Nordic Journal of Computing

IMPLEMENTING JAVA COMPILERS USING ReRAGs

Anders Nilsson   Anders Ive   Torbjörn Ekman
Görel Hedin

Department of Computer Science
Lund University
Box 118, SE-221 00 Lund, Sweden

(andersn|ive|torbjorn|gorel)@cs.lth.se

Abstract. Rewritable Reference Attributed Grammars (ReRAGs) is a recently developed compiler-compiler technology based on object-orientation, aspect-orientation, reference attributed grammars, and conditional rewrites. In this paper we describe our experiences from using ReRAGs for implementing Java compilers. We illustrate how the usage of ReRAGs renders a rather compact, yet easy-to-understand and modular compiler specification where code analysis, restructurings, and optimizations can be conveniently described as aspects performing computations and transformations on the abstract syntax tree. Currently, we have implemented two compilers: one that generates C code with real-time support, and one that generates Java bytecode. Both share the same front end.

ACM CCS Categories and Subject Descriptors: D.1.5 Object-oriented Programming, D.2.13 Reusable Software, D.3.4 Processors (Code generation, Translator writing systems and compiler generators)

Key words: Java compiler, modular implementation, declarative implementation, aspect-oriented programming, conditional rewrites, attribute grammars, reference attributes

1. Introduction

Compilers are complex systems, and implementing them in a modular way is a challenging task. In this paper we describe our experiences from using Rewritable Reference Attributed Grammars (ReRAGs) [2] for generating Java compilers. ReRAGs is a conditional rewrite formalism that is based on object-orientation, aspect-orientation, and reference attributed grammars. In [4] it was illustrated how ReRAGs can be used for specifying a front end for Java in a modular high-level manner. In this paper, we look at how these techniques can be applied also to the compiler backend and code generation.

A major challenge in implementing compilers in a modular way is how to deal with contextual information. For example, a syntactic source construct may have several different interpretations depending on context, and should result in correspondingly different code instruction sequences. Many optimizations also depend on contextual information. For example, a reference to a constant variable may be replaced by a constant literal.

ReRAGs support rewriting of an abstract syntax tree (AST) using contextual information. An AST for a program can be transformed in a series of steps, allowing each computation to be expressed on the most suitable form of AST. The Java front end uses these techniques to transform the syntax-oriented source AST (produced by the parser) into an attributed AST, which reflects the static semantics of the program. Our backends

Received 8th December 2004.

use ReRAGs to further transform the AST in a series of steps, to prepare, optimize, and generate code.

We have developed two Java compilers using these techniques: one complete compiler, Java2C, that generates C code with real-time support, and one experimental compiler, Java2Bytecode, that generates Java bytecode.

The rest of this paper is structured as follows. Section 2 gives an introduction to ReRAGs. Section 3 describes the architecture of our compilers. Section 4 describes the general parts of the back end, and section 5 the specific parts of the backends, with a focus on the Java2C compiler. Section 6 evaluates our approach and compares it to compilers handwritten in Java, section 7 discusses future work, and section 8 concludes the paper.

2. ReRAGs

ReRAGs is a declarative conditional rewrite formalism based on reference attributed grammars. There are two features of ReRAGs that are particularly important and that distinguish them from many other rewriting systems. The first is that the rewrite conditions can make use of attributes to easily describe complex contextual conditions for the rewrites. The second is that the formalism is declarative: attributes and rewrites are evaluated automatically, driven by the dependencies, rather than by explicitly given evaluation orders. This makes the evaluation transparent: whenever a syntax node or attribute is inspected, it will automatically be in its final form according to the grammar.

ReRAGs have been implemented in a compiler compiler tool, JastAdd II [3]. The specification language used is an extension to Java. There is strong support for separation of concerns through the combined use of object-orientation, aspect-orientation, reference attributed grammars, and rewrites.

The JastAdd tool does not include any support for parsing, but relies on the use of an external parser. Any parser generator that can generate Java code will do. We have used JavaCC [15] and CUP [10] for different versions of our compilers.

2.1 Object orientation

The AST is described in an object-oriented fashion by a class hierarchy, as in the Interpreter pattern [6]. Behavior common to a group of language constructs can be specified in a common superclass and specialized in subclasses. An AST class corresponds to a nonterminal or a production (or a combination thereof) and may define a number of descendents and their declared types. The AST nodes must be *type consistent* according to the normal type checking rules of Java. Support for lists, optionals, and lexical items are also provided. An example of some AST classes is shown in Figure 1.

2.2 Aspect orientation

Rather than defining the AST behavior directly in the AST classes, it is defined in *aspect modules*, supporting static aspect-oriented programming similar to open classes or static introduction in AspectJ [12]. These modules allow behavior that crosscuts the AST class hierarchy to be specified together. For example, name analysis, type analysis, error checking, code generation, can be specified in different modules, although they add attributes and rewrites that belong to the same classes. In addition to enhancing readability, this makes it possible to add or remove specific aspect modules during implementation or debugging, and to reuse modules for different versions of a compiler.

The behavior defined in the aspect modules is typically in the form of rewrites, attributes, and equations. However, it is also possible to add ordinary Java code that make

```
// Expr is an abstract AST class and corresponds to a nonterminal
abstract ast Expr;

// Id is a subclass of Expr and corresponds to a production
// id is a String-valued token
ast Id : Expr ::= <String id>;

// BinExpr is an abstract subclass of Expr representing binary expressions
abstract ast BinExpr : Expr ::= Expr left, Expr right;

// ArithmeticBinExpr is an abstract subclass of BinExpr
abstract ast ArithmeticBinExpr : BinExpr;

// AddExpr is a subclass of ArithmeticBinExpr
ast AddExpr : ArithmeticBinExpr;

// RelationalBinExpr is an abstract subclass of BinExpr
abstract ast RelationalBinExpr : BinExpr;

// LessThanExpr is a subclass of RelationalBinExpr
ast LessThanExpr : RelationalBinExpr;
```

Figure 1: The grammar is expressed as an object-oriented class hierarchy.

use of the attributes. If desired, such Java code can be added in aspect modules that extend the AST classes with variables and methods. This is useful, e.g., for printing the generated code to a file.

Aspects are much more powerful than the Visitor design pattern [6]. First, they allow instance variables and interface implementation clauses to be modularized, and not only methods. Second, the aspects allow the types of method arguments and return values to be retained, whereas the Visitor pattern requires these to be upcast to some common type, typically `Object`. Third, aspects can group declarations in an arbitrary manner. For example, if a new language construct is added, its combined behavior concerning name analysis, type analysis, etc. can be grouped together in an aspect if desired. Visitors can only group implementations of the same method.

2.3 Reference attributed grammars

ReRAGs are based on Reference Attributed Grammars (RAGs) [8], which is an object–oriented extension to Attribute Grammars (AGs) [13]. RAGs extend plain AGs by allowing attributes to be references to nodes in the AST. This allows simple and straightforward representation of non-local relationships in the AST, such as cross-references, superclass-subclass relations, etc.

Attributes are defined by equations residing either in the node itself (synthesized attributes) or in an ancestor node (inherited attributes). The attributes defined by a certain module constitutes an API that can be used by other modules. In particular, the code generation modules use attributes defined by the name and type analysis modules.

2.4 Rewrites

ReRAGs extend RAGs with rewrite rules 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 an ancestor node, in some imperative code traversing the AST, or even during other rewrites. 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.

A rewrite step is specified by a rewrite rule that defines the conditions when the rewrite is applicable, as well as the resulting tree. After the application of one rewrite rule, more rewrite rules may become applicable. This allows complex rewrites to be broken down into a series of simple small rewrite steps. Since rewrite rules are declarative, their lexical order is irrelevant.

A rewrite rule has the following form:

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

This specifies that a subtree of type *N* will be replaced by the subtree *result* of type *R*, if the condition *cond* holds. To maintain type consistency, *R* must be the same type or a subtype of *N*. The rewrite may be destructive in that it may reuse nodes from the original *N* tree to build the result tree. Therefore, a rewrite that involves changes to several nodes must be placed in a common ancestor of these nodes. The expression computing *result* may be written using imperative code, but must in that case contain no side effects apart from changing the structure of the rewritten tree.

For a given node, there may be several rewrite rules that apply at the same time. The rewrite rules should be written so that they are confluent, i.e., that the order of evaluation is irrelevant. If the rules are not confluent, their conditions should normally be made more specific so that they do in fact not apply at the same time. This is discussed in more detail in [4].

2.5 Node specialization

A particularly common way of rewriting in ReRAGs is to *specialize* a node, i.e., to replace the node of a class *A* by a node of another class *B*, where *B* is a subclass to *A*. This is useful for turning an AST generated from a context-free grammar into a context-sensitive AST, i.e., one where the nodes are selected based on their context. For example, a use of a name can be specialized into a variable use or a method use, depending on the meaning of the name. Because this rewrite is so common we have introduced a special shorthand syntax for it:

```
specialize N {
  when (cond)
  to result;
}
```

This is a shorthand for a normal rewrite rule with an additional condition that the node is *exactly* of type *N* (and not of any subtype of *N*), and where the result is a subtype of *N*.

3. Compiler architecture

The architecture of our compiler differs from traditional compilers in that there is no explicit symbol table or other large external data structures. Instead, the attributed AST is itself used as the main representation.

A concrete grammar description is used to create a parser, while an abstract grammar describes the AST class hierarchy. A collection of aspect modules, including name- and type analysis, optimizations and code generation, are woven into the AST classes. The generated parser, the woven AST classes, and auxiliary hand-written Java code make up the compiler, see Figure 2.

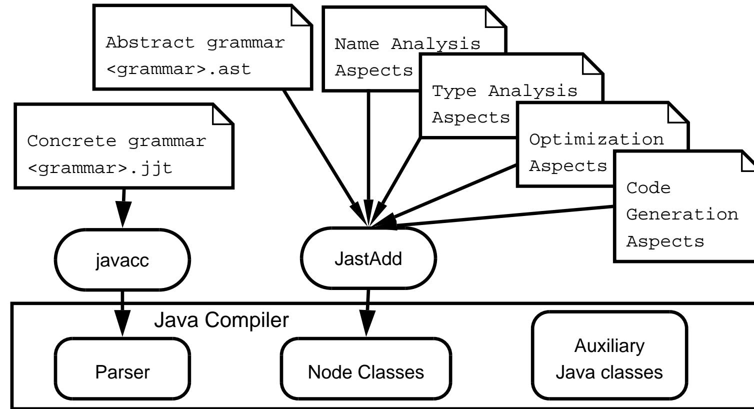

Figure 2: Compiler generation.

The compilation behavior is divided into a front end, which performs parsing and static-semantic analysis, and a back end, which performs optimizations and generates code. Figure 3 shows a typical ReRAG compiler architecture. The boxes represent APIs defined by ReRAG aspects, and the arrows dependencies: either generational (hollow arrows) or rewrites (arched arrows). The boxes can also be interpreted as conceptual states and the arrows as phases. Thus, we can think of the figure as showing a source file that is parsed into a syntactic AST; then semantic analysis is performed resulting in a semantic AST; then the AST is prepared for code generation and optimized; then an intermediate representation AST is generated which is further optimized; and finally the target code is output. In reality, all the steps (except for the first and the last) are applied on demand rather than monolithically, and different parts of the program may be in different states at a given point during evaluation.

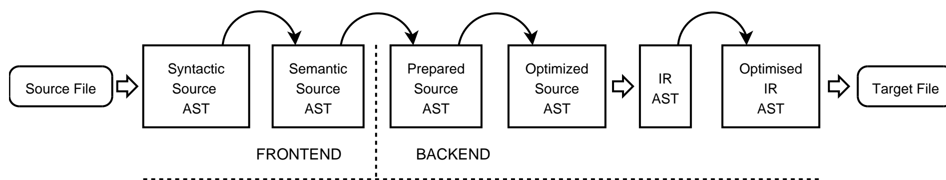

Figure 3: Typical ReRAG architecture showing different conceptual states of the compilation resulting from rewrites (arched arrows) or generation (hollow arrows).

In the front end, the parser reads the source file and produces a *syntactic source AST*, i.e., an undecorated AST that closely follows the context-free grammar used by the parser. The static-semantic analyzer is a ReRAG that transforms the syntactic AST into a *semantic source AST* that reflects the static semantics of the program. The semantic AST is attributed, and it is rewritten to capture context-sensitive properties that could not be detected by the context-free grammar. For example, each name node (an AST node representing the use of an identifier) is bound directly to its corresponding declaration node through a reference attribute. Furthermore, each syntactic name node is rewritten to a specialized node, e.g., corresponding to a variable name or a method name. The AST is also rewritten to be simplified, eliminating various ”shorthand” constructs in the language. For example, a declaration clause listing many variables of the same type can be rewritten to

a list of declaration clauses, one for each variable.

The backend uses ReRAGs to go from the semantic AST and optimize it and generate code. This is done in several conceptual phases. First, the semantic AST is prepared for code generation by further specializing and rewriting some AST nodes. The goal of this *preparation phase* is to allow straightforward code generation, and results in a *prepared source AST*. For example, the semantic AST contains general variable nodes, while the prepared AST has semantically specialized them into instance variables, class variables, local variables, parameters, etc., since the code will differ for accessing these different kinds of variables. Second, the prepared AST is optimized into an *optimized source AST* by performing high-level transformations of the AST. In our Java2C backend, the C code is generated directly from the optimized source AST. In our Java2Bytecode backend, we first generate a bytecode AST (as the intermediate representation). The bytecode AST initially contains a basic set of bytecodes. A subsequent phase replaces some of the basic bytecodes with more efficient ones, resulting in an *optimized bytecode AST*, which is traversed and printed to a target classfile.

The overall view of the architecture given in Figure 3 is simplified: each of the phases actually consists of several smaller aspects. For example, the semantic analysis phase consists of both name and type analysis aspects, and the preparation phase consists of both general preparation aspects used for both backends, and target specific preparation aspects. The linear arrangement of the phases is also a simplification, aimed only at giving an overall picture of the compiler architecture. For example, some of the optimization aspects depend only on the semantic AST and not on the preparation phase, and could as well be placed before the preparation phase.

4. General parts of the back end

Many parts of the back end are general and applicable regardless of the kind of target code generated. This includes general preparations of the AST in order to simplify the code generation, and also many optimizations.

4.1 Removing redundant variety

The Java language permits many different ways of expressing the same behavior. This variety is desirable for programmers, but not for the compiler implementor. One way of preparing the AST for code generation is to eliminate such redundant variety. This can be done by rewriting the AST to express the programs in a simplified and more homogeneous form, while preserving the semantics.

4.1.1 Simplifications in the front end

Some of the simplifications are done already in the front end in order to simplify the semantic analysis and to provide a simpler API for subsequent phases like back ends and analysis tools. Figure 4 shows some typical simplifications carried out in the front end.

4.1.2 Splitting declarations and initialization code

The back end contains additional simplifications. An example is the splitting of declarations from their initialization code as shown in figure 5. Note that the initializations of instance variables are moved into those constructors that directly call a super constructor.

Figure 6 shows the rewrite rule that splits the declaration and initialization code for local variables. The relevant part of the abstract grammar is shown on lines 1–4 (but are actually part of a separate file). The rewrite rule (starting on line 6) applies to VariableDecl

Rewrite	*Example*
Compound declaration splitting	`int x, y;` ➔ `int x; int y;`
Default constructor insertion	`class A{}` ➔ `class A{A(){}}`
Default super insertion	`A(){}` ➔ `A(){ super(); }`
Implicit typecast insertion	`9*3.14F;` ➔ `(float)9*3.14F;`
Implicit `this` insertion	`f(p);` ➔ `this.f(this.p);`

Figure 4: Typical simplifications in the front end.

Rewrite		*Example*
Declaration and initialization splitting	local var.	`int i=9;` ➔ `int i; i=9;`
	instance variable	`class A {int i=9; A() {super();}}` ➔ `class A {int i; A() {super(); i=9;}}`
	class variable	`class A {static int i=9;}` ➔ `class A {static int i; static {i=9;}}`

Figure 5: Examples of splitting declarations and initializations.

objects that are in the context of a Block's statement list (line 6), and where there is an initialization part (line 7). When the rewrite is applied, the VariableDecl is replaced by a list of two statements (line 18): one which is the old VariableDecl (`this`) without its initialization part, and one which is a new assignment, initializing the variable.

```
1 ast VariableDecl : Stmt ::= Type type, IdDecl idDecl, [Expr init];
2 ast AssignExpr   : Expr ::= Expr dest, Expr source;
3 ast Block        : Stmt ::= Stmt stmt*;
4 ast ExprStmt     : Stmt ::= Expr expr;
5 ...
6 rewrite VariableDecl in Block.stmt() {
7   when (hasInit())
8   to List {
9     Expr init = init();
10    removeOptInit();
11    Stmt assign =
12      new ExprStmt(
13        new AssignExpr(
14          new VarAccess(new IdUse(idDecl().id())),
15          init
16        )
17      );
18    return new List().add(this).add(assign);
19  }
20 }
```

Figure 6: Rewrite rule specifying splitting of variable declaration and initialization.

4.2 *Differentiating nodes that will require different code generation*

In many cases, the same kind of language construct will generate quite different code, depending on context. In order to prepare for the code generation, these different cases can be split up by rewriting them to more specialized nodes. Subsequent code generation can then be specified separately for the different cases, associated with the appropriate AST type, thus allowing simpler and more modular specification of the code generation.

As was discussed earlier, the front end specializes the AST nodes for names, so that different AST types will be used depending on if a name refers to, e.g., a variable or a method. Further semantic specialization is carried out in the back end in order to differentiate between accesses to different kinds of variables and methods. For example a general VarAccess node (representing variable access) can be specialized to ClassVariableAccess if its declaration is a class variable, and to InstanceVariableAccess if its declaration is an ordinary instance variable. Subsequent code generation can then be specified separately for the different cases, associated with the appropriate AST type.

In order to specialize these nodes, the abstract grammar needs to be extended with new subclasses. This is done simply by adding an additional abstract grammar module, as shown in Figure 7.

```
ast ClassVarAccess    : VarAccess;
ast InstanceVarAccess : VarAccess;
ast LocalVarAccess    : VarAccess;

ast ClassVarAssignExpr    : AssignExpr;
ast InstanceVarAssignExpr : AssignExpr;
ast LocalAssignExpr       : AssignExpr;

ast ClassMethodAccess    : MethodAccess;
ast InstanceMethodAccess : MethodAccess;
```

Figure 7: Additional AST classes that are used for semantic specialization in the preparation phase.

Figure 8 shows the rewrite rule that performs the specialization of variable access nodes. The API of the semantic AST is used for specifying the appropriate rewrite conditions. This is very simple: the reference attribute `decl` refers to the appropriate declaration node in the AST. That node in turn has attributes `isClassVariable`, etc., in order to find out what kind of variable it is.

```
specialize VarAccess {
  when (decl().isClassVariable())
  to new ClassVarAccess(idUse());

  when (decl().isInstanceVariable())
  to new InstanceVarAccess(idUse());

  when (decl().isLocalVariable())
  to new LocalVarAccess(idUse());
}
```

Figure 8: Rewrite rule for specializing variable accesses.

A similar semantic specialization is done for assignment statements. Depending on the assignment destination (local, instance, or class variable), different code will be generated. To prepare the AST, the assignment nodes are specialized to differ between these

kinds of storage locations. The rewrite rule is shown in Figure 9. The rewrite rule uses the API from the aspect specifying the variable access specialization, with methods like `isClassVarAccess`, etc., to specify the appropriate rewrites.

```
specialize AssignExpr {
  when ((dest().isClassVarAccess())
  to new ClassVarAssignExpr(dest(), source());
  when ((dest().isInstanceVarAccess())
  to new InstanceVarAssignExpr(dest(), source());
  when ((dest().isLocalVarAccess()
  to new LocalVarAssignExpr(dest(), source());
}
```

Figure 9: Rewrite rule for specializing assignment statements.

Method accesses (i.e., method calls) are specialized in the same way as the variable accesses in order to differentiate between different kinds of calls.

4.3 Optimizations on the source AST

By describing source level optimizations as aspect modules with rewrites, it should be possible to easily combine and select different optimizations. So far, we have only done experiments with some small local optimizations using this technique. An example is constant-expression evaluation (constant folding). Consider the following fragment of Java code:

```
final int a=1;
x=a+2;
```

The expression `a+2` operates only on constants and can therefore be evaluated at compile time and replaced by the expression `3`. The rewrite rule in Figure 10 specifies constant folding for the primitive types in Java. The rule makes use of the semantic AST API with attributes like `isConstant()`, `constantIntegerValue()`, etc. These attributes are already defined in the semantic AST API since they are needed to check that the cases in a switch statement have constant disjoint values.

```
rewrite Expr {
  when (isConstant() && type().isInteger())
  to IntegerLiteral new IntegerLiteral(constantIntegerValue());
  when (isConstant() && type().isLong())
  to LongLiteral new LongLiteral(constantLongValue());
  when (isConstant() && type().isFloat())
  to FloatLiteral new FloatLiteral(constantFloatValue());
  when (isConstant() && type().isDouble())
  to DoubleLiteral new DoubleLiteral(constantDoubleValue());
  when (isConstant() && type().isBoolean())
  to BooleanLiteral new BooleanLiteral(constantBooleanValue());
  when (isConstant() && type().isString())
  to StringLiteral new StringLiteral(constantStringValue());
}
```

Figure 10: Constant folding.

5. The Java2C backend

The development of our Java2C compiler is motivated by our research on languages for small embedded systems with hard real-time constraints [17]. In order to run Java on such systems, special support is needed for memory management, in particular garbage collection. Depending on the system constraints, different memory management schemes can be used, e.g., different kinds of usual (non-real-time) garbage collection, or garbage collection for systems with hard real-time constraints [9] or combinations thereof. In order to separate these constraints from the compilation, we have previously developed a Garbage Collection Interface (GCI) to the runtime system [11]. Before generating the C code, we have an additional preparation phase that rewrites the AST to make code generation to this interface simple.

5.1 Simplifying to a Java subset

In the runtime system for the Java2C compiler, the GCI is implemented as a set of macros for assignment and referencing variables, involving at most one indirect reference for each macro. To simplify the generation of code for this runtime system, we have defined a Java language subset [14] and added an additional preparation phase to the compiler that rewrites an arbitrary Java AST to this Java subset. For example, a complex expression will be broken down into a sequence of simple assignments, using temporary variables, similar to three-address code.

Expressions in Java may be rather complex, as for example in the code fragment with corresponding AST in Figure 11.

```
a.b().c().d = e().f.g();
```

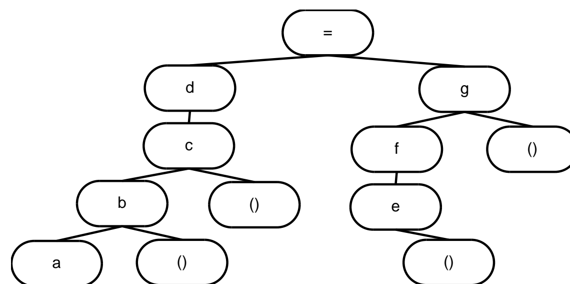

Figure 11: Java code fragment and corresponding AST.

The mapping from the full Java language specification [7] to the simpler subset can be conveniently described as a set of transformation on the AST, as will be shown in the following sections.

Names

Most of the simplifying transformations needed to perform on the AST are consequences of real-time memory management, see [11, 17] for details. Memory operations on references are performed via side-effect macros, only allowing one level of indirection at each step. It is therefore necessary to transform all Java expressions with more than one level

of indirection into lists of statements each containing at most one level of indirection. For example, the Java statement

```
a.b = c;
```

contains one indirection, whereas

```
a.b = c.d;
```

has two indirections, and must therefore be transformed into something like the following.

```
tmp_1 = c.d;
a.b = tmp_1;
```

Figure 12 shows the corresponding AST transformation.

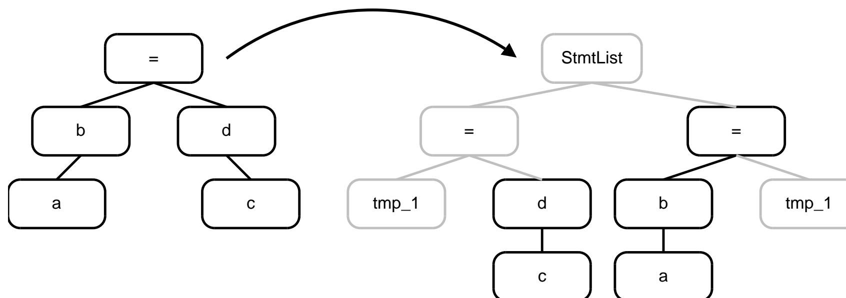

Figure 12: Simplifying names by means of an AST transformation. Grey nodes and egdes are inserted as result of the transformation.

The situation becomes a little more complicated with method calls, since arguments passed in the call may contain arbitrarily complex expressions. For example, for the following method call

```
a(b(c()),d());
```

the evaluation order must be

```
c(), b(), d(), a()
```

A suitable simplifying transformation for the above expression, to meet the indirection requirements, could then be expressed as AST transformations or as code as in Figure 13.

Unary Expressions

Unary expressions which as a side effect change the value of the operand, may need to be simplified in order to meet indirection requirements. For example, the simple statements

```
a++;
b.a++;
```

should be read as

```
a = a+1;
b.a = b.a+1;
```

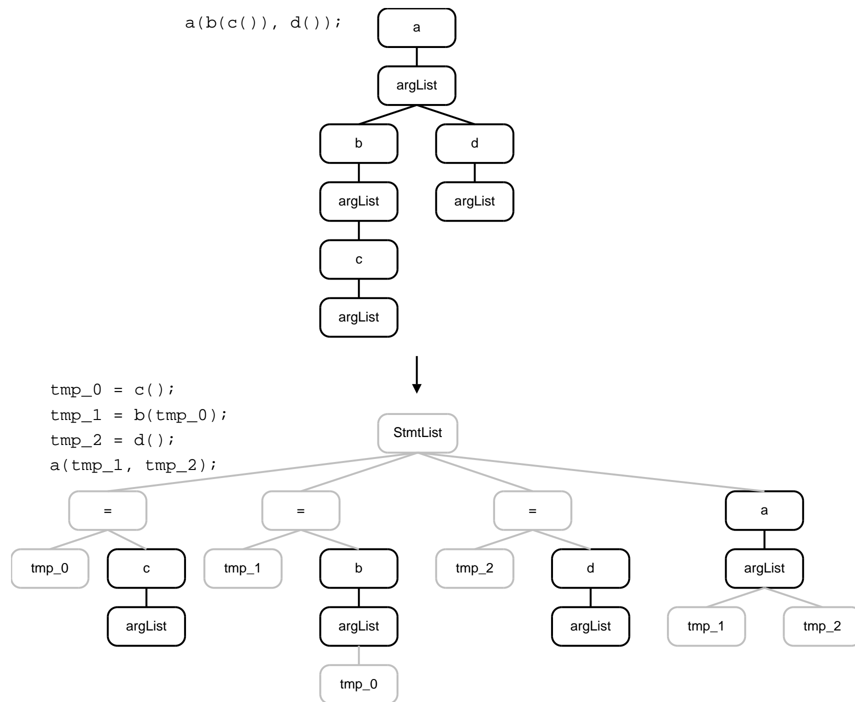

Figure 13: Simplifying a complex method call. Grey nodes and egdes are inserted as result of the transformation.

which poses no problem in the first statement, with zero indirections, but the latter statement now has two indirections and must be simplified to something like

```
tmp_0 = b.a;
b.a = tmp_0+1;
```

However, things get more complicated as such unary expressions may be used inside other expressions. For example, the seemingly simple statement

```
a[k.i++] = b[++k.i];
```

has a non-trivial evaluation order. A simplification of the above statement which meet indirection requirements can be written as:

```
tmp_0 = k.i;
++tmp_0;
k.i = tmp_0;
tmp_1 = b[tmp_0];

tmp_2 = k.i;
k.i = tmp_2 + 1;

a[tmp_2] = tmp_1;
```

Note that the evaluation of a *PreIncrement* expression differs from the evaluation of a *PostIncrement* expression to maintain semantic correctness.

Control-Flow Statements

Expressions that are part of control-flow statements require special care in the simplification process, so as not to alter the semantics of the program. The for statement is the most complicated loop statement in Java, and serves well to illustrate these issues.

A Java for-statement, as defined by the abstract grammar for Java, is represented by the AST subtree in Figure 14. As defined in the Java language specification [7], the

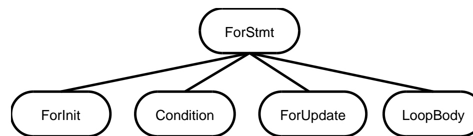

Figure 14: Subtree representing a for-statement.

ForInit and *ForUpdate* nodes may hold a list of StatementExpressions, and the *ForInit* may alternatively contain a local VariableDeclaration. An example of a complex for-statement could be

```
for(a=b(c(1),d),e=f[g()];a[h++]<i;a=b(c(h++),d))
  // ... loop body ...
```

The solution to simplifying complex for-statements is to create while-statements by moving the *ForInit* ahead of the statement and move the *ForUpdate* last inside the *Stmt* node (which has been transformed to a *Block*). A simplified for-statement subtree is shown in Figure 15. The resulting code after simplifying the example for-statement above

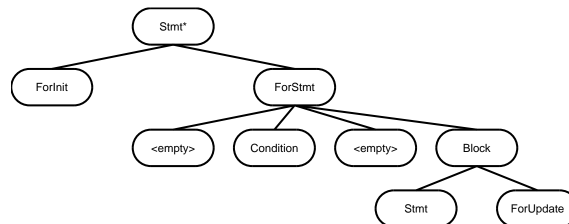

Figure 15: Subtree representing a simplified for-statement.

would then be

```
tmp_0 = c(1);
a = b(tmp_0,d);
tmp_1 = g();
e = f[tmp_1];

tmp_2 = a[h++];
for ( ; tmp_2<i ; ) {
  // Loop body

  tmp_3 = c(h++);
  a = b(tmp_3,d);

  tmp_2 = a[h++];
}
```

Similar techniques are used to simplify the other Java control-flow statements.

5.2 Code Generation

When the AST has been prepared to fulfill the indirection requirement, the task of generating the C code becomes straight-forward.

For each used class in the AST, a C header file is generated. The header file contains the type declarations of the object model, i.e., structs for representing the class, its instances, and its virtual method tables (vtables). The actual code is generated into a single C implementation file that contains the implementations of all constructors and methods, as well as class initialization code. Handwritten C code, such as native method implementations, can include appropriate class header files.

The process of using the Java2C compiler to compile a Java program to an executable machine code image is sketched in Figure 16.

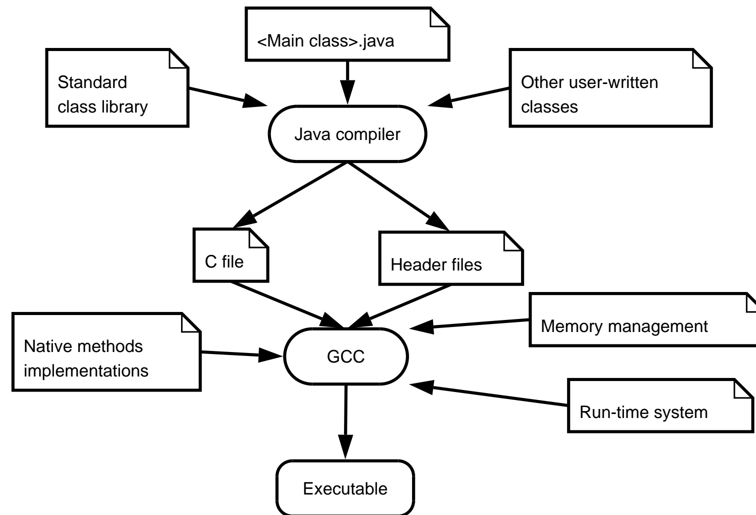

Figure 16: Flowchart of compilation process.

5.3 The Java2Bytecode backend

In addition to the Java2C backend we are developing Java2Bytecode, a compiler for generating ordinary Java bytecode. This compiler is motivated primarily by a research project in pervasive systems and programming applications for communicating devices [18]. The Java2Bytecode compiler is only a first step, and will be continued with the development of alternative bytecode generation and extended source languages for pervasive systems. This effort is still in an experimental stage: we have some tiny programs that compile and run, but the complete Java bytecode set is not yet implemented. The Java2Bytecode backend uses the same front end aspect modules and the same general preparation phase modules as the Java2C compiler.

In the Java2Bytecode backend, an Intermediate Representation AST (IR AST) is generated from the source AST, as shown earlier in Fig. 3. The full Java bytecode instruction set consists of 201 instructions. Of these, 59 are optimized variants of the remaining 142 basic instructions. In the generation of the IR AST from the source AST, only the basic instruction set is generated. Because of the preparation phase on the source AST, this generation is completely straight forward. In an additional optimization phase on the IR

AST, rewrites are used for replacing basic instructions by optimized instructions when possible. This architecture makes each of the phases very simple.

As a concrete example, the basic set contains an instruction `ldc_w i` which pushes a constant value on the stack by accessing its value from the constant pool (an array of constants in the class file), using index `i`. If the constant is small, this instruction can be replaced by optimized instructions like `iconst_2` which pushes the constant 2, or `bipush v` which pushes the immediate integer byte value `v`. Usually, the optimized instructions reduce the code size. E.g., the basic instruction `ldc_w i` is three bytes (one for the instruction code and two for the 2-byte index). The optimized instruction `bipush v` is two bytes, and `iconst_2` is a single byte. On an interpreting JVM, the optimizing instructions will usually also run faster.

6. Evaluation

6.1 Compiler architecture

To evaluate our compiler architecture we compare our Java compiler to two compilers and one front-end; javac included in j2sdk 1.4.2, the Java compiler in Eclipse 2.1.3, and the polyglot 1.3 extensible Java frontend. All four compilers implement version 1.4 of the Java programming language. We first give a brief description of the used compiler architectures and then describe the benefit of our architecture from a modularization and extensibility perspective.

6.1.1 J2SDK 1.4.2 javac

Javac builds an AST modeled using the interpreter pattern and consists of approximately 40 node types. To model the entire language with only 40 node types some nodes are fairly abstract and contain constants to describe the actual language element. The nodes only model the structure of the language and contextual information is passed around as parameters to the methods accessed during tree traversal.

Four main visitors are used to perform various computations on the AST; attribution, dataflow analysis, flattening, and code generation. Attribution performs the main contextual computations and calls utility functions in separate modules to add implict tree nodes, build a symbol table, resolve syntactically ambiguous names, constant folding, and type checking. The various activities are implemented in separate modules but their invocation is scheduled imperatively in the attribution visitor. The dataflow analysis visitor handles exceptions, definite assignment, and reachability computations. These computations are mixed in one module. A full Java AST with nested classes is then transformed to a flat Java AST using a tree translator. This simplifies code generation in that the tree structure better fits the target bytecode structure.

6.1.2 Eclipse Java compiler

The Java compiler in Eclipse builds an AST similar to javac but has approximately twice as many node types. Visitors are used but to a lesser extent than in javac: some computations are placed directly in the node types and thus scattered over the type hierarchy, e.g. code generation, type evaluation, and generic data flow computations. The generic data flow analysis computations are then, however, used by visitors that modularize computations such as exception targets, definite assingment and various reachability computations.

The code generation is performed on an AST supporting nested types. This, in combination with nodes that represent multiple semantically different language elements, re-

sults in quite a few case blocks in the code generation to choose the appropriate code generation.

6.1.3 Polyglot

Polyglot differs from the other compilers in that it is merely an extensible front-end that performs semantic analysis and does not emit bytecode but merely Java source. This is mainly used to experiment with source code analysis and language extensions that can be transformed into plain Java. The AST is constructed from approximately 95 node types. Rigorous use of interfaces and factories makes it easy to extend the language including new language elements, changed type system, modified scoping rules etc. The various computations in the compiler are separated using a set of visitors. The tree is also transformed in several steps to simplify later computations, e.g., to add implicit nodes and to resolve contextually ambiguous names.

6.2 Modularization techniques

The architectures in the described compilers, including our own, are similar in that they are based on an object-oriented AST modeled using the interpreter pattern. Visitors are used to separate the various computations on the tree structure except for our solution that is based on AOP techniques and open classes. Another difference is that our approach is declarative and need thus not order the computations in a list of traversals over the AST using visitors. The declarative attributes are computed on-demand according to their dependences.

All solutions rewrite the AST, to some extent, during compilation to simplify later computations, e.g. add implicit nodes such as default constructors, and translate nested Java classes to flat Java classes. Polyglot and our compiler also rewrites the tree to reflect the semantic meaning of names, e.g. variable name, field name, or type name. Our compiler takes this approach one step further and tries to create a semantic AST where each node maps to a single semantic language entity. This results in an AST class hierarchy of over 200 node types that in combindation with open classes enables excellent modularization through object-oriented techniques such as virtuality and overriding. Another difference is that our solution is declarative and need thus not be scheduled explicitly but is demand driven and transparent to other computations.

6.3 Extensibility

The visitor pattern is a common technique to add computations on an existing class hierarchy in a modular fashion. All the described compilers can thus easily be extended with a new computation module by adding a new visitor. The same is, however, not true for new language elements. It is not easy to both extend the data model and the computations performed on it in a modular fashion as noted in amongst others [20].

Polyglot is designed to allow both added computations and language elements using a technqiue based on extensive use of interfaces and factories. Our compiler does not need a similar framework, but can easily be extended directly due to its declarative nature where equations and rewrites can be combined freely and placed in modules as desired.

6.4 Size and Speed

We have compared the size of the implementation for the Java compilers. While this comparison is not completely fair since they are somewhat different, e.g. Polyglot has no backend but includes an extensible framework, and the Eclipse compiler is somewhat

incremental, it gives some understanding of how much code that is needed for a Java compiler. We took the source folders and removed some utility code not concerned with the compilers and gzip:ed the source code. This technique was chosen instead of lines of code to remove differences in formatting. The specifications from smallest to largest:

- our compiler 150 kbyte
- javac 225 kbyte
- polyglot 295 kbyte
- Eclipse Java compiler 460 kbyte

Our compiler development has focused on modularity, and so far we have done little effort to improve the compiler speed. We have, however, run some initial benchmarks to compare compilation speed of our compiler to the javac compiler. Compiling the java.lang, java.util, and java.io packages in j2sdk1.4.2 consisting of roughly 100.000 lines of code takes approximately 24 seconds which is four times slower than javac. This comparison shows that the ReRAG-based compiler is only a few times slower than javac and can be used for large-scale practical applications.

7. Future Work

In our future work we are addressing both the development of ReRAGs as such and the development of the real-time Java2C compiler and compilers for bytecode.

Within ReRAGs we are looking at optimizing the ReRAG evaluation. This will allow us to bring down the compilation times for our Java compilers. We are also working on merging Circular Reference Attributed Grammars (CRAGs) [16] with ReRAGs in order to support cyclic dependencies and iterative fixed-point evaluation of attributes and possibly also rewrites. Many code optimizations make use of such iterative algorithms, and with this support in ReRAGs we hope to be able to specify many of these optimizations in a modular and declarative way.

For the Java2C compiler, the focus is on using Java for small embedded systems with hard real-time constraints. Generating code that will function properly in all possible executions will result in conservative code. In particular, the read and write barriers used for real-time GC algorithms are quite expensive. We are therefore looking at ways of enhancing the performance for specific critical parts of the program. E.g., increasing the performance of high priority regulator threads in order to be able to increase sampling rate. We are also interested in achieving improved performance in general, using some of the many well-known general optimizations techniques for OO languages like [1, 5, 21]. E.g., method call de-virtualization and class in-lining should be very profitable in the closed world applications we are looking at, in particular if applied in a way that does not increase the memory consumption too much. We plan to specify some of these optimizations using ReRAGs. The modular way these optimizations can be introduced will allow us to easily experiment with different combinations of optimizations, to investigate their effect on the real-time applications we are targeting. A related issue is finding techniques for tuning the GC behavior by analyzing the source code, as an alternative to relying on experimental dynamic measurements. Initial work in this direction has been carried out by Persson [19] on implementing worst-case memory usage and worst-case execution time (WCET) analysis on Java.

For the Java2Bytecode compiler we will proceed with completing implementation of the backend and with adding more bytecode optimizations. This will allow us to evaluate our approach in a more thorough way and compare it to others, e.g., other Java compilers, and other optimization and backend frameworks. We also plan to experiment with implementing other bytecode instruction sets. It would be interesting to study how well the different aspect modules can be reused.

8. Conclusions

We have presented our experiences from generating Java compilers using ReRAGs. So far, our experience is very positive. We have been able to successfully divide the compilers into modules that separate the different concerns and make each module simple to write and understand on its own:

- The preparation modules contain a few simple rewrites that make the AST streamlined for code generation.
- The optimization modules are optional and more optimization modules can be added later without affecting other parts of the backend.
- The generation modules consist of very simple and straight-forward code.

The rewrites in the preparation and optimization modules were easy to write due to the information available in the attributed AST, where the attribute API from the semantic analysis could be used to specify the context-dependent rewrite conditions in an easy way. The declarative approach provides automatic rewrite scheduling and thereby allows the different modules to be written independently of each other.

From our experience of implementing the compilers, we have drawn the following conclusions:

- The use of ReRAGs allows compact compiler implementation. The size of our ReRAG specifications is significantly lower than the size of the Java code of handwritten compilers based on visitors.
- The use of ReRAGs allows the compiler to be better modularized than handwritten Java code. This is because of the aspect-oriented and declarative approach, which allows the compiler implementor to group attributes, equations, and rewrites that address the same concern, without having to take evaluation order into account, and without having to follow the class hierarchy.
- The use of node specialization and rewriting to simpler forms allows more straightforward code generation.
- Our generated compilers are around four times slower than compilers handwritten in Java, like javac. We find this sufficient for practical use when modularity and speed of compiler development is of primary importance. We are also working on improving this performance.

Acknowledgments

This work has been carried out within the LUCAS[1] applied software research center with financing from the Swedish Agency for Innovation Systems (VINNOVA), the Swedish Foundation for Strategic Research (SSF), and the Palcom integrated project in the EU 6th Framework Programme.

[1] `http://www.lucas.lth.se`

References

[1] Matthew Arnold, Michael Hind, and Barbara G. Ryder. An empirical study of selective optimization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing. (LCPC '00)*, August 2000.

[2] Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In *Proceedings of ECOOP 2004: 18 European Conference on Object-Oriented Programming*, 2004.

[3] Torbjörn Ekman. A casy study of separation of concerns in compiler construction using JastAdd II. In *Proceedings of the Third AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACI4IS)*, 2004.

[4] Torbjörn Ekman. *Rewritable Reference Attribute Grammars*. Licentiate thesis LU-CS-LIC:2004-3, Lund Institute of Technology, Lund University, Sweden, June 2004.

[5] Robert Fitzgerald, Todd B. Knoblock, Erika Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for java. Technical report, Microsoft Research, 1 Microsoft Way Redmond, WA 98052, June 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1995.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1st edition, August 1996.

[8] Görel Hedin. Reference attribute grammars. In *Informatica (Slovenia)*, 24(3), 2000.

[9] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, July 1998.

[10] Scott E. Hudson, Frank Flannery, C. Scott Anaian, Dan Wang, and Andrew W. Appel. Cup parser generator for java. *http://www.cs.princeton.edu/ appel/modern/java/CUP/*, 1999.

[11] Anders Ive, Anders Blomdell, Torbjörn Ekman, Roger Henriksson, Anders Nilsson, Klas Nilsson, and Sven Gestegård-Robertz. Garbage collector interface. In *Proceedings of NWPER 2002*, August 2002.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001. *http://eclipse.org/aspectj/*.

[13] Donald E. Knuth. Semantics of context–free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp.95–96 (March 1971).

[14] Fransisco Menjíbar. Portable Java compilation for Real-Time Systems. Master's thesis, Dep. of Computer Science Lund University, September 2003.

[15] Java-cc parser generator. Metamata Inc. *http://www.metamata.com*.

[16] Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. In *Proceedings of the Third Workshop on Language Descriptions, Tools and Applications (LDTA 2003)*, Warsaw, Poland, April 2003.

[17] Anders Nilsson. Compiling Java for Real-Time Systems. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, May 2004.

[18] Palpable computing. *http://www.palcom.dk/*, 2004.

[19] Patrik Persson. Predicting time and memory demands of object-oriented programs. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, April 2000.

[20] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. pages 13–23, 1994.

[21] Frank Tip, Peter F. Sweeney, and Chris Laffra. Extracting library-based java applications. *Communications of the ACM*, 46(8):35–40, August 2003.