

# Extending Attribute Grammars with Collection Attributes – Evaluation and Applications

Eva Magnusson  
Dept. of Computer Science  
Lund University  
Lund, Sweden  
Eva.Magnusson@cs.lth.se

Torbjörn Ekman  
Computing Laboratory  
Oxford University  
Oxford, United Kingdom  
torbjorn@comlab.ox.ac.uk

Görel Hedin  
Dept. of Computer Science  
Lund University  
Lund, Sweden  
Gorel.Hedin@cs.lth.se

## Abstract

*Collection attributes, as defined by Boyland, can be used as a mechanism for concisely specifying cross-reference-like properties such as callee sets, subclass sets, and sets of variable uses. We have implemented collection attributes in our declarative meta programming system JastAdd, and used them for a variety of applications including devirtualization analysis, metrics, and flow analysis. We propose a series of evaluation algorithms for collection attributes, and compare their performance and applicability. The key design criteria for our algorithms are 1) that they work well with demand evaluation, i.e., defined properties are computed only if they are actually needed for a particular source code analysis problem and a particular source program, and 2) that they work in the presence of circular (fixed-point) definitions that are common for many source code analysis problems, e.g., flow analysis. We show that the best algorithms work well on large practical problems, including the analysis of large Java programs.*

## 1. Introduction

Attribute grammars have recently received renewed interest due to the emergence of practical meta programming tools such as JastAdd [1] and Silver [21] that can handle analysis, transformation and compilation of complex programming languages like Java. Main advantages of these systems are that they make use of declarative specifications, allowing high-level concise specifications, and that they support extensibility, for example, allowing advanced analyses to be added modularly to a compiler. Good performance can be achieved, as shown for our own tool, JastAdd, with which we have built an extensible Java compiler that can compile programs in the order of 100 k lines of code and that runs within a factor of three of javac [6, 7].

The practicality of recent attribute-grammar based systems relies on the development of a variety of extensions [3, 10, 11, 14, 20, 9, 17, 8, 19] to the original Knuth style attribute grammars [16]. In this paper we investigate applications and implementation of *collection attributes*, as defined by Boyland [3]. A collection attribute is the declarative specification of a combination of properties of an unbounded number of abstract syntax tree nodes. Simple examples are the set of calls of a procedure, and the set of subclasses of a class. While such combined properties can be computed by ordinary Knuth-style synthesized and inherited attributes, the use of collection attributes makes their specification much more simple and concise, and opens for more efficient implementations.

Collection attributes are often whole-program properties, i.e., they combine information from, potentially, the whole program. A naive way to implement them is to simply traverse the whole program, find all the contributing AST nodes of the program, and that way compute the combined value. This is potentially very expensive. In this paper we propose a series of evaluation algorithms and compare them, both with regards to applicability and to performance. Our evaluation algorithms are all based on *demand evaluation*, i.e., attributes are not evaluated until their value is demanded by some other computation.

We have implemented collection attributes in our system JastAdd and we give several examples of their use, including devirtualization analysis and metrics for Java programs, and flow analysis for grammars. Of particular interest is the combination of collection attributes with *circular attributes* [9, 17]. A circular attribute is an attribute that is defined, potentially, in terms of itself, and is evaluated through fixed-point iteration.

The rest of this paper is structured as follows. In section 2 we give background on the JastAdd system and discuss a motivating example. Section 3 discusses the definition of collection attributes and how they are used in JastAdd.

Section 4 gives a series of algorithms for evaluation of collection attributes. Section 5 discusses application examples, and Section 6 gives experimental results from the different evaluation algorithms on these applications. Section 7 discusses related work, and Section 8 concludes the paper.

## 2. Motivation

### 2.1 The JastAdd system

The JastAdd system [12, 1] allows source code analyses to be built in a concise way as extensions on top of other analyses, typically on top of the name and type analyses that are core parts of a compiler. The analyses are formulated as attribute grammars (AGs) that include the basic AG mechanisms of synthesized and inherited attributes [16], as well as several extensions, including *reference attributes* [11] that are of key importance to this paper.

A reference attribute of an abstract syntax tree (AST) node is an attribute that refers to another node in the AST. They are used to bind different parts of the AST together, e.g., to bind a use of a variable to its declaration, a class to its superclass, a call to its method declaration, an expression to a type declaration denoting its type, etc. Using JastAdd, such attributes are typically specified in name and type analysis modules, and a compiler is composed by combining these with a code generation module.

For many source code analysis problems it is useful to reuse the reference attributes computed by the name and type analysis modules. In addition, there is often a need for the reverse information, i.e., the cross references. For example, for a metrics problem, we might be interested in finding all the uses of a particular instance variable declaration, all the calls of a method, or all the subclasses of a class. Cross-reference problems are often whole program in the sense that the cross references may be located in any part of the program. For example, a public instance variable declaration can, through qualified use, be used from any other class in the program.

### 2.2 A motivating example

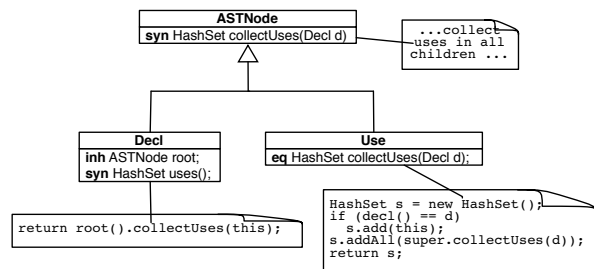
Later in this paper we will see how cross-reference sets can be specified by means of collection attributes. But first, as a motivation and for comparison, we will look at how they can be specified using ordinary synthesized and inherited attributes<sup>1</sup>.

As an example, consider cross references for name bindings. The name analysis module has defined that each *Use* node has an attribute *decl* which refers to the appropriate *Decl* node. We now want to define that each *Decl*

<sup>1</sup>A synthesized attribute is defined by an equation in the same AST node. An inherited attribute is defined by an equation in an ancestor node.

node has an attribute *uses* which is a set of cross references, i.e., it contains references to all the *Use* nodes whose *decl* refers to that particular *Decl* node. This can be done by using attributes that in effect traverse the complete AST and collect all the relevant *Use* nodes. The specification is shown in Fig 1. The *uses* attribute accesses the *collectUses* attribute of the root. The *collectUses* attribute is defined for all AST nodes (in the superclass *ASTNode*), and by default collects the uses of its children. This in effect results in a traversal of the complete AST. For *Use* nodes, the *collectUses* attribute in addition adds a reference to that *Use* node, if appropriate, i.e., if its *decl* refers to the *Decl* in question.

Figure 1. Finding all uses of a declaration.



The JastAdd system evaluates attributes through *demand evaluation*. This means that the value of an attribute is not computed until its value is needed. This is important for efficiency because there may be many attributes defined whose values are not needed for a particular application. For our example we can note that there is one instance of the *uses* attribute for each declaration in the analyzed program. It might be the case that we are interested in a *uses* attribute instance only if some condition holds. For example, if the declaration has a particular name.

In analyzing the specification of the *uses* attribute above, we can notice some drawbacks. First and foremost, the evaluation is inefficient if several instances of the attribute are demanded since a complete tree traversal is performed for every instance. Furthermore, the user has to explicitly express the tree traversal using auxiliary attributes like *collectUses*. In the next section, we will see how both these drawbacks can be overcome through the use of collection attributes.

## 3. Collection Attributes

### 3.1. Definitions

The value of an ordinary synthesized attribute of an AST node *n* is defined locally by an equation in node *n*. In con-

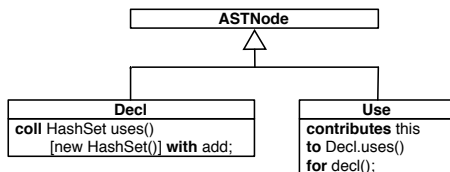
trast, the value of a *collection attribute*, as defined by Boyland [3], is defined through a number of partial definitions, located in arbitrary nodes in the AST. More precisely, the value is defined as a combination of an *initial value* and zero or more applications of a *combination operation*. The collection attribute declaration contains the initial value and the combination operation. The partial definitions, in the form of applications of the combination operation, can be located in arbitrary AST nodes. Following the declarative paradigm of AGs, the order of specification is irrelevant. Therefore, the combination operation must be such that the order of application does not affect the resulting value of the collection attribute. Typical examples of collection attribute types are *sets*, with the empty set as the initial value and *add element* as the combination operation; *booleans*, with false and *or*; and *integers*, with zero and *+*.

In order to facilitate description of evaluation algorithms we introduce some additional terms. A node containing a partial definition for some collection attribute  $c()$  is said to be a *contributor* to its final value. Alternatively, we say that the node *contributes* to the value of  $c()$ . The value of the partial definition is said to be its *contribution*.

### 3.2. Motivating example revisited

Consider again the problem of finding all uses of a declaration, as was described in Section 2. After introducing collection attributes, we can replace the synthesized attribute *uses* with a collection attribute as shown in Fig 2. It has the initial value `new HashSet()` and the combination operator `add`. The nodes of type `Use` are contributors of `Decl.uses` and their respective contribution is `this`, i.e., a reference to the `Use` node. The reference attribute `decl` (that was defined in the name analysis module) points out the appropriate `Decl` node to contribute to.

Figure 2. Defining uses with a collection



This definition is much more concise than the one using ordinary attributes. We can also notice that the combination operator `add` is an ordinary Java method that updates the state of the `HashSet` object. This is fine, since the value of *uses* will not be used until its final value is computed. In contrast, the solution shown in Section 2 exposes all the partial collections as attributes, and therefore needs to represent

them as separate objects. This difference will contribute to better efficiency for the collection attribute solution.

### 3.3. JastAdd collection attribute syntax

The JastAdd syntax for declaring a collection attribute  $c$  of type  $T$  in a node class  $N1$  is:

```
coll T N1.c() [' initial ']' with op;
```

The declaration of the attribute  $c$  is an *intertype declaration* in that it allows the declaration to be expressed in a module textually separate from class  $N1$ , similar to intertype declarations of methods and fields in, for example, AspectJ [15]. The expression *initial* is the initial value of the collection attribute, before applying the contributions, and thereby also the final value in the case of zero contributions. The `op` should be a Java method for class  $T$  that serves as the combination operator and updates the  $T$  object by adding a contribution. The method `op` should be void and have a single parameter of the same type as the contributions<sup>2</sup>. Given a set of contributions  $E$ , the final value of  $c$  is computed as follows (pseudo code):

```
T c = initial;
for all e in E do
  c.op(e);
end for
```

The uses example in Fig 2 illustrated a very simple example of declaring contributions. In more complex cases it can be desirable to express conditional contributions, and contributions for a set of collection attributes, not just for a single attribute. Below, the general syntax for declaring contributions from a node  $N2$  to collection attributes  $c$  in  $N1$  nodes is shown.

```
N2 contributes
  contr1 [when cond1],
  contr2 [when cond2],
  ...
  contrN [when condN]
to N1.c()
for [each] ref();
```

The contributions `contr1`, `contr2`, etc., should be expressions that have the same type as the parameter of the combination operator of  $c$ . The semantics of a specification with *when*-clauses is that all contributions for which the corresponding condition holds are valid. The expression `ref` should be a reference to an  $N1$  object, or, if the optional keyword `each` is used, `ref` should be a *set* of references to  $N1$  objects. In the latter case, the contribution is added to the collection  $c$  of *all* those  $N1$  objects.

<sup>2</sup>If collections of primitive types like `int` and `boolean` are desired, they currently have to be implemented by wrapper classes.

Below, an example of using `for each` in a contribution is given. The class `MethodDecl` has a collection attribute `calls` (defined elsewhere) which contains references to all methods called inside its body. We now want to define a cross-referencing collection, `callers`, that should contain references to all methods that call the `MethodDecl`. This is accomplished concisely by letting the `MethodDecl` contribute itself to all the `caller` attributes of each of the members in its `calls` attribute.

```
coll HashSet MethodDecl.callers() =
  new HashSet() with add;

MethodDecl contributes this
to MethodDecl.callers()
for each calls();
```

## 4. Evaluation of Collection Attributes

### 4.1. Attribute evaluation in JastAdd

In JastAdd, a demand driven evaluation technique is used. This means that an attribute instance is not computed until its value is needed. If the value depends on other attribute instances, these instances are demanded and evaluated as well. The evaluator code for ordinary synthesized and inherited attributes is realized by translating their declarations and equations into Java methods, as described in more detail in [12]. For efficiency reasons, a caching technique is also available. Circular attributes are always cached since the iterative technique used for their evaluation requires values from a previous iteration to be cached for convergence check. When the iterative process is over, these cached values contain the final values of all attributes involved in the cycle. Collection attributes are also always cached since their computation involves traversing the whole AST, and is thus inherently expensive.

In implementing collection attributes, we want to keep with the demand-driven approach, so that the evaluation of the attributes demanded in a particular application is not slowed down by the existence of attributes that are not demanded.

### 4.2. Naive evaluation

A simple way to implement a collection attribute is to represent the collection attribute by a method that traverses the complete AST, finds the appropriate contributors, and returns the final value, i.e., the combination of the contributions. We refer to this evaluation scheme as the *naive evaluation algorithm*. This algorithm has the advantage that it is purely demand driven: if a single collection attribute instance is demanded, there is no extra work involving the

evaluation of other instances. But if several instances are demanded, the tree will be traversed over and over again, leading to overall inefficiency.

### 4.3. One-phase joint evaluation

To obtain better overall efficiency, it is possible to compute *all* instances of a given collection attribute when the first instance is demanded. One traversal of the tree is sufficient to find all contributing nodes to any instance of the attribute and to perform the proper computations for combining contributions. We call this technique *one-phase joint evaluation*. This scheme deviates from the demand-driven technique. If only a single attribute instance is actually demanded, it will be less efficient than the naive algorithm. But if more instances are demanded, it will quickly become much more efficient.

A shortcoming of the one-phase algorithm is that it is less general than the naive one: If a condition or contribution for one collection attribute instance depends on another instance of the attribute, the scheme fails. The dependency will then trigger a new tree traversal, during which the same dependency will be encountered, and the algorithm will end up in a loop. The naive algorithm does not have this problem: the dependency will simply trigger the evaluation of the other collection attribute instance and then continue with the evaluation of the first one. The naive technique will fail only if the dependencies between the collection attribute instances are in fact circular, i.e., if a collection attribute instance depends (possibly indirectly) on itself.

### 4.4 Two-phase joint evaluation

In order to avoid the inefficiency of the naive technique and to avoid the shortcomings of the one-phase technique, we propose an alternative technique, *two-phase joint evaluation*. In the following we assume that `N1` is a node class declaring a collection attribute `c`.

**Survey phase** The first time any instance of the `c` attribute is demanded, a traversal of the AST is triggered. During this traversal, contributors to *all* instances of `c` are collected into *contributor sets*, one set for each instance of `c`, and stored in an auxiliary attribute `c_contributors`, in class `N1`. A flag is set to indicate that this *survey phase* has been performed.

**Combination phase** To compute the value of an instance of `c`, the flag is first checked to see if the survey phase has already been run. If not, this phase is performed first. Then the *combination phase* is run: the attribute value is computed by iterating over the elements in the corresponding `c_contributors` set, and combining the contributions using the combination operation.

The final value of the  $c$  instance is cached so that subsequent demands for it can return the value directly.

For conditions attached to contributions we have two variants of the two-phase algorithm: to evaluate the conditions during the survey phase, *early condition evaluation*, or to postpone these computations until the combination phase, *late condition evaluation*. If conditions are evaluated during the survey phase, and there is more than one when-clause, they will have to be checked again during the combination phase.

### Qualitative analysis of running time

Like the one-phase technique, the two-phase technique avoids repeated traversals of the AST. It is more demand-driven than the one-phase technique: the combination phase is done on demand for individual collection attribute instances. We can therefore expect that it is faster than the one-phase technique if sufficiently few instances are actually demanded. We can also expect it to be slower than the one-phase algorithm if sufficiently many instances are demanded, since the contribution sets are computed explicitly and stored, something which is not needed in the one-phase algorithm.

The late or early evaluation of conditions will affect the size of the contributor sets computed during the survey phase. In order to keep these sets small, it seems desirable to use early condition evaluation. Small sets will also improve the efficiency of the combination phase when method calls are made for all members of the sets. If there is a sufficiently large difference in size between the contributor sets for the early and late variants, we can therefore expect the early variant to be more efficient. However, early condition evaluation may fail under certain circumstances, as discussed below.

### Applicability of the algorithms

We noted that the one-phase technique can fail if a contribution for one collection attribute instance is dependent on another instance of the collection attribute. The two-phase technique overcomes this shortcoming since the contributions are not evaluated in the joint survey phase. They are evaluated in the combination phase when each attribute is evaluated individually on demand.

In a similar way, we can note that the two-phase technique with early condition evaluation can fail, causing looping evaluation, if a contribution *condition* for one collection attribute instance is dependent on another instance of the collection attribute. The two-phase technique with *late* condition evaluation cannot fail this way, since the conditions are evaluated in the on-demand combination phase rather than in the jointly evaluated survey phase.

The two-phase algorithm with late condition evaluation can fail only if there is a true circular dependency. To see this, we first observe that the only expressions that are evaluated in the survey phase are the references to the nodes with collections. A cyclic evaluation can therefore occur only if such a reference expression is dependent on one of the collection attribute instances. But each instance of the collection attribute depends on all instances of the reference expressions, since all these references need to be evaluated in order to examine whether a contribution is valid for the collection attribute instance being computed. Thus, if any of the instances of the reference expressions depends on any instance of the collection attribute, the dependency graph is cyclic. The attributes on the cycle must then be evaluated using algorithms capable of dealing with cyclic dependencies. Such algorithms are more expensive, and are discussed in Section 4.5.

To conclude this analysis, different algorithms may be the fastest for different applications. The one-phase algorithm and the two-phase with early condition evaluation may fail for certain applications, even if there are no circular dependencies. However, we have not found any realistic applications of this kind. The two-phase algorithm with late condition evaluation and the naive algorithm will work for all non-circular problems. If there is a true circular dependency, an algorithm that can deal with circularities is needed.

### Additional variants

**Grouped joint evaluation** The discussed one-phase and two-phase algorithms evaluate all or parts of all instances of a single collection attribute. It is also possible to evaluate instances of more than one collection attribute jointly. We call this *grouped joint evaluation*. The grouping can be done in combination with either of the joint evaluation algorithms: one-phase, two-phase with early conditions, or two-phase with late conditions.

**Pruned traversal** Through a simple analysis of the abstract grammar, it can be determined which AST types can be derived from a given AST type  $T$ . If none of these AST types contain contributions for a given collection attribute  $c$ , then the AST traversal for  $c$  can be pruned at  $T$  nodes, thereby speeding up the traversal. However, for Java, such prunings cannot be expected to be other than marginal due to the very recursive structure of the language. For example, a Java expression can derive an anonymous class, which means that most AST types can be derived from expressions.

## 4.5. Circular collection attributes

In [17] we showed how the combined formalism CRAG (Circular Reference Attributed Grammar), which supports both reference attributes and circularly defined attributes, enhances the expressiveness of AGs. When introducing collection attributes it then becomes natural to explore the possibility to handle collection attributes with circular dependencies. In this section we will briefly discuss this extension and describe evaluation algorithms.

The requirement for non-circularity in traditional AGs is a sufficient but not necessary condition to guarantee that the AG is well defined, i.e., that all semantic rules can be satisfied. It actually suffices that all attribute instances involved in circular dependencies have a fixed point that can be computed with a finite number of iterations. Conditions to ensure this for ordinary attributes can easily be carried over to collection attributes. Evaluation algorithms that work well if these conditions are met can be built by combining the ideas for CRAGs in [17] and the ideas for evaluating non-circular collection attributes as described earlier.

We have developed and implemented two different algorithms for circular collection attributes. The first algorithm builds on the ideas of the two-phase joint evaluation technique used for non-circular collection attributes. The survey phase is carried out as before while the combination phase is iterated until a fixed point is reached. It is applicable for cases where no dependencies between instances of the attribute being computed appear during the survey phase. As for non-circular attributes the algorithm has two variants. One evaluates conditions during the survey phase and the other postpones that computation until the combination phase. Hence, the first alternative cannot handle interdependencies caused by conditions, but is more efficient as condition evaluation can be omitted in the iterative combination phase. The second alternative handles all cases except where dependencies on other instances are caused by the reference attributes, which seems to be extremely rare in practice.

When the second alternative above fails another algorithm has to be used. It builds on the ideas of the naive algorithm for non-circular collection attributes. It is an iterative process where each iteration traverses the AST to combine contributions to the demanded instance. It is more expensive than the two-phase technique described above and should therefore only be used when the other alternatives fail. This algorithm can also be used when attempt to use the non-circular variant with late condition evaluation fails as described in Section 4.4.

## 4.6. Implementation of the algorithms

In JastAdd, we have implemented the following algorithms for evaluation of noncircular collection attributes:

naive, one-phase, two-phase with early condition evaluation, two-phase with late condition evaluation, and grouped joint evaluation (that can be combined with any of the joint evaluation algorithms). Pruned traversal has not been implemented. For circular collection attributes we have implemented both variants of the two-phase technique and also the naive algorithm. For each of the algorithms, failures (due to real circularities for attributes declared to be non-circular or to shortcomings of the algorithm) will be detected dynamically. The evaluator will then throw an exception and stop the execution. The default algorithm for noncircular as well as for circular collection attributes is non-grouped two-phase with late condition evaluation. It is possible to annotate individual collection attributes to select another algorithm for that attribute. This can be useful if the first algorithm failed, or to achieve faster evaluation.

## 5. Application Examples

This section discusses some applications that we have implemented using collection attributes.

### 5.1. Devirtualization

For object-oriented languages it is possible to improve execution speed by applying devirtualization techniques. The aim is to determine statically which virtual method calls can be replaced by static method calls. There are many different techniques for devirtualization based on analyzing the class hierarchy and the call graph of the program.

The simplest condition for devirtualization is that a class has no subclasses. If for a methodcall `a.m()` the declared type of `a` is a class `A` with this property, then `m()` can be devirtualized. This is also possible if `A` has subclasses but `m()` is not overridden in any of them. Part of a devirtualization analysis based on these simple criteria can be specified using collection attributes. Assume that `ClassDecl` is a node class modelling class declarations and that a set-valued attribute `superClasses()` has been specified to contain references to all its superclasses. The set of subclasses can then be modelled as a collection attribute:

```
coll HashSet ClassDecl.subClasses()
  [new HashSet()] with add;

ClassDecl contributes this
to ClassDecl.subClasses()
for each superClasses();
```

A collection attribute `overrides()` can be specified for methods in the same simple manner provided that an ordinary set-valued attribute `overrides()` containing references to all overriding methods is available.

Devirtualization can be improved if reachability is taken into account as in the well-known RTA algorithm. This algorithm uses information about global class instantiation and class hierarchy. In order to decide whether a class is instantiated we need to find *new* expressions inside reachable methods. The property for a method of being reachable can be defined in a recursive manner: the `main` method is reachable and other methods are reachable if any of their callers are reachable. Therefore it can be modelled as a circular attribute `reachable()` using an auxiliary attribute `callers()` for its specification. The `callers()` attribute is naturally expressed as a collection attribute cross-referencing an ordinary attribute `calls()` modelling the set of methods called from inside a method body.

## 5.2. Metrics

To evaluate the use of collection attributes in a real-life application we implemented Chidamber and Kemerer’s set of object-oriented metrics [5]. These metrics include structural properties such as the height of inheritance tree and number of subclasses, but also more global properties such as the coupling between classes. Internal properties such as the lack of cohesion of methods within a class and the number of weighted methods per class are also computed.

The metrics are implemented as a modular extension to our Java 1.4 checker and consists of 7 collection attributes, 17 contribution declarations, and 12 synthesized utility attributes. The entire specification, including code for printing the metrics for each type, is 165 lines of code excluding comments, and is available on the JastAdd web site [1]. We compare our implementation to the program *ckjm* [18] which provides an alternative implementation for the same set of metrics but that is using a set of visitors on top of the Byte Code Engineering Library (BCEL). That implementation is also completely modular but more than twice as large, being 380 lines of code without comments. The collection attribute based implementation has a number of improvements compared to the visitor based solution. The implementation is not as tangled, i.e., each attribute computes a single metric rather than interleaving multiple ones within a visitor. Another improvement is that each metric is implemented by a set of equations in a single module rather than being scattered across multiple visitors. The declarative attributes also alleviates the programmer from the manual scheduling of visitor passes, and the temporary storage of intermediate state.

## 5.3. Grammar flow problems

Specifications using ordinary circular attributes can, in some cases, be expressed in a more simple and concise way using circular collection attributes. An example is the

task of computing the *nullable* property and the *first* and *follow* sets for the nonterminals of context-free grammars. In [17] it is shown how the recursive definitions of these concepts can be translated into specifications using circular attributes. This is straightforward for *nullable* and *first* but somewhat more laborious for *follow*. The reason is that all places where a nonterminal is used in production right-hand sides contribute to its *follow* set and these places are distributed all over the grammar. Using circular collection attributes the specification becomes trivial. A reference attribute `decl()` from nonterminal use sites `NUse` to nonterminal “declaration” sites (production left-hand sides) `NDecl` from the name analysis is reused for this purpose:

```
coll HashSet NDecl.follow() circular
    [new HashSet()] with addAll;

NUse contributes followContribution()
to NDecl.follow()
for decl();
```

`followContribution()` is an ordinary synthesized attribute defined to contain all terminals that can immediately follow an applied occurrence of a nonterminal.

## 6. Experimental results

### 6.1. Emulated, naive, and joint evaluation

We start by comparing the performance of emulated collection attributes (using ordinary attributes) with real collection attributes using the naive algorithm and the two-phase algorithm with late condition evaluation (2Ph-LC). The 2Ph-LC variant of the joint evaluation algorithms is chosen because it is the most general one (it can handle all non-circular collections), and is therefore functionally equivalent to both the naive algorithm and to the emulated solution.

As an extension to the front end of our JastAdd extensible Java compiler [7], we have specified two cross-reference attributes: `varUses` which is similar to the computation of `uses` in Section 3.2, but uses a condition in the contribution to cover uses of variables only; and `subClasses` as shown in Section 5.1. To measure performance, we have run a number of Java programs in this extended front end. Table 1 shows the results for demanding all instances of `varUses`, and Table 2, the same for `subClasses`.

The tests have been run on three sample Java programs. The first one is a typical student program with about 750 lines of code. The second program has 15.000 lines and has been constructed especially to test a case with several contributions for each collection attribute instance. The third program is the source code for the `javac` compiler which comprises about 36.500 lines.

Time is measured in milliseconds and is given for the following different evaluation alternatives: Emulated collection attributes (Emul attrs), the naive algorithm (Naive alg), and two-phase joint evaluation with late evaluation of conditions (2Ph-LC).

For each test, the following information concerning the size is given: number of lines of Java code in the program (lines), number of collection attribute instances (coll inst), number of contributing nodes (contr inst) and number of applicable contributions (appl contr), i.e., the number of contributions for which the attached condition, if any, is true. Note that the number of applicable contributions can be greater than the number of contributing nodes when there are for-each clauses or when-clauses in the contributions.

As expected, the joint evaluation technique (2Ph-LC) outperforms emulated collection attributes as well as the naive algorithm. The reason is that the tree has to be traversed for each instance of the collection attribute in the latter cases. It is also evident that the 2Ph-LC algorithm scales well, even sublinearly, with program size. The reason is that even small programs bring in large parts of the standard libraries.

**Table 1. Computation of varUses**

lines	Size			Time (ms)		
	coll inst	contr inst	appl contr	Emul attrs	Naive alg	2Ph-LC
750	73	337	184	4100	1050	50
15000	30	14565	14565	3100	730	135
36500	1969	29180	6464	675000	165000	200

**Table 2. Computation of subclasses**

lines	Size			Time (ms)		
	coll inst	contr inst	appl contr	Emul attrs	Naive alg	2Ph-LC
750	8	8	0	550	160	35
15000	77	77	126	6300	1600	60
36500	180	180	153	110000	15000	200

## 6.2. 1-phase versus 2-phase evaluation

In Section 4 we stated that 2-phase evaluation should be faster than 1-phase joint evaluation (1Ph) if sufficiently few instances of a collection attribute were demanded. A reason why instances can be un-demanded is that for some analysis tasks only instances appearing in certain contexts are of interest. This is the case, for example, in devirtualization analysis.

Table 3 shows results for a very simple devirtualization analysis. It checks two simple conditions for possible devirtualization. The first condition is that for each virtual methodcall  $\text{exp.m}()$  the formal type  $F$  of  $\text{exp}$  is a class

with no subclasses. The second condition is that  $m()$  is not overridden in classes below  $F$ . The specification involves two collection attributes modelling the set of subclasses for a class and the set of overriders of a method. As indicated in the table, only a subset of these attribute instances are demanded. If a method is never called or if it only appears in static method calls, the value of its collection attribute instance is not needed. In the 2Ph-LC evaluation technique the combination phase will be performed only for demanded instances, while the 1Ph technique always evaluates all instances. As a consequence, the 2Ph-LC is somewhat faster in this case.

We have also measured the difference between the 2Ph-LC and 2Ph-EC (two-phase with early condition evaluation) variants, and found the differences to be very marginal.

**Table 3. Devirtualization computations**

Size				Time (ms)	
Coll attr instances		Demanded instances		2Ph	1Ph
subClasses	overriders	subClasses	overriders	LC	joint
645	6769	139	1219	700	760

## 6.3. Grouped evaluation

We have performed measurements on our implementation of the Chidamber and Khemerer metrics, described in Section 5.2. Since these are defined using 7 different collection attributes, this gives the opportunity to measure grouped evaluation, i.e., evaluating several different collection attributes jointly.

As a sample application to compute the metrics on, we have used the source code of the Jigsaw web server. Jigsaw is the W3C’s web server platform, consisting of more than 100 k lines of Java code excluding comments.

We have performed one suite of tests computing metrics for *all* types in the Jigsaw application, and another suite of tests for only the types that are in packages starting with *org.w3c.www*. While this subset of types accounts for roughly a fifth of the source code, all source code for Jigsaw is needed to perform the computations, since some of the metrics take contributions from types in other Jigsaw packages. This allows us to evaluate whether we can exploit the demand driven evaluation of the collection attributes for this particular application: For the first suite, all collection attribute instances are demanded. For the *org.w3c.www* case, only a subset of the instances are demanded.

Table 4 shows the results of our experiments for different evaluation algorithms. We see that the one-phase algorithm (1Ph) is somewhat faster than the two-phase algorithm (2Ph-LC), even in the *org.w3c.www* case when not all collection attribute instances are demanded. The number of

demanded instances is thus not small enough for the two-phase algorithm to get an advantage. The contributions are actually all very simple, typically adding one to a counter or adding a reference to a set. The main execution cost is thus related to AST traversal.

The table also shows results from grouped joint evaluation, both for one-phase (1Ph-Grp) and for two-phase (2Ph-LC-Grp). We could not place all collection attributes in the same group, due to dependencies between the collection attributes. But we could place six of them in one group and the seventh in a group of its own. We can see that there are significant performance improvements for this grouped evaluation.

The times in Table 4 are in milliseconds and *Decr* is the decrease in execution time. Times include only the execution time for computing the metrics. The entire analysis also includes lexing, parsing, AST building, and error checking, which adds another 12 seconds to the overall analysis time. It is somewhat unfair to compare this result to *ckjm*, which takes 3.4 seconds, since it processes bytecode which requires much less static analysis, e.g., all names are bound before hand. Processing source will therefore always be more expensive than bytecode, but we still find the performance perfectly reasonable for a fairly large project, and we notice that for this particular application the bottleneck is not the collection attributes.

**Table 4. Metrics computation (ms)**

Jigsaw types	1Ph	1Ph-Grp	Decr	2Ph-LC	2Ph-LC-Grp	Decr
all	2207	1535	24%	2585	1948	25%
org.w3c.www	1320	693	47%	1639	1002	39%

#### 6.4. Circular evaluation algorithms

For circular collection attributes the two phase and the naive schemes have been implemented. Table 5 shows execution times when these are used to compute the *follow* sets for the Java grammar. For comparison we show results for two solutions using ordinary circular synthesized attributes. The first (Emul-naive) specifies the `follow` attribute as a tree traversal during which contributions to follow sets are collected. The evaluator will in this case perform iterations over the entire AST. The second (Emul-improved) is the one used in [17]. It emulates the two-phase technique for collection attributes by introducing auxiliary ordinary set valued attributes for non terminals containing references to their use sites. The circular `follow` attribute is specified as the union of contributions from these sites. As a consequence, iterations will only involve these sites.

Specifications using collection attributes outperforms the naive emulated solution. The improved emulated specifica-

tion is, however, faster than the naive algorithm for collection attributes. The reason being that the latter performs iterations over the entire AST. Also, the improved emulated solution is only marginally slower than the 2Ph-LC algorithm. The experiment thus indicates that circular collection attributes give modest performance improvement. The main advantage being that they yield much simpler specifications.

**Table 5. Circular collection attributes**

Size		Time			
Coll inst	Contr inst	Emul-naive	Emul-improved	Naive	2Ph-LC
155	280	22500	320	480	270

## 7. Related Work

Restricted forms of collection attributes were introduced by Knuth [16] who allowed global sets in the start symbol, and by Kaiser [13] and Beshers [2] who allowed collection attributes associated with subtrees. Hedin introduced general collection attributes with contributions via reference attributes, but with partly manual implementation techniques [10].

The collection attributes as presented in this paper were introduced by Boyland in his PhD-thesis, [3]. His APS system also supports circular collection attributes. The focus of Boyland’s thesis is on the attribute specification mechanisms and how they can be applied. There is only a brief sketch of the implementation, and no performance results are reported. Just like JastAdd, the APS system uses a demand-driven evaluation technique. The APS technique for evaluating collection attributes is based on a concept called *guards*, i.e., artificial attributes that are added by the APS compiler. Consider a collection attribute *c* in node type *N*. Each instance of *c* is made dependent on a guard, and the guard is in turn made dependent on all the reference expressions of type *N*. Each instance of *c* is evaluated on demand, but not until all reference expressions of type *N* are evaluated. The implementation sketch does not give the details of how this is done, but the effect seems similar to our two-phase algorithm.

In his later work on collection attributes, e.g., [4], Boyland investigates static evaluation algorithms and incremental versions of them. In this work, circular dependencies are no longer supported, and the evaluation technique is based on static analysis of the grammar rather than demand evaluation. This work focuses on theory and contains no reports on practical applications or performance results.

Silver [21] is a recent AG system supporting collection attributes. It supports several extensions to traditional AGs such as higher-order attributes, forwarding and pattern

matching. There is, however, no support for circular attributes. In Silver, attributes are evaluated by translating the AG specifications into Haskell. The system is modular, consisting of a core attribute grammar language which serves as the host language for specifying extensions. Collection attributes have been implemented as one such language extension. The work focuses on the composability of language constructs and the application of Silver to extensible and domain-specific languages. No performance results or specific evaluation algorithms are reported.

## 8. Conclusions

We have shown how cross-reference-like properties can be specified very concisely using collection attributes. We have presented several evaluation algorithms, and found that the joint evaluation algorithm works very well for large practical applications. Our implementation of the Chidamber and Kemerer metrics increased the compilation time with only 1-2 seconds for Java programs of 100 k lines of code. As shown by our smaller examples, the naive algorithm is several orders of magnitude slower. Emulation, using ordinary inherited and synthesized attributes, is much slower still.

We have presented a series of variants on the joint evaluation algorithm. The 2Ph-LC variant is a general algorithm and can handle all non-circular collection attributes. It is used by default in the JastAdd system. The other variants: 1Ph and 2Ph-EC, can be faster for some applications, but they are not completely general: contrived non-circular examples can be constructed that these algorithms cannot handle. For our example applications, the 2Ph-EC variant was only very marginally faster than the 2Ph-LC algorithm. The 1Ph variant was marginally slower on some examples and marginally faster on others. These variants can be selected by annotating individual collection declarations.

Using grouped joint evaluation led to decreased execution time from 25% to almost 50% on the metrics application, depending on source program and on algorithm variant. However, grouping cannot be done automatically: the user has to explicitly annotate the collection attribute declarations with the desired group.

We have also extended the algorithms to circular variants that can handle collection attributes explicitly declared as circular. The 2Ph-LC circular algorithm is used as the default. On our example application in grammar flow analysis, the use of circular collection attributes gave a much simpler specification than the corresponding non-collection attribute solution, without leading to decreased performance.

## References

[1] JastAdd, 2007. <http://jastadd.cs.lth.se/web/>.

- [2] G. M. Beshers and R. H. Campbell. Maintained and constructor attributes. In *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, pages 34–42, New York, NY, USA, 1985. ACM Press.
- [3] J. T. Boyland. *Descriptive Composition of Compiler Components*. PhD thesis, University of California, Berkeley, Sept. 1996.
- [4] J. T. Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.
- [5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.
- [6] T. Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Sweden, June 2006.
- [7] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. Accepted for publication at OOPSLA’07.
- [8] T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of LNCS, pages 144–169. Springer, 2004.
- [9] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of CC’86*, pages 85–98. ACM Press, 1986.
- [10] G. Hedin. An overview of door attribute grammars. In *Proceedings of CC’94*, volume 786 of LNCS, pages 31–51, Edinburgh, Apr. 1994.
- [11] G. Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.
- [12] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
- [13] G. E. Kaiser. *Semantics for structure editing environments*. PhD thesis, Carnegie Mellon University, 1985.
- [14] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.
- [15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of LNCS, pages 327–355. Springer, 2001.
- [16] D. 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).
- [17] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Electr. Notes Theor. Comput. Sci.*, 82(3), 2003.
- [18] D. D. Spinellis. ckjm - Chidamber and Kemerer Java Metrics, 2007. <http://www.spinellis.gr/sw/ckjm/>.
- [19] E. Van Wyk, O. d. Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of CC 2002*, volume 2304 of LNCS, pages 128–142. Springer, 2002.
- [20] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings PLDI’89*, pages 131–145. ACM Press, 1989.
- [21] E. V. Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute Grammar-based Language Extensions for Java. In *Proceedings of ECOOP’07*, LNCS. Springer, 2007.