Concurrent Circular Reference Attribute Grammars (Extended Version)

Öqvist, Jesper; Hedin, Görel

2017

Document Version:
Publisher's PDF, also known as Version of record

Concurrent Circular Reference Attribute Grammars (Extended Version)

Jesper Öqvist

Görel Hedin

Technical report, LU-CS-TR:2017-254
ISSN 1404-1200, Report 103, 2017

Lund University

Concurrent Circular Reference Attribute Grammars (Extended Version)

Jesper Öqvist
Lund University
Sweden
jesper.oqvist@cs.lth.se

Görel Hedin
Lund University
Sweden
gorel.hedin@cs.lth.se

Abstract

Reference Attribute Grammars (RAGs) is a declarative executable formalism used for constructing compilers and related tools. Existing implementations support concurrent evaluation only with global evaluation locks. This may lead to long latencies in interactive tools, where interactive and background threads query attributes concurrently.

We present lock-free algorithms for concurrent attribute evaluation, enabling low latency in interactive tools. Our algorithms support important extensions to RAGs like circular (fixed-point) attributes and higher-order attributes.

We have implemented our algorithms in Java, for the JastAdd metacompiler. We evaluate the implementation on a JastAdd-specified compiler for the Java language, demonstrating very low latencies for interactive attribute queries, on the order of milliseconds. Furthermore, initial experiments show a speedup of about a factor 2 when using four parallel compilation threads.

CCS Concepts • **Theory of computation → Concurrency**; **Parallel algorithms**; • **Software and its engineering → Compilers**; **Translator writing systems and compiler generators**; *Concurrent programming structures*;

Keywords Reference Attribute Grammars, Concurrency, Parallelization, Memoization, Circular Attributes

1 Introduction

Reference Attribute Grammars (RAGs) [9] have proven useful for generating extensible compilers for languages like Java [6, 28] and Modelica [1]. They are supported in several attribute grammar systems, for example, JastAdd [9], Silver [27], Kiama [22], JavaRAG [8], and RACR[3].

Typically, attributes are evaluated sequentially, in a single thread. To extend such evaluation to several threads requires a global evaluation lock, leading to attribute value queries in one thread being blocked by ongoing attribute evaluation in other threads. This blocking can cause long latencies in interactive tools, like Integrated Development Environments (IDEs). In IDEs, it is typically desired to keep the response time below 0.1 seconds, to ensure that users perceive the tool as reacting instantaneously [20]. By using concurrency, an interactive task can be performed within this time limit even while longer-running analysis tasks run in the background. Concurrent evaluation also enables parallelization, which could speed up regular compilation tasks.

A challenge in supporting concurrent attribute evaluation is to safely handle circular attributes, i.e., attributes that are in a dependency cycle, and evaluated by fixed-point iteration [7, 11, 18]. Circular attributes are useful for many complex problems in compilers, like definite assignment (a dataflow problem) and type inference. However, concurrent evaluation in the presence of circular attributes is nontrivial because approximations for each attribute value in a dependency cycle need to be stored and updated safely between multiple threads. An implementation based on locking individual attributes would cause deadlocks whenever two or more threads attempt to evaluate (and lock) attributes on the same dependency cycle.

We solve the latency problem by developing concurrent algorithms for RAG attribute evaluation. The algorithms are lock-free and safe to use with circular attributes without risk of deadlock. Our algorithms support other common extensions to RAGs like higher-order attributes, collection attributes, and attribute-controlled rewrites. The supported attribute kinds are described in Section 2.

Our contributions are:

- Lock-free concurrent evaluation algorithms for extended RAGs (Section 4), including circular attributes (Section 5).
- Correctness proofs for the concurrent attribute evaluation algorithms for extended RAGs (Section 4) and for the circular attribute evaluation (Appendix A).
- Relaxed requirements on circular attributes to make their specification simpler (Section 6).
- Implementation of the concurrent algorithms in the JastAdd metacompiler. For a correctly specified JastAdd project, our implementation can be used without further modification. We validate this by using our implementation on a full Java compiler specified with JastAdd attributes (Section 7.1). We also validate the implementation by using it in an interactive tool for exploring Java programs.
- Empirical evaluation of attribute evaluation latency in interactive tasks, comparing our concurrent implementation to a sequential one. The results show significant

This is an extended version of a paper to appear at SLE 2017.

latency improvements using concurrent evaluation (Section 7.2).
- Empirical evaluation of speedup using parallel attribute evaluation, with results of about a factor 2 in speedup when compiling large Java programs (Section 7.3).

2 Circular Reference Attribute Grammars

In a RAG [9], an abstract grammar is viewed as a set of node classes representing the nonterminals of the grammar. Attributes are specified for node classes, and an Abstract Syntax Tree (AST) defined by the grammar has attribute instances attached to its nodes. We refer to attribute instances as simply attributes, unless otherwise noted.

An attribute is defined by a *semantic function* of an AST node. For example, an attribute x with semantic function f can be written as $x = f(n)$, where n is an AST node. The attribute x belongs to either n or one of its children. If x is an attribute of n, we say that x is *synthesized*, and if it is an attribute of one of n's children, we say that x is *inherited*.[1] Synthesized attributes are typically used for propagating information upwards in the AST, such as propagating the type of an expression to its surrounding statement. Inherited attributes are typically used for propagating information downwards in the AST, such as propagating the set of visible declarations from a block to its inner statements.

Unlike the original definition of attribute grammars by Knuth [15], RAGs allow attributes to be *references* to nodes in the AST. For example, a variable access may have a reference to the declaration of the variable. In RAGs, the semantic functions can access information in remote nodes via reference attributes. For example, the declared type of a variable can be accessed via a reference to the declaration.

RAGs also support *parameterized* attributes, where the semantic function depends not only on the node, but also on arguments supplied when using the attribute. A typical example is comparing two types by a parameterized attribute on one of the types, where the type to compare against is the argument.

The typical way to evaluate a Knuth AG is to statically analyze attribute dependencies, and use a static schedule to evaluate all attributes in dependency order, for example using Ordered AGs [14]. For RAGs, this does not work, since attribute dependencies are not statically known due to the use of reference attributes and parameterized attributes. Instead, RAGs use recursive dynamic attribute evaluation that memoizes attributes to make subsequent accesses fast [12].

Extensions to RAGs supported in the JastAdd system include circular attributes, higher-order attributes, collection attributes and attribute-controlled rewrites.

[1] It can be noted that the attribute grammar concept of *inherited* is independent of the object-oriented concept with the same name.

Circular attributes, useful for data-flow problems, may depend upon themselves, and are evaluated using a fixed-point iteration algorithm [7]. For RAGs, the fixed-point algorithm is recursive [18].

A *higher-order* attribute is an attribute whose value is a fresh AST subtree, and which can itself have attributes [26]. Example uses are computation of transformed structures and macro-like expansions. In RAGs, it is important that even if a higher-order attribute is evaluated more than once, each time creating a fresh subtree, only one result reference should become visible to the rest of the program.

Collection attributes allow compound values to be defined by a combination of contributions in an AST [2, 17]. A typical use is to collect all error messages in the program.

Attribute-controlled *rewrites* allow AST nodes to be conditionally rewritten [5]. They have recently been shown to be equivalent to circular higher-order attributes [23]. Typical uses include specialization of nodes depending on context, for example, replacing a field access node by a static or instance field access, depending on its declaration.

For attribute evaluation to work correctly, certain well-formedness conditions must be met. The following are of particular importance for concurrent evaluation:

WF1: Pure semantic functions. Each semantic function must be *observationally pure* [19], meaning that it always computes the same value, does not modify the AST, and does not rely on external mutable data.

WF2: Terminating semantic functions. Each semantic function must terminate, given that access to other attributes terminates.

WF3: Circular attributes are computable. To guarantee a computable least fixed point, we require the semantic function of circular attributes to be monotonic and yield values in a lattice of finite height. This is the condition used by Jones [11].

3 Correctness

We will prove correctness properties for the concurrent attribute evaluation algorithms presented in this report. The main correctness properties we wish to prove are soundness and lock-freedom.

Soundness means that the algorithms compute the correct attribute values. Lock-freedom is important to ensure that the algorithms do not cause deadlocks when used in circular attribute evaluation.

To show lock-freedom we prove a stronger progress guarantee: that the algorithms terminate in a finite number of steps. This means that most of our algorithms are actually wait-free. However, some of the data structures used in our implementation are not wait-free, only lock-free. If wait-free implementations of those data structures were used, our algorithms would be wait free.

For higher-order attributes we must prove an additional correctness property: that the attribute can only compute a single reference. A higher-order attribute creates a new AST node object each time it is computed, but only one result node must be attached to the AST and become visible to the rest of the program. By proving that the evaluation algorithm for higher-order attributes only allows a single reference to be computed, we ensure that only a single node object is shared between multiple threads.

4 Non-Circular Attribute Implementation

A recursive attribute evaluator computes an attribute by calling its semantic function, and memoizing the result for fast future accesses. Calling the semantic function leads to other attributes being evaluated recursively. For non-circular attributes, implementation of lock-free concurrent evaluation is fairly straightforward. The main problem is to make sure that memoization is done in a thread-safe way. For higher-order attributes, it must be ensured that all threads will share the same resulting reference to the new subtree.

A template attribute evaluator algorithm, Eval, is shown in Algorithm 1. The Eval procedure takes as parameter an attribute instance to be evaluated. Computation and memoization have been abstracted out of Eval as four separate procedures:

Compute Compute the value of an attribute.
Memoized Test if an attribute has been memoized.
Store Memoize a value for an attribute.
Load Retrieve a previously memoized value of an attribute.

Algorithm 1 Template attribute evaluation algorithm for memoized non-circular attributes.

```
procedure Eval(x)                          ▷ Evaluate attribute x.
    if Memoized(x) then                    ▷ Test if already memoized.
        return Load(x)                     ▷ Return memoized value.
    else
        u ← Compute(x)                     ▷ Compute attribute value.
        return Store(x, u)                 ▷ Memoize and return result.
    end if
end procedure
```

The Eval procedure can be trivially translated to Java as a method of an AST node class that the attribute it evaluates was declared on [9]. In the following sections we present Java methods implementing the procedures used in Eval for non-circular attributes.

The attribute instance is not explicitly passed to the methods, as it is not a concrete Java object. Instead, the implicit `this` parameter of the Java methods separates the evaluation of different attribute instances.

By proving that the procedures used by Eval fulfill certain requirements we can show that the resulting Eval implementation is sound and lock-free. For soundness, the procedures must fulfill the following soundness requirements:

Memoization Requirement
In one thread, if Memoized(x) returns true before Load(x), then Load(x) returns a value v stored by some thread executing Store(x, v).

Store Requirement
Executing Store(x, _) returns some value v stored by some thread executing Store(x, v).

We will later show that our implementations of the memoization procedures fulfill these requirements. Given that these requirements are fulfilled, Eval computes the right value for any non-circular attribute:

Theorem 4.1 (Eval Sound). *If Memoized, Store and Load fulfill the Memoization Requirement and Store Requirement, then, for an attribute x, Eval(x) (Algorithm 1) computes the value of x.*

Proof. Consider a thread that executes Eval(x). The **if**-statement in Eval has two branches:

- If Memoized(x) returned true, then by the Memoization Requirement the returned value, v, was stored by some call Store(x, v). Because all calls to Store(x, _) store a computed value of x, and because x is well-formed (WF1), the returned value is the value of x.
- Otherwise, Memoized(x) returned false. The returned value is the result of Store(x, u). According to the Store Requirement, the result v was stored by some call Store(x, v). Because all calls to Store(x, _) store a computed value of x, the returned value is the value of x.

□

For a non-higher-order attribute the semantic function always computes an identical value. However, for a higher-order attribute this is not the case, as the attribute computes a freshly created AST node reference. It is important that only one reference becomes visible to the rest of the program. For higher-order attributes we add the following soundness requirement:

Higher-Order Memoization Requirement
For a higher-order attribute instance x, each call to Store(x, _) returns a single reference, and any call to Load(x) that happens after some call to Store(x, _) returns the same value as Store(x, _).

The requirement ensures that Eval only returns a single result reference.

Theorem 4.2. *Consider a higher-order attribute instance x. If Memoized, Store and Load fulfill the Higher-Order Memoization Requirement, then Eval(x) (Algorithm 1) returns a single reference.*

Proof. Consider a thread that executes Eval(x). The **if**-statement in Eval has two branches that return either the result of Store(x, _) or Load(x). The first case is trivially true, since Store(x, _) is required to return a single reference. In the second case, Memoized(x) returned true, so according to the Higher-Order Memoization Requirement, Load returns a single reference. Additionally, the Higher-Order Memoization Requirement specifies that Load(x) returns the same reference as Store(x, _), so only one reference can be returned from Eval(x). □

To ensure that Eval is lock-free, we require that Compute, Memoized, Store, and Value are lock-free:

Theorem 4.3 (Eval Lock-Free)**.** *Consider a non-circular attribute instance x. If Compute, Memoized, Store and Load are lock-free, then Eval(x) (Algorithm 1) is lock-free.*

Proof. Eval itself uses no iteration and no self-recursion (because x is not circular). Thus, Eval is lock-free because all called procedures are lock-free by assumption. □

4.1 Synthesized and Inherited Attributes

For synthesized attributes, the Compute procedure is a direct translation of the semantic function into an executable form, where other attribute uses are replaced by calls to Eval. Because JastAdd attributes are specified with Java code, the translation of the Compute procedure is just a Java method containing the code of the semantic function.

For inherited attributes, the Compute procedure that computes an inherited attribute on a node n must find the semantic function for the inherited attribute. This is done by accessing the parent of n, and determining which semantic function should be computed for the child node n.

The equation used for an inherited attribute a on a node n depends on the child position of n in its parent node. The original definition of inherited attributes by Knuth requires the equation for an inherited attribute to be defined on the direct parent of the node that the attribute belongs to. An example of this is the b attribute in Figure 1: it belongs to Z, and the equation for it is defined by X. RAGs allow the equation to be defined further up in the AST, as in Figure 2, where the equation for b on X is propagated down to Z. Conceptually, inherited attributes in RAGs work by using implicit copy attributes on all intermediate nodes between the node defining the equation for the attribute, and the node that owns the attribute, as illustrated in Figure 3.

Concurrent memoization for synthesized and inherited attributes can be implemented using a simple cache field and volatile flag in Java, as shown in Listing 1. It is not necessary to exclude concurrent `store` calls, because well-formed synthesized and inherited attributes always compute the same value, so concurrent `store` calls will only store the same value. Note, however, that the order of assignments inside the `store` procedure is critical: the write to the volatile cache flag must be after the write to the value field to ensure that the value is safely published to other threads.

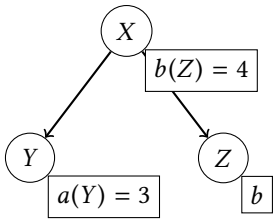

Figure 1. An AST, rooted at the node X. Nodes Y and Z are children of X. The attribute a is synthesized: it belongs to Y and its equation is defined on Y. The attribute b is inherited: it belongs to Z, but its equation is defined on X.

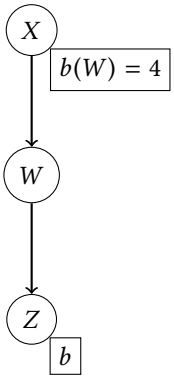

Figure 2. An example of elided inherited attribute equations: X defines the value of b for Z, despite X not being the direct parent of Z.

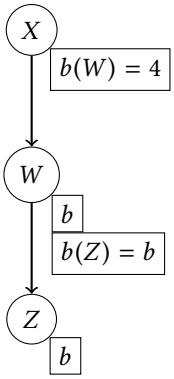

Figure 3. An implicit copy attribute is used on W to copy the value of the attribute b from X to Z.

4.1.1 Correctness

For synthesized attributes, the Compute procedure is lock-free, due to well-formedness condition WF2. For inherited attributes, locating the semantic function is implemented by a loop that iterates over parent references in the AST. Since

Listing 1. Simple attribute memoization.

```
T value;
volatile boolean cached = false;

boolean memoized() {
  return cached;
}

T store(T v) {
  value = v;
  cached = true;
  return v;
}

T load() { return value; }
```

Listing 2. Parameterized attribute memoization.

```
ConcurrentMap map =
    new ConcurrentHashMap();

boolean memoized(Object args) {
  return map.containsKey(args);
}

T store(Object args, T v) {
  map.putIfAbsent(args, v);
  return map.get(args);
}

T load(Object args) {
  return (T) map.get(args);
}
```

Listing 3. Non-parameterized higher-order attribute memoization.

```
AtomicReference value =
    new AtomicReference(nil);

boolean memoized() {
  return value.get() != nil;
}

T store(T v) {
  value.compareAndSet(nil, v);
  return value.get();
}

T load() {
  return (T) value.get();
}
```

the AST has a finite height, this loop terminates in a finite number of steps. The number of children for any node in an AST is finite, so each iteration of the loop performs a finite amount of work and is thus lock-free. The procedure for finding the equation for an inherited attribute is thread-safe because the AST is not modified after construction, making it effectively immutable during attribute evaluation.

Theorem 4.4. *The methods in Listing 1 fulfill the Memoization Requirement.*

Proof. The Memoization Requirement entails that if `memoized` returns TRUE before `load`, then `load` returns a value `v` stored by some execution of `store(v)`.

The `cached` flag starts out as FALSE. Hence, `memoized` returns TRUE only after some write has set `cached` to TRUE. The `cached` flag is declared as `volatile`. According to the Java Memory Model, a previous write to `value` must therefore be visible to the thread that observed `cached` having the value TRUE, so a following call to `load` will return this value or some other value of x stored by `store`. □

Theorem 4.5. *The methods in Listing 1 fulfill the Store Requirement.*

Proof. The Store Requirement entails that `store(u)` returns a value `v` stored by some execution of `store(v)`. The requirement is fulfilled because `store(u)` always returns `u`. □

Theorem 4.6. *The methods in Listing 1 are lock-free.*

Proof. All of the operations used are lock-free according to the Java specification. There exists no iteration or recursion, hence the methods are lock-free. □

4.2 Parameterized Attributes

Synthesized and inherited attributes can be optionally parameterized. To compute a parameterized attribute, additional arguments are sent to the COMPUTE procedure.

Parameterized attributes are memoized by mapping argument values to result values. To this end we use a concurrent map. For unary attributes, the single argument value is used as map key, and for 2+ arity attributes a list of the argument values is used as map key. Map keys of primitive type are converted to object types like `java.lang.Integer`.

Our parameterized attribute implementation, in Listing 2, uses the class `ConcurrentHashMap` from the Java standard library as map structure. The method `putIfAbsent` is used in `store` to atomically associate an argument list with an attribute value. Using `putIfAbsent` allows only one thread to succeed in updating the map for any given argument list.

The `ConcurrentHashMap` impelementation is lock-free but the rest of our implementation is wait-free. If full wait-freedom is required, a Java implementation of a wait-free map could be used.

4.2.1 Correctness

For parameterized attributes, the memoization methods in Listing 2 are used.

Theorem 4.7. *The methods in Listing 2 fulfill the Memoization Requirement.*

Proof. The Memoization Requirement entails that if `memoized(p)` returns TRUE before `load(p)`, then `load(p)` returns a value `v` stored by some execution of `store(p,v)`.

The `map` is initially empty, with no key associated to a value. A call to `map.containsKey(p)` then only returns TRUE if some call to `putIfAbsent(p,_)` inserted a value for the given key previously. Keys are never disassociated in the map, so `map.get(p)` is guaranteed to return an inserted value `v`, which was inserted by `store(p,v)`. □

Theorem 4.8. *The methods in Listing 2 fulfill the Store Requirement.*

Proof. The Store Requirement entails that `store(p,u)` returns a value `v` stored by some execution of `store(p,v)`.

Keys are never disassociated in the map, and because `store(p,_)` either inserts a value for some key `p`, or does not because the key was already associated to some value, and because the call to `map.get(p)` occurs after that in program order, `map.get(p)` returns an inserted value `v`, which was inserted by `store(p,v)`. □

Theorem 4.9. *The methods in Listing 2 are lock-free if the* `ConcurrentMap` *implementation is lock-free.*

Proof. The methods do not use iteration or recursion, so if the methods implemented by the `ConcurrentMap` object are lock-free (`containsKey`, `putIfAbsent`, and `get`), then the methods in Listing 2 are lock-free. □

4.3 Higher-Order Attributes

A higher-order attribute can be either synthesized or inherited, and optionally parameterized. In either case, the only difference in computing the attribute is that, at the end of the COMPUTE procedure, the result node is attached to the AST by setting its parent reference.

For non-parameterized higher-order attributes we cannot reuse the memoization method in Listing 1 because two threads can race to write a value with `store`, making it possible for two separate AST nodes to be shared with the rest of the program. This is a consensus problem: concurrent threads calling `store` must agree on a single value. A standard solution for *n*-thread consensus is to use *Compare-And-Set* (CAS) [10]. CAS is a lock-free and atomic operation that atomically tests the value of a variable and conditionally updates it to a new value if it had the expected value. In our implementation, in Listing 3, we use the Java standard library class `AtomicReference`, that implements CAS by the `compareAndSet` method. The first argument is the expected value, and the second is the new value.

We use `nil` to represent an illegal attribute value (not equal to `null`). This value is used to indicate that the attribute has not yet been computed and thus replaces the `cached` flag from the synthesized memoization methods (Listing 1).

For parameterized higher-order attributes, we reuse the parameterized memoization methods in Listing 2.

4.3.1 Correctness

For non-parameterized higher-order attributes, the implementation in Listing 3 is used. As mentioned, the COMPUTE method for a higher-order attribute sets the parent reference of the result node. Updating the parent reference does not affect the lock-freedom of COMPUTE: it still terminates in a finite number of steps. However, we must show the that the implementation fulfills the Higher-Order Memoization Requirement:

Theorem 4.10. *The methods in Listing 3 fulfill the Higher-Order Memoization Requirement.*

Proof. The `value` field is only updated by the CAS in `store`, with the expected value NIL. Only one CAS is able to succeed, because the attribute value is never equal to NIL. Because `store` returns the single successful CAS value, it always returns the same value for a single attribute instance.

Note that `memoized` returns TRUE only if a previous CAS has succeeded, and then `load` must return the stored value of the single successful CAS. □

Theorem 4.11. *The methods in Listing 3 are lock-free.*

Proof. All methods of `AtomicReference` are lock-free. No other method calls are used, and no iteration or recursion is used, so the methods in Listing 3 are lock-free. □

For parameterized higher-order attributes, the parameterized memoization implementation in Listing 2 is used. The following theorem proves that it is sound for higher-order attribute memoization.

Theorem 4.12. *The methods in Listing 2 fulfill the Higher-Order Memoization Requirement.*

Proof. The associated value for some key `p` is only updated by `putIfAbsent(p,_)` in `store(p,_)`. Because keys are never disassociated, only one `putIfAbsent(p,_)` is able to succeed. Because `store(p,_)` returns the single successful `putIfAbsent(p,_)` value, it always returns the same reference, for a single attribute instance.

Note that `memoized(p)` returns TRUE only if a previous `putIfAbsent(p,v)` call has succeeded, and then `load(p)` must return the value `v` stored by the single successful `putIfAbsent(p,v)` call. □

4.4 Collection Attributes

Collection attributes [17] collect values from multiple nodes in a subtree of the AST. Each node that potentially contributes a value to a collection attribute is called a *contributor*. A contributor has a semantic function to compute the contributed value, and it may additionally have a contribution condition, i.e., a boolean expression that restricts the node to contribute a value to the collection only if some condition holds.

Collection attribute computation is divided into two phases [17]:

- **Survey phase** A subtree of the AST is traversed, starting from some predetermined *collection root*. All nodes that are potential contributors to the collection attribute are added to a worklist for the next phase.
- **Collection phase** For each node in the worklist from the previous phase, the contribution condition is

checked to determine if the node actually should contribute a value. If the node is contributing to the collection then its semantic function is computed and its value is added to the result.

A simple method of computing collection attributes is to perform a depth-first traversal for the survey phase, and then use a loop to iterate over the resulting list of contributors in the collection phase.

Collection attributes are only computed using the base AST, excluding higher-order attributes. Computing a non-parallelized collection attribute is lock-free because each contribution is computed by a semantic function that must terminate in a finite number of steps, and there are a finite number of contributions because the base AST has a bounded height and each AST node has a finite number of children.

A sequentially evaluated collection attribute is safe for concurrent use if it does not memoize the result. If memoization is needed the memoization scheme for concurrent synthesized attributes can be used.

4.5 Rewrites

JastAdd provides automatic AST rewriting controlled by attributes. This is a powerful tool for transforming the AST, but the original implementation in JastAdd was not safe for concurrent evaluation, as it modified the AST whenever a rewrite was evaluated [5]. JastAdd provides an alternative rewrite mechanism that implements rewrites using higher-order attributes, based on the work of Söderberg and Hedin [23].

For concurrent evaluation, we use the higher-order attribute implementation of rewrites in order to avoid modification of the base AST by rewrite evaluation. This makes rewrites safe for concurrent evaluation by keeping the AST immutable after construction, and using concurrent attributes to implement the rewrites.

5 Circular Attribute Implementation

Circular attributes in RAGs are evaluated recursively, by fixed-point iteration. The current approximations of attributes need to be stored in order for the attribute values to be successively refined. This could be done by using only thread-local storage, or by using a global evaluation lock to prevent concurrency problems in sharing attribute approximations, but neither of these solutions is attractive. We instead investigate an algorithm that allows multiple threads to safely cooperate in computing attribute approximations in a recursive fixed-point iteration.

A circular attribute can be seen as a fixed-point function f. Usually, a fixed-point function is evaluated by repeated application, starting from some bottom value. However, there may be multiple mutually dependent attributes. Therefore, f does not necessarily correspond to a single semantic function, rather it represents multiple simultaneously applied semantic functions. Furthermore, it is possible to apply the individual semantic functions one at a time, in any order, and reach the same simultaneous least fixed-point. This is true because the values of each attribute can be arranged in a lattice, and a combination of attribute approximations, for example a vector of approximations, is also a value in a lattice. Since each semantic function is monotonic, according to well-formedness condition WF3 in Section 2, updating one approximation is a monotonic operation on the combined approximation vector.

We will now illustrate how a circular attribute can be evaluated in practice. Let x be some circular attribute (instance), with $D(x)$ being the set of attribute (instances) that x transitively depends on. For now, we assume that all attributes in $D(x)$ are circular and mutually transitively dependent. We discuss how to relax these requirements later, in Section 6.

Let S be a vector of attribute approximations for the attributes $D(x)$. The S vector forms the state of a fixed-point computation of the attributes $D(x)$. A successor state S' is found by updating one approximation $S'_y = f_y(S)$, where y is an attribute in $D(x)$. If the new approximation of y is not equal to the previous approximation, i.e., $S'_y \neq S_y$, then since f_y is monotonic, S' is greater than S.

Consider a starting state $S^\perp$, where each approximation is equal to the bottom value of the corresponding attribute. By repeatedly updating approximations of attributes in $D(x)$ as above, in any order, starting in state $S^\perp$, the approximations will eventually reach a simultaneous fixed point in which all approximations are equal to the fixed-point value of the corresponding attribute.

A state S^{fp} is a simultaneous fixed point of the attributes in $D(x)$ if, for all $y \in D(x)$, $S^{fp}_y = f_y(S^{fp})$.

5.1 Concurrent Circular Attribute Algorithm

Our algorithm for concurrent evaluation of circular attributes is shown in Algorithm 2(see Appendix C for example Java code generated by our implementation). The algorithm works by using a fixed-point loop (in Case1) to iteratively refine attribute approximations for all attributes in a dependency cycle. The algorithm terminates when all approximations have reached a simultaneous fixed point.

Each iteration of the fixed-point loop starts by computing some attribute x in the dependency cycle (by calling Case2), and recursively computing the attributes used by x, and so on. Recursion stops when an attribute that was previously visited in the current iteration is encountered (Case3). The previous approximation is then reused. To track if an attribute has been visited in the current iteration of the fixed-point loop, attributes are tagged with an index identifying the current iteration. These iteration tags are stored in the thread-local map *tls.iter*.

A change flag, *tls.change*, is used to track if any attribute approximation changes in the fixed-point loop. When the

change flag remains FALSE after an iteration, the loop is exited and the attribute is memoized.

If a single thread is used, our algorithm executes similarly to the sequential algorithm of Magnusson and Hedin [18]. If multiple threads are working in the same dependency cycle, they will exchange attribute approximations through global atomic variables. An invariant in both cases is that a thread can only update attribute approximations to monotonically increasing values (WF3).

The main procedure of our algorithm, CEVAL, uses three CASE subroutines similarly to the formulation of the sequential algorithm of Söderberg and Hedin [23]. Each thread starts in CASE1, which starts a new fixed-point loop. During the loop, CASE2 is used to update approximations of attributes on the dependency cycle. When an attribute is recursively revisited during a particular iteration of the loop, CASE3 is used to fetch the most recent approximation.

To illustrate, assume there are two threads T_1 and T_2 computing mutually dependent attributes x and y respectively. The control flow then looks like this:
($\longrightarrow$ call, - -› loop)

$$T_1\text{: CEVAL}(x) \longrightarrow \text{CASE1}_x \dashrightarrow \text{CASE2}_x \longrightarrow \text{CASE2}_y \longrightarrow \text{CASE3}_x$$

$$T_2\text{: CEVAL}(y) \longrightarrow \text{CASE1}_y \dashrightarrow \text{CASE2}_y \longrightarrow \text{CASE2}_x \longrightarrow \text{CASE3}_y$$

If T_2 memoizes y, then T_1 will not call CASE2 or CASE3 for y upon the next use of y. Instead, the memoized value of y is used directly by T_1.

The next sections describe the state variables used in the algorithm, and how the individual procedures of the algorithm work in more detail.

5.1.1 Shared State

All threads share a global approximation for each attribute x, stored in the atomic variable gv_x. This atomic variable is updated using Compare-And-Set (*CAS*). All reads and CAS of gv_x are lock-free and atomic.

The value of gv_x is a tuple of an attribute value and a *done* flag, indicating if the attribute is memoized. The attribute is only memoized when it has reached its fixed-point value. If the *done* flag is FALSE, the value is either uninitialized (NIL), or an approximation of the attribute x.

The notation used for updating gv_x is $CAS(gv_x, p, n)$, where p is the expected previous value and n is the value to update to. When the *CAS* is performed, if the value of gv_x is indeed p then it is atomically updated to n. In Java, the gv_x field can be implemented by `AtomicReference` as in the higher-order attribute memoization from Section 4.3.

5.1.2 Thread-Local State

Each thread stores thread-local state (TLS), that is not visible to other threads, in the *tls* object. The purpose of each *tls* field is described below:

tls.change A flag indicating if the current thread observed any attribute approximation change during the current iteration of the fixed-point loop.

tls.iter A map from attributes to iteration indices. Unassociated keys are mapped to an unused non-zero iteration index.

In each thread, the iterations of the fixed-point loop in CASE1 are assigned unique indices. Each time a thread computes an approximation of some attribute x, it updates $tls.iter(x)$ to the current iteration index. Thus, $tls.iter(x)$ is used to tag which iteration an attribute was last computed in. This works like a visit flag, except that the iteration tags do not need to be reset on each iteration. Instead, the current iteration index is updated for each iteration. The iteration index is always updated to a unique value, to ensure that iteration indices are unique across all CASE1 invocations, not only across iterations of a single CASE1 invocation.

5.1.3 The CEval Procedure

The CEVAL procedure takes two arguments: an attribute to be evaluated, x, and an iteration index, i. The iteration index identifies separate iterations of the fixed-point loop in the current thread.

CEVAL is called with $i = 0$ when there is no ongoing fixed-point computation. In this case, CEVAL will return the fixed-point value of x. If $i \neq 0$, then CEVAL returns an approximation of x.

The execution of CEVAL starts by testing if x has already been memoized, in which case the memoized value is returned. Otherwise, if the global approximation was not initialized (equal to NIL), the global approximation is updated to the bottom value of x. Next, execution continues to either CASE 1, 2, or 3:

- If $i = 0$, CASE1(x) is called to start a new fixed-point loop.
- Otherwise, if the current thread has not computed a value for x during the current iteration i, CASE2(x, i) is called to compute a new approximation of x.
- Otherwise, CASE3(x) is called to reuse the current approximation of x.

5.1.4 Case1 (Fixed-point Loop)

In CASE1, a new fixed-point computation for an attribute x is started. The computation is performed by a loop, and an iteration index i is used to identify each iteration of the loop.

An iteration of the loop starts by updating i to a new unique, non-zero value, and clearing the *tls.change* flag. Next, CASE2(x, i) is called to compute a new approximation of x. The loop is exited at the end of an iteration i if *tls.change* remains unset. The *tls.change* flag remains unset only if, during an iteration of the CASE1 loop, no attribute approximation was updated to a new value via CASE2(x, i).

Algorithm 2 Concurrent evaluation algorithm for circular attributes.

```
▷ Shared global value of attribute x:
gv_x : Value × Boolean ← (NIL, FALSE)

▷ Thread-local state:
tls.change : Boolean
tls.iter : (Attribute → Integer)

▷ Main procedure for computing an attribute x.
▷ Current iteration index is passed as the i parameter.
procedure CEVAL(x, i)
    (value, done) ← read(gv_x)
    if done then
        return value
    else if value = NIL then
        ▷ Initialize gv_x by Compare-And-Set:
        CAS(gv_x, (NIL, FALSE), (⊥_x, FALSE))
    end if
    if i = 0 then
        return CASE1(x)
    else if tls.iter(x) ≠ i then
        return CASE2(x, i)
    else
        return CASE3()
    end if
end procedure

▷ Run a fixed-point computation of attribute x.
procedure CASE1(x)
    repeat
        i ← uniqueId()
        tls.change ← FALSE
        CASE2(x, i)
    until ¬tls.change
    ▷ Memoize x by marking gv_x as done:
    (result, _) ← read(gv_x)
    CAS(gv_x, (result, FALSE), (result, TRUE))
    return result
end procedure

▷ Compute a new approximation of attribute x.
procedure CASE2(x, i)
    (prev, _) ← read(gv_x)
    tls.iter(x) ← i
    next ← COMPUTE(x, i)
    if next ≠ prev then
        tls.change ← TRUE
        CAS(gv_x, (prev, FALSE), (next, FALSE))
    end if
    return next
end procedure

▷ Read most recent approximation of x.
procedure CASE3(x)
    (prev, _) ← read(gv_x)
    return prev
end procedure
```

After the loop is exited, the stored global value of x is equal to the fixed-point value of x, so the current thread attempts to memoize the attribute by updating the *done* flag for x.

5.1.5 Case2 (Compute Approximation)

CASE2 computes a new approximation of an attribute x by calling COMPUTE(x, i). The COMPUTE procedure is an executable translation of the semantic function of attribute x, where each access to some other attribute y is translated as a call to CEVAL(y, i).

CASE2 starts by reading the current approximation of x. The current approximation is stored in a temporary variable to be able to detect if the newly computed value differs from the previous approximation.

Before computing a new approximation, the current thread tags the attribute x with the current iteration index i. This causes recursive CEVAL(x, i) calls in the current thread to enter CASE3 rather than CASE2, thereby avoiding unbounded recursion.

A new approximation of x is computed by COMPUTE(x). If the new value is different from the previous approximation, then *tls.change* is set to TRUE and the shared approximation of x is updated using *CAS*.

5.1.6 Case3 (Recursion Termination)

CASE3 returns the most recent approximation computed for the attribute x. This case is necessary to terminate the recursion when called via CASE2.

5.2 Correctness

We will here show informal outlines for proofs of soundness and lock-freedom of CEVAL. The full proofs are in Appendix A.

Soundness. CEVAL is sound if, for a well-formed circular attribute x, CEVAL$(x, 0)$ computes the fixed-point value of x. Well-formedness is defined in Section 2. Specifically, the semantic function of x must be monotonic (WF3).

Consider a single thread executing CEVAL. It will always enter CASE1 initially, then perform iterations until the *tls.change* flag remains unset. For this to work, CASE2

should be called for each attribute that x depends on, that can change value, in each iteration of Case1. It can be shown that in each iteration of Case1, either all attributes that x transitively depends on have reached their fixed-point value, or Case2 is executed for all attributes that x transitively depends on(Lemma A.11).

It is important that a single thread only advances the global state of an attribute to a monotonically increasing value. This is both ensured by the monotonicity of semantic functions, and by the fact that the current approximation is read before computing a new approximation, and used as the expected value before updating to a new approximation(Lemma A.13).

Lock-freedom. CEval is lock-free if it terminates in a finite number of steps. Proving this is mostly straight-forward. The only challenging parts are to show that the loop in Case1 performs a finite number of iterations, and that each one performs a finite amount of work. This relies on the fact that the attributes are well-formed and thus have terminating semantic functions (WF2), and a finite greatest possible value (WF3) which is eventually reached by successive approximation in Case1. Since all attribute approximation states form a lattice of finite height, and approximation updates are monotonic, the algorithm will always terminate in a finite number of iterations of the Case1 loop if at least one approximation is updated on each iteration. If not at least one approximation update happens, then Case1 would terminate anyway.

5.3 Parameterized Circular Attributes

Like most other kinds of attributes, circular attributes can be parameterized. To support parameterized circular attributes, we need a few modifications to Algorithm 2. A new parameter p is added to the CEval procedure. The parameter p is a list of the attribute argument values, and it is passed to Case1, Case2, Case3, and Compute.

The global value of a non-parameterized circular attribute is stored in an atomic variable with a CAS operation and atomic read. To store global values for a parameterized circular attribute we instead use a concurrent map that maps attributes to atomic variables. The global value map is indexed by p, i.e. $gv_x(p)$ gives the atomic variable for the global value of x with arguments p.

A parameterized circular attribute x is initialized by using `putIfAbsent` to insert a new atomic variable containing the bottom value of x in the global value map. The other uses of gv_x from the non-parameterized algorithm are replaced by map lookups $gv_x(p)$. Because we use `putIfAbsent`, we ensure that an attribute is only initialized once. The rest of the uses of gv_x will all act as before, but on different atomic variables for different argument combinations.

The local iteration map *tls.iter* needs to be indexed by both attribute and argument values. We implement this by using tuple objects containing the attribute and argument value list as map key.

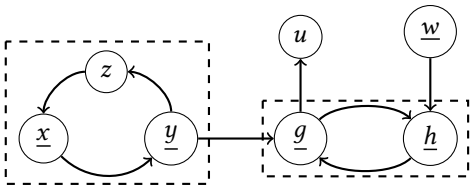

Figure 4. An attribute dependency graph. Each circle is an attribute instance. Attributes with underlined names are circularly evaluated (with bottom value). Attributes inside the dashed rectangles are strongly connected and effectively circular. The attributes u and $\underline{w}$ are effectively non-circular.

6 Relaxed Requirements on Circular RAGs

Up until now we have used some simplifying assumptions about the structure of circular attributes in RAGs. In this section we review these assumptions, and we show why some of them are not necessary for correctness, and how others can be relaxed by simple additions to our algorithms.

By relaxing assumption 1 below, we allow more general combinations of circular and non-circular attributes than were previously allowed in Circular RAGs according to Magnusson and Hedin [18]. This relaxed requirement is useful in practice since it is common that attributes are on a cycle only for a small fraction of typical ASTs. Requiring these attributes to be declared as circular would cause expensive fixed-point computations during runtime even for many ASTs where there actually is no cycle.

To concisely discuss these assumptions we will first need two auxiliary definitions:

Circularly evaluated attribute An attribute instance x is *circularly evaluated* if it has a bottom value, and CEval is used to compute its value.

Effectively circular attribute An attribute instance is *effectively circular* if it depends transitively on itself. Otherwise it is said to be *effectively non-circular*.

An attribute declaration can have both effectively circular and effectively non-circular instances.[2]

These are the assumptions we have used so far:

1. A circularly evaluated attribute instance depends only on circularly evaluated attribute instances.
2. All circularly evaluated attribute instances are effectively circular.
3. All effectively circular attribute instances are circularly evaluated.
4. If an effectively circular attribute instance x transitively depends on an attribute instance y, then y transitively depends on x.

Assumption (1) can be relaxed. Figure 4 shows two attribute instances breaking this assumption: $\underline{g}$ and $\underline{u}$ are both

[2]Magnusson and Hedin [18] refer to circularly evaluated attributes as *potentially circular* and to effectively circular attributes as *actually circular*.

circularly evaluated, but depend on attributes that are not, namely u and z. Evaluating u with Algorithm 1 works as it should, because u is not effectively circular and always computes the same value. For z, however, Algorithm 1 does not work correctly, because z is effectively circular and can compute different values based on an approximation of $\underline{x}$.

A problem arises because Algorithm 1 memoizes attributes on the first computation, but during a fixed-point computation, effectively circular attributes return approximations, which are not safe to memoize. To avoid premature memoization, an attribute depending (transitively) on a circularly evaluated attribute should not memoize its result during a fixed-point computation.

Assumption (1) can be relaxed by replacing Algorithm 1 by Algorithm 3 for non-circular attributes. This changes memoization of attributes to be conditional, so that Eval only memoizes an attribute when it is called outside any ongoing fixed-point computation. An additional field, *tls.i*, is added to the thread-local state to track the current fixed-point iteration index. At the start of each iteration of the Case1 loop, *tls.i* is updated to the current iteration index, and at the end of Case1, *tls.i* is set to 0. A memoized attribute must ensure that $tls.i = 0$ before memoizing a result. The updated algorithm works even for higher-order attributes, because the result node is not memoized by any other attribute that depends on the higher-order attribute before the circular evaluation has reached the fixed point, thereby different AST nodes do not become visible to the rest of the program.

Algorithm 3 can be further improved by memoizing attribute values and tagging the memoized value with the current iteration index. This allows the attribute value to be reused in a single Case1 iteration. This improved algorithm is the one that we implemented in JastAdd.

Algorithm 3 Evaluation algorithm for memoized non-circular attributes with circular dependees.

```
procedure Eval(x)
    if Memoized(x) then
        return Load(x)
    else
        u ← Compute(x)
        ▷ Test if called in circular evaluation.
        if tls.i ≠ 0 then
            ▷ In circular evaluation: not safe to memoize.
            return u
        else
            ▷ Memoize the computed value as usual.
            return Store(x, u)
        end if
    end if
end procedure
```

Assumption (2) is not necessary for correctness. In Figure 4, $\underline{w}$ is circularly evaluated, but it is not effectively circular. In general, if some attribute a is not effectively circular, but circularly evaluated with CEval, then there are two cases: a transitively depends on some circularly evaluated attribute instance, or it does not.

If a depends on some circularly evaluated attribute instance, then, as long as that attribute changes approximation, the Case1 loop for a will not terminate, thereby the circular attribute reaches its fixed-point.

If a does not depend on a circular attribute, then in the first Case1 iteration, there is no approximation of a so the change flag is set in Case2. Since a is not circular it does not compute a new approximation in the second Case1 iteration, so the fixed-point loop completes after the second iteration, and the value of a is memoized.

Assumption (3) is not necessary for correctness of Algorithm 2. It is sufficient that at least one distinguished attribute in each dependency cycle is circularly evaluated. The non-distinguished attributes can then be seen as non-memoized circular attributes with bottom values computed using the bottom values of the distinguished circular attributes. Recursive computation terminates in Case3 at the distinguished attributes, preventing unbounded recursion.

Assumption (4) is not necessary for correctness. It implies that the dependency graph of each circular attribute is strongly connected. If it is not strongly connected, our algorithm works without modification. However, the algorithm could potentially be modified to improve performance by separately evaluating the connected components in topological order and memoizing each component separately, similar to the method used by Magnusson and Hedin [18]. Future work could investigate extending our concurrent circular evaluation algorithms to improve runtime performance for separate components.

7 Empirical Evaluation

The research questions we want to answer in the evaluation are:

RQ1 Does our implementation of the concurrent algorithms work on existing well-formed JastAdd projects?
RQ2 Can the implementation be used for interactive tools with both interactive and long-running tasks?
RQ3 Does our concurrent implementation give sufficiently low latency for interactive tasks?

Section 7.1 addresses the applicability of the approach (RQ1 and RQ2). Latency (RQ3) is addressed in Section 7.2. Speedup is discussed in Section 7.3. Threats to validity are discussed in Section 7.4.

7.1 Concurrent ExtendJ and Interactive Applications

We applied our concurrent implementation[3] to ExtendJ, a full-featured Java compiler [6]. The ExtendJ specification is complex, with more than 3000 attributes, including all attribute kinds discussed in this paper.

Initially, running ExtendJ concurrently did not work because its specification was not completely well-formed, with some semantic functions being non-pure (WF1). Most of these problems happened to be masked in sequential evaluation, but resulted in errors when running concurrently. In one case there was also an error when running sequentially caused by purity issues.

Substantial work was required to find and fix attribute purity problems, but the result benefits the sequential compiler by removing cases where it could compute incorrect results when attributes were evaluated in a certain order. [4]

After fixing the identified well-formedness problems, we successfully ran both the sequential and the concurrent implementations on all regression tests for ExtendJ using the same JastAdd specification. Based on this, we can answer RQ1 affirmatively.

To address RQ2 we implemented an extension of an interactive AST debugging tool named DrAST [16]. DrAST has a Graphical User Interface in which the user can explore a JastAdd AST for a program and interactively inspect/compute attribute values of nodes in the AST. In our extension to DrAST, we integrated ExtendJ and added a few features: We added a source editor for the program, and changes to the program are reflected in the AST view. A screenshot of our version of the tool is shown in Figure 6 in Appendix B. We also added a computation of ExtendJ's `problems` attribute containing compile-time error and warning messages so that these messages are displayed by DrAST. The user can interactively inspect/compute attribute values while the long-running `problems` attribute is computed. Any interactive attribute queries are run concurrently with error-checking tasks using our concurrent attribute evaluator. The tool thus works similarly to a typical Integrated Development Environment, and we can thereby answer RQ2 affirmatively.

7.2 Latency

The independent variable in studying latency is the attribute evaluator implementation. We measure two different attribute evaluators: the sequential implementation from JastAdd, and our concurrent implementation presented in this paper (Algorithms 2 and 3). Attribute evaluation time is the measured dependent variable. Confounding variables are the compiler (ExtendJ) on which we measure, and the attributes measured.

[3] Our concurrent attribute implementation is available in JastAdd version 2.3.0. See http://jastadd.org.

[4] The fixes are available in ExtendJ version 8.0.1-183-g812e434, from the public Git repositories. See http://extendj.org.

Table 1. The thread-task mapping for each benchmark.

Benchmark	Thread			
	1	2	3	4
1	Task P	Task VD	–	–
2	Task P	Task MT	–	–
3	Task P	–	–	–
4	Task P	Task P	Task P	Task P

7.2.1 Setup

We designed benchmarks to measure attribute evaluation latency and overall overhead and speedup of concurrent attribute evaluation. For evaluation latency we measure two relatively short-running attributes that are evaluated concurrently with a long-running attribute. For overhead and speedup we measure the evaluation time of the long-running attribute when evaluated sequentially and in parallel.

We use four benchmark configurations, as shown in Table 1. Each benchmark runs some combination of three tasks executed in separate concurrent threads:

Task P Evaluates the long-running attribute `problems` on all `CompilationUnit` nodes.

Task VD Evaluates the short-running attribute `decl` (variable declaration) on 500 stochastically selected `VarAccess` nodes.

Task MT Evaluates the short-running attribute `type` (method type) on 500 stochastically selected `MethodDecl` nodes from classes and interfaces.

The first two benchmarks are used to measure attribute evaluation latency in an interactive setting. They run many short-running attributes in one thread, and a long-running attribute in a separate concurrent thread. Benchmark 3 is used to measure sequential performance by running a long-running attribute in a single thread. Benchmark 4 is used to measure parallelization performance by running long-running attributes in parallel in four threads.

All benchmarks are run both with the sequential and concurrent implementation. In the concurrent mode, task threads are allowed to evaluate attributes concurrently, but in the sequential mode, we use a lock to ensure that only one thread at a time is evaluating any attribute.

Each benchmark configuration is executed 15 times in a single Java process. The results of the first three iterations are discarded to reduce the impact of warm-up effects in the Java environment.

Before Benchmark 1 and 2 are executed we first search the AST of the subject program to find all `VarAccess` or `MethodAccess` nodes, then the list of nodes is shuffled and the first 500 nodes are used in the benchmark.

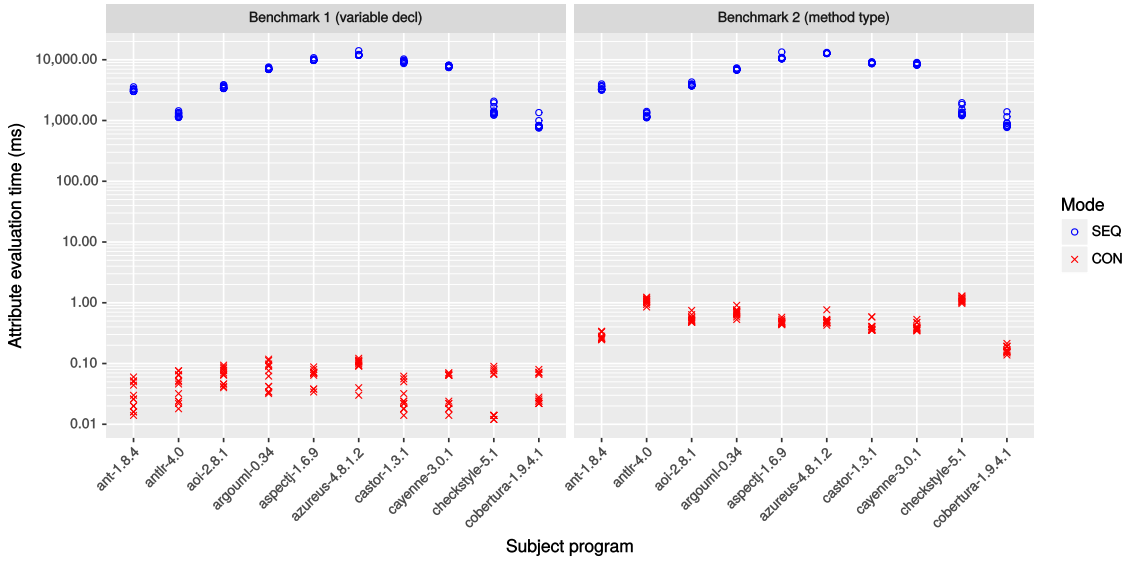

Figure 5. Latency results from Benchmark 1 and 2. Red ×:s show the latency for interactive tasks when using the concurrent implementation. Each × shows the average time for computing a variable declaration (left) or a method type (right) attribute, when running concurrently with the long-running `problems` attribute. Blue ○:s show the time it took to complete the long-running task of computing the `problems` attribute when using the sequential algorithms with locking. This is the minimum latency for interactive tasks that would occur when the interactive task is started right after starting the long-running task.

Subject programs we used for the benchmarks are taken from the Qualitas Corpus, Version 20130901 [25]. We measured the first 10, in alphabetical order, of the subject programs in the Qualitas Corpus that were written for Java 5 or higher.

The benchmark suite was run on an Intel Core i7-3820 CPU at 3.60GHz, running 64-bit Linux Mint, with Java version 1.8.0_112 (Oracle JDK). A relatively large Java heap size of 32Gb was used, more than 10× the minimum requirement to compile each subject program in sequential mode, in order to limit runtime garbage collection.

7.2.2 Results

RQ3 asks whether our concurrent implementation gives sufficiently low latency for interactive tasks. Benchmark 1 and 2 address this question by measuring the time it took to evaluate 500 instances of two kinds of attributes: a variable declaration attribute (Benchmark 1) and a method type attribute (Benchmark 2). In both benchmarks, the attributes are evaluated while concurrently computing compile-time errors and warnings for the whole subject program via a long-running attribute.

Our results show that when running the concurrent implementation, for any of the 10 programs, the highest average latency for finding a variable declaration is below 0.5 ms, and the highest average latency for computing a method type is below 5 ms. This is far below the acceptable threshold of 100 ms, so this answers RQ3 affirmatively.

If the sequential implementation with locks is used instead, the lower bound for the latency of an interactive task that starts right after the start of a long-running task will be the time it takes to complete the long-running task. This could in principle be a very long time. In our experiments, we used the computation of the `problems` attribute as a typical representative of a long-running task. Our experiments show that the average time for this computation is between 500 ms and 11 seconds for the 10 different programs. The latency in the sequential case is thus clearly too high for interactive tasks. Figure 5 shows the average latency of the short-running attributes in Benchmark 1 and 2 compared to the long-running attribute.

7.3 Speedup

Although it is not one of our research questions, we were interested to investigate the overhead and speedup of concurrent evaluation. We make some observations here based on the results of Benchmark 3 and 4 (Section 7.2.1).

For overhead, we use the time for evaluating a long-running attribute in a single thread by using both the concurrent and sequential implementation. The overhead is computed as the concurrent evaluation time divided by the sequential time. Speedup of parallelization is measured by taking the time to find all compile-time errors in a program using four parallel threads, divided by the running time of the same computation in a single thread, using the concurrent implementation. We measured the overhead and speedup for the 10 subject programs, see Table 2. On average, the

overhead was less than 20%, and the speedup was around 2 for most programs.

Table 2. Overhead of one thread running the concurrent algorithms, compared to the sequential algorithms. Speedup on a 4-core processor when running four threads in parallel, compared to running only one thread, all running the concurrent algorithms. Note that overhead is independent of code size (NCLOC, non-comment lines of code).

Program	NCLOC	Overhead	Speedup
ant-1.8.4	105,007	1.10	1.95
antlr-4.0	21,919	1.16	2.04
aoi-2.8.1	111,725	1.22	2.07
argouml-0.34	192,410	1.33	2.24
aspectj-1.6.9	412,394	1.17	2.43
azureus-4.8.1.2	484,739	1.22	2.32
castor-1.3.1	115,543	1.18	2.03
cayenne-3.0.1	127,529	1.16	1.98
checkstyle-5.1	23,316	1.07	2.27
cobertura-1.9.4.1	51,860	1.19	1.52
average		*1.18*	*2.15*

7.4 Threats to Validity

The general applicability of our results is limited by the fact that we have measured only three attributes in a single JastAdd-specified compiler, ExtendJ. However, in our opinion, ExtendJ is representative of a typical JastAdd compiler. Also, ExtendJ is one of the largest JastAdd projects freely available, and it uses all different attribute kinds discussed in this paper.

An alternative compiler we considered is JModelica: a compiler for the Modelica language. However, JModelica currently uses several difficult to remove side-effects in the specification that would need to be fixed in order to run it concurrently.

Our results of course depend on the subject programs that were used. We selected these programs in a systematic manner from a well-known corpus in order to avoid bias.

8 Related Work

There are many algorithms for concurrent and parallel evaluation of Knuth attribute grammars, see, e.g., the surveys by Jourdan [13] and Paakki [21]. However, that work is based on tree-walking evaluators which are not applicable to RAGs. First, tree-walking evaluators take only local dependencies into account, and can therefore not deal with the non-local dependencies arising from the use of reference attributes. Second, the tree-walking algorithms evaluate *all* attributes in an AST, whereas in RAGs, the only attributes that are evaluated are those needed for the computation of some goal attribute. Third, the tree-walking algorithms do not work for circularly dependent attributes. We have not found any previous attempts to parallelize the demand-driven evaluation algorithms used in RAGs, neither for circular nor for non-circular attributes.

In dynamic programming, results to subproblems are memoized, typically in a hash table, so that they only need to be computed once. In top-down dynamic programming, subproblems are computed and memoized recursively, similar to demand-driven evaluation of RAGs. Stivala et al. [24] have developed lock-free parallel algorithms for top-down dynamic programming. The basic idea is to let several threads solve the complete problem in parallel, and let them store and share the memoized subproblems through a global lock-free hash table. Randomization is used to encourage different threads to work on different subproblems. This approach is not sufficient for concurrent evaluation of RAGs, with their different kinds of attributes and fixed point computations. However, the idea of using randomization is interesting to investigate in future work for RAGs in order to gain better speed-up when running threads in parallel.

Ditter et al. [4] develop a method for evaluating fixed-points in parallel, with the goal of speeding up software verification using boolean equation systems. They observe that in a fixed point iteration, the order of evaluating the different equations does not matter, and the equations can therefore be evaluated in parallel. We also make use of this observation in order to let several threads cooperatively evaluate a circular attribute. Our demand-driven fixed point algorithm is, however, substantially different from the traditional fixed-point algorithm used by Ditter. In the traditional algorithm, it is assumed that both the equations and the variables to be solved are known a-priori, and it is therefore straight-forward to view this as a homogeneous data-parallel problem. For RAGs, neither the equations nor the variables are known a-priori, but are discovered during the recursive evaluation algorithm, and the fixed-point problem is heterogeneous, involving attributes associated with many different node types and which are defined by many different equations.

9 Conclusions

The goal of this work was to develop safe concurrent algorithms for Circular Reference Attribute Grammars, in order to reduce latency in interactive tools. To this end, we designed new lock-free attribute evaluation algorithms that can be run concurrently by several threads. Our algorithms support synthesized, inherited, parameterized, higher-order, collection, and circular attributes, as well as attribute-controlled rewrites. Furthermore, we have generalized the algorithms to work with relaxed requirements on circular attributes.

We implemented our algorithms in the JastAdd metacompiler, and the implementation can be used directly for any well-formed JastAdd project. With our implementation, it

is straightforward to evaluate attributes concurrently in an interactive tool, for example to perform attribute computations in a background thread at the same time as computing attribute query results in an interactive thread.

Through empirical evaluation, we demonstrated that our algorithms significantly reduce the attribute evaluation latency, from seconds to milliseconds for an interactive thread. Our results are well below the threshold of 0.1 seconds strived for in interactive systems, and we conclude that concurrent RAGs is a very attractive implementation technology for interactive tooling.

We also did initial experiments on using the concurrent algorithms for improving performance using parallelization. We found that the overhead of the concurrent algorithm over the sequential algorithm is under 20% on average and that it is outweighed when running in parallel: We measured a speedup of about 2 on average when running four parallel threads.

Our results are very encouraging, and exploring how to improve and take advantage of parallel evaluation is a very interesting direction of future work. Possibilities for performance improvement include tuning which attributes are memoized, and improving work distribution between threads, for example using randomization and work stealing. Refactoring attributes to be more long/short-running could also affect parallel performance: Short-running attributes reduce the risk of duplicate work when running in parallel, while long-running attributes reduce the relative concurrent memoization overhead.

Acknowledgments

This work was partially funded by a 2015 Google Faculty Research Award on supporting concurrent analyses in interactive programming tools. We would like to thank to Emma Söderberg, Niklas Fors, Alfred Åkesson, Axel Mårtensson, and the anonymous reviewers for valuable feedback.

References

[1] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. 2010. Modeling and optimization with Optimica and JModelica.org - Languages and tools for solving large-scale dynamic optimization problems. *Computers & Chemical Engineering* 34, 11 (2010), 1737–1749.

[2] John Tang Boyland. 2005. Remote attribute grammars. *J. ACM* 52, 4 (2005), 627–687.

[3] Christoff Bürger. 2015. Reference attribute grammar controlled graph rewriting: motivation and overview. In *SLE '15*. ACM, Pittsburgh, PA, USA, 89–100.

[4] Alexander Ditter, Milan Ceska, and Gerald Lüttgen. 2012. On Parallel Software Verification Using Boolean Equation Systems. In *SPIN 2012 (LNCS)*, Vol. 7385. Springer, Oxford, UK, 80–97.

[5] Torbjörn Ekman and Görel Hedin. 2004. Rewritable Reference Attributed Grammars. In *ECOOP 2004 (LNCS)*, Vol. 3086. Springer, Oslo, Norway, 144–169.

[6] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd extensible Java compiler. In *OOPSLA '07*. ACM, Montreal, Canada, 1–18.

[7] Rodney Farrow. 1986. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. ACM, Palo Alto, CA, USA, 85–98.

[8] Niklas Fors, Gustav Cedersjö, and Görel Hedin. 2015. JavaRAG: a Java library for reference attribute grammars. In *MODULARITY '15*. ACM, Fort Collins, CO, USA, 55–67.

[9] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.

[10] Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann, San Francisco, CA, USA.

[11] Larry G. Jones. 1990. Efficient Evaluation of Circular Attribute Grammars. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 429–462.

[12] Martin Jourdan. 1984. An Optimal-time Recursive Evaluator for Attribute Grammars. In *International Symposium on Programming (LNCS)*, Vol. 167. Springer, Toulouse, France, 167–178.

[13] Martin Jourdan. 1991. A Survey of Parallel Attribute Evaluation Methods. In *Attribute Grammars, Applications and Systems (LNCS)*, Vol. 545. Springer, Prague, Czechoslovakia, 234–255.

[14] Uwe Kastens. 1980. Ordered Attributed Grammars. *Acta Inf.* 13 (1980), 229–256.

[15] Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145.

[16] Joel Lindholm, Johan Thorsberg, and Görel Hedin. 2016. DrAST: An Inspection Tool for Attributed Syntax Trees (Tool Demo). In *SLE '16*. ACM, Amsterdam, The Netherlands, 176–180.

[17] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. 2009. Demand-driven evaluation of collection attributes. *Autom. Softw. Eng.* 16, 2 (2009), 291–322.

[18] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars - their evaluation and applications. *Sci. Comput. Program.* 68, 1 (2007), 21–37.

[19] David A. Naumann. 2005. Observational Purity and Encapsulation. In *FASE '05 (LNCS)*, Vol. 3442. Springer, Edinburgh, UK, 190–204.

[20] Jakob Nielsen. 1993. *Usability engineering*. Academic Press, San Francisco, CA, USA.

[21] Jukka Paakki. 1995. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Comput. Surv.* 27, 2 (1995), 196–255.

[22] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. 2013. A pure embedding of attribute grammars. *Sci. Comput. Program.* 78, 10 (2013), 1752–1769.

[23] Emma Söderberg and Görel Hedin. 2015. Declarative rewriting through circular nonterminal attributes. *Computer Languages, Systems & Structures* 44 (2015), 3–23.

[24] Alex D. Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel V. Hermenegildo, and Anthony Wirth. 2010. Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput.* 70, 8 (2010), 839–848.

[25] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *APSEC '10*. IEEE Comp. Soc., Sydney, Australia, 336–345.

[26] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher-Order Attribute Grammars. In *PLDI '89*. ACM, Portland, Oregon, USA, 131–145.

[27] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Sci. Comput. Program.* 75, 1-2 (2010), 39–54.

[28] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. 2007. Attribute Grammar-Based Language Extensions for Java. In *ECOOP '07 (LNCS)*, Vol. 4609. Springer, Berlin, Germany, 575–599.

A Circular Attribute Correctness Proofs

This section gives the proofs of soundness and termination for CEval, i.e., the concurrent algorithm for circular attributes (Algorithm 2). We start by giving several technical lemmas with proofs. The main theorems and proofs appear at the end of this section.

The lemmas below state properties about iterations of Case1 as executed in one single thread. There may be concurrently executing threads, running their own Case1 iterations, but threads never share Case1 executions. The only interaction between threads happens through reading and writing shared attribute approximations (gv_x, gv_z, etc.). Iteration indices can, without loss of generality, be thought of as being globally unique between all threads.

We will often talk about properties such as the *done* flag being set before some point in execution. This means that given some linearization of several threads executing the algorithm concurrently, the linearization point of a write to gv_x, setting *done* to TRUE, was linearized as happening before the given point in the current thread being discussed.

Also note that the lemmas only deal with attribute instances, though we sometimes refer to them as just attributes.

First, we need a few definitions:

Definition A.1 (Attribute Set)**.** $\mathcal{A}$ is the set of attribute instances in some attributed AST.

Definition A.2 (Direct Dependencies)**.** For an attribute $x \in \mathcal{A}$, $d(x)$ is the set of attributes that x directly depends on.

Definition A.3 (Transitive Dependencies)**.** For an attribute $x \in \mathcal{A}$, $D(x)$ is the set of attributes that x transitively depends on, including x.

We assume here that $D(x)$ is strongly connected, in other words, for all $y \in D(x)$, $D(y) = D(x)$.

The following observation restates a consequence of how semantic functions are translated into Compute procedures:

Observation 1. *For an attribute $x \in \mathcal{A}$ and an iteration i of the loop in Case1(x), executing Case2(x,i) leads to executing CEval(y,i) for all $y \in d(x)$.*

We will need to reason about what happens during an iteration i of the loop in Case1. The following definitions introduce boolean functions to succinctly reason about this.

Definition A.4 (Execution of Case2)**.** Let $x \in \mathcal{A}$ be an attribute, and i an iteration of the loop in Case1(x). Then, *case2*(x,i) is true iff Case2(x,i) was executed during iteration i.

Definition A.5 (Fixed-Point Value)**.** Let $x \in \mathcal{A}$ be an attribute and i an iteration of the loop in Case1(x). Then, *fix*(x,i) is true iff, before the end of iteration i, the shared approximation for x is equal to the fixed-point value of x.

Definition A.6 (Memoized Value)**.** Let $x \in \mathcal{A}$ be an attribute and i an iteration of the loop in Case1(x). Then, *done*(x,i) is true iff the *done* flag in the tuple gv_x is TRUE before the end of iteration i.

In other words, *done*(x,i) implies that x was memoized before or during iteration i.

Note that *fix*(x,i) is not equivalent to *done*(x,i), though if the algorithm is correct, *done*(x,i) implies *fix*(x,i). We prove this in Lemma A.10.

Lemma A.7 (Case2 on Direct Dependency)**.** *Let $x \in \mathcal{A}$ be an attribute, and i an iteration of the loop in Case1(x), and let $y \in d(x)$ be a direct dependency of x. If Case2(x,i) is executed during iteration i, and y is not marked as done before the end of the iteration, then Case2(y,i) is executed at some point during the iteration.*

Proof. According to Observation 1, CEval(y,i) is executed as a direct consequence of executing Case2(x,i).

When CEval(y,i) is executed, there are three cases:

- y is already marked as done, contradicting the premise of the lemma, or,
- no previous approximation has been stored for y during iteration i, so then Case2(y,i) is executed, or,
- a local approximation of y has been previously stored during iteration i. Note that local approximations are only stored in Case2, so then Case2(y,i) was executed at some point during i.

□

We now define another helper function to reason about paths in the dependency graph of an attribute:

Definition A.8 (Dependency Paths)**.** Let $x \in \mathcal{A}$ be an attribute, with a transitive dependency on $y \in D(x)$. The function *paths*(x,y) gives all acyclic paths from x to y following the attribute dependency graph.

In other words, for each $p = (a_1, a_2, \cdots, a_n)$ in *paths*(x,y), the following holds:

- $x = a_1$,
- $y = a_n$,
- and $a_{i+1} \in d(a_i)$, where $1 \leq i < n$.

Lemma A.9 (Case2 on All if None Done)**.** *Let $x \in \mathcal{A}$ be an attribute, transitively depending on $y \in D(x)$, and let i be an iteration of the loop in Case1(x). For each path $p \in$ paths(x,y), from x to y, where none of the attributes in p are marked as done by the end of i, Case2(z,i) is executed for each attribute z in p during i.*

Proof. By induction on prefixes of p. The one-length prefix of p is equal to x, and since i is an iteration of Case1(x), Case2(x,i) is directly executed in the loop body.

Assuming that the lemma holds for an n-length prefix of p, we must show that it holds for a prefix of length $n+1$. Let a_n be the n^{th} element of p, then it follows from the induction hypothesis that $\neg$*done*(a_n), and by the definition of *paths*(x,y) it follows that $a_{n+1} \in d(a_n)$. By the induction

hypothesis it also follows that $case2(a_n, i)$, and together with the conclusion that $\neg done(a_n)$, Lemma A.7 gives the goal: $case2(a_{n+1}, i)$.

By induction the lemma holds for any length prefix of $p \geq 1$, so the lemma holds for p. □

Lemma A.10 (Done $\implies$ Fix)**.** *Let $x \in \mathcal{A}$ be an attribute, and i an iteration of the loop in Case1(x). If x is marked as* done *before the end of i, then for each transitive dependency $y \in D(x)$, the shared approximation of y is equal to the fixed-point value of y before the end of i.*

Proof. Note that an attribute can only be marked as *done* by the CAS after the Case1 loop. CAS is linearizable, so the effect of several (concurrent) CAS calls is identical to the effect of some sequential ordering of the CAS operations. Consequently, among the CAS calls that mark attributes in $D(x)$ as *done*, there exists a first one.

Consider the first attribute $y \in D(x)$ that is marked as *done* by the CAS at the end of Case1. The loop always takes at least one iteration, so let k denote the last iteration before y was marked as *done*.

From Lemma A.9 and the assumption that y is the first attribute in $D(y)$ which is marked as *done*, it follows that Case2 was executed for all $z \in D(y)$. The loop condition implies that none of the attributes in $D(y)$ changed approximation, thus a simultaneous fixed-point has been reached and $fix(z, k)$ is true for all $z \in D(y)$.

Because $D(x)$ is strongly connected, and $y \in D(x)$, then $D(x) = D(y)$. Substituting $D(y)$ for $D(x)$ gives the goal: for all $z \in D(x)$ it holds that $fix(z, k)$ is true. □

Lemma A.11 (All Fix Or Case2)**.** *Let $x \in \mathcal{A}$ be an attribute, and i an iteration of the loop in Case1(x). Then one of the following properties hold:*

- *all attributes in $D(x)$ have reached their fixed-point value before the end of iteration i, or,*
- *Case2 is executed for all attributes in $D(x)$ during iteration i.*

Proof. There are two cases:

- some attribute $z \in D(x)$ was marked as done before the end of iteration i, or,
- none of the attributes in $D(x)$ were marked as done before the end of iteration i.

In the first case, there exists some $z \in D(x)$ such that $done(z, i)$ is true, and by Lemma A.10 we have the fact that all attributes $w \in D(z)$ have reached their fixed-point values before the end of iteration i. Additionally, $D(x)$ is strongly connected, and $z \in D(x)$ means that $D(x) = D(z)$. Substituting $D(z)$ for $D(x)$ gives the goal: for all $w \in D(x)$, $fix(w, i)$ is true.

In the second case, there does not exist an attribute $z \in D(x)$ such that $done(z, i)$ is true. Consequently, for each path $p \in paths(x, y)$ it most hold that $done(z, i)$ is false for each element z of p. By Lemma A.9 it follows that Case2(y, i) is executed during iteration i for all $y \in D(x)$. □

Lemma A.12 (Case1 Sound)**.** *Let $x \in \mathcal{A}$ be an attribute, with a transitive dependency on some attribute $y \in D(x)$, and let i be the last iteration of an execution of Case1(x). Then, the shared approximation of y is equal to y's fixed-point value before the end of iteration i, i.e. $fix(y, i)$ is true.*

Proof. By Lemma A.11 there are two cases:

- all attributes in $D(x)$ have reached their fixed-point values before the end of iteration i, or,
- Case2 is executed for all attributes in $D(x)$ during iteration i.

In the first case, the goal follows directly from the premise, $y \in D(x)$.

In the second case, Case2(z, i) is executed during iteration i for all $z \in D(x)$. Since i was the last iteration, and the loop is only exited if *tls.change* = FALSE, it must be that all attributes $z \in D(x)$ were computed to the same value as their previous approximations. According to our definition of simultaneous fixed-point, all attributes in $D(x)$ must then have reached their fixed-point value. Again, according to the premise, $y \in D(x)$ leads to the conclusion that y has reached its fixed-point value. □

Lemma A.13 (Case2 is Monotonic)**.** *Let $x \in \mathcal{A}$ be an attribute and i an iteration of the loop in Case1(x). Then, executing Case2(x, i) does not update the shared approximation for x to a new value that is lower in the value lattice of x.*

Proof. First note that the shared approximations of attributes are updated only by using CAS. The CAS operation is linearizable, so the calls take effect as if executed in some sequential order. Consequently, there exists a first shared approximation update among any set of shared approximation updates for any set of attributes.

Proof by contradiction. Assume that there exists an attribute $z \in \mathcal{A}$ which is the first attribute whose approximation gv_z is updated to a lower value in the value lattice of z by Case2(z, k) during some iteration k.

At the start of Case2(z, k), the value v_0 was read from gv_z. Note that the approximation v_0 was computed by applying the semantic function of z to some state S_0. Later in Case2, computing z gives a value v_1 applying the semantic function to a state S_1. Because the semantic function is monotone, according to well-formedness condition WF3, the value v_1 can only be lower than v_0 in the attributes value lattice if S_1 is lower than S_0 in the state lattice. However, since S_1 is read after S_0, there must exist some other attribute y whose approximation has been updated to a lower value, but this contradicts the assumption that z was the first attribute to update approximation to a lower value. □

Lemma A.14. *For a well-formed attribute $x \in \mathcal{A}$, the fixed-point loop in Case1(x) performs a finite number of iterations.*

Proof. For each iteration i of Case1, by Lemma A.11, there are two cases:

- all attributes in $D(x)$ have reached their fixed-point values before the end of iteration i, or,
- Case2 is executed for all attributes in $D(x)$ during iteration i.

If the first case holds for some iteration i, then i is either the last or penultimate iteration. There can not be more than one additional iteration after i because the next iteration will not be able to update any attribute approximation to a new value, causing *tls.change* to remain FALSE after the assignment at the start of the next iteration, and then leading to the loop exiting after that iteration.

Now, assume that there is an unbounded number of iterations. This implies that the second case must hold for all iterations: Case2 is executed for all attributes in $D(x)$ in each iteration k. However, executing Case2 for all attributes means that *tls.change* is set to TRUE only if at least one attribute changed approximation. Since attribute values are in a lattice, there are only a finite number of possible value updates until no value can be further updated. Additionally, Lemma A.13 shows that all approximation updates are monotonic. Thus, after a finite number of iterations it will not be possible to update any approximation and *tls.change* remains FALSE and the loop ends. □

Lemma A.15. *For a well-formed attribute $x \in \mathcal{A}$, and an iteration i of the loop in Case1(x), Case2(x,i) does not cause unbounded recursion.*

Proof. If there exists unbounded recursion, executing Case2(x,i) leads to a call to Case2(x,i). However, the condition of the **if**-statement for calling Case2 in CEval tests if x has already been computed during the iteration i, by reading $tls.iter(x)$ and comparing against i. In the first execution of Case2(x,i), $tls.iter(x)$ is assigned i before the Compute call, which is the only control flow path that could lead to recursion. Thus, Case2 does not lead to unbounded recursion. □

Now we can finally present the main correctness theorems and proofs using the above lemmas. There are two things we must prove: that CEval always terminates, and that it returns the correct value.

Theorem A.16 (Termination)**.** *For a well-formed circular attribute x, CEval$(x,0)$ terminates.*

Proof. Some of the operations used by CEval terminate due to Java semantics, and the remaining can be ensured to terminate using appropriate library implementations (thread-local data can use non-concurrent data structures). We assume that the following operations terminate:

- reads and writes of thread-local data,
- updating and reading shared approximations,
- unpacking tuples.

To show that CEval terminates, we must show that only a finite number of these operations are performed. For this, it suffices to show that CEval, Case1, Case2, and Case3 are executed a finite number of times. Except the initial call to CEval$(x,0)$, all calls to CEval, Case2, and Case3 are executed via Case1. Additionally, Case2 causes recursion. So, we must show that Case1 executes a finite number of iterations and Case2 never leads to unbounded recursion. These properties are provided by Lemma A.14 and A.15. □

Theorem A.17 (Fixpoint Sound)**.** *For a well-formed circular attribute x, CEval$(x,0)$ computes the least fixed-point value of x.*

Proof. Lemma A.12 shows that Case1(x) only terminates after the shared approximations of all $y \in D(x)$ have reached their fixed-point values. The approximations of all attributes in $D(x)$ then form a simultaneous fixed point. Because $x \in D(x)$, this means that x has reached a fixed-point value. To show that the value of x is the *least* fixed-point value, we must show that the initial approximations of all attributes in $D(x)$ were their respective bottom values. This is ensured by the first **if**-statement at the start of CEval. Because the approximation of each attribute is initially set to (NIL, FALSE), any thread executing CEval will attempt to initialize gv_x to $(\bot_x, \text{FALSE})$ if it sees the initial state. This happens before using the shared approximation, because all uses of the shared approximation are in Case2 which only happens after the first **if**-statement of CEval.

It remains to show that multiple threads can concurrently execute the algorithm without affecting the correctness of each other's results. For this we only need to look at the points in the algorithm where threads communicate - that is, via $read(gv_x)$ and $CAS(gv_x, \cdots)$.

The CAS used to initialize the shared value of x to the bottom value, $\bot_x$, only has an effect for a single invocation of the CAS, and it ensures that all threads read the bottom value of x as the first approximation of x.

In Case1, the shared value is accessed when the fixed-point computation is finished and a thread tries to mark the shared approximation as the final result. The CAS fails if another thread has marked the same result as final. Since the returned value is read from the shared approximation two separate threads will return the same result regardless of which thread performed the successful CAS.

In Case2, the shared approximation is read and used as the local approximation in case it was different from the local approximation. A fundamental property of Case2 is that it should only be able to update the shared approximation to a higher value, which is proven in Lemma A.13. □

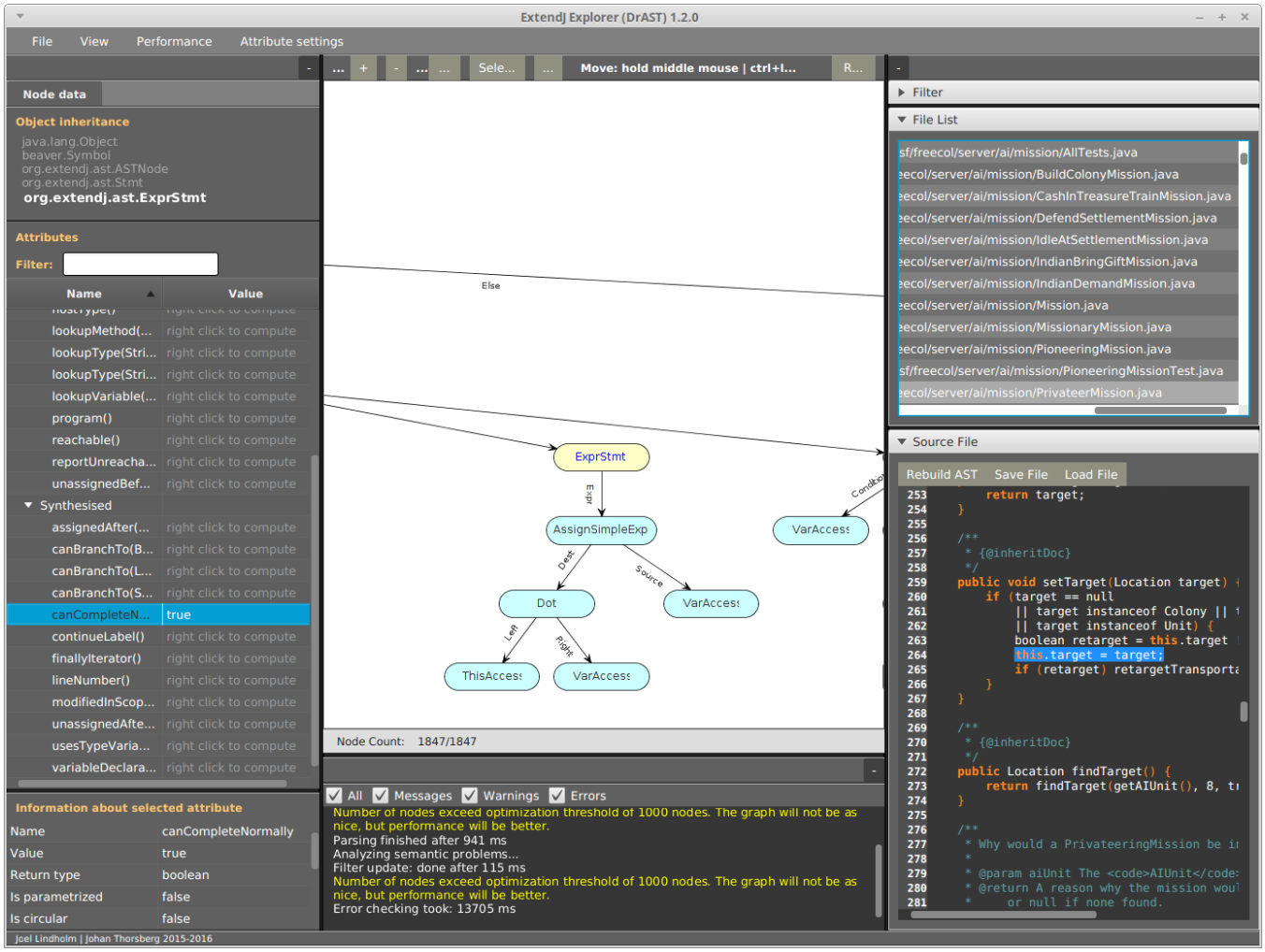

Figure 6. Screenshot of DrAST extended with ExtendJ. The center part of the window contains a graph of the AST of one source file in freecol-0.10.7. The left part of the window contains a list of all attributes in the currently selected node. The `canCompleteNormally` attribute has been selected and manually computed by the user. The right side of the window contains a list of all files in freecol-0.10.7, and below that is a source file view of the currently selected file. The bottom center part of the window shows status messages from DrAST. Note the message about compile-time error checking at the end. No compile-time errors or warnings were present in freecol-0.10.7.

B DrAST Extension

DrAST is a tool for inspecting ASTs in JastAdd-generated compilers. We extended the tool for interactive attribute queries in ExtendJ, using concurrent attribute evaluation. Semantic warnings and errors are computed concurrently in the background, while interactive queries can be issued by the user via the graphical interface. A screenshot of the DrAST Graphical User Interface is seen in Figure 6.

C Generated Code

The algorithms we have shown were designed to be simple to present and prove correct. The actual implementations are slightly different, in order to improve performance. For the most part the control flow is identical, but we try to reduce redundant object creation where possible. We will note a few small but notable differences in the implementation for circular attributes. Our implementation adds a new option in the JastAdd metacompiler which enables concurrent code generation for attribute evaluation.

With JastAdd, attributes are computed by a Java method that JastAdd generates for the node class that the attribute belongs to. To demonstrate generated attribute code, we will use a very simple attribute a, which is specified with the following JastAdd declaration:

```
syn X A.a() circular [new X()] = new X();
```

The attribute a is a simple synthesized attribute that computes an instance of class X. Note that a is not effectively circular. However, the declaration uses the `circular` keyword, so the generated code will evaluate it as if it could be effectively circular. The bottom value for the attribute is computed by the expression in square brackets. The attribute computes the same value as the bottom value expression, so the attribute evaluation will stop after one fixed-point iteration.

Figure 7 shows a slightly simplified version of the generated code for the attribute a. The generated code implements Algorithm 2, with some differences. The largest difference is in how attribute approximations are stored: Shared approximations are stored in an instance of the `CircularAttributeValue` class. This class has a volatile done flag, and an atomic reference to the current approximation value. This is done to avoid creating new tuple objects for each approximation update, and it works similarly to the synthesized attribute memoization in Listing 1. A tuple was used in the algorithm to simplify the presentation and correctness proofs. Our implementation preserves correctness while trying to improve performance slightly. Smaller differences to note about the generated code, compared to the algorithm:

- The Case1, Case2, and Case3 calls have all been inlined into the main attribute method.
- The attribute method does not take an iteration index as parameter. Instead, the iteration index is stored in the thread-local state. The iteration index is reset to zero at the end of each fixpoint iteration.
- All thread-local state is accessed via methods. For example, the `inCircle` method is used to test if the current iteration index is zero, and the `testAndClearChangeFlag` method is used to test and reset the thread-local change flag. The iteration map is also accessed via methods on the thread-local state.
- Attribute value equality tests are done with a static method in the `AttributeValue` class.
- Attributes are identified by the object storing the attribute value, which is unique to each attribute instance, and does not change during attribute evaluation. The `updateIteration` method is used to update the iteration index for an attribute, using the `value` reference as unique attribute identifier. The `observedInCycle` method used to test if an attribute has been tagged with the current iteration index in the thread-local iteration map.
- In practice, data used during fixpoint iterations can be discarded when the fixpoint loop is finished. The `enterCircle` and `leaveCircle` methods are used to initialize and clean up the iteration map.

```
class A extends ASTNode {
  CircularAttributeValue value = new CircAttrVal();

  public X a() {
    if (value.done) {
      return value.get();
    }
    Object prev = value.get();
    if (prev == NIL) {
      // Compute bottom value expression:
      X bottom = new X();
      if (!value.compareAndSet(NIL, bottom)) {
        bottom = value.get();
      }
      prev = bottom;
    }
    ASTState tls = state();
    if (!tls.inCircle()) {
      // CASE1 - start fixed-point loop.
      tls.enterCircle();
      do {
        tls.nextIteration();
        tls.updateIteration(value);
        X next = a_compute();
        if (!AttributeValue.equals(prev, next)) {
          tls.setChangeFlag();
          if (!value.compareAndSet(prev, next)) {
            next = value.get();
          }
        }
        prev = next;
      } while (tls.testAndClearChangeFlag());
      tls.leaveCircle();
      value.done = true;
      return (X) prev;
    } else if (!tls.observedInCycle(value)) {
      // CASE2 - compute new approximation.
      tls.updateIteration(value);
      X next = a_compute();
      if (!AttributeValue.equals(prev, next)) {
        tls.setChangeFlag();
        if (!value.compareAndSet(prev, next)) {
          next = value.get();
        }
      }
      return next;
    } else {
      // CASE3 - reuse prev approximation.
      return (X) prev;
    }
  }
}
```

Figure 7. Generated code for circular attribute a.