# Automated Selective Caching for Reference Attribute Grammars

Emma Söderberg and Görel Hedin

Department of Computer Science, Lund University, Sweden
{emma.soderberg,gorel.hedin}@cs.lth.se

**Abstract.** Reference attribute grammars (RAGs) can be used to express semantics as super-imposed graphs on top of abstract syntax trees (ASTs). A RAG-based AST can be used as the in-memory model providing semantic information for software language tools such as compilers, refactoring tools, and meta-modeling tools. RAG performance is based on dynamic attribute evaluation with caching. Caching all attributes gives optimal performance in the sense that each attribute is evaluated at most once. However, performance can be further improved by a selective caching strategy, avoiding caching overhead where it does not pay off. In this paper we present a profiling-based technique for automatically finding a good cache configuration. The technique has been evaluated on a generated Java compiler, compiling programs from the Jacks test suite and the DaCapo benchmark suite.

## 1 Introduction

Reference attribute grammars (RAGs) [11] provide a means for describing semantics as super-imposed graphs on top of an abstract syntax tree (AST) using *reference attributes*. Reference attributes are defined by functions and may have values referring to distant nodes in the AST. RAGs have been shown useful for the generation of many different software language tools, including Java compilers [31,9], Java extensions [13,14,22], domain-specific language tools [16,2], refactoring tools [24], and meta-modeling tools [7]. Furthermore, they are being used in an increasing number of meta-compilation systems [12,30,25,18].

RAG evaluation is based on a dynamic algorithm where attributes are evaluated on demand, and their values are cached (memoized) for obtaining optimal performance [15]. Caching all attributes gives optimal performance in the sense that each attribute is evaluated at most once. However, caching has a cost in both compilation time and memory consumption, and caching does not pay off in practice for all attributes. Performance can therefore be improved by *selective caching*, caching only a subset of all attributes, using a *cache configuration*. But determining a good cache configuration is not easy to do manually. It requires a good understanding of how the underlying attribute evaluator works, and a lot of experience is needed to understand how different input programs can affect the caching inside the generated language tool. Ideally, the language engineer should not need to worry about this, but let the system compute the configuration automatically.

In this paper we present a profiling-based approach for automatically computing a cache configuration. The approach has been evaluated experimentally on a generated compiler for Java [9], implemented using JastAdd [12], a meta-compilation system based on RAGs. We have profiled this compiler using programs from Jacks (a compiler test suite for Java) [28] and DaCapo (a benchmark suite for Java) [4]. Our evaluation shows that it is possible to obtain an average compilation speed-up of 20% while only using the profiling results from one application with a fairly low attribute coverage of 67%. The contributions of this paper include the following:

- A profiling-based approach for automatic selective caching of RAGs.
- An implementation of the approach integrated with the JastAdd meta-compilation system.
- An evaluation of the approach, comparing it both to full caching (caching all attributes) and to an expert cache configuration (produced manually).

The rest of this paper is structured as follows. Section 2 gives background on reference attribute grammars and their evaluation, explaining the JastAdd caching scheme in particular. Section 3 introduces the concept of an AIG, an attribute instance graph with call information, used as the basis for the caching analysis. Section 4 introduces our technique for computing a cache configuration. Section 5 presents an experimental evaluation of the approach. Section 6 discusses related work, and Section 7 gives a conclusion along with future work.

## 2 Reference Attribute Grammars

Reference Attribute Grammars (RAGs) [11], extend Knuth-style attribute grammars [19] by allowing attributes to be references to nodes in the abstract syntax tree (AST). This is a powerful notion because it allows the nodes in an AST to be connected into the graphs needed for compilation. For example, reference attributes can be used to build a type graph connecting subclasses to superclasses [8], or a control-flow graph between statements in a method [20]. Similar extensions to attribute grammars include Poetzsch-Heffter's occurrence algebras [21] and Boyland's remote attribute grammars [6].

In attribute grammars, attributes are defined by equations in such a way that for any attribute instance in any possible AST, there is exactly one equation defining its value. The equations can be viewed as side-effect-free functions which make use of constants and of other attribute values.

In RAGs, it is allowed for an equation to define an attribute by following a reference attribute and accessing its attributes. For example, suppose node $n_1$ has attributes $a$ and $b$, where $b$ is a reference to a node $n_2$, and that $n_2$ has an attribute $c$. Then $a$ can be defined by an equation as follows:

$$a = b.c$$

For Knuth-style attribute grammars, dependencies are restricted to attributes in parents or children. But the use of references gives rise to non-local dependencies, i.e., dependencies that are independent of the AST hierarchy: $a$ will be dependent on $b$ and $c$, where the dependency on $b$ is local, but the dependency on $c$ is non-local: the node $n_2$ referred to by $b$ could be anywhere in the AST. The resulting attribute dependency graph cannot be computed without actually evaluating the reference attributes, and it is therefore difficult to statically precompute evaluation orders based on the grammar alone. Instead, evaluation of RAGs is done using a simple but general dynamic evaluation approach, originally developed for Knuth-style attribute grammars, see [15]. In this approach, attribute access is replaced by a recursive call which evaluates the equation defining the attribute. To speed up the evaluation, the evaluation results can be cached (memoized) in order to avoid evaluating the equation for a given attribute instance more than once. Caching all attributes results in optimal evaluation in that each attribute instance is evaluated at most once. Because this evaluation scheme does not require any pre-computed analysis of the attribute dependencies, it works also in the presence of reference attributes.

Caching is necessary to get practical compiler performance for other than the tiniest input programs. But caching also implies an overhead. Compared to caching all attributes, *selective caching* may improve performance, both concerning time and memory.

## 2.1 The JastAdd Caching Scheme

In JastAdd, the dynamic evaluation scheme is implemented in Java, making use of an object-oriented class hierarchy to represent the abstract grammar. Attributes are implemented by method declarations, equations by method implementations, and attribute accesses by method calls. Caching is decided per attribute declaration, and cached attribute values are stored in the AST nodes using two Java fields: one field is a flag keeping track of if the value has been cached yet, and another field holds the value. Figure 1 shows the implementation of the equation $a = b.c$, both in a non-cached and a cached version. It is assumed that $a$ is of type $A$. The example shows the implementation of a so called *synthesized attribute*, i.e., an attribute defined by an equation in the node itself. The implementation of a so called *inherited attribute*, defined by an equation in an ancestor node, is slightly more involved, but uses the same technique for caching. The implementation in Figure 1 is also simplified as compared to the actual implementation in JastAdd which takes into account, for example, circularity checking. These differences are, however, irrelevant to the caching problem.

This caching scheme gives a low overhead for attribute accesses: a simple test on a flag. On the other hand, the caching pays off only after at least one attribute instance has been accessed at least twice. Depending on the cost of the value computation, more accesses than that might be needed for the scheme to pay off.

JastAdd allows attributes to have parameters. A *parameterized attribute* has an unbounded number of values, one for each possible combination of parameter values. To cache accessed values, the flag and value fields are replaced by a map where the actual parameter combination is looked up, and the cached values are stored. This is a substantially more costly caching scheme, both for accessing attributes and for updating the cache, and more accesses per parameter combination will be needed to make it pay off.

**Non-cached version**:

```
class Node {
  A a() {
    return b().c();
  }
}
```

**Cached version**:

```
class Node {
  boolean a_cached = false;
  A a_value;
  A a() {
    if (! a_cached) {
      a_value = b().c();
      a_cached = true;
    }
    return a_value;
  }
}
```

**Fig. 1.** Caching scheme for non-parameterized attributes

## 3 Attribute Instance Graphs

In order to decide which attributes that may pay off to cache, we build a graph that captures the attribute dependencies in an AST. This graph can be built by instrumenting the compiler to record all attribute accesses during a compilation. By analyzing such graphs for representative input programs, we would like to identify a number of attributes that are likely to improve the compilation performance if left uncached. We define the *attribute instance graph* (AIG) to be a directed graph with one vertex per attribute instance in the AST. The AIG has an edge $(a_1, a_2)$ if, during the evaluation of $a_1$, there is a direct call to $a_2$, i.e., indirect calls via other attributes do not give rise to edges. Each edge is labeled with a *call count* that represents the number of calls. This count will usually be 1, but in an equation like $c = d + d$, the count on the edge $(c, d)$ will be 2, since $d$ is called twice to compute $c$.

The main program of the compiler is modeled by an artificial vertex `main`, with edges to all the attribute instances it calls. This may be many or few calls, depending on how the main program is written.

To handle parameterized attributes, we represent each accessed combination of parameter values for an attribute instance by a vertex. For example, the evaluation of the equation $d = e(3) + e(4) + e(4)$ will give rise to two vertices for $e$, one for $e(3)$ and one for $e(4)$. The edges are, as before, labeled by the call counts, so the edge $(d, e(3))$ is labeled by 1, and the edge $(d, e(4))$ by 2, since it is called twice. Figure 2 shows an example AIG for the following equations:

$$\begin{aligned} a &= b.c \\ c &= d + d \\ d &= e(3) + e(4) + e(4) \end{aligned}$$

and where it is assumed that $a$ is called once from the main program.

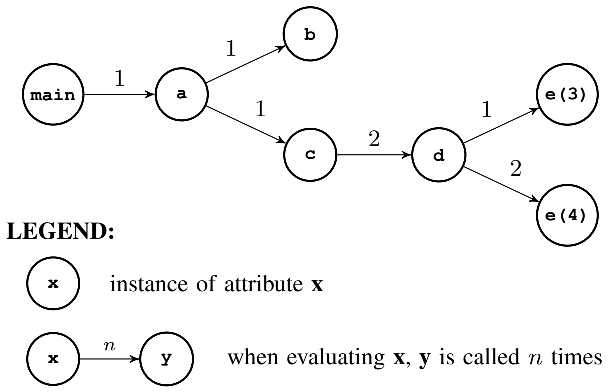

**Fig. 2.** Example AIG

## 3.1 An Example Grammar

Figure 3 shows parts of a typical JastAdd grammar for name and type analysis. The abstract grammar rules correspond to a class hierarchy. For example, `Use` (representing a use of an identifier) is a subclass of `Expr`. The first attribution rule:

```
syn Type Expr.type();
```

declares a synthesized attribute of type `Type`, declared in `Expr` and of the name `type`. All nodes of class `Expr` and its subclasses will have an instance of this attribute.

```
abstract Expr;
Use : Expr ::= ...;
Literal : Expr ::= ...;
AddExpr : Expr ::=
    e1:Expr e2:Expr;

Decl ::= Type ... ;

abstract Type;
Integer : Type;
Unknown : Type;

...
```

```
syn Type Expr.type();
syn Type Decl.type() = ...;

eq Literal.type() =
  stdTypes().integer();
eq Use.type() = decl().type();
eq AddExpr.type() =
  (left.type().sameAs(right.type()) ?
  left.type() : stdTypes.unknown();

syn Decl Use.decl() = lookup(...);
inh Decl Use.lookup(String name);
inh Type Expr.stdTypes();

syn boolean Type.sameAs(Type t) = ...;
...
```

**Fig. 3.** Example JastAdd attribute grammar

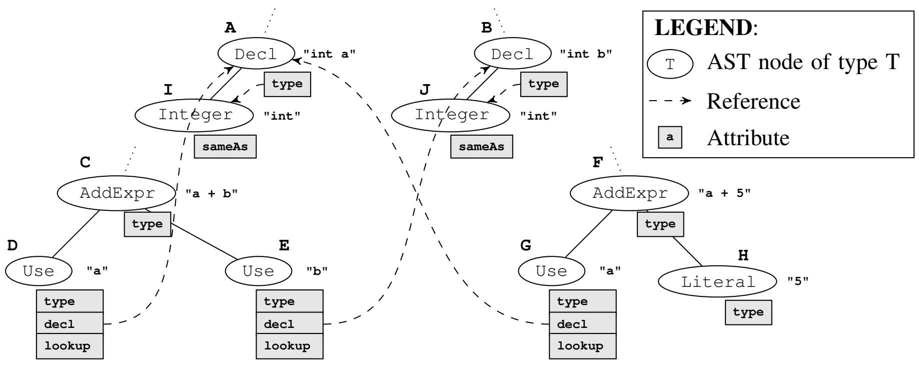

**Fig. 4.** An example attributed AST

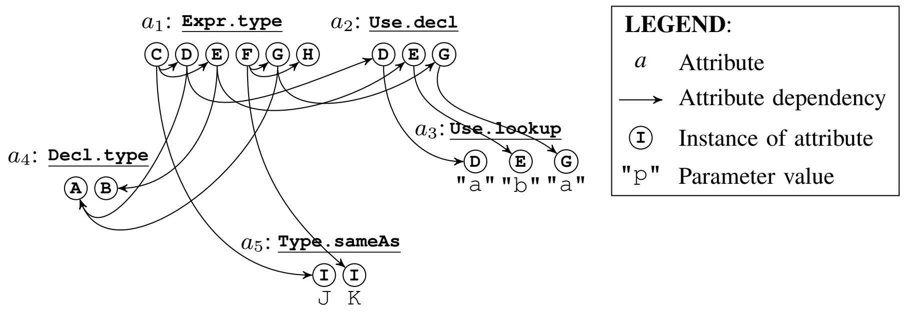

**Fig. 5.** Parts of the AIG for the example

Different equations are given for it in the different subclasses of `Expr`. For example, the equation

```
eq Use.type() = decl().type();
```

says that for a `Use` node, the value of `type` is defined to be `decl().type()`. The attribute `decl()` is another attribute in the `Use` node, referring to the appropriate declaration node, possibly far away from the `Use` node in the AST. The `decl()` attribute is in turn defined using a parameterized attribute `lookup`, also in the `Use` node. The `lookup` attribute is an inherited attribute, and the equation for it is in an ancestor node of the `Use` node (not shown in the grammar). For more information on name and type analysis in RAGs, see [8].

Figure 4 shows parts of an attributed AST for the grammar in Figure 3. The example program contains two declarations: `"int a"` and `"int b"`, and two add expressions: `"a + b"` and `"a + 5"`. For the `decl` attributes of `Use` nodes, the reference values are shown as arrows pointing to the appropriate `Decl` node. Similarly, the `type` attributes of `Decl` nodes have arrows pointing to the appropriate `Type` node. The nodes have been labeled `A`, `B`, and so on, for future reference.

Figure 5 shows parts of the AIG for this example. In the AIG we have grouped together all instances of a particular attribute declaration, and labeled each attribute instance with the node to which it belongs. For instance, since the node `D` has the three attributes (`type`, `decl`, and `lookup`), there are three vertices labeled `D` in the AIG. For parameterized attribute instances, there is one vertex per actual parameter combination, and their values are shown under the vertex. For instance, the `sameAs` attribute for `I` is called with two different parameters: `J` and `K`, giving rise to two vertices. (`K` is a node representing integer literal types and is not shown in Figure 4.) All call counts in the AIG are 1 and have therefore been omitted.

## 4 Computing a Cache Configuration

Our goal is to automatically compute a good cache configuration for a RAG specification. A cache configuration is simply the set of attributes configured to be cached. Among the different attribute kinds, there are some that will always be cached, due to properties of the kind. For example, circular attributes [10], which may depend on themselves, and non-terminal attributes (NTAs) [29], which may have ASTs as values. There is no cache decision to make for these attributes, i.e., they are *unconfigurable*. We let PRE denote the set of unconfigurable attributes. Since the attributes in the PRE set are always cached, we exclude them from remaining definitions in this paper. We let ALL denote the remaining set of *configurable* attributes. This ALL set can further be divided into two disjoint sets PARAM and NONPARAM, for parameterized and non-parameterized attributes respectively. For the rest of this paper we will refer to configurable attributes when we write attributes.

As a basis for our computation, we do profiling runs of the compiler on a set of test programs, producing the AIG for each program. These runs are done with all attributes cached, allowing us to use reasonably large test programs, and making it easy to compute the AIG which reflects the theoretically optimal evaluation with each attribute instance evaluated at most once. We will refer to these test programs as the *profiling input* denoted by the set P. Further, a certain profiling input ($p \in$ P) will, depending on its structure, require that a certain number of attributes are evaluated. We call this set of attributes the $\text{USED}_p$ set. However, it cannot be assumed that a single profiling input uses all attributes. We define the set of unused attributes for a profiling input $p$ as follows:

$$\text{UNUSED}_p = \text{ALL} \setminus \text{USED}_p \tag{1}$$

### 4.1 The ONE Set

The calls label on the edges in the AIG reflects the number of attribute calls in a fully cached configuration. To find out if a certain attribute is worth uncaching, we define $\text{extra\_evals}(a_i)$, i.e., the number of extra evaluations of the attribute instance $a_i$ that will be done if the attribute $a$ is not cached:

$$\text{extra\_evals}(a_i) = \begin{cases} \text{calls}(a_i) - 1, & \text{if } a \in \text{NONPARAM} \\ \sum_{c \in \text{params}(a_i)} (\text{calls}(c) - 1), & \text{if } a \in \text{PARAM} \end{cases} \tag{2}$$

where $\mathrm{params}(a_i)$ is the set of vertices in the AIG representing different parameter combinations for the parameterized attribute instance $a_i$. The number of extra evaluations is a measure of what is lost by not caching an attribute. The total number of extra evaluations for an attribute $a$ is simply the sum of the extra evaluations of all its instances:

$$\mathrm{extra\_evals}(a) = \sum_{a_i \in \mathrm{I}_{\mathrm{called}}(a)} \mathrm{extra\_evals}(a_i); \tag{3}$$

where $\mathrm{I}_{\mathrm{called}}(a)$ is the set of attribute instances of $a$ that are called at least once. Of particular interest is the set of attributes for which all instances are called at most once. These should be good candidates to leave uncached since they do not incur any extra evaluations for a certain profiling input ($p$). We call this set the $\mathrm{ONE}_p$ set, and for a profiling input $p$ it is constructed as follows:

$$\mathrm{ONE}_p = \{a \in \mathrm{USED}_p | \mathrm{extra\_evals}(a) = 0\} \tag{4}$$

The $\mathrm{USED}_p \setminus \mathrm{ONE}_p$ set contains the remaining attributes in the AIG, i.e., the attributes which may gain from being cached, depending on the cost of their evaluation.

## 4.2 Selecting a Good Profiling Input

To obtain a good cache configuration, it is desirable to use profiling input that is realistic in its attribute usage, and that has a high *attribute coverage*, i.e., as large a $\mathrm{USED}_p$ set as possible. We define the attribute coverage (in percent) for a profiling input, $p \in \mathrm{P}$, as follows:

$$\mathrm{coverage}(p) = (|\mathrm{USED}_p|/|\mathrm{ALL}|) * 100 \tag{5}$$

Furthermore, for tools used in an interactive setting with continuous compilation of potentially erroneous input, it is important to also take incorrect programs into account. To help fulfill these demands, different profiling inputs can be combined. In particular, a compilation test suite may give high attribute coverage and test both correct and erroneous programs. But test suites might contain many small programs that do not use the attributes in a realistic way. In particular, attributes which most likely should be in the $\mathrm{USED}_p - \mathrm{ONE}_p$ set for an average application may end up in the $\mathrm{ONE}_p$ sets of the test suite programs because these are small. By combining the test suite with a large real program, better results may be obtained. Still, even with many applications and a full test suite, it may be hard to get full coverage. For example, there may be semantic checks connected to uncommon language constructs and, hence, attributes rarely used.

## 4.3 Choosing a Cache Configuration

In constructing a good cache configuration we want to consider the $\mathrm{USED}_p$, $\mathrm{UNUSED}_p$, $\mathrm{ONE}_p$ and $\mathrm{ALL}$ sets. From these sets we can experiment with two interesting configurations:

$$\mathrm{ALLONE}_p = \mathrm{ALL} \setminus \mathrm{ONE}_p \tag{6}$$

$$\mathrm{USEDONE}_p = \mathrm{USED}_p \setminus \mathrm{ONE}_p \tag{7}$$

Presumably, the first configuration, which includes the $\textsc{unused}_p$ set, will provide robustness for cases where the profiling input is insufficient, i.e., the $\textsc{used}_p$ set is too small. In contrast, the second configuration may provide better performance in that it uses less memory for cases where the profiling input is sufficient.

### 4.4 Combining Cache Configurations

In order to combine the results of several profiling inputs, for example, A, B and C in P, we need to consider each of the resulting sets $\textsc{used}_p$ and $\textsc{one}_p$. One attribute might be used in the compilation of program A but not in the compilation of program B. If an attribute is used in both B and C, it might belong to $\textsc{one}_B$, but not to $\textsc{one}_C$, and so on. We want to know which attributes that end up in a total $\textsc{one}_\mathrm{P}$ set for all profiling inputs ($p \in$ P), i.e., the attributes that are used by at least one profiling input, but that, if they are used by a particular profiling input, they are in its $\textsc{one}_p$ set. More precisely:

$$\textsc{one}_\mathrm{P} = \bigcup_{p \in \mathrm{P}} \textsc{used}_p \setminus \bigcup_{p \in \mathrm{P}} (\textsc{used}_p \setminus \textsc{one}_p) \tag{8}$$

These attributes should be good candidates to be left uncached. By including or excluding the $\textsc{unused}_\mathrm{P}$ set, we can now construct the following combined cache configuration, for a profiling input set P, in analogy to Definition 6 and Definition 7:

$$\textsc{AllOne}_\mathrm{P} = \textsc{all} \setminus \textsc{one}_\mathrm{P} \tag{9}$$

$$\textsc{UsedOne}_\mathrm{P} = \textsc{used}_\mathrm{P} \setminus \textsc{one}_\mathrm{P} \tag{10}$$

## 5 Evaluation

To evaluate our approach we have applied it to the frontend of the Java compiler JastAddJ [9]. This compiler is specified with RAGs using the JastAdd system. We have profiled the compilation with one or several Java programs as profiling input, and used the resulting AIGs to compute different cache configurations. We have divided our evaluation into the following experiments:

**Experiment A:** The effects of no caching
**Experiment B:** The effects of profiling using a compiler test suite
**Experiment C:** The effects of profiling using a benchmark application
**Experiment D:** The effects of combining B and C

Throughout our experiments we use the results of caching all attributes and the results of using a manual configuration, composed by an an expert, for comparison.

### 5.1 Experimental Setup

All measurements were run on a high-performing computer with two Intel Xeon Quad Core @ 3.2 GHz processors, a bus speed of 1.6 GHz and 32 GB of primary memory. The operating system used was Mac OS X 10.6.2 and the Java version was Java 1.6.0._15.

*The JastAddJ compiler.* The frontend of the JastAddJ compiler (for Java version 1.4 and 1.5) has an $\textsc{all}$ set containing 740 attributes and a $\textsc{pre}$ set containing 47 unconfigurable attributes (14 are circular and 33 are non-terminal attributes). The compiler comes with a configuration MANUAL, with 281 attributes manually selected for caching by the compiler implementor, an expert on RAGs, making an effort to obtain as good compilation speed as possible. The compiler performs within a factor of three as compared to the standard javac compiler, which is good considering that it is generated from a specification. MANUAL is clearly an expert configuration, and it cannot be expected that a better one can be obtained manually.

*Measuring of performance.* The JastAddJ compiler is implemented in Java (generated from the RAG specification), so measuring its compilation speed comes down to measuring the speed of a Java program. This is notoriously difficult, due to dynamic class loading, just-in-time compilation and optimization, and automatic memory management [4]. To eliminate as many of these factors as possible, we use the multi-iteration approach suggested in [5]. We start by warming up the compiler with a number of non-measured compilations (5), thereby allowing class loading and optimization of all relevant compiler code to take place, in order to reach a steady state. Then we turn off the just-in-time compilation and run a couple of extra unmeasured compilations (2) to drain any JIT work queues. After that we run several (20) measured compilation runs for which we compute 95% confidence intervals. In addition to this, we start each measured run with a forced garbage collection (GC) in order to obtain as similar conditions as possible for each run. Memory usage is measured by checking of available memory in the Java heap after each forced GC call and after each compilation. The memory measurements are also given with a 95% confidence interval. We present a summary of these results in Figure 7, Figure 8, Figure 9 and Figure 10. All results have a confidence interval of less than $\pm 0.03\%$. These intervals have not been included in the figures since they would be barely visible with the resolution we need to use. A complete list of results are available on the web [27].

*Profiling and test input.* As a basis for profiling input, we use the Jacks test suite [28], the DaCapo benchmark suite [4,26] and a small hello world program. We use 4170 tests from the Jacks suite, checking frontend semantics, and the following applications from the DaCapo suite (lines of code (LOC)):

**ANTLR:** an LL(k) parser generator (ca 35 000 LOC).
**Bloat:** a program for optimization and analysis of Java bytecode (ca 41 000 LOC).
**Chart:** a program for plotting of graphs and rendering of PDF files (ca 12 000 LOC).
**FOP:** parses XSL-FO files and generates PDF files (ca 136 000 LOC).
**HsqlDb:** a database application (ca 138 000 LOC).
**Jython:** a Python interpreter (ca 76 000 LOC).
**Lucene:** a program for indexing and searching of large text corpuses (ca 87 000 LOC).
**PMD:** a Java bytecode analyzer for a range of source code problems (ca 55 000 LOC).
**Xalan:** a program for transformation of XML documents into HTML (ca 172 000 LOC).

In our experiments, we use different combinations of these applications and tests as profiling input. We will denote these profiling input sets as follows:

**J:** The Jacks test suite profiling input set
**D:** The DaCapo benchmarks profiling input set
**"APP":** The benchmark APP of the DaCapo benchmarks.
For example, ANTLR means the ANTLR benchmark.
**HELLO:** The hello world program

We combine these profiling input sets in various ways, for example, the profiling input set J+ANTLR means we combine the Jacks suite with the benchmark ANTLR. Finally, as test input for performance testing we use the benchmarks from the DaCapo suite and the hello world program.

*Cache configurations.* We want to compare the results of using the cache configurations defined in Section 4. In addition, the JastAddJ specification comes with a manual cache configuration (MANUAL) which we want to compare to. We also have the option to cache all attributes (ALL), or to cache no attributes (NONE):

**MANUAL:** This expert configuration is interesting to compare to, as it would be nice if we could obtain similar results with our automated methods.
**ALL:** The ALL configuration is interesting as it is easily obtainable and robust with respect to performance: there is no risk that a particular attribute will be evaluated very many times for a particular input program, and thereby degrade performance.
**NONE:** The least possible configuration is interesting as it provides a lower bound on the memory needed during evaluation. However, this configuration will in general be useless in practice, leading to compilation times that increase exponentially with program size.

From each profiling input set P, we compute $\text{USED}_P$, and $\text{ONE}_P$, and construct the configurations $\text{USEDONE}_P$ and $\text{ALLONE}_P$ (according to Definition 9 and 10):

**$\text{USEDONE}_P$:** This is an interesting cache configuration as it should give good performance by avoiding caching of unused attributes and attributes used only once by P. The obvious risk with this configuration is that other programs might use attributes unused by P, causing performance degradation. There is also a risk that the attributes in the $\text{ONE}_P$ set may belong to another program's $\text{USED} \setminus \text{ONE}$ set, also causing a performance degradation. However, if attributes in $\text{ONE}_P$ are only used once in a typical application, they are likely to be used only once in most applications.
**$\text{ALLONE}_P$:** This configuration is more robust than the $\text{USEDONE}_P$ configuration in that also unused attributes are cached, which prevents severe performance degradation for those attributes.

*Attribute coverage.* Figure 6 gives an overview of the $\text{USED}_P \setminus \text{ONE}_P$, $\text{ONE}_P$ and $\text{UNUSED}_P$ sets for the profiling inputs from the DaCapo suite. The figure also includes the combined sets for DaCapo (D) and Jacks (J). Not surprisingly, Hello World has the lowest attribute coverage. Still, it covers as much as 29%. The high coverage is due to analysis of standard library classes needed to compile the program. The combined results for the DaCapo suite and two of its applications have better or the same coverage

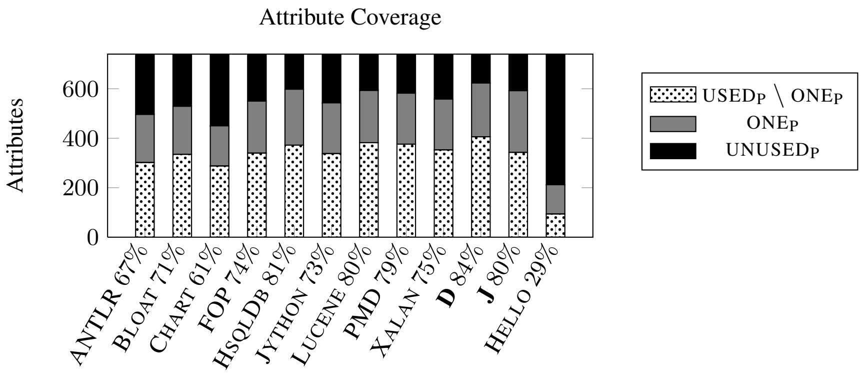

**Fig. 6.** Attribute coverage for the benchmarks in the DaCapo suite, the combined coverage for all the DaCapo applications (D), the combined coverage for the programs in the Jacks suite (J) and for a hello world program. The attribute coverages are given next to the names of the application/combination.

as the Jacks suite, i.e., the USED$_J$ set of Jacks *does not* enclose the USED$_D$ set of DaCapo neither does it have an empty UNUSED$_J$. These observations are interesting since they might indicate that additional tests could be added to Jacks. We can also note that the attribute coverage is not directly proportional to the size of an application, as shown by PMD and Lucene which both are smaller than Xalan and FOP in regard to LOC. This may not be surprising since the actual attribute coverage is related to the diversity of language constructs in an application rather than to the application size.

## 5.2 Experiment A: The Effects of No Caching

To compare the behavior of *no caching* with various other configurations, we profiled a simple Hello World program (HELLO) and then tested performance by compiling the same program using the configurations ALL, NONE, MANUAL, USEDONE$_{\text{HELLO}}$ and ALLONE$_{\text{HELLO}}$. The results are shown in Figure 7. It is clear from these results that

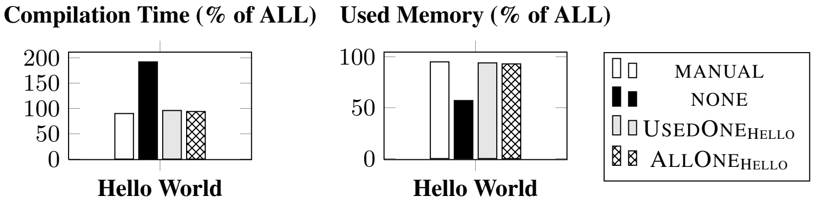

**Fig. 7. Results from Experiment A:** Compilation of Hello World using static configurations along with configurations obtained using Hello World (HELLO) as profiling input. The average compilation time / memory usage when compilating with the ALL configuration were 50.0 ms / 14.7 kb. The corresponding values for the NONE configuration were 95.9 ms / 8.4 kb.

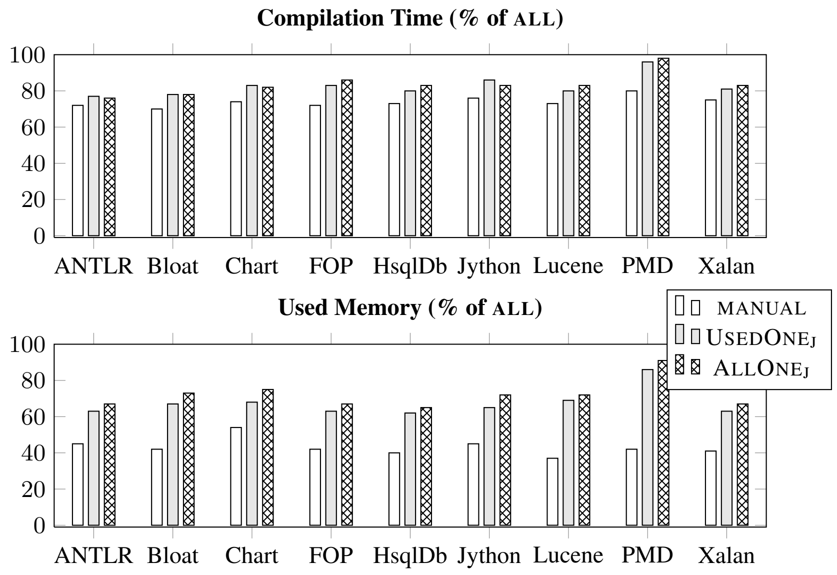

**Fig. 8. Results from Experiment B:** Compilation of DaCapo benchmarks using configurations from the Jacks suite. All results are given in relation to the compilation time and memory usage of the ALL configuration and results for MANUAL are included for comparison.

the minimal NONE configuration is not a good configuration, not even on this small test program. Even though it provides excellent memory usage, the compilation time is more than twice as slow as any of the other configurations. For a larger application the NONE configuration would be useless.

### 5.3 Experiment B: The Effects of Profiling Using a Compiler Test Suite

To show the effects of using a compiler test suite we profiled the compilation of the Jacks suite and obtained the two configurations USEDONEJ and ALLONEJ. We then measured performance when compiling the DaCapo benchmarks using these configurations. The results are shown in Figure 8 and are given as percent in relation to the compilation time and memory usage of the ALL configuration[1]. The results for the MANUAL configuration are included for comparison.

Clearly, the MANUAL configuration performs better with regard to both compilation time and memory usage, with an average compilation time / memory usage of 75% / 47% in relation to the ALL configuration. The average results for USEDONEJ is 83% / 67%. The ALLONEJ configuration has the same average compilation time 83% / 72%, but higher average memory usage. It is interesting to note that USEDONEJ is robust enough to handle the compilation of all the DaCapo benchmarks. So it seems that the

[1] The absolute average results for ALL are the following: Antlr (1.462s/0.270Gb), Bloat (1.995s/0.339Gb), Chart (0.928s/0.177Gb), FOP (8.328s/1.362Gb), HsqlDb (6.054s/1.160Gb), Jython (3.257s/0.611Gb), Lucene (4.893s/0.930Gb), PMD (3.921s/0.691Gb), Xalan (6.606s/1.141Gb).

Jacks test suite has a sufficiently large coverage, i.e., we can use the $\text{USEDONE}_{\text{J}}$ configuration rather than the $\text{ALLONE}_{\text{J}}$ configuration. In doing so, we can use less memory with the same performance and robustness.

## 5.4 Experiment C: The Effects of Profiling Using a Benchmark Program

To show the effects of using benchmarks we profiled using each of the DaCapo benchmarks obtaining the USEDONE and ALLONE configurations for each benchmark. We also combined the profiling results for all the benchmarks to create the combined configurations $\text{USEDONE}_{\text{D}}$ and $\text{ALLONE}_{\text{D}}$. We then measured performance when compiling the DaCapo benchmarks using these configurations. A selected set of the results are shown in Figure 9, including the combined results and the best USEDONE and ALLONE configurations from the individual benchmarks. All results are given as percent in relation to the compilation time and memory usage of the ALL configuration. The results for the MANUAL set are also included for comparison. Note that not all results are shown in Figure 9. Two of the excluded configurations $\text{USEDONE}_{\text{ANTLR}}$ and $\text{USEDONE}_{\text{CHART}}$ performed worse than full caching (ALL). These results validate the concern that the $\text{USEDONE}_p$ configuration would have robustness problems for insufficient profiling input. In this case, neither ANTLR nor Chart were sufficient as profiling

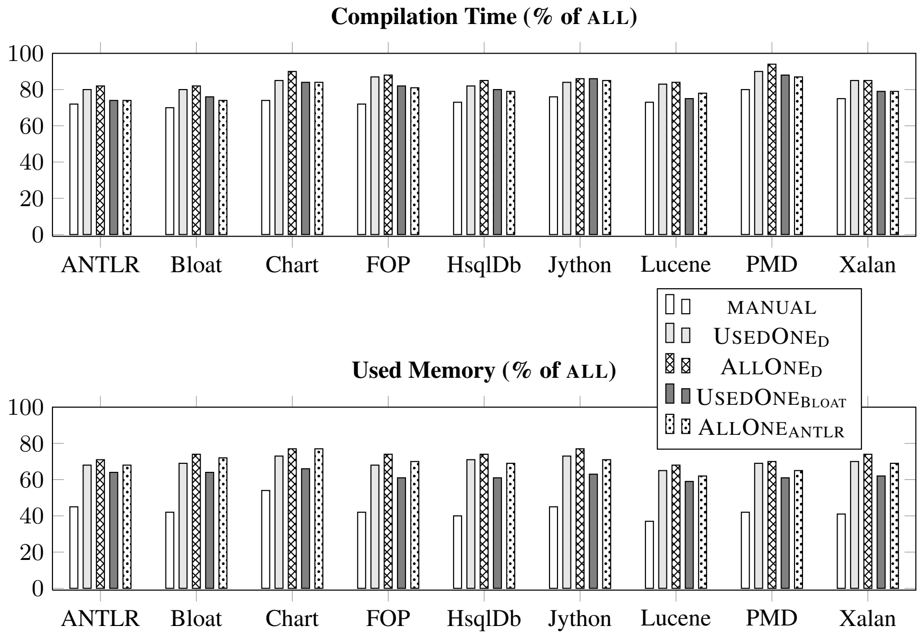

**Fig. 9. Results from Experiment C:** Compilation of DaCapo benchmarks using configurations from the DaCapo benchmarks. All results are shown as compilation time and memory usage as percent of the results for the ALL configuration and results for MANUAL are included for comparison.

inputs on their own. We can note that these two applications have the least coverage among the applications from the DaCapo suite (67% for ANTLR and 61% for Chart).

**The UsedOne configurations.** The UsedOne configurations for ANTLR and Chart perform worse than the ALL configuration for several of the applications in the DaCapo benchmarks: FOP, Lucene and PMD. The remaining UsedOne configurations can be sorted with regard to percent of compilation time, calculated as the geometric mean of the DaCapo benchmark program compilation times (each in relation to the ALL configuration), as follows:

80%: UsedOne$_{\text{Bloat}}$ (mem. 62%)
82%: UsedOne$_{\text{FOP}}$ (mem. 63%), UsedOne$_{\text{Xalan}}$ (mem.66%)
83%: UsedOne$_{\text{HsqlDb}}$ (mem. 68%), UsedOne$_{\text{Jython}}$ (mem. 65%),
  UsedOne$_{\text{Lucene}}$ (mem. 68%), UsedOne$_{\text{PMD}}$ (mem. 66%)
84%: UsedOne$_{\text{D}}$ (mem. 70%)

These results indicate that a certain coverage is needed in order to obtain a robust UsedOne configuration. It is also interesting to note that the combined UsedOne$_{\text{D}}$ configuration for DaCapo performs the worst (except for the non-robust configurations). One possible explanation to this performance might be that some attributes ending up in the UsedOne$_{\text{D}}$ set might be used rarely or not at all in several compilations. Still, these attributes are cached which leads to more memory usage.

**The AllOne configurations.** The AllOne configurations generally perform worse than the UsedOne configurations. This result might be due to the fact that these configurations include unused attributes for robustness. However, this strategy for robustness pays off in that all AllOne configurations become robust, i.e., they compile all the DaCapo benchmarks faster than the ALL configuration. The AllOne configurations can be sorted as follows, with regard to percent of compilation time:

80%: AllOne$_{\text{ANTLR}}$ (mem. 69%)
82%: AllOne$_{\text{Bloat}}$ (mem. 69%), AllOne$_{\text{FOP}}$ (mem. 69%)
83%: AllOne$_{\text{Chart}}$ (mem. 71%)
84%: AllOne$_{\text{HsqlDb}}$ (mem. 73%), AllOne$_{\text{Jython}}$ (mem. 70%),
  AllOne$_{\text{Xalan}}$ (mem. 70%)
85%: AllOne$_{\text{Lucene}}$ (mem. 73%), AllOne$_{\text{PMD}}$ (mem. 71%)
86%: AllOne$_{\text{D}}$ (mem. 73%)

These results indicate that a profiled application does not necessarily need to be large, or have the best coverage, for the resulting configuration to provide good performance. The best individual AllOne configuration is obtained from profiling ANTLR which is remarkable since ANTLR has the next lowest attribute coverage, while the combined AllOne$_{\text{D}}$ configuration for DaCapo performs the worst on average. This result might be due to the fact that the combined configuration caches attributes that might be in the ONE set of an individual application. This fact is also true for several of the individual configurations but apparently the complete combination takes the edge off the configuration.

### 5.5 Experiment D: The Effects of Combining B and C

To show the effects of profiling using a compiler test suite (B) together with profiling a benchmark program (C) we combine the cache configurations from experiment B and C. We then measured performance when compiling the DaCapo benchmarks using these configurations. A selected set of the results are shown in Figure 10, including the fully combined results and the best USEDONE and ALLONE configurations, obtained from combining configurations from ANTLR and Jacks. We can sort the USEDONE configurations, with regard to their percent of compilation time, as follows:

81%: $\text{USEDONE}_{\text{J+ANTLR}}$ (mem. 64%), $\text{USEDONE}_{\text{J+BLOAT}}$ (mem. 67%), $\text{USEDONE}_{\text{J+CHART}}$ (mem. 65%)
82%: $\text{USEDONE}_{\text{J+FOP}}$ (mem. 68%), $\text{USEDONE}_{\text{J+XALAN}}$ (mem. 69%)
83%: $\text{USEDONE}_{\text{J+JYTHON}}$ (mem. 69%)
84%: $\text{USEDONE}_{\text{J+HSQLDB}}$ (mem. 70%), $\text{USEDONE}_{\text{J+LUCENE}}$ (mem. 70%), $\text{USEDONE}_{\text{J+PMD}}$ (mem. 70%)
85%: $\text{USEDONE}_{\text{J+D}}$ (mem. 71%)

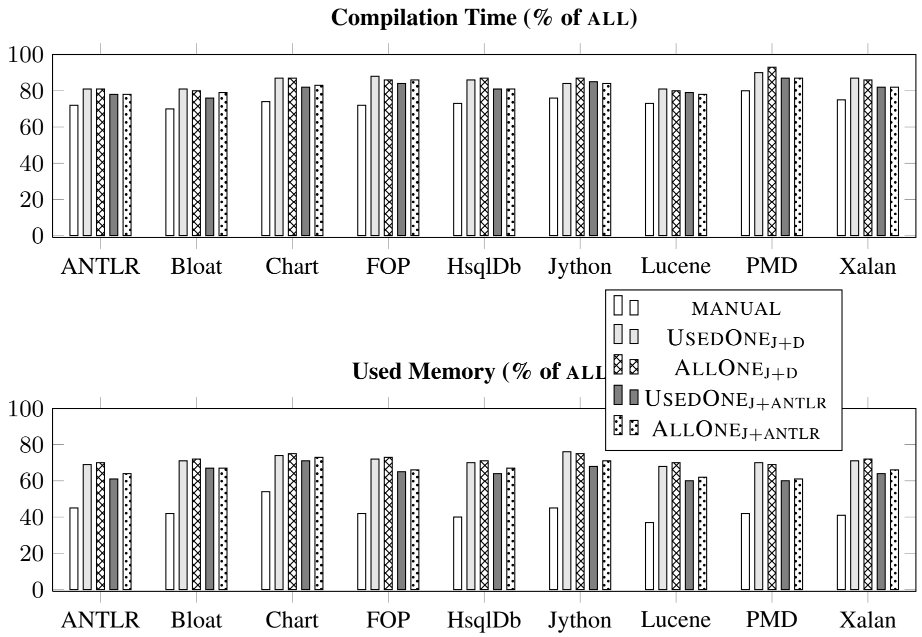

**Fig. 10. Results from Experiment D:** Compilation of DaCapo benchmarks using combined configurations from the Jacks and DaCapo suites. All results are shown as compilation time and memory usage as percent of the results for the ALL configuration and results for MANUAL are included for comparison.

We can sort the ALLONE configurations in the same fashion:

82%: $\text{ALLONE}_{\text{J+ANTLR}}$ (mem. 66%), $\text{ALLONE}_{\text{J+BLOAT}}$ (mem. 68%)
84%: $\text{ALLONE}_{\text{J+CHART}}$ (mem. 66%)
85%: $\text{ALLONE}_{\text{J+D}}$ (mem. 72%)
88%: $\text{ALLONE}_{\text{J+FOP}}$ (mem. 69%), $\text{ALLONE}_{\text{J+JYTHON}}$ (mem. 69%),
$\text{ALLONE}_{\text{J+XALAN}}$ (mem. 70%)
89%: $\text{ALLONE}_{\text{J+LUCENE}}$ (mem. 71%), $\text{ALLONE}_{\text{J+PMD}}$ (mem. 71%)

We note that the influence of the benchmarks improve the average performance of the Jacks configurations (83%) with one or two percent. It is interesting to note that the benchmarks providing the best performance on average for Jacks, independent of configuration, are those with small coverage and few lines of code. These results indicate that it is worth combining a compiler test suite with a normal program, but that the program should not be too large or complicated. This way, we will end up with a configuration that caches attributes that end up in the USEDONE set of any small intricate program, as well as in the USEDONE set of larger programs, but without caching attributes that seem to be less commonly used many times. Further, it should be noted that the memory usage results for the combined $\text{ALLONE}_{\text{J+D}}$ and $\text{USEDONE}_{\text{J+D}}$ present unexpected results when compiling Jython. Presumably, the first configuration should use more memory than the second configuration, but the results show the reverse. The difference is slight but still statistically significant. At this point we have no explanation for this unexpected result.

## 6 Related Work

There has been a substantial amount of research on optimizing the performance of attribute evaluators and to avoid storing all attribute instances in the AST. Much of this effort is directed towards optimizing static visit-oriented evaluators, where attribute evaluation sequences are computed statically from the dependencies in an attribute grammar. For RAGs, such static analysis is, in general, not possible due to the reference attributes. As an example, Saarinen introduces the notion of *temporary attributes* that are not needed outside a single visit, and shows how these can be stored on a stack rather than in the AST [23]. The attributes we have classified as ONE correspond to such temporary attributes: they are accessed only once, and can be seen as stored in the stack of recursive attribute calls. Other static analyses of attribute grammars are aimed at detecting *attribute lifetimes*, i.e., the time between the computation of an attribute instance until its last use. Attributes whose instances have non-overlapping lifetimes can share a global variable, see, e.g., [17]. Again, such analysis cannot be directly transfered to RAGs due to the use of reference attributes.

*Memoization* is a technique for storing function results for future use, and is used, for example, in dynamic programming [3]. Our use of cached attributes is a kind of memoization. Acar et al. present a framework for selective memoization in a function-oriented language [1]. However, their approach is in a different direction than ours, intended to help the programmer to use memoized functions more easily and with more control, rather than to find out which functions to cache. There are also other differences between memoization in function-oriented programming, and in our object-oriented evaluator. In function-oriented programming, the functions will often have many and complex arguments that can be difficult or costly to compare, introducing substantial overhead for memoization. In contrast, our implementation is object-oriented, reducing most attribute calls to parameterless functions which are cheap to cache. And for parameterized attributes, the arguments are often references which are cheap to compare.

## 7 Conclusions and Future Work

We have presented a profiling technique for automatically finding a good caching configuration for compilers generated from RAG specifications. Since the attribute dependencies in RAGs cannot be computed statically, but depend on the evaluation of reference attributes, we have based the technique on profiling of test programs. We have introduced the notion of an attribute dependency graph with call counts, extracted from an actual compilation. Experimental evaluation on a generated Java compiler shows that by profiling on only a single program with an attribute coverage of only 67%, we reach a mean compilation speed-up of 20% and an average decrease in memory usage of 38%, as compared to caching all configurable attributes. This is close to the average compilation speed-up obtained for a manually composed expert configuration (25%). The corresponding average decrease in memory usage for the manual configuration (53%) is still significantly better. Our evaluation shows that we get similar performance improvements for both tested cache configuration approaches. Given these results, we would recommend the ALLONE configuration due to its higher robustness.

We find these results very encouraging and intend to continue this work with more experimental evaluations. In particular, we would like to study the effects of caching, or not caching, parameterized attributes, and to apply the technique to compilers for other languages. Further, we would like to study the effects of analyzing the content of the attribute equations. Most likely there are attributes which only return a constant or similar and, hence, should not benefit from caching independent of the number of calls. Finally, it would be interesting to further study the differences between the cache configurations from the profiler and the manual configuration.

## Acknowledgements

We are grateful to Torbjörn Ekman for fruitful discussions and for implementing support in JastAdd for profiling and separate cache configurations. Thanks also to anonymous reviewers for comments on an earlier version of this paper.

## References

1. Acar, U.A., Blelloch, G.E., Harper, R.: Selective memoization. In: POPL, pp. 14–25. ACM, New York (2003)
2. Åkesson, J., Ekman, T., Hedin, G.: Implementation of a Modelica compiler using JastAdd attribute grammars. Science of Computer Programming 75(1-2), 21–38 (2010)

3. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
4. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 169–190. ACM, New York (2006)
5. Blackburn, S.M., McKinley, K.S., Garner, R., Hoffmann, C., Khan, A.M., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: Wake up and smell the coffee: evaluation methodology for the 21st century. Communications of the ACM 51(8), 83–89 (2008)
6. Boyland, J.T.: Remote attribute grammars. Journal of the ACM 52(4), 627–687 (2005)
7. Bürger, C., Karol, S., Wende, C.: Applying attribute grammars for metamodel semantics. In: Proceedings of the International Workshop on Formalization of Modeling Languages. ACM Digital Library (2010)
8. Ekman, T., Hedin, G.: Modular Name Analysis for Java Using JastAdd. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 422–436. Springer, Heidelberg (2006)
9. Ekman, T., Hedin, G.: The Jastadd Extensible Java Compiler. In: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007), pp. 1–18. ACM, New York (2007)
10. Farrow, R.: Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In: SIGPLAN 1986: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, pp. 85–98. ACM, New York (1986)
11. Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) 24(3), 301–317 (2000)
12. Hedin, G., Magnusson, E.: JastAdd: an aspect-oriented compiler construction system. Science of Computer Programming 47(1), 37–58 (2003)
13. Huang, S.S., Hormati, A., Bacon, D.F., Rabbah, R.M.: Liquid metal: Object-oriented programming across the hardware/Software boundary. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 76–103. Springer, Heidelberg (2008)
14. Ibrahim, A., Jiao, Y., Tilevich, E., Cook, W.R.: Remote batch invocation for compositional object services. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 595–617. Springer, Heidelberg (2009)
15. Jourdan, M.: An optimal-time recursive evaluator for attribute grammars. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 167–178. Springer, Heidelberg (1984)
16. Jouve, W., Palix, N., Consel, C., Kadionik, P.: A SIP-based programming framework for advanced telephony applications. In: Schulzrinne, H., State, R., Niccolini, S. (eds.) IPTComm 2008. LNCS, vol. 5310, pp. 1–20. Springer, Heidelberg (2008)
17. Kastens, U.: Lifetime analysis for attributes. Acta Informatica 24(6), 633–651 (1987)
18. Kats, L.C.L., Sloane, A.M., Visser, E.: Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 142–157. Springer, Heidelberg (2009)
19. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory 2(2), 127–145 (1968); Correction: Mathematical Systems Theory 5(1), 95–96 (1971)
20. Nilsson-Nyman, E., Ekman, T., Hedin, G., Magnusson, E.: Declarative intraprocedural flow analysis of Java source code. In: Proceedings of the Eight Workshop on Language Description, Tools and Applications (LDTA 2008). Electronic Notes in Theoretical Computer Science, Elsevier B.V., Amsterdam (2008)

21. Poetzsch-Heffter, A.: Prototyping realistic programming languages based on formal specifications. Acta Informatica 34(10), 737–772 (1997)
22. Rajan, H.: Ptolemy: A language with quantified, typed events (2010), `http://www.cs.iastate.edu/~ptolemy/`
23. Saarinen, M.: On constructing efficient evaluators for attribute grammars. In: Ausiello, G., Böhm, C. (eds.) ICALP 1978. LNCS, vol. 62, pp. 382–397. Springer, Heidelberg (1978)
24. Schäfer, M., Ekman, T., de Moor, O.: Sound and Extensible Renaming for Java. In: Kiczales, G. (ed.) 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008). ACM Press, New York (2008)
25. Sloane, A.M., Kats, L.C.L., Visser, E.: A Pure Object-Oriented Embedding of Attribute Grammars. In: Proceedings of the 9th Workshop on Language Descriptions, Tools and Applications, LDTA 2009 (2009)
26. The DaCapo Project. The DaCapo Benchmarks (2009), `http://dacapobench.org`
27. The JastAdd Caching Trac. Performance results for JastAddJ (2010), `http://svn.cs.lth.se/trac/jastadd-caching`
28. The Mauve Project. Jacks (Jacks is an Automated Compiler Killing Suite) (2009), `http://sources.redhat.com/mauve`
29. Vogt, H., Doaitse Swierstra, S., Kuiper, M.F.: Higher-order attribute grammars. In: PLDI, pp. 131–145 (1989)
30. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Science of Computer Programming 75(1-2), 39–54 (2010)
31. Van Wyk, E., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute grammar-based language extensions for java. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 575–599. Springer, Heidelberg (2007)