly defined as part of the functional decomposition of the transformation ruleset.

The plain RAG evaluation scheme is demand driven, evaluating an attribute only when its value is read. The ReRAG evaluation scheme extends this basic approach by rewriting parts of the AST as needed during the evaluation. We have designed different caching strategies to achieve performance optimization and evaluated the approach using a subset of the J2SDK 1.4.2 class library as our benchmark suite.

ReRAGs are implemented in our tool JastAdd II, a successor to our previous tool JastAdd that supported plain RAGs [8]. Several grammars have been developed for JastAdd II, the largest one being our Java grammar that implements static-semantics checking as specified in the Java Language Specification [9].

In addition to RAG/ReRAG support, the JastAdd systems support static aspect-oriented specification and integration with imperative Java code. Specifications are aspect-oriented in that sets of attributes and equations concerning a particular aspect, such as name analysis, type checking, code generation, etc., can be specified in modules separate from the AST classes. This is similar to the static introduction feature of AspectJ [10] where fields, methods, and interface implementation clauses may be specified in modules separate from the original classes.

Integration with imperative Java code is achieved by simply allowing ordinary Java code to read attribute values. This is useful for many problems that are more readily formulated imperatively than declaratively. For example, a code emission module may be written as ordinary Java code that reads attribute values from the name and type analysis in order to emit the appropriate code. These modules are also specified as static introduction-like aspects that add declarations to the existing AST classes. The ReRAG examples given in this paper are taken from our experience with the Java grammar and utilize the separation of concerns given by the aspect-oriented formulation, as well as the possibility to integrate declarative and imperative modules.

The rest of this paper is structured as follows. Section 2 introduces some typical examples of when AST rewriting is useful. Section 3 gives background information on RAGs and ASTs. Section 4 introduces ReRAG rewriting rules. Section 5 discusses how

ReRAGs are evaluated. Section 6 describes the algorithms implemented in JastAdd II. Section 7 discusses ReRAGs from both an application and a performance perspective. Section 8 compares with related work, and Section 9 concludes the paper.

2 Typical examples of AST rewriting

From our experience with writing a static-semantics analyzer for Java, we have found many cases where it is useful to rewrite parts of the AST in order to simplify the compiler implementation. Below, we exemplify three typical situations.

2.1 Semantic specialization

In many cases the same context-free syntax will be used for language constructs that carry different meaning depending on context. One example is names in Java, like `a.b`, `c.d`, `a.b.c`, etc. These names all have the same general syntactic form, but can be resolved to a range of different things, e.g., variables, types, or packages, depending on in what context they occur. During name resolution we might find out that `a` is a class and subsequently that `b` is a static field. From a context-free grammar we can only build generic `Name` nodes that must capture all cases. The attribution rules need to handle all these cases and therefore become complex. To avoid this complexity, we would like to do *semantic specialization*. I.e., we would like to replace the general `Name` nodes with more specialized nodes, like `ClassName` and `FieldName`, as shown in Figure 1. Other computations, like type checking, optimization, and code generation, can benefit from this rewrite by specifying different behavior (attributes, equations, fields and methods) in the different specialized classes, rather than having to deal with all the cases in the general `Name` class.

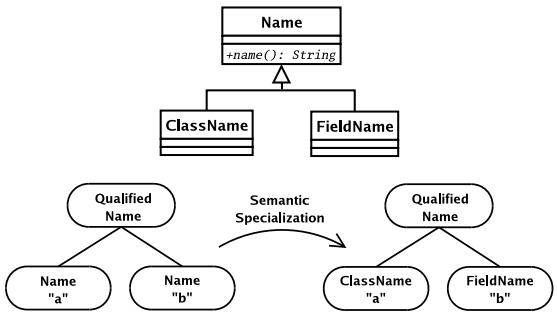


Fig. 1. Semantic specialization of name references.

2.2 Make implicit behavior explicit

A language construct sometimes has *implicit behavior* that does not need to be written out by the programmer explicitly. An example is the implicit constructors of Java classes. If a class in Java has no constructors, this corresponds to an implicit constructor taking no arguments. The implicit behavior can be made explicit by rewriting the AST, see Figure 2. This simplifies other computations, like code generation, which do not have to take the special implicit cases into account.

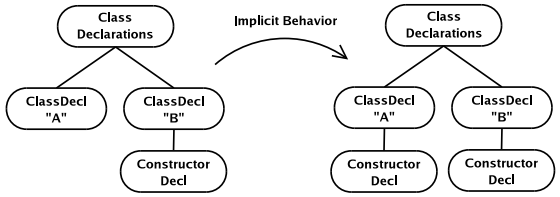


Fig. 2. The implicit constructor in class "A" is made explicit.

2.3 Eliminate shorthands

Some language constructs are shorthands for specific combinations of other, more basic, constructs. For example, string concatenation in Java can be written using the binary addition operator (e.g., `a+b`), but is actually implemented as an invocation of the `concat` method in the `String` class (e.g., `a.concat(b)`). The AST can be rewritten to eliminate such shorthands, as shown in Figure 3. The AST now reflects the semantics rather than the concrete syntax, which simplifies other computations like optimizations and code generation.

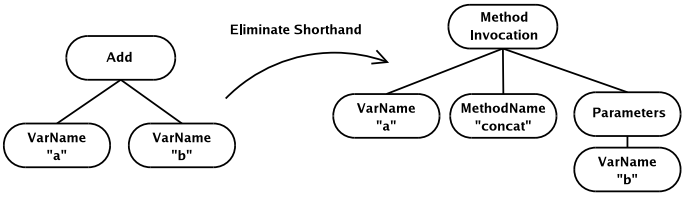


Fig. 3. Eliminate shorthand and reflect the semantic meaning instead.

3 Background

3.1 AGs and RAGs

ReRAGs are based on Reference Attributed Grammars (RAGs) which is an object-oriented extension to Attribute Grammars (AGs) [11]. In plain AGs each node in the AST has a number of *attributes*, each defined by an *equation*. The right-hand side of the equation is an expression over other attribute values and defines the value of the left-hand side attribute. In a consistently attributed tree, all equations hold, i.e., each attribute has the same value as the right-hand side expression of its defining equation.

Attributes can be *synthesized* or *inherited*. The equation for a synthesized attribute resides in the node itself, whereas for an inherited attribute, the equation resides in the parent node. From an OO perspective we may think of attributes as fields and of equations as methods for computing the fields. However, they need not necessarily be implemented that way. Note that the term *inherited attribute* refers to an attribute defined in the parent node, and is thus a concept unrelated to the inheritance of OO languages. In this article we will use the term *inherited attribute* in its AG meaning.

Inherited attributes are used for propagating information downwards in the tree (e.g., propagating information about declarations down to use sites) whereas synthesized attributes can be accessed from the parent and used for propagating information upwards in the tree (e.g. propagating type information up from an operand to its enclosing expression).

RAGs extend AGs by allowing attributes to have reference values, i.e., they may be object references to AST nodes. AGs, in contrast, only allow attributes to have primitive or structured algebraic values. This extension allows very simple and natural specifications, e.g., connecting a use of a variable directly to its declaration, or a class directly to its superclass. Plain AGs connect only through the AST hierarchy, which is very limiting.

3.2 The AST class hierarchy

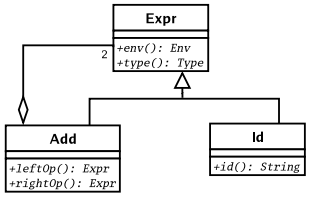
The nodes in an AST are viewed as instances of Java classes arranged in a subtype hierarchy. An AST class correponds to a nonterminal or a production (or a combination thereof) and may define a number of children and their declared types, corresponding to a production right-hand side. In an actual AST, each node must be *type consistent* with its parent according to the normal type-checking rules of Java. I.e., the node must be an instance of a class that is the same or a subtype of the corresponding type declared in the parent. Shorthands for lists, optionals, and lexical items are also provided. An example definition of some AST classes in a Java-like syntax is shown below.

```
// Expr corresponds to a nonterminal
ast Expr;
// Add corresponds to an Expr production
ast Add extends Expr ::= Expr leftOp, Expr rightOp;
// Id corresponds to an Expr production, id is a token
ast Id extends Expr ::= <String id>;
```

Aspects can be specified that define attributes, equations, and ordinary Java methods of the AST classes. An example is the following aspect for very simple type-checking.

```
// Declaration of an inherited attribute env of Expr nodes
inh Env Expr.env();
// Declaration of a synthesized attribute type of Expr nodes
// and its default equation
syn Type Expr.type() = TypeSystem.UNKNOWN;
// Overriding the default equation for Add nodes
eq Add.type() = TypeSystem.INT;
// Overriding the default equation for Id nodes
eq Id.type() = env().lookup(id()).type();
```

The corresponding Java API is shown in the following UML diagram. It includes methods for accessing child nodes like `leftOp` and `rightOp`, tokens like `id` and user-defined attributes like `env` and `type`. This API can be used freely in the right-hand sides of equations, as well as by ordinary Java code.



4 Rewrite rules

ReRAGs extend RAGs by allowing rewrite rules to be written that automatically and transparently rewrite nodes. The rewriting of a node is triggered by the first access to it. Such an access could occur either in an equation in the parent node, or in some imperative code traversing the AST. In either case, the access will be captured and a reference to the final rewritten tree will be the result of the access. This way, the rewriting process is transparent to any code accessing the AST. The first access to the node will always go via the reference to it in the parent node. Subsequent accesses may go via reference attributes that refer directly to the node, but at this point, the node will already be rewritten to its final form.

A rewrite step is specified by a rewrite rule that defines the conditions when the rewrite is applicable, as well as the resulting tree. For a given node, there may be several rewrite rules that apply, which are then applied in a certain order. It may also be the case that after the application of one rewrite rule, more rewrite rules become applicable. This allows complex rewrites to be broken down into a series of simple small rewrite steps.

A rewrite rule for nodes of class *N* has the following general form:

```
rewrite N  {
  when (cond)
  to R result;
}
```

This specifies that a node of type *N* may be replaced by another node of type *R* as specified in the result expression *result*. The rule is applicable if the (optional) boolean condition *cond* holds and will be applied if there are no other applicable rewrite rules of higher priority (priorites will be discussed later). Furthermore, all rewrite rules must be type consistent in that the replacement will result in a type consistent AST regardless of the context of the node, as will be discussed in Section 4.2. In a consistently attributed tree, all equations hold and all rewrite conditions are false.

4.1 A simple example

As an example, consider replacing an `Add` node with a `StringAdd` node in case both operands are strings [1]. This can be done as follows.

```
ast StringAdd extends Expr ::= Expr leftOp, Expr rightOp;
rewrite Add {
   when (childType().equals(TypeSystem.STRING))
   to StringAdd new StringAdd(leftOp(), rightOp());
}
syn Type Add.childType() = ...;
```

Note that in the creation of the new right-hand side, the previous children `leftOp()` and `rightOp()` are used. These accesses might trigger rewrites of these nodes in turn.

Avoiding repeated applications. `StringAdd` nodes might have much in common with `Add` nodes, and an alternative way of handling this rewrite would be to define `StringAdd` as a subclass of `Add`, rather than as a sibling class. In this case, the rewrite should apply to all `Add` nodes, except those that are already `StringAdd` nodes, and can be specified as follows.

```
ast StringAdd extends Add;
rewrite Add {
   when (childType().equals(TypeSystem.STRING)
      and !(this instanceOf StringAdd))
   to StringAdd new StringAdd(leftOp(), rightOp());
}
syn Type Add.childType() = ...;
```

Note that the condition includes a type test to make sure that the rule is not applied to nodes that are already of type StringAdd. This is necessary since the rule would otherwise still be applicable after the rewrite, resulting in repeated applications of the same rule and thereby nontermination. In general, whenever the rewrite results in the same type or a subtype, it is advicable to reflect over if the condition might hold also after the rewrite and in that case if the condition should be tightened in order to avoid nontermination.

[1] In Section 4.4 we will instead rewrite addition of strings as method calls.

Solutions that refactor the AST class hierarchy. A third alternative solution could be to keep `Add` and `StringAdd` as sibling classes and to factor out the common parts into a superclass as follows.

```
ast Expr:
ast GeneralAdd extends Expr ::= Expr leftOp, Expr rightOp;
ast Add extends GeneralAdd;
ast StringAdd extends GeneralAdd;
```

This solution avoids the type test in the rewrite condition. However, it requires that the grammar writer has access to the original AST definition of `Add` so that it can be refactored.

4.2 Type consistency

As mentioned above, rules must be *type consistent*, i.e., the replacing node must always be type consistent with any possible context. This is checked statically by the JastAdd II system. Consider the rewrite rule that replaces an `Add` node by a sibling `StringAdd` node using the grammar described above. The expected child type for all possible contexts for `Add` nodes is `Expr`. Since both `Add` and `StringAdd` are subclasses of `Expr` the rule is type consistent. However, consider the addition of the following AST class.

```
ast A ::= Add:
```

In this case the rewrite rule would not be type consistent since the rewrite could result in an `A` node having a `StringAdd` node as a child although an `Add` node is expected. Similarly, in the second rewrite example in Section 4.1 where `StringAdd` is a subclass of `Add`, that rewrite rule would not be type consistent if the following classes were part of the AST grammar.

```
ast B ::= C:
ast C extends Add;
```

In this case, the rewrite rule could result in a `B` node having a `StringAdd` node as a child which would not be type consistent.

Theorem 1. *A rule* `rewrite` *A...*`to` *B... is type consistent if the following conditions hold: Let C be the first common superclass of A and B. Furthermore, let $\mathcal{D}$ be the set of classes that occur on the right-hand side of any production class. The rule is type consistent as long as there is no class D in $\mathcal{D}$ that is a subclass of C, i.e., $D \not< C$.*

Proof. The rewritten node will always be in a context where its declared type D is either the same as C, or a supertype thereof, i.e. $C \leq D$. The resulting node will be of a type $R \leq B$, and since $B \leq C$, then consequently $R \leq D$, i.e., the resulting tree will be type consistent. □

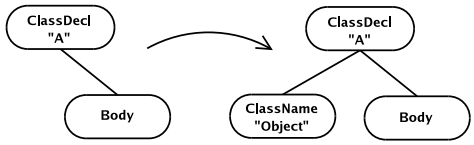
4.3 Rewriting descendent nodes

The tree resulting from a rewrite is specified as an expression which may freely access any of the current node's attributes and descendents. Imperative code is permitted, using the syntax of a Java method body that returns the resulting tree. This imperative code may reuse existing parts in the old subtree in order to build the new subtree, but may have no other externally visible side effects. This can be used to rewrite descendent nodes, returning an updated version of the node itself as the result.

As an example, consider a Java class declaration `class A { ... }`. Here, `A` is given no explicit superclass which is equivalent to giving it the superclass `Object`. In order to simplify further attribution (type checking, etc.), we would like to change the AST and insert the superclass as an explicit node. This can be done by the following rewrite rule:

```
ast ClassDecl extends Decl ::=
   <String classId>, [ TypeRef superClass ], Body body;
rewrite ClassDecl {
   when (!hasSuperClass() && !name().equals("Object"))
   to ClassDecl {
      setSuperClass(new TypeRef("Object"));
      return this;
   }
}
```

Note that the rewrite rule updates a descendent node and returns itself, as illustrated in the figure below.



As seen from the specification above, the condition for doing this rewrite is that the class has no explicit superclass already, and that it is another class than the root class `Object`. The result type is the same as the rewritten type, which means we should reflect on possible nontermination due to repeated applications of the same rule. However, it is clear that the rewrite will not be applicable a second time since the rewrite will result in a node where the condition is no longer met.

4.4 Combining rules

It is often useful to rewrite a subtree in several steps. Consider the following Java-like expression

```
a+"x"
```

Supposing that `a` is a reference to an non-null `Object` subclass instance, the semantic meaning of the expression is to convert `a` into a string, convert the string literal `"x"` into a string object, and to concatenate the two strings by the method `concat`. It can thus be seen as a shorthand for the following expression.

```
a.toString().concat(new String(new char[ ] {'x'} ))
```

To simplify code generation we would like to eliminate the shorthand notation by rewriting the AST. This can be accomplished by a number of rewrite rules, each taking care of a single subproblem:

1. replace the right operand of an Add node by a call to `toString` if the left operand is a string, but the right is not
2. replace the left operand of an Add node by a call to `toString` if the right operand is a string, but the left is not
3. replace an Add node by a method call to `concat` if both operands are strings
4. replace a string literal by an expression creating a new string object

Suppose the original `Add` node is accessed from its parent. This will cause the AST to be rewritten in the following steps. First, it will be checked which rules are applicable for `Add`. This will involve accessing its left and right operands, which triggers the rewrite of these nodes in turn. In this case, the right operand will be rewritten according to rule 4. It is now found that rule 2 is applicable for `Add`, and it is applied, replacing the left operand by a `MethodCall`. This causes rule 3 to become applicable for `Add`, replacing it too by a `MethodCall`. Now, no more rules are applicable for the node and a reference is returned to the parent. Figure 4 illustrates the steps applied in the rewrite.

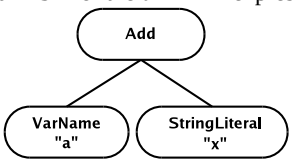
Rule priority. In general, it is possible that more than one rule applies to a node. Typically, this happens when there are two rewrite rules in a node, each rewriting different parts of the substructure of the node. For example, in a class declaration there may be one rewrite rule that takes care of making an implicit constructor explicit, and another rule making an implicit superclass explicit. Both these rules can be placed in the `ClassDecl` AST class, and may be applicable at the same time. In this particular case, the rules are *confluent*, i.e., they can be applied in any order, yielding the same resulting tree. So far, we have not found the practical use for nonconfluent rules, i.e., where the order of application matters. However, in principle they can occur, and in order to obtain a predictable result also in this case, the rules are prioritized: Rules in a subclass have priority over rules in superclasses. For rules in the same class, the lexical order is used as priority.

5 ReRAG evaluation

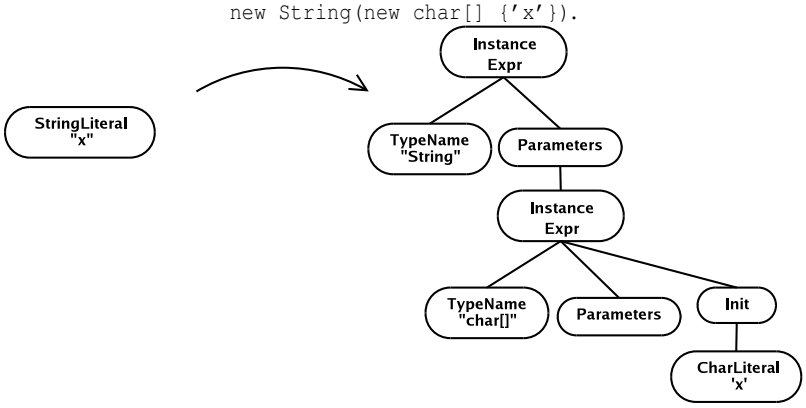
5.1 RAG evaluation

An attribute evaluator computes the attribute values so that the tree becomes consistently attributed, i.e., all the equations hold. JastAdd uses a demand-driven evaluation mechanism for RAGs, i.e., the value of an attribute is not computed until it is read [8]. The implementation of this mechanism is straight-forward in an object-oriented language [12]. Attributes are implemented as methods in the AST classes where they are declared. Accessing an attribute is done simply by calling the corresponding method. Also equations are translated to methods, and are called as appropriate by the attribute

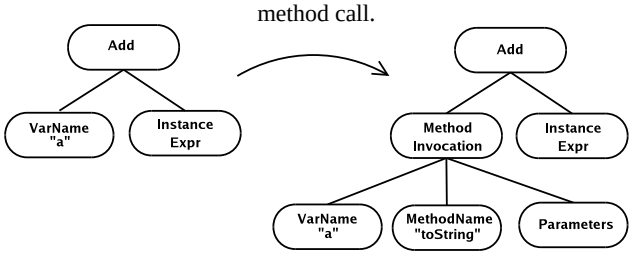
Initial AST for the `a + "x"` expression



Rule 4: Replace the `"x"` string literal by a new string instance expression `new String(new char[] {'x'})`.



Rule 2: Make the implicit Object to String type conversion explicit by adding a `"toString"` method call.



Rule 3: Replace add by a method call to `"concat"`.

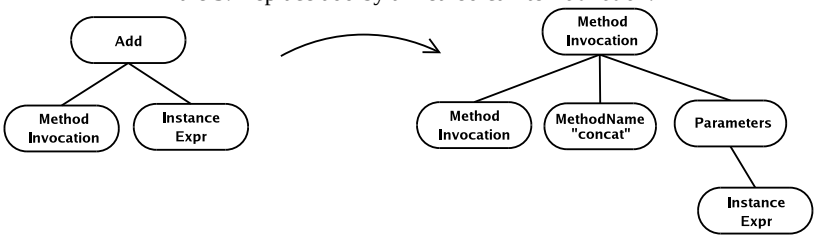


Fig. 4. Combine several rules to eliminate the shorthand for String addition and literals in a Java like language.

methods: The method implementing an inherited attribute will call an equation method in the parent node. The method implementing a synthesized attribute calls an equation method in the node itself. JastAdd checks statically that all attributes in the grammar have a defining equation, i.e., that the grammar is well-formed. For efficiency, the value of an attribute is cached in the tree the first time it is computed. All tree nodes inherit generic accessor methods to its parent and possible children through a common superclass. As a simple example, consider the following RAG fragment:

```
ast Expr;
ast Id extends Expr ::= <String id>;
inh Env Expr.env();
syn Type Expr.type();
eq Id.type() = env().lookup(id()).type();
```

This is translated to the following Java code:

```
class Expr extends ASTNode {    // inherit generic node access
  Env env_value = null;         // cached attribute value
  boolean env_cached = false;   // flag true when cached
  Env env() {                  // method for inherited attribute
    if(!env_cached) {
      env_value = ((HasExprSon)parent()).env_eq(this);
      env_cached = true; }
    return env_value; }
  Type type_value = null;       // cached attribute value
  boolean type_cached = false;  // flag true when cached
  Type type() {              // method for synthesized attribute
    if(!type_cached) {
      type_value = type_eq();
      type_cached = true; }
    return type_value; }
  abstract Type type_eq(); }
interface HasExprSon {
  Env env_eq(Expr son); }
class Id extends Expr {
  String id() { ... }
  Type type_eq() {              // method for equation defining
    return env().lookup(id()).type()  // synthesized attribute
  } }
```

This demand-driven evaluation scheme implicitly results in topological-order evaluation (evaluation order according to the attribute dependences). See [1] for more details.

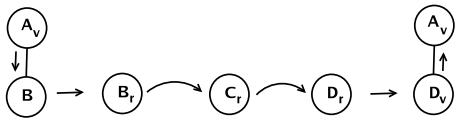
Attribute evaluation using this scheme will often follow complex tree traversal patterns, often visiting the same node multiple times in order to evaluate all the attributes that a specific attribute depends on. For example, consider the evaluation of the attribute *Id.type* above. This involves finding the declaration of the identifier, then finding the declaration of the type of the identifier, and during this process, possibly finding the declarations of classes in the superclass chain where these declarations may be located. In this process, the same block nodes and declaration nodes may well be visited several times. However, once a certain attribute is evaluated, e.g., the reference from a class to

its superclass, that computation does not need to be redone since the attribute value is cached. The traversals do therefore not always follow the tree structure, but can also follow reference attributes directly, e.g., from subclass to superclass or from variable to declaration.

5.2 Basic rewrite strategy

To handle ReRAGs, the evaluator is extended to rewrite trees in addition to evaluating attributes, resulting in a consistently attributed tree where all equations hold and all rewrite conditions are false. A demand-driven rewriting strategy is used. When a tree node is visited, the node is rewritten iteratively. In each iteration, the rule conditions are evaluated in priority order, and the first applicable rule will be applied, replacing the node (or parts of the subtree rooted at the node). The next iteration is applied to the root of the new subtree. The iteration stops when none of the rules are applicable (all the conditions are false), and a reference to the resulting subtree is then returned to the visiting node. The subtree may thus be rewritten in several steps before the new subtree is returned to the visiting node. Since the rewrites are applied implicitly when visiting a node, the rewrite is transparent from a node traversal point of view.

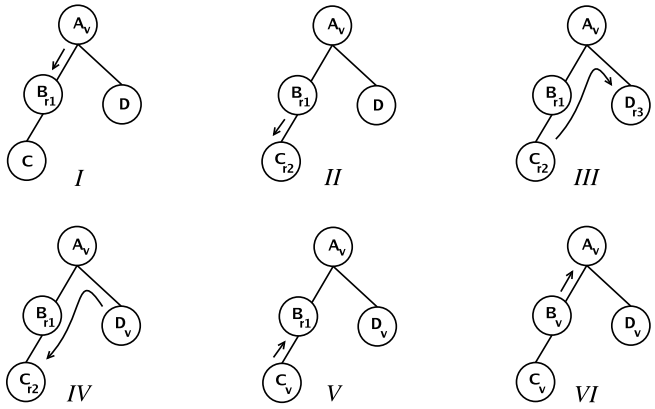
The figure below shows how the child node *B* of *A* is accessed for the first time and iteratively rewritten into the resulting node *D* that is returned to the parent *A*. The subscript *v* indicates that a node has been visited and *r* that a rewrite is currently being evaluated. When *B* is visited a rewrite is triggered and the node is rewritten to a *C* node that in turn is rewritten to a *D* node. No rewrite conditions for the *D* node are true, and the node is returned to the parent *A* that need not be aware of the performed rewrite.



5.3 Nested and multi-level rewrites

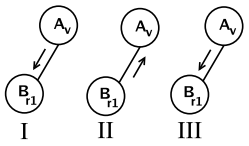
When evaluating a condition or a result expression in a rewrite rule, attributes may be read that trigger a visit to another node. That visit may in turn trigger a second rewrite that is executed before the first may continue its evaluation. This nesting of rewrites results in several rewrites being active at the same time. Since attributes may reference distant subtrees, the visited nodes could be anywhere in the AST, not necessarily in the subtree of the rewritten tree.

The following figure shows an example of nested rewrites. The subscript *v* indicates that a node has been visited and *r* that a rewrite is currently being evaluated. The rewrites are numbered in the order they started.



An initial rewrite, *r1*, is triggered when *A* visits its child *B* in stage *I*. A visit to *C*, that is caused by accessing a synthesized attribute during rewrite condition checking, triggers a second rewrite *r2* in stage *II*. That rewrite triggers a visit to a distant node *D* by reading an inherited attribute and initiates a third rewrite *r3* in stage *III*. When no conditions in *D* are true the result of the inherited attribute is calculated and returned to *C* in stage *IV*. The synthesized attribute is calculated and returned to *B* in stage *V*. The resulting node *B* is finally returned to *A* in stage *VI*. Notice that the rewrites terminate in the opposite order that they were initiated.

As discussed in Section 5.1, most non-trivial attribute grammars are multi-visit in that a node may have to be visited multiple times to evaluate an attribute. A common situation is when a child node has an inherited attribute, and the equation in the parent node depends on a synthesized attribute that visits the child node again. The situation is illustrated in the figure below. *A* visits *B* and a rewrite is initiated in stage *I*. During condition evaluation the inherited attribute *y()* is read and *A* is visited to evaluate its equation in stage *II*. That equation contains the synthesized attribute *x()* that in turn depends on *z()* in *B* and a second visit is initiated in stage *III*.




```
ast A ::= B child;
inh Type B.y();
eq A.child().y() = x();
syn Type A.x() = child().z();
syn Type B.z() = ... ;
```

Such multi-visits complicate the rewrite and attribute evaluation process somewhat. Should the second visit to a node that is being rewritten start a second rewrite? No. The attributes read in a node that is being rewritten should reflect the current tree structure. Otherwise, the definition of rewrites would be circular and evaluation would turn into an endless loop. Therefore, when visiting a node that is already being rewritten, the current node is returned and no new rewrite is triggered.

Note that attribute values that depend on nodes that are being rewritten, might have different values during the rewrite than they will have in the final tree. Therefore, such

attributes will not be cached until all the nodes they depend on have reached their final form. We will return to this issue in Section 6.3.

Note also that a node may well be rewritten several times, provided that the previous rewrite has completed. This can happen if the rewrites are triggered by the rewriting of another node. For example, suppose we are rewriting a node *A*. During this process, we visit its son node *S* which is then rewritten to S'. After this rewrite of *S*, the conditions of S' are all false (the rewrite of S completes). We then complete one iteration of rewriting *A*, replacing it with a new node A'(but keeping the son S'). In the next iteration of rewriting A', it may be found that S' needs to be rewritten again since the conditions of S' may give other results after replacing *A* by A'. This will also be discussed more in Section 6.2.

6 Implementation algorithm

6.1 Basic algorithm

As discussed in Section 3.2, a Java class is generated for each node type in the AST. All classes in the class hierarchy descend from the same superclass, *ASTNode*, providing generic traversal of the AST by the generic *parent()* and *child(int index)* methods. These methods are used in the implementation of attribute and equation methods, as discussed in Section 5.1.

We have implemented our rewriting algorithm by extending the existing JastAdd RAG evaluator as an AspectJ [10] aspect. In particular, the *child* method is extended to trigger rewrites when appropriate. To start with, we consider the case when no RAG attributes are cached. The handling of cached attributes in combination with rewriting is treated in Section 6.3.

Rewrite rules for each node type are translated into a corresponding Java method, *rewriteTo()*, that checks rewrite rule conditions and returns the possibly rewritten tree. This method is iteratively invoked until no conditions are true. If all conditions in one node's rewriteTo() method are false, then rewriteTo() in the node's superclass is invoked. The generated Java method for the first example in Section 4 is shown below.

```
ASTNode Add.rewriteTo() {
  if(childType().equals(TypeSystem.STRING))
    return new StringAdd(leftOp(), rightOp())
  return super.rewriteTo();
}
```

To determine when no conditions are true and iteration should stop, a flag is set when the rewriteTo() method in ASTNode is reached, indicating that no overriding rewriteTo method has calculated a result tree. A flag is used since a simple comparison of the returned node is not sufficient because the rewrite may have rewritten descendent nodes only. In order to handle nested rewrites, a stack of flags is used.

Figure 5 shows an AspectJ aspect implementing the above described behaviour:

(1) The stack used to determine when no conditions are true
(2) Iteratively apply rewrite until no conditions are true

(3) Push *false* on the stack to guess that a rewrite will occur
(4) Bind the rewritten tree as a child to the parent node.
(5) Set top value on stack to *true* when rewriteTo in ASTNode is reached (no rewrite occurred)
(6) Define a pointcut when the child method is called.
(7) Each call to child is extended to also call rewrite.

```
   public aspect Rewrite {
(1)  protected static Stack noRewrite = new Stack();
(2)  ASTNode rewrite(ASTNode parent, ASTNode child, int index) {
       do {
(3)      noRewrite.push(Boolean.FALSE);
         child = child.rewriteTo();
(4)      parent.setChild(index, child);
       } while(noRewrite.pop() == Boolean.FALSE);
       return child; }
(5)  ASTNode ASTNode.rewriteTo() {
       noRewrite.pop();
       noRewrite.push(Boolean.TRUE);
       return this; }
(6)  pointcut child(int index, ASTNode parent) :
       call(ASTNode ASTNode.child(int)) &&
       args(index) && target(parent);
(7)  ASTNode around (int index, ASTNode parent) :
            child(index, parent) {
       ASTNode child = proceed(index, parent):
       return rewrite(parent, child, index); }
   }
```

Fig. 5. Aspect Rewrite: Iteratively rewrite each visited tree node

As discussed in Section 5.3 a tree node currently in rewrite may be visited again during that rewrite when reading attributes. When a node that is in rewrite is visited, the current tree state should be returned instead of initiating a new rewrite. That behaviour is implemented in the aspect shown in Figure 6:

(1) A flag, inRewrite, is added to each node to indicate whether the node is in rewrite or not.
(2) Add advice around each call to the rewriteTo method.
(3) The flag is set when a rewrite is initiated.
(4) The flag is reset when a rewrite is finished.
(5) Add advice around the rewrite loop in the previous aspect.
(6) When a node is in rewrite then the current tree is returned instead of initiating a new rewrite.

```
    public aspect ReVisit {
(1)   boolean ASTNode.inRewrite = false;
(2) ASTNode around(ASTNode child)
     : execution(ASTNode ASTNode+.rewriteTo()) && target(child) {
(3)   child.inRewrite = true;
      ASTNode newChild = proceed(child);
(4)   child.inRewrite = false;
      return newChild; }
(5) ASTNode around(ASTNode child)
      : execution(ASTNode Rewrite.rewrite(ASTNode, ASTNode, int)
      && args(*, child, *) {
(6)   if(child.inRewrite)
        return child;
      return proceed(child);
    }
```

Fig. 6. Aspect ReVisit: Pass through re-visit to a node already in rewrite

6.2 Optimization of final nodes

As mentioned, a node may be rewritten several times. We are interested in detecting when no further rewriting of it is possible so we know that it has reached its final identity. By detecting final nodes, we can avoid the needless checking of their rewrite conditions (since they will all be false). This performance improvement can be significant for nodes with expensive conditions, e.g., when extracting a property by visiting all the children of the node. We can also use the concept of final nodes to cache attributes, as will be discussed in Section 6.3.

Definition 1. *A node is said to be* final *when i) all its rewrite conditions evaluate to false, and ii) future evaluations of its rewrite conditions cannot yield other values, and iii) it cannot be rewritten by any other node.*

Clearly, no further rewriting of final nodes is possible: i) and ii) guarantee that the node itself cannot trigger any rewriting of it, and iii) that it cannot be rewritten by any other node.

To find out when a node is final, we first recall (from Section 4) which nodes may be changed by a rewrite rule. Consider a node N which is the root of a subtree T. The rewrite rule will result in replacing T by T', where T' consists of a combination of newly created nodes and old nodes from T. I.e., the rewrite may not change nodes outside T. From this follows that a node can only be rewritten by rules in the node itself or rules in nodes on the path to the AST root node.

This allows us to state that

Lemma 1. *If a node is final, all its ancestor nodes are final.*

Proof. Otherwise the node may be rewritten by an ancestor node, in which case it is not final.

From Lemma 1 follows that at any point during evaluation, the final nodes of the AST will constitute a connected region that includes a path to the root, the *final region*. Initially, the evaluator visits only nodes in the final region, and is said to be in *normal* mode. But as soon as a non-final node is accessed from normal mode, the evaluator enters *rewrite* mode and that non-final node is said to be a *candidate*. When the iterative rewriting of the candidate has finished it turns out that it is final (see Theorem 2, and the evaluator returns to normal mode, completing the rewrite session. This way the final region is successively expanded. During a rewrite session, other non-final nodes may be visited and rewritten, but these are not considered candidates and will not become final during that rewrite session. There is only one candidate per rewrite session.

Note that during a rewrite session, the evaluator may well visit non-final nodes outside of the candidate subtree, and non-final nodes may be visited several times, the candidate included. For example, let us say we are rewriting a class String to add an explicit superclass reference to class Object. This means we will visit and trigger a rewrite of class Object. The rewrite of Object includes adding an explicit constructor. This involves searching through the methods of Object for a constructor. Suppose there is a method String toString() in Object. When this method is traversed, this will trigger rewriting of the identifier String to a type reference that directly refers to the String class. This in turn will involve a second visit to the String class (which was the candidate).

Theorem 2. *At the end of a rewrite session, the candidate c is final.*

Proof. At the end of the rewrite session, all rewrite conditions of *c* have just been evaluated to false. Furthermore, all ancestors of *c* are final, so no other node can rewrite *c*. What remains to be shown (see Definition 1) is that future evaluations of the rewrite conditions cannot yield other values. To see this we must consider the set of all other non-final nodes *N* that were visited in order to evaluate the rewrite conditions of *c*. This has involved evaluating all the rewrites conditions of these nodes in turn, also yielding false for all these conditions, and without triggering any rewrites of those nodes. Otherwise, another iteration of rewrite of *c* would have been triggered and we would not be at the end of the rewriting session. Since all these conditions evaluate to false, and there is no other node that can rewrite any of the nodes in *N* (since their ancestors outside *N* are final), none of these conditions can change value, and not only *c*, but in fact all nodes in *N* are final. □

In keeping track of which nodes are final, we add a flag isFinal to each node. In principle, we could mark both *c* and all the nodes in *N* as final at the end of the rewriting session. However, it is sufficient to mark *c* since any subsequent visits to a node in *N* will immediately mark that node as final, since all its rewrite conditions are false. An aspect introducing the isFinal flag is implemented in the aspect shown in Figure 7:

(1) A flag, isFinal, is added to each node to indicate whether the node is final or not.
(2) Add advice around the rewrite loop in the Rewrite aspect.
(3) When a node is final no rule condition checking is necessary and the node is returned immediately.

(4) When a node is entered during normal mode it becomes the next node to be final and we enter rewrite mode. On condition checking completion the node is final and we enter normal mode.
(5) A rewrite during rewrite mode continues as normal.

```
   public aspect FinalNodes {
(1)  boolean ASTNode.isFinal = false;
(2)  boolean normalMode = true;
(2)  ASTNode around(ASTNode parent, ASTNode child)
       : execution(ASTNode Rewrite.rewrite(ASTNode, ASTNode, int))
       && args(parent, child, *) {
(3)    if(child.isFinal)
         return child;
(4)    if(normalMode) {
         normalMode = false;
         child = proceed(parent, child);
         child.isFinal = true;
         normalMode = true;
         return child; }
(5)    return proceed(parent, child); }
   }
```

Fig. 7. Aspect FinalNodes: Detect final nodes and skip condition evaluation

6.3 Caching attributes in the context of rewrites

In plain RAGs, attribute caching can be used to increase performance by ensuring that each attribute is evaluated only once. When introducing rewrites the same simple technique cannot be used. A rewrite that changes the tree structure may affect the value of an already cached attribute that must then be re-evaluated. There are two principle approaches to ensure that these attributes have consistent values. One is to analyze attribute dependences dynamically in order to find out which attributes need to be reevaluated due to rewriting. Another approach is to cache only those attributes that cannot be affected by later rewrites. In order to avoid extensive run-time dependency analysis, we have chosen the second approach.

We say that an attribute is *safely cachable* when its value cannot be affected by later rewrites. Because final nodes cannot be further rewritten, an attribute will be safely cachable if all nodes visited during its evaluation are final.

A simple solution is to only cache attributes whose evaluation is started when the evaluator is in normal mode, i.e., not in a rewriting session. These attributes will be safely cachable. To see this, we can note that

i) the node where the evaluation starts is final (since the evaluator is in normal mode)

ii) any node visited during evaluation will be in its final form before its attributes are accessed, since any non-final node encountered will cause the evaluator to enter rewrite mode, returning the final node after completing that rewriting session.

It is possible to cache certain attributes during rewriting, by keeping track dynamically of if all visited nodes are final. However, this optimization has not yet been implemented.

As mentioned earlier, the ReRAG implementation is implemented as aspects on top of the plain RAG implementation. The RAG implementation caches attributes, so we need to disable the caching whenever not in normal mode in order to handle ReRAGs. This is done simply by advice on the call that sets the cached-flag. Figure 8 shows how this is done.

```
public aspect DisableCache {
  Object around() : set(boolean ASTNode+.*_cached) {
    if(!FinalNodes.normalMode)
      return false;
    return proceed(); }
}
```

Fig. 8. Aspect DisableCache: Disable caching of attributes when not in normal mode

7 Implementation evaluation

7.1 Applicability

We have implemented ReRAGs in our tool JastAdd II and performed a number of case studies in order to evaluate their applicability.

Full Java static-semantics checker Our largest application is a complete static-semantic analyzer for Java 1.4. The grammar is a highly modular specification that follows the Java Language Specificaton, second edition[9], with modules like name binding, resolving ambiguous names, type binding, type checking, type conversions, inheritance, access control, arrays, exception handling, definite assignment and unreachable statements.

An LALR(1) parser using a slightly modified grammar from the Java Language Specification [9], is used to build the initial abstract syntax tree. The AST is rewritten during the analysis to better capture the semantics of the program and simplify later computations. Some examples where rewrites were useful are:

- for resolving ambiguous names and for using semantic specialization for bound name references.
- for making implicit constructs explict by adding (as appropriate) empty constructors, supertype constructor accesses, type conversions and promotions, and inheritance from *java.lang.Object*.

– for eliminating shorthands such as splitting compound declarations of fields and variables to a list of single declarations.

Java to C compiler Our collegue, Anders Nilsson, has implemented a Java to C compiler in ReRAGs [13], based on an older version of the Java checker. The generated C code is designed to run with a set of special C runtime systems that support real-time garbage collection, and is interfaced to through a set of C macros. ReRAGs are used in the back end for adapting the AST to simplify the generation of code suitable for these runtime systems. For example, all operations on references are broken down to steps of only one indirection, generating the macro calls to the runtime system. ReRAGs are also used for optimizing the generated code size by eliminating unused classes, methods, and variables. They are also used for eliminating shorthands, for example to deal with all the variants of loops in Java.

Worst-case execution time analyzer The Java checker was extended to also compute worst-case execution times using an annotation mechanism. The extension could be done in a purely modular fashion.

Automation Language The automation language *Structured Text* in IEC-61131-3 has been modeled in ReRAGs and extended with an object-oriented type system and instance references. The extended language is translated to the base language by flattening the class hierarchies using iterative rewriting. Details will appear in a forthcoming paper.

7.2 Performance

We have implemented ReRAGs in our aspect-oriented compiler compiler tool JastAdd II. To give some initial performance measurements we benchmark our largest application, a complete static-semantic analyzer for Java 1.4. After parsing and static-semantic analysis the checked tree is pretty printed to file. Since code generation targeted for the Java virtual machine, [14], is fairly straight forward once static-semantic analysis is performed we believe that the work done by our analyzer is comparable to the work done by a java to byte-code compiler. We therefore compare the execution time of our analyzer to the standard java compiler, javac, in J2SE JDK.

Two types of optimizations to the basic evaluation algorithm were discussed in Section 6.2 and Section 6.3. The first disables condition checking for nodes that are final and the second caches attribute values that only depend on attributes in final nodes. To verify that these optimizations improve performance we benchmark our analyzer with and without optimizations. The execution times when analysing a few files of the *java.lang* package are shown in Figure 9. These measurements show that both attribute caching and condition checking disabling provide drastic performance improvements when applied individually and even better when combined. Clearly, both optimizations should be used to get reasonable execution times.

The execution times do not include parsing that took 3262ms without attribute caching and slightly more, 3644ms, when caching attributes. We believe the increase is due to the larger tree nodes used when caching attributes.

To verify that the ReRAG implementation scales reasonably we compare execution times with a traditional Java compiler, javac, see Figure 10. We are using a subset of the Java class library, the *java.lang, java.util, java.io* packages, as our benchmarks.

	condition checking	no condition checking
no attribute caching	546323 ms	61882 ms
attribute caching	21216 ms	2016 ms

Fig. 9. Comparison of analysis execution time with and without optimizations

Roughly 100.000 lines of java source code from J2SE JDK 1.4.2 are compiled, and the ReRAG-based compiler uses both the optimizations mentioned above. The comparison is not completely fair because javac generates byte code whereas the ReRAG compiler only performs static-semantic analysis and then pretty-prints the program. However, generating byte code from an analyzed AST is very straight-forward and should be roughly comparable to pretty-printing. The comparison shows that the ReRAG-based compiler is only a few times slower than javac. Considering that the ReRAG-based compiler is generated from a declarative specification, we find this highly encouraging. This shows that ReRAGs are already useful for large-scale practical applications.

	total	JVM init	parsing	analysis and prettyprinting
ReRAG compiler	22801ms	600ms	7251ms	14950ms
javac	6112ms			

Fig. 10. Compile time for the *java.lang, java.util, java.io* packages using the ReRAG-based compiler and javac.

8 Related work

Higher-ordered Attribute Grammars ReRAGs are closely related to Higher-ordered Attribute Grammars (HAGs) [2], [3] where an attribute can be *higher-order*, in that it has the structure of an AST and can itself have attributes. Such an attribute is also called an *ATtributable Attribute* (ATA). Typically, there will be one equation defining the bare AST (without attributes) of the ATA, and other equations that define or use attributes of the ATA, and which depend on the evaluation of the ATA equation.

In ReRAGs each node in the AST is considered to be the root of a *rewritable attribute* of its parent node and may be rewritten to an alternative subtree during attribute evaluation. The rewriting is done conditionally, in place (replacing the original subtree during evaluation), and may be done in several steps, each described by an individual rewrite rule. This is contrast to the ATAs of HAGs which are constructed unconditionally, in one step, and where the evaluation does not change previously existing parts of the AST (the new tree is stored as a previously unevaluated attribute).

A major difference lies in the object-oriented basis of ReRAGs, where reference attributes are kept as explicit links in the tree and subtrees are rewritten in place.

HAGs, in contrast, have a functional programming basis, viewing the AST as well as its attributes as structured values without identity. This is in our view less intuitive where, for instance, cross references in the AST have to be viewed as infinite values.

HAGs + Forwarding Forwarding [4] is an attribute grammar technique used to forward attribute equations in one node to an equation in another node. This is transparent to other attribute equations and when combined with ATAs that use contextual information it allows later computations to be expressed on a more suitable model in a way similar to ReRAGs. To simulate a nested and multi-level rewrite there would, however, conceptually have to be a new tree for each step in the rewrite.

Visitors The Visitor pattern is often used in compiler construction for separation of concerns when using object-oriented languages. Visitors can only separate cross-cutting methods while the weaving technique used in JastAdd can be used for fields as well. This is superior to the Visitor pattern in that there is no need to rely on a generic delegation mechanism resulting in a cleaner more intuative implementation and also provide type-safe parameter passing during tree traversal. ReRAGs also differ in that traversal strategies need not be specified explicitly since they are implicitly defined by attribute dependences. The use of attributes provide better separation of concerns in that contextual information need not be included in the traversal pattern but can be declared separately.

Rewrite Systems ReRAGs also have similarities to tree transformation systems like *Stratego* [5], *ASF+SDF* [6], and *TXL* [7] but improves data acquisition support through the use of RAGs instead of embedding contextual data in rewrite rules or as global variables. *Stratego* uses Dynamic Rewrite Rules [15] to separate contextual data acquisition from rewrite rules. A rule can be generated at run-time and include data from the context where it originates. That way contextual data is included in the rewrite rule and need not be propagated explicitly by rules in the grammar. ReRAGs provide an even cleaner separation of rewrite rule and contextual information by the use of RAGs that also are superior in modeling complex non-local dependences. The rewrite application order differs in that ReRAGs only support the described declarative approach while the other systems support user defined strategies. In *Stratego* and *ASF+SDF* the user can define explicit traversal strategies that control rewrite application order. Transformation rules in *TXL* are specified through a pattern to be matched and a replacement to substitute for it. The pattern to be matched may be guarded by conditional rules and the replacement may be a defined as a function of the matched pattern. A function used in a transformation rule may in turn be a composed from other functions. The rewrite application strategy in *TXL* is thus implicitly defined as part of the functional decomposition of the transformation ruleset, which controls how and in which order subrules are applied. Dora [16] supports attributes and rewrite rules that are defined using pattern matching to select tree nodes for attribute definitions, equation, and as rewrite targets. Attribute equations and rewrite results are defined through Lisp expressions. Composition rules are used to define how to combine and repeat rewrites and the order the tree is traversed. The approach is similar to ReRAGs in that attribute dependences are computed dynamically at run-time but there is no

support for remote attributes and it is not clear how attributes read during rewriting are handled.

Dynamic reclassification of objects Semantic specialization is similar to dynamic reclassification of objects, e.g. Wide Classes, Predicate Classes, FickleII, and Gilgul. All of these approaches except Gilgul differ from ReRAGs in that they may only specialize a single object compared to our rewritten sub-trees. *Wide Classes* [17] demonstrates the use of dynamic reclassification of objects to create a more suitable model for compiler computations. The run-time type of an object can be changed into a super- or a sub-type by explicitly passing a message to that object. That way, instance variables can be dynamically added to objects when needed by a specific compiler stage, e.g., code optimization. Their approach differs from ours in that it requires run-time system support and the reclassification is explicitly invoked and not statically type-safe. In *Predicate Classes* [18], an object is reclassified when a predicate is true, similar to our rewrite conditions. The reclassification is dynamic and lazy and thus similar to our demand-driven rewriting. The approach is, however, not statically type-safe. *FickleII* [19] has strong typing and puts restrictions on when an object may be reclassified to a super type by using specific state classes that may not be types of fields. This is similar to our restriction on rewriting nodes to supertypes as long as they are not used in the right hand side of a production rule as discussed in Section 4.2. The reclassification is, however, explicitly invoked compared to our declarative style. *Gilgul* [20] is an extension to Java that allows dynamic object replacement. A new type of classes, implementation-only classes, that can not be used as types are introduced. Implementation-only instance may not only be replaced by subclass instances but also by instances of any class that has the same least non implementation-only superclass. Object replacement in Gilgul is similar to our appraoch in that no support from the run-time system is needed. Gilgul uses an indirection scheme to be able to simultaneously update all object references through a single pointer re-assignment. The ReRAGs implementation uses a different approach and ensures that all references to the replaced object structure are recalculated dynamically on demand.

9 Conclusions and Future Work

We have introduced a technique for declarative rewriting of attributed ASTs, supporting conditional and context-dependent rewrites during attribution. The generation of a full Java static-semantic analyzer demonstrates the practical use of this technique. The grammar is highly modular, utilizing all three dimensions of separation of concerns: inheritance for separating the description of general from specific behavior of the language constructs (e.g., general declarations from specialized declarations like fields and methods); aspects for separating different computations from each other (e.g., type checking from name analysis); and rewriting for allowing the computations to be expressed on suitable forms of the tree. This results in a specification that is easy to understand and to extend. The technique has been implemented in a general system that generates compilers from a declarative specification. Attribute evaluation and tree transformation are performed automatically according to the specification.The running times

are sufficiently low for practical use. For example, parsing, analyzing, and prettyprinting roughly 100.000 lines of Java code took approximately 23 seconds as compared to 6 seconds for the javac compiler on the same platform.

We have identified several typical ways of transforming an AST that are useful in practice: Semantic Specialization, Make Implicit Behavior Explicit, and Eliminate Shorthands. The use of these transformations has substantially simplified our Java implementation as compared to having to program this by hand, or having to use a plain RAG on the initial AST constructed by the parser.

Our work is related to many other transformational approaches, but differs in important ways, most notably by being declarative, yet based on an object-oriented AST model with explicit references beween different parts. This gives, in our opinion, a very natural and direct way to think about the program representation and to describe computations.

Many other transformational systems apply transformations in a predefined sequence, making the application of transformations imperative. In contrast, the ReRAG transformations are applied based on conditions that may read the current tree, resulting in a declarative specification.

There are many interesting ways to continue this research.

Optimization The caching strategies currently used can probably be improved in a variety of ways, allowing more attributes to be cached, resulting in better performance.

Termination Our current implementation does not deal with possible non-termination of rewriting rules (i.e., the possibility that the conditions never become false). In our experience, it can easily be seen (by a human) that the rules will terminate, so this is usually not a problem in practice. However, techniques for detecting possible non-termination, either statically from the grammar or dynamically, during evaluation, could be useful for debugging.

Circular ReRAGs We plan to combine earlier work on Circular RAGs [21] with our work on ReRAGs. We hope this can be used for running various fixed-point computations on ReRAGs, with applications in static analysis.

Language extensions Our current studies on generics indicate that the basic problems in GJ [22] can be solved using ReRAGs. Extending our Java 1.4 to handle new features in Java 1.5 like generics, autoboxing, static imports, and type safe enums is a natural next step. This will also further illustrate how language extensions can be modularized using ReRAGs.

Acknowledgements

We are grateful to John Boyland and to the other reviewers (anonymous) for their valuable feedback on the first draft of this paper.

References

1. Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) **24** (2000)

2. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher order attribute grammars. In: Proceedings of the SIGPLAN '89 Programming language design and implementation, ACM Press (1989)
3. Saraiva, J.: Purely functional implementation of attribute grammars. PhD thesis, Utrecht University, The Netherlands (1999)
4. Van Wyk, E., Moor, O.d., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Proceedings of Compiler Construction Conference 2002. Volume 2304 of LNCS., Springer-Verlag (2002) 128–142
5. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In: Proceedings of Rewriting Techniques and Applications (RTA'01). Volume 2051 of LNCS., Springer-Verlag (2001) 357–361
6. van den Brand et al., M.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In: Proceedings of Compiler Construction Conference 2001. Volume 2027 of LNCS., Springer-Verlag (2001)
7. Cordy, J.R.: Txl: A language for programming language tools and applications. In: Proceedings of the 4th Workshop on Language Descriptions, Tools, and Applications (LDTA'04) at ETAPS 2004. (2004)
8. Hedin, G., Magnusson, E.: JastAdd: an aspect-oriented compiler construction system. Science of Computer Programming **47** (2003) 37–58
9. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass. (2000)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. LNCS **2072** (2001) 327–355
11. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2** (1968) 127–145 Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).
12. Hedin, G.: An object-oriented notation for attribute grammars. In: the 3rd European Conference on Object-Oriented Programming (ECOOP'89). BCS Workshop Series, Cambridge University Press (1989) 329–345
13. Nilsson, A.: Compiling Java for Real-Time Systems. Licentiate thesis, Department of Computer Science, Lund Institute of Technology (2004) In preparation.
14. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc. (1999)
15. Visser, E.: Scoped dynamic rewrite rules. Electronic Notes in Theoretical Computer Science **59** (2001)
16. Boyland, J., Farnum, C., Graham, S.L.: Attributed transformational code generation for dynamic compilers. In Giegerich, R., Graham, S.L., eds.: Code Generation - Concepts, Tools, Techniques. Workshops in Computer Science. Springer-Verlag (1992) 227–254
17. Serrano, M.: Wide classes. In: Proceedings of ECOOP'99. Volume 1628 of LNCS., Springer-Verlag (1999) 391–415
18. Chambers, C.: Predicate classes. In: Proceedings of ECOOP'93. Volume 707 of LNCS., Springer-Verlag (1993) 268–296
19. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: More dynamic object reclassification: FickleII;. ACM Trans. Program. Lang. Syst. **24** (2002) 153–191
20. Costanza, P.: Dynamic object replacement and implementation-only classes. In: 6th International Workshop on Component-Oriented Programming (WCOP 2001) at ECOOP 2001. (2001)
21. Magnusson, E., Hedin, G.: Circular reference attributed grammars - their evaluation and applications. Electronic Notes in Theoretical Computer Science **82** (2003)
22. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In: Proceedings of Object Oriented Programming: Systems, Languages, and Applications (OOPSLA. (1998) 183–200