LUND UNIVERSITY

Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking

Söderberg, Emma; Hedin, Görel

2012

Total number of authors:
2

Incremental Evaluation of
Reference Attribute Grammars using
Dynamic Dependency Tracking

Emma Söderberg

Görel Hedin

Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking

Emma Söderberg and Görel Hedin

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

Abstract. Reference attribute grammars (RAGs) have proven practical for generating production-quality compilers from declarative descriptions, as demonstrated by the JastAdd system. Recent results indicate their applicability also to generating semantic services in interactive editors. For use in editors, it is necessary to update the attribution after edit operations. Earlier algorithms based on statically scheduled incremental attribute evaluation are, however, not applicable to RAGs, as they do not account for the dynamic dependencies that reference attributes give rise to. In this report, we introduce a notion of consistency for RAG attributions, along with an algorithm for maintaining consistency after edit operations, based on dynamic dependency tracking. That is, we introduce a means to do incremental evaluation of RAGs using dynamic dependency tracking.

1 Introduction

Today's industry-standard language-based editors, like the Eclipse JDT or IntelliJ IDEA for Java, have become indispensable tools for efficient software development. More languages would benefit from this kind of editor support, but to develop an editor from scratch is a major endeavor. Especially, since such interactive tools need support for efficient updating of their internal representation.

To instead generate such editors from declarative descriptions is an area of research that has been extensively investigated. Much focus has been put on editor descriptions building on the formalism of attribute grammars (AGs) [16], and the development of incremental evaluation algorithms for such grammars [22]. This research has resulted in several generator systems [21, 15, 17].

However, a major deficiency of the pure AG approach is its limitation of only supporting attribute dependencies running along the structure of the syntax tree. A restriction to such dependencies, leads to shuffling of large aggregated values along the syntax tree. Also, these restrictions make it difficult to express graph-like properties, like use-def chains or object-oriented inheritance. Many efforts have been made to remove these obstacles through extensions to AGs [4, 20].

One such recent extension, *reference attribute grammars* [11] (RAGs), allows attributes to be references to other abstract syntax tree (AST) nodes. In effect, this allows graphs to be super-imposed on top of the AST, making it easy to, for instance, express use-def chains. RAGs have further been extended with *parameterized attributes*, which

remove the need to shuffle large aggregated values up and down the syntax tree. In addition, RAGs have been extended with so called ReRAGs or *rewrites*, that is, demand-driven transformations depending on attribute values. ReRAGs can, for instance, be used for syntax normalization, or context-based specialization of nodes in the syntax tree. Together, these extensions, tackle the earlier mentioned practicality issues of pure AGs, making it possible to more easily express complex semantics. This expressive property has been clearly demonstrated by the JastAdd system [12], where a full Java compiler has been generated from a RAG specification, performing within a factor of three from handwritten compilers [9].

The graph properties of RAGs make them highly attractive for the generation of interactive services in language-based editors, such as refactorings, name completion, and cross-references [23, 25]. However, there is so far no general algorithm for updating of RAG attributions after edits. Earlier developed incremental algorithms for AGs are based on static analysis of attribute dependencies, where dependencies follow the tree structure of the AST. RAGs, in contrast, are more general and may have attribute dependencies that follow a graph structure emanating from reference attributes, making these static algorithms inapplicable. Instead, RAGs are evaluated dynamically on-demand using a recursive algorithm, originally formulated for AGs [14], during which attribute values may be cached to prevent unnecessary re-computations.

In this report, we explore different approaches for maintaining consistency of RAG attributions after edit operations in an interactive setting. We consider the spectrum of possible approaches: from the crude batch solution, which restores consistency by rebuilding the AST with a complete re-parsing of the source code, to the fine-grained incremental solution, that seeks to retain as many valid attribute values as possible. Previous incremental algorithms statically compute an evaluation order based on static dependencies. These algorithms work under the assumption that all attributes should be evaluated, and that all dependencies are known before evaluation, or a at least a good approximation thereof.

In our setting, neither is true: the exact set of evaluated attributes depends on the syntax tree, and the set of dependencies depends on the values of references attributes, and is therefore not known before evaluation. To find the exact dependencies we are left to using a dynamic algorithm where we construct a dependency graph during evaluation. Once we have this dependency graph, we can react to change and restore consistency after edits. In addition to the batch and the incremental approach, we consider a so called full flush approach, where we restore consistency by removing all computed attribute values, but avoid the re-parsing needed in the batch approach. In this full flush approach, and in the fine-grained incremental approach, we incorporate support for reversal of rewrites.

To our knowledge, this is the first work on consistency maintenance of RAGs. The main contributions of this report are the following:

- Basic notions of consistency and attribute dependencies for RAGs.
- A dynamic dependency tracking approach for dependencies in RAGs.
- An incremental algorithm for consistency maintenance for RAGs.

The rest of this report is structured as follows: We start with a brief introduction to RAGs in Section 2 and a description of the concept of a consistent RAG attribution in

Section 3. This is followed by an explanation of how the dynamic dependency tracking works in Section 4, and how it is used to maintain a consistent RAG attribution in Section 5. Related work is covered in Section 6 and, finally, conclusions and a summary of future work ends the report in Section 7.

2   Reference Attribute Grammars

This section describes RAGs and the problems in applying a statically scheduled attribute evaluation.

2.1   Traditional Attribute Grammars

Attribute grammars (AGs) [16] provide context-sensitive information by associating *attributes* to nodes of an abstract syntax tree (AST) defined by a context-free grammar. The attribute values are defined using so called *semantic functions* of other attributes. Traditionally, there are two kinds of attributes: **inh**erited and **syn**thesized, propagating information downwards and upwards in the AST. The following example shows a synthesized attribute propagating the sum of an addition upwards in the AST:

```
// Grammar: Add ::= Left Right
syn int Add.sum = Left.val + Right.val;
```

In the example, the comment shows a simplified grammar with an AST node `Add` with two children – `Left` and `Right`. The attribute `sum` is declared as an **int**eger defined as the sum of its children's `val` attributes (definitions of `val` are not included in the example).

The definition of an attribute value is called an *equation*, whose left hand side is the defined attribute, and whose right hand side is an application of a semantic function to other attributes. In this case, the semantic function is "+", and it is applied to the attributes `Left.val` and `Right.val`. To propagate information in the reverse direction, downwards, we can use inherited attributes. The following example shows an inherited attribute which is used to compute the nesting depth:

```
// Grammar: Program ::= Block
// Grammar: Block ::= Assign | Block
inh int Assign.nesting;
inh int Block.nesting;
eq Block.Block.nesting = nesting + 1;
eq Block.Assign.nesting = nesting + 1;
eq Program.Block.nesting = 0;
```

Again, the comment provides a simplified grammar, this time with a `Block` node which may have either an assignment (`Assign`) or another block as a child. Each of these possible children are defined to have an inherited attribute `nesting` returning an integer. In contrast to synthesized attributes, the equation for an inherited attribute is not given in the node in which the attribute is declared, instead it is provided by an ancestor in

the AST. With this in mind, we provide equations for the attribute in `Block` – one for the `Assign` child and one for the `Block` child. In these equations, we make use of the `nesting` attribute in `Block` itself to increase the value for its children. Finally, the root of the AST (`Program`) provides an equation for the `nesting` attribute of its `Block` child.

2.2 Reference Attributes and Parameters

In traditional AGs, attribute types are value types, for example, integers and booleans. RAGs extend AGs with *reference attributes*, that is, by allowing attribute values to be direct references to distant nodes in the AST. Reference attributes allow for easy specification of structures that do not follow the AST tree structure, like call graphs and inheritance relations useful in compiler construction. A reference attribute can be used to access information in a distant node, as in the following example:

```
// Grammar: Decl ::= <Type:String> <Name:String>
// Grammar: Use ::= <Name:String>
syn Decl Use.decl = ...
syn String Use.type = decl.Type;
```

Here, we have two AST nodes, `Decl` and `Use`, representing declarations and uses of names in a language. The `Decl` node has two terminals of type `String` while the `Use` node has one. Two synthesized attributes are defined: the first providing a reference to the declaration of a use, and the second providing the type of a use as a string. In the latter equation, `Type` is a terminal of the `Decl` object referred to by the `decl` attribute.

```
// Grammar: Program ::= Block
// Grammar: Block ::= Decl (Use | Block)
// Grammar: Decl ::= <Type:String> <Name:String>
// Grammar: Use ::= <Name:String>
syn String Use.type = decl.Type;
syn Decl Use.decl = lookup(Name);
inh Decl Use.lookup(String name);
inh Decl Block.lookup(String name);
syn boolean Decl.declares(String name) =
    Name == name;
eq Block.Use.lookup(String name) =
    Decl.declares(name) ? Decl : lookup(name);
eq Block.Block.lookup(String name) =
    Decl.declares(name) ? Decl : lookup(name);
eq Program.Block.lookup(String name) =
    ''Unknown Decl''
```

Fig. 1. Lookup example, illustrating parameterized attributes.

To complete the example and provide an equation for the `decl` attribute, we need a means to look up the declaration corresponding to the name of the use. Traditionally, AGs use inherited aggregate-valued attributes, usually named `environment`, to propagate information about all visible names to each use. With RAGs, we can instead provide a distributed symbol table [8] using *parameterized attributes*, another central extension in RAGs. Using parameterized attributes, we define inherited attributes, typically called `lookup`, that take a name as a parameter and return a reference to the appropriate declaration node. Parameterized attributes allow nodes to be queried for information, rather than having to construct large aggregate attribute values with all potentially interesting information.

An example is shown in Figure 1. The top comments in the figure show a simplified grammar with `Decl` and `Use` nodes as presented earlier, a `Block` node with a `Decl` node followed by a `Block` or `Use` node, and a root `Program`. The previous `decl` attribute is here defined by an equation calling an inherited attribute `lookup`, taking the name of the use node as parameter. Three equations are provided for the `lookup` attribute: two in `Block` and one final in `Program`, returning a representation of an "unknown declaration". The equations in `Block` check whether the `Decl` child declares `name`, in which case the `Decl` node is returned, or calls `lookup` of `Block` itself.

2.3 Attribute Evaluation and Dependencies

If the value of an attribute b is used when evaluating the right hand side of the equation for another attribute a, we say that a *depends on* b. For traditional AGs, all attribute dependencies are *static*, in the sense that they can be deduced from the AG alone, without taking the attribute values of a particular AST into account. Most incremental algorithms for AGs, e.g., the well-known optimal algorithm by Reps [22], make use of this fact.
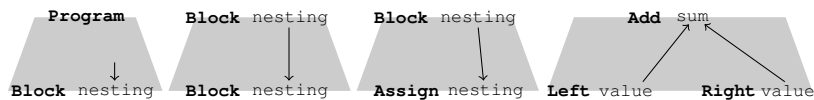
Fig. 2. Example of production-based static dependency graphs. An arrow from b to a indicates that a depends on b.

These algorithms do not apply to RAGs since the possibility to access attributes of distant nodes, via reference attributes, makes the dependencies dependent on the values of individual reference attributes. I.e., some of the dependencies are *dynamic*, in that they can be decided only after actually evaluating some of the attributes.

Considering the examples in Section 2.1, the static evaluation order is quite clear: to compute `Add.sum` we must first compute `Left.val` and `Right.val`, and to compute `Assign.nesting` we must first compute the `nesting` of its parent block.
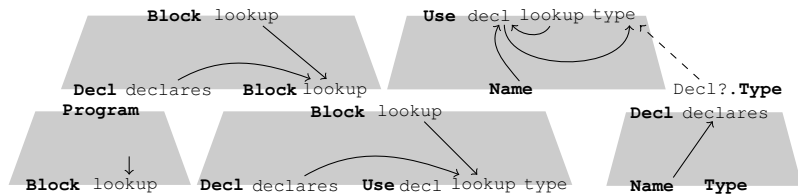
Fig. 3. Production-based static dependency graphs for the lookup example listed in Figure 1. Dashed arrows show dependencies, which cannot be captured by these graphs, due to reference attributes.
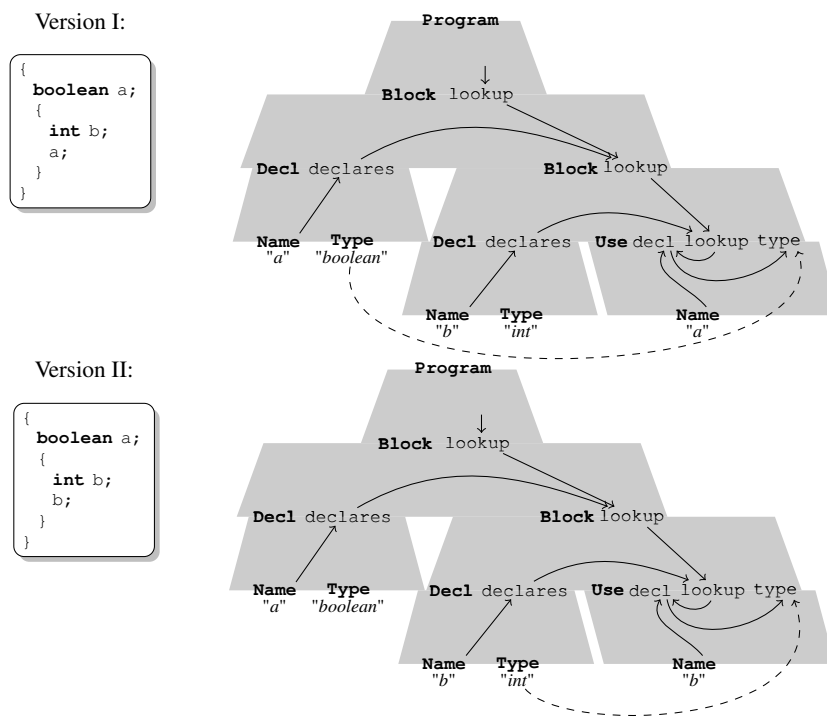
Fig. 4. Two example showing how the dynamic dependency graph is obtained for a derivation tree by pasting together instances of the production-based static dependency graphs. The dashed arrows show dynamic dependencies not captured by these graphs.

These static dependencies can be illustrated using production-based static dependency graphs, also used in [6], as illustrated in Figure 2.

Using the same notation, we can capture the static dependencies of the lookup example presented in Section 2.2, as shown in Figure 3. However, here we have trouble capturing the dependency of the `type` attribute, as indicated by the dashed arrow. We know that `type` depends on the `Type` terminal of a `Decl` node but exactly which node depends on the AST. Figure 4 illustrates these dynamic dependencies with two examples of possible ASTs. The two examples are identical except for the `Use` node: in version I there is a use of the name `a` and in version II there is a use of the name `b`.

2.4 Demand-driven Transformations

In addition to the previous mentioned extensions, reference attributes and parameterized attributes, RAGs support demand-driven transformations called *rewrites* [7]. Rewrites are defined as conditional transformations on node types, and are triggered and evaluated on first access to a node of that type. During traversal of an AST, on each access to a child, potential rewrites will be evaluated on that child before it is returned. At the point where a child is returned, it is considered to be *final*. Initially, only the root node is considered final, but this final "region" of the root node will spread downwards in the AST as new nodes are accessed and evaluated. In practice, this means that rewrites are evaluated top-down, from parent to child, starting at the root of the AST.

There is no limit to the number of rewrites in an AST. In theory, all nodes except the root node may have rewrites, but in practice rewrites are mainly used for smaller transformations. For example, desugaring of syntax or specialization of access nodes based on context. The extent to which rewrites are evaluated depend on which AST nodes are that accessed, and in the set of accessed nodes, the actual set of rewritten nodes depend on which rewrite conditions that have become true. A rewrite condition may contain attribute values, and these values may depend on the syntax tree. That is, a rewrite may happen in one syntax tree but not in another.

In order to incrementally update an AST constructed using rewrites, we need to know the dependencies of rewrite conditions and we need a means to reverse rewrites if their conditions turn to false after an update. Finding the dependencies of rewrite conditions, boils down to the finding of dependencies between reference attributes, and the reversal, or *flushing*, of rewrites, requires knowledge of which value to reverse back to. Regardless of approach, the solution to these problems needs to be integrated with the tracking of attribute dependencies and flushing of attribute values.

3 Consistent Attribution

In this section we describe what is meant by a RAG attribution, and what it means for it to be consistent.

3.1 Attribution

The value of an attribute instance is found by evaluating the right-hand side of its defining equation, and recursively evaluating any attribute instances used in this equation.

For efficiency, the value can be *cached*, i.e., stored at the first access, so that subsequent accesses can return the value directly, rather than have to recompute it [14]. In theory, all attribute values should be cached, to minimize the number of computations. However, in practice, there are performance gains in selecting only a subset of attributes to be cached [24].

We will refer to attributes that store their value as *cacheable* and attributes that do not as *uncacheable*. A cacheable attribute instance is either in the state *cached*, meaning it has a currently stored value, or *decached*, meaning it does not. Initially, all cacheable attributes are decached. Evaluation of a decached attribute computes its value, stores it, and takes the attribute to the cached state. A cached attribute can also be *flushed*, removing the value and taking the attribute back to the decached state.

To be able to reason about edits of the AST, we will regard the child and parent links as *intrinsic* reference attributes, and terminals as intrinsic value attributes. Intrinsic attributes have a stored value that is given a priori, when the AST is constructed. They are similar to cached attributes in that they have a stored value, but different in that they generally have no defining equation, and are not flushed. The collective state of all intrinsic and cacheable attributes is called an *attribution*.

Adding rewrites to a RAG system is then like adding equations to certain intrinsic attributes, in this case child links. Rewrites use the values that are given a priori as *base values* for their evaluation: the rewrite condition will be evaluated based on this value and, if the condition is true, the final value of the rewrite will be constructed using this value. Attribution-wise when using rewrites, the intrinsic child attribute can be considered as a cached attribute with a more complex attributed value: Flushing the rewritten child attribute will bring it back to its base value.

3.2 Consistency

When accessing an attribute we expect that we will get the same value as we would if we evaluated the right-hand side of its defining equation. If this is the case for all attributes, we say that the attribution is *consistent*. For the different kinds of attributes, we define consistency as follows:

intrinsic attributes are by definition consistent, rewrites are here not considered to be pure intrinsic attributes, but cached intrinsic attributes.

uncacheable attributes are by definition consistent

decached attributes are by definition consistent

cached attributes are consistent if all cached attributes they (transitively) depend on are consistent, and if their stored value is equal to the value computed by evaluating the right-hand side of their defining equation.

rewrites are considered to be cached intrinsic attributes. In their decached state, they are consistent if they have their base value, and in their cached state, their consistency follows from the definition for cached attributes.

It follows that an initial AST is consistent since it has no cached attributes. Evaluation of cacheable attributes and caching of their values will keep the attribution consistent, since the expressions in the right-hand side of equations are side-effect free. These conditions also hold for rewrites, which behave like cached attributes.

However, after editing an AST, i.e., changing the value of an intrinsic attribute, cached attributes may have become inconsistent. With knowledge of dependencies, potentially inconsistent attributes can be found and consistency can be restored, either by decaching attributes or by re-caching attributes, i.e., by re-evaluating attributes. In this paper, we focus on decaching of attributes, i.e., flushing.

4 Dependency Tracking

This section describes how dynamic dependencies are found by tracking during evaluation of attributes and rewrites.

4.1 Stack-based Dependency Tracking

Reference attributes are evaluated recursively using an evaluation stack [14]. To find dynamic dependencies, dependency tracking is done during evaluation, recording how attribute instances depend on each other. The example below shows the stack during the evaluation of an attribute `f`.

```
int f = g + h;
int g = h;
int h = 4;
```

The evaluation stack is in effect a call stack, where each call reflects that the callee is dependent on the caller, giving the following dependencies: $f \leftarrow g \leftarrow h$, and $f \leftarrow h$. In this example, the dependencies are static and could have been deduced from the attribute definitions alone. However, in the case of dynamic dependencies, the call stack is needed to capture the exact dependencies. Consider the following example involving a terminal `term` (i.e., an intrinsic attribute) and a conditional equation right-hand side:

```
int m =
  term > 2 ? g : h;
int g = 5;
int h = 4;
```
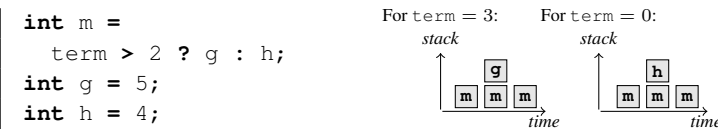
Here, the stack depends on the value of `term`, resulting in different dependencies: for $\text{term} = 3$, we get $m \leftarrow g$, and for $\text{term} = 0$ we get $m \leftarrow h$. Here, a static approach would lead to an approximation of the exact dependencies: $m \leftarrow \{g, h\}$.

The need for dynamic dependencies is even more apparent in examples using reference attributes. Figure 5 shows the call stacks for the examples in Figure 4. Here, we show the stacks only at points during evaluation when the stack is about to decrease. Version I of the program gives rise to the following dependencies: $\text{Use}_a.\text{type} \leftarrow \text{Use}_a.\text{decl} \leftarrow \{\text{Use}_a.\text{Name}, \text{Use}_a.\text{lookup}_a\}, \text{Use}_a.\text{lookup}_a \leftarrow \{\text{Decl}_b.\text{declares}_a, \text{Block}.\text{lookup}_a\}, \text{Block}.\text{lookup}_a \leftarrow \text{Decl}_a.\text{declares}_a$, and Version II give rise to the following dependencies: $\text{Use}_b.\text{type} \leftarrow \text{Use}_b.\text{decl} \leftarrow \{\text{Use}_b.\text{Name}, \text{Use}_b.\text{lookup}_b\}, \text{Use}_b.\text{lookup}_b \leftarrow \text{Decl}_b.\text{declares}_b$. We can note that Version II induces fewer dependencies, due to the closeness of the declaration to the use.

Version I:

Version II:
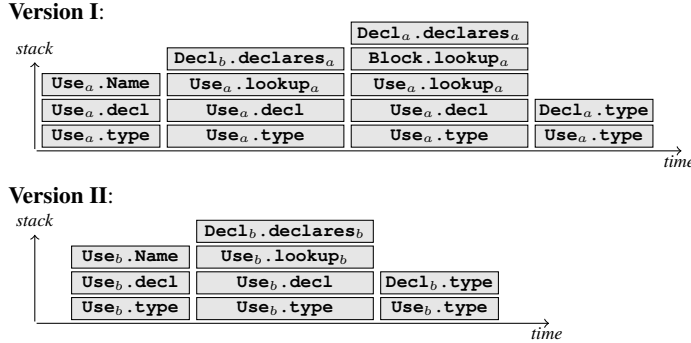
Fig. 5. The stacks correspond to the evaluation stack at points during the evaluation where the stack is about to decrease. The examples being evaluated are taken from Figure 4.

4.2 Tracking of Attribute Dependencies

Each attribute is implemented as a method containing evaluation code. For cacheable and intrinsic attributes this code accesses the stored state (computing and storing the value in case the cacheable attribute was previously decached). For uncacheable attributes, the evaluation code simply computes the value according to the equation right-hand side.

We represent each intrinsic and cacheable attribute instance a by a *dependency handler object* that keeps a set of dependents, i.e., a set of references to handler objects for the attribute instances that depend on a. Initially, before any attribute evaluation starts, all dependents sets are empty. For cacheable parameterized attributes, the cached value is stored for each used combination of parameter values, and a handler is created for each new such combination used.

To track dependencies during evaluation, we instrument the evaluation code of these attributes to maintain a global stack of handlers, adding and removing the handler for the evaluated attribute instance to this stack as the evaluation code is entered and exited. Furthermore, at each evaluation code entry, the previous top of stack is added to the dependents set of the new top of stack.

4.3 Tracking of AST Structure and Rewrites

Accesses to child and parent links also give rise to dependencies. Even more so, when a child has rewrites, since then there is a rewrite condition potentially depending on attribute values. Dependencies for these rewrite conditions need to be tracked like for any cached attribute. In fact, this tracking needs to be done for all intrinsic attributes that may be changed. Rewritten children may be changed due to an update of a dependency, and non-rewritten children may be changed due to an AST edit. In this sense, we may consider parent and child links as reference attributes, with all child links of a node as one parameterized child attribute.

The dependency graphs in Figure 4 are thus actually incomplete. In particular, dependencies to child links are missing, since equations may return references to children.

For example, the equations for `lookup` in `Block` in Figure 1 may return a reference to the `Decl` child. Possibly less apparent, is that each time an attribute of a child is used in an equation, there is actually also a dependency on the child link. Also, each inherited attribute actually depends on the parent link up to the node holding its defining equation.

5 Consistency Maintenance

During development, a developer makes changes to a program. These changes will result in a sequence of AST edits handled by the editor. After each edit, the previously consistent attribution of the AST, corresponding to the program being edited, may have become inconsistent. The goal of consistency maintenance is to bring the AST into a consistent state after each edit. In practice, this means keeping track of dependencies and notifying affected cached attributes of change when needed, so that these attribute values may be flushed.

In our setting, we assume the following: 1) that the AST is initially syntactically correct with a consistent attribution before any edits have taken place, 2) that the AST is syntactically correct after an edit, and 3) that any new nodes or subtrees added to the AST are consistent before the addition, typically with all cacheable attributes in the decached state.

5.1 AST Edits

There are several possible AST edits, for example, a child link may be replaced, added, removed or inserted, or a terminal may be replaced. Edits to child links may be considered to be the most complex edits given that its an edit to a list structure, where succeeding children may be affected. In contrast, a parent link or an intrinsic terminal value, like the name of a variable, can only be replaced.

As an example, consider the removal of a child k in a list of n children (from 0 to $n-1$), dependants to the removed child must be notified, but also dependants to child $k+1$ to $n-1$, since these children are being moved as a consequence. In comparison, the replacement of a child is a simpler edit, since then only the dependencies of the child being replaced needs to be notified.

In general, an edit can be described as changing the values of a set I of intrinsic attributes, i.e., parent and child links, and terminals, followed by a notification of dependencies which are (transitively) dependent on the set I. Clearly, these are the set of cached attributes that can become inconsistent due to the edit.

Notably, edits to rewritten children are not very different than edits to children without rewrites. For example, if a rewritten child is replaced, then the new value is used as the base value of the rewrite, and the rewrite is considered to be decached.

5.2 Flushing

If a cached attribute is notified of a change, with the current approach, it should be flushed. A flush means marking the attribute as decached and returning the value of the attribute to its base value given at AST construction.

Base values for rewrites To flush a rewrite, the base value needs to be stored. The trivial approach for storing base values is to make an *base copy* of the value of a rewrite before it is evaluated. This approach is, however, quite memory demanding, especially when rewrites are nested, or if rewrites occur for larger subtrees, and not useful in practice. A slightly trimmed alternative, is for rewrites to share base copies when possible. That is, nested rewrites, or *inner rewrites*, share their base copy with enclosing rewrites, or *outer rewrites*. In practice, this means that during evaluation outer rewrites make copies while inner rewrites do not.

Flushing of rewrites Given that we use the slightly trimmed copying of base values, we get a situation where we have flushing of inner and outer rewrites. The flushing of an outer rewrite, then includes the following steps: 1) setting the rewrite to decached, 2) setting all inner rewrites to decached, 3) restoring the value of the base value, and 4) notifying dependencies of the rewrite. In contrast, the flushing of an inner rewrite involves the following steps: 1) setting the rewrite to decached, and 2) locating the enclosing outer rewrite and notifying it of change.

5.3 Algorithm for Consistency Maintenance

A general technique for maintaining RAG consistency after edits is to flush all attributes that transitively depend on the edited set of intrinsic attributes I. We represent I by the set of corresponding handler objects, handling dependencies and acting as nodes in the dependency graph. The algorithm for restoring consistency can then be expressed as follows:

RESTORE-CONSISTENCY(I)
1 **for** each intrinsic handler $h \in I$
2   **do** TRANSITIVE-FLUSH(h)

FLUSH(h)
1 ▷ Flush the cacheable
2 attribute handled by h

TRANSITIVE-FLUSH(h)
1 $deps \leftarrow dependents(h)$
2 $dependents(h) \leftarrow \emptyset$
3 **for** each cacheable handler $h \in deps$
4   **do** FLUSH(h)
5   TRANSITIVE-FLUSH(h)

Notably, the *dependents* set of h is cleared before dependents are transitively flushed. This clean up prevents the algorithm from going into endless recursion if there are circular dependencies between attributes. Circular attributes [10, 19] are excluded from the examples in this paper, but are nonetheless supported by this approach.

5.4 Aborting Transitive Flush

The simple algorithm above does not take the attribute values into account: if an attribute happens to have the same value after the change, all its dependent attributes will be flushed. In principle, it would be possible to abort the transitive flush for attributes that are known to have the same value after the change. However, this would require that a new value is computed before the flush is done. To compute this new value, we cannot, however, use the cached values it depends on since they might be inconsistent.

In principle, new values can be computed without using cached values, i.e., by evaluating corresponding equations rather than using cached values. However, in general, this can become extremely expensive, since evaluating attributes without using caching may lead to exponentially growing evaluation times. Therefore, in general, such abortion is not likely to be profitable. In specific cases, however, it can still pay off.

In particular, if an attribute does not depend (transitively) on any cacheable attributes, it can be evaluated in the same amount of time before or after the flush. We call such an attribute *cache-independent*. To take such cache-independent attributes into account, the algorithm for TRANSITIVE-FLUSH would be altered as follows:

TRANSITIVE-FLUSH(h)

```
 1   deps ← dependents(h)
 2   dependents(h) ← ∅
 3   for each cacheable handler h ∈ deps
 4       do if CACHE-INDEPENDENT(h)
 5           then
 6                   value_new ← EVALUATE-ATTRIBUTE-OF(h)
 7                   if value_new! = CACHED-VALUE-OF(h)
 8                     then
 9                           SET-VALUE-OF(h, value_new)
10                           FLUSH(h)
11                           TRANSITIVE-FLUSH(h)
12
13           else
14                   FLUSH(h)
15                   TRANSITIVE-FLUSH(h)
```

The identification of attributes that are cache-independent could either be done statically, by annotating the attributes as such (and checking this property), or dynamically, by keeping track of which attributes are cache-independent during the dependency tracking.

In practice, abortion of transitive flush will be particularly important for parameterized attributes that check terminal values, like the attribute `Decl.declares` in Figure 1. Suppose the `Name` terminal of a `Decl` is edited. Calls to `Decl.declares` will have the same value for all parameters that are different from the old and new `Name` terminal. In practice, there may be many such calls due to the block structure in a program.

5.5 Implementation

The algorithm described in Section 5.3 has been implemented and tested in the JastAdd system [1]. The implementation supports incremental consistency maintenance for all AST edits allowed by the system, that is, removal, addition and insertion of children. The JastAddJ extensible Java compiler [9] has been used as a test platform for the implementation, and all attributes and transformations occurring in the JastAddJ compiler are supported. This includes synthesized, inherited, and parameterized attributes,

higher-order attributes [26], circular attributes [10], and rewrites [7]. The abortion of transitive flush is currently under implementation.

6 Related Work

There is an extensive amount of previous work on the incremental evaluation of attribute grammars with the goal of supporting interactive language-based editors.

For classical AGs, Demers, Reps, and Teitelbaum presented a two-pass algorithm, which first nullifies dependent attributes and then reevaluates them [5]. The dependency analysis is done based on the static dependencies in the AG. Reps improved this approach by presenting an optimal *change propagation* algorithm [22], where old and new attribute values are compared, and avoiding to propagate the change to dependents if the values are equal. This algorithm was proven optimal in the sense that it does work proportional to the number of affected attributes, i.e., the attributes that actually do get new values.

A problem with classical AGs is that, even if the algorithm is optimal, the number of affected attributes becomes very large: to handle complex computations, large amounts of information, typically symbol tables, are bundled together into single aggregate-valued attributes that are copied throughout the AST. A small change to one declaration thereby causes all the copies to become affected, even if very few attributes actually make use of the changed declaration. A number of different solutions to these problems were proposed, focusing on special support for these aggregate-valued attributes, e.g., [13].

Other work focused on extending the classical AGs themselves, to make the complex computations more straightforward to express. This includes work by Poetzsch-Heffter [20], Boyland's Remote AGs [2], and our RAGs [11]. All these formalisms make use of some kind of mechanism for remote access of attributes, thereby inducing dependencies that are difficult to deal with by static analysis of the AG.

In Boyland's Remote AGs, AST nodes can have local objects with fields, and attributes can be references to such objects. This allows graph structures to be built, and equations can read the fields of an object remotely. In Remote AGs, an AST node can also have collection fields, which are aggregate-valued attributes like sets, and where the definition can be spread out on multiple sites in the AST, each contributing to the collection, e.g., adding a particular element.

Boyland developed a static algorithm for the evaluation of Remote AGs, where control attributes are automatically added to take care of scheduling of remote attributes and collections [2]. He also developed an incremental algorithm for remote AGs that combines static scheduling for "ordinary" attributes with dynamic scheduling for remote attributes and collections [3]. Preliminary experiments with this algorithm on a small procedural toy language and synthetic benchmark programs, showed substantial speedups for edits of declarations as compared to reevaluating all attributes.

Boyland uses collection attributes for solving name analysis problems, representing local symbol tables as collections of declaration objects. Although JastAdd does support collection attributes (whose incremental evaluation is not treated in this paper), name

analysis in RAGs is typically solved using parameterized attributes, as in the examples in this paper.

All the algorithms mentioned above are based on data-driven attribute evaluation, i.e., all attributes are evaluated, regardless of if they are actually used or not. After an edit, all affected attributes are updated. In RAGs, the attribute evaluation is instead demand-driven [14], evaluating only attributes whose values are actually needed. After an edit, we decache all attributes that might be inconsistent. This might well be a larger set than the actually affected set. However, because aggregate values are avoided, we do not get the inflated affected sets that classical AGs suffer from.

7 Conclusion and Future Work

We have presented a basic fine-grained algorithm for incremental evaluation of RAGs. The algorithm restores consistency after edits to the abstract syntax tree, and is based on dynamic dependency tracking to flush all possibly affected attributes. We have also discussed how the flush propagation can be aborted by comparing old and new attribute values, and how this can be done without extra cost for cache-independent attributes.

As future work, we will evaluate the algorithm experimentally, and investigate several optimizations. The basic algorithm has extensive overhead due to the fine-grained dependency information that is maintained. We expect that more coarse-grained approaches will perform better in practice, and we are investigating approaches based on partitioning the AST and the attribute set. Furthermore, some practical RAGs such as the JastAddJ extensible Java compiler [9], make use of large attribute values for collecting local declarations into a map data structure. This is done in order to avoid that multiple queries for declarations repeatedly search the AST. Unfortunately, this use of large values causes the same kind of dependency imprecision that ordinary AGs suffer from. To obtain both the higher performance of the maps and fine-grained dependencies, we are investigating the introduction of a new kind of *bound* parameterized attribute, where all results (for all possible parameter values) can be computed at the first call to the attribute.

There is also a potential for improving the evaluation of rewrites. In our current implementation, attributes depending on rewritable parts of the AST are not cached until those parts are rewritten [7]. By using the incremental evaluation it might be possible to cache such attributes already during rewrite. Other possible improvements include support for incrementally updating rather than recomputing affected higher-order attributes [26], and support for incremental evaluation of collection attributes [18].

References

1. JastAdd, 2012. http://jastadd.org.
2. John Boyland. Analyzing direct non-local dependencies in attribute grammars. In *Compiler Construction, 7th International Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 31–49. Springer, 1998.
3. John Boyland. Incremental evaluators for remote attribute grammars. *Electr. Notes Theor. Comput. Sci.*, 65(3), 2002.

4. John Tang Boyland. Remote attribute grammars. *Journal of the ACM*, 52(4):627–687, 2005.

5. Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL*, pages 105–116, 1981.

6. Pierre Deransart, Martin Jourdan, and Bernard Lorho. A Survey on Attribute Grammars, Part I: Main Results on Attribute Grammars. Technical report, INRIA, Rocquencourt, France, January 1986. Rapport de Recherche 485.

7. Torbjörn Ekman and Görel Hedin. Rewritable reference attributed grammars. In Martin Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 144–169. Springer, 2004.

8. Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE 2005)*, volume 4143 of *LNCS*. Springer, 2006.

9. Torbjörn Ekman and Görel Hedin. The Jastadd Extensible Java Compiler. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 1–18. ACM, 2007.

10. Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 85–98, New York, NY, USA, 1986. ACM.

11. Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

12. Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

13. Roger Hoover and Tim Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 39–50. ACM, 1986.

14. Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 1984.

15. Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the fnc-2 attribute grammar system. *SIGPLAN Notices*, 25(6):209–222, 1990.

16. Donald E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (1971).

17. Matthijs F. Kuiper and João Saraiva. Lrc - a generator for incremental language-oriented tools. In *CC*, pages 298–301, 1998.

18. Eva Magnusson, Torbjörn Ekman, and Görel Hedin. Demand-driven evaluation of collection attributes. *Automated Software Engineering*, 16(2):291–322, 2009.

19. Eva Magnusson and Görel Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Science of Computer Programming*, 68(1):21–37, 2007.

20. Arnd Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34(10):737–772, 1997.

21. Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. *SIGPLAN Notices*, 19(5):42–48, May 1984.

22. Thomas W. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *POPL*, pages 169–176, 1982.

23. Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and Extensible Renaming for Java. In Gregor Kiczales, editor, *23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*. ACM Press, 2008.

24. Emma Söderberg and Görel Hedin. Automated selective caching for reference attribute grammars. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering (SLE 2010)*, volume 6563 of *LNCS*, pages 2–21, 2010.

25. Emma Söderberg and Görel Hedin. Building semantic editors using jastadd. In *Proceedings of the of the 11th Workshop on Language Descriptions, Tools and Applications, LDTA 2011*. ACM, 2011.
26. Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.