Efficient Demand Evaluation of Fixed-Point Attributes using Static Analysis*

Idriss Riouak
idriss.riouak@cs.lth.se
Lund University
Lund, Sweden

Niklas Fors
niklas.fors@cs.lth.se
Lund University
Lund, Sweden

Jesper Öqvist
jesper.oqvist@cognibotics.com
Cognibotics AB
Lund, Sweden

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

Christoph Reichenbach
christoph.reichenbach@cs.lth.se
Lund University
Lund, Sweden

Abstract

Declarative approaches to program analysis promise a number of practical advantages over imperative approaches, from eliminating manual worklist management to increasing modularity. Reference Attribute Grammars (RAGs) are one such approach. One particular advantage of RAGs is the automatic generation of on-demand implementations, suitable for query-based interactive tooling as well as for client analyses that do not require full evaluation of underlying analyses. While historically aimed at compiler frontend construction, the addition of circular (fixed-point) attributes also makes them suitable for dataflow problems. However, prior algorithms for on-demand circular RAG evaluation can be inefficient or even impractical for dataflow analysis of realistic programming languages like Java. We propose a new demand algorithm for attribute evaluation that addresses these weaknesses, and apply it to a number of real-world case studies. Our algorithm exploits the fact that some attributes can never be circular, and we describe a static meta-analysis that identifies such attributes, and obtains a median steady-state performance speedup of ~2.5x and ~22x for dead-assignment and null-pointer dereference analyses, respectively.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; **Formal language definitions**.

Keywords: Attribute Grammars, Circular Attributes, Static Analysis, Demand Analysis, Fixpoint Computations

*This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

SLE '24, October 20–21, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695644

ACM Reference Format:
Idriss Riouak, Niklas Fors, Jesper Öqvist, Görel Hedin, and Christoph Reichenbach. 2024. Efficient Demand Evaluation of Fixed-Point Attributes using Static Analysis. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3687997.3695644

1 Introduction

Static program analysis is a key part of modern software tools, including compilers and static checkers. After first deriving facts from program code, many analyses rely on a fixed-point computation over some lattice to find a solution to a mutually dependent equation system. Typically, this computation is either *data-driven*, exhaustively computing all derivable facts, or *on demand*, computing only the facts necessary to answer a particular query. Demand evaluation can substantially outperform data-driven exhaustive analysis when the analysis client asks for only a subset of the analysis results, e.g., for a dead code elimination that uses constant folding only for branch conditions or for interactive tools that scan only code portions that are visible in the editor.

Reference Attribute Grammars (RAGs) [14] are a high-level declarative formalism for specifying static program analyses in terms of *attributes*, i.e., properties associated with program nodes. These specifications take the form of equations (sometimes called semantic functions) that may introduce dependencies between attributes. RAGs extend Knuth Attribute Grammars (AGs) [20] to allow attribute equations to describe and traverse *references* to other AST nodes, and contemporary RAG frameworks provide facilities to reify additional structures [40] such as Control Flow Graphs (CFGs), and to compute fixed points with the help of *circular attributes* [22] to solve typical dataflow problems [28, 31].

Contemporary RAG compilers [15, 30, 39] translate these equations into attribute evaluation engines that answer attribute queries by recursively evaluating the equations on demand, memoizing intermediate results. When an attribute has a self-dependency, evaluation iterates until the result no longer changes. This evaluation strategy is practical for many applications; for example, the ExtendJ Java compiler, which is specified using RAGs, executes within 3× the execution time of the handwritten reference compiler `javac` [12].

However, for applications that make heavy use of fixed point computations, efficient evaluation may hinge on identifying strongly-connected components (SCCs) over the dependency graph and evaluating them in topological order [17]. This is non-trivial for RAGs as the dependency graph depends on reference attribute values, and is therefore not known before evaluation. Fixed point support in contemporary RAG compilers uses either a heavy-weight algorithm for all potentially cyclic attributes that can distinguish SCCs separated by non-circular attributes [22], or a light-weight algorithm that operates on a subset of the potentially cyclic attributes but cannot distinguish between different SCCs [26].

We here propose a novel evaluation algorithm that overcomes the limitations of the earlier algorithms by combining their insights with a technique that statically identifies attributes that are guaranteed to never be on a cycle. We have implemented and validated our algorithm in the JastAdd metaprogramming system [15], which compiles RAG specifications to Java code. Our approach constructs a call graph from the generated Java code and maps it back into attribute declaration dependencies, which we use to conservatively overapproximate dynamic evaluation dependency cycles. We then feed this information back into JastAdd to allow it to generate more efficient evaluation code.

While our work builds on RAGs, the algorithms are general in that they can be applied to implement any system that exposes a demand analysis as an observationally pure query API on a graph of nodes (e.g., an abstract syntax tree or an abstract syntax graph). We believe that our algorithms could therefore also be useful for compilers built around other query-based architectures, including Microsoft's Roslyn platform and the `rustc` compiler for Rust.

We start by giving a brief background on RAGs and circular attributes (Section 2). We then present our contributions:

- We introduce our new attribute evaluation algorithm (Section 3).
- We propose a novel approach to conservatively identify dependencies in RAGs based on the call graph of the generated evaluation code and explain how our new algorithm uses this information to speed up evaluation (Section 4).
- We evaluate our approach on a set of real-world case studies. Our evaluation shows that our approach can significantly improve the performance of the generated evaluator (Section 5).

We then discuss related work (Section 6) before concluding the paper (Section 7).

```
state S1;
state S2;
state S3;
trans S1 -> S2;
trans S2 -> S1;
trans S2 -> S3;
```

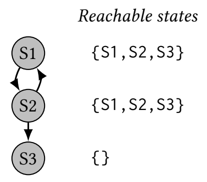


Figure 1. State machine example. Textual syntax (left), visual depiction (middle), and reachable states (right).

```
/* — Abstract Syntax Definitions — */
Machine ::= State* Transition*;
State ::= <Label:String>;
Transition ::= <SourceLabel:String> <TargetLabel:String>;

/* — RAG Attribute Definitions — */
syn Set<State> State.successors();
eq  State.successors() { ... };

syn Set<State> State.reachable()
        circular [new HashSet<State>()];
eq  State.reachable() {
    Set<State> result = new HashSet<State>();
    for (State s: successors()) {
        result.add(s);
        result.addAll(s.reachable());
    }
    return result;
}
```

Figure 2. State machine language definition, comprising the abstract grammar and the RAG specification of `reachable`.

2 Reference Attribute Grammars with Circular Attributes

In this section we introduce Reference Attribute Grammars (RAGs) and circular attributes by defining reachability for a simple state machine language. Consider the example state machine in Figure 1. For each of the machine's three states, we want to compute the set of transitively reachable states.

We demonstrate the analysis in the metacompilation system JastAdd, eliding the concrete syntax definition for brevity. We first parse input state machine programs such as the one in Figure 1 into an abstract syntax tree (AST). Figure 2 (top) shows the abstract grammar: a state machine (`Machine`) consists of a list of states (`State`) and a list of transitions (`Transition`). Each state has a label, and each transition has a source and a target label.

Figure 2 (bottom) shows the definitions of two attributes, `successors` and `reachable`. Each definition specifies the attribute name on its left-hand side and gives a Java method body on the right-hand side that must have no observable side effects. For example, the attribute `successors` defines the set of all successor states for `State`, though we elide the implementation for brevity. Each non-terminal instance (AST node) has its own set of attribute instances. For a state n, our definition of `reachable` computes the following:

$$n.reachable = \bigcup_{s \in n.successors} \{s\} \cup s.reachable$$

Figure 2 writes the right-hand side of this equation in plain Java code, looping over all successors to construct their union. For each attribute, JastAdd generates a namesake method, which here allows `reachable` to access `successors` directly.

The keyword `syn` marks both attributes as so-called *synthesized* attributes, meaning that we evaluate their defining equations in the context of the node that the attribute belongs to. Other kinds of attributes use other contexts; e.g., *inherited* attributes use the parent node context, though this distinction is inessential for the work that we present here.

Since state machines may contain cycles, an instance of `reachable` may depend on itself. We must thus declare `reachable` as `circular`, which tells JastAdd to evaluate it with a fixed-point iteration algorithm (shown in Section 3). Circular attributes must have explicit *bottom values*: here, we use the empty set (`[new HashSet<State>()]`). Since the set of `States` forms a finite lattice on which set union is monotonic, iteration terminates. JastAdd requires attribute definitions to ensure that there can be no infinite chains of updates, e.g. via finite-height lattices and monotonic updates.

When we access an attribute from Java, JastAdd will now compute it on demand and memoize the result. For example, suppose that we have parsed the program from Figure 1 into an AST and have a reference `s1` to the `S1` state node. By calling `s1.reachable()` we will execute the right-hand side of the `reachable` attribute equation, which in turn will call `s1.successors()`, and then recurse by calling `s.reachable()` for each `s` from `s1.successors()`. If we call `reachable()` on the S3 state node, its `successors` attribute will be the empty set, so `reachable` will return the empty set without recursing. For any attribute evaluation, the exact set of attribute instances that we need to evaluate thus depends on the structure of the AST, the equations, and the values of attributes that we have evaluated so far.

3 Circular Attribute Algorithms

We now describe our on-demand algorithms for attribute evaluation in the presence of circular attributes. We first discuss a general framework that captures commonalities across the algorithms, then detail the constituent subalgorithms.

3.1 Preliminaries

In our approach, each non-terminal X is implemented by a corresponding class `X`, and each attribute $X.attr$ by a method `X.attr()`. If `x` is an instance of class `X`, we can thus access the value of an attribute instance $x.attr$ by calling `x.attr()`.

We consider only well-formed RAGs for which each attribute instance will have exactly one defining equation for any possible AST. The defining equation and the attribute may be located in different AST nodes, e.g., if the attribute is inherited rather than synthesized. We abstract away the equation location by introducing a method `X.attr_compute()` for each attribute declaration `X.attr()`. This method will locate the equation in the AST and call a method corresponding to the right-hand side of the equation. In this process, the method will call a number of other attribute instances.

The calls form a dynamic dependency graph where each edge $\langle a, b\rangle$ represents a call from attribute instance a to attribute instance b. When a can transitively reach an attribute instance c along the edges of this dependency graph, we say that c is *downstream* from a, and when a is downstream from a itself, we say that a is *effectively circular*. Since the dynamic dependency graph can depend on dynamically computed reference attributes, we cannot precisely predict whether a given attribute instance is circular in the general case.

3.2 Attribute Declarations and Main Algorithms

To avoid unnecessary fixed-point computation, we require a declaration for each attribute that selects one of three subalgorithms for evaluating that attribute's instances:

Circular An instance of an attribute declared as Circular is allowed to be effectively circular. A Circular attribute instance, $x.attr$, will be evaluated by a fixed-point computation, and has an explicit bottom value, computed by the method `x.attr_bottom_value()`.

NonCircular An instance of an attribute declared as NonCircular is not allowed to be effectively circular. If it is, evaluating the instance will yield a runtime error.

Agnostic An instance of an attribute declared as Agnostic is allowed to be effectively circular, if there is at least one attribute declared as Circular on each cycle it is part of. An Agnostic attribute does not have any explicit bottom value. Instead, its first approximation will be computed based on the approximations of its downstream attributes. If it is on a cycle without any intervening Circular attribute, attempting to evaluate it will yield a runtime error.

We say that a Circular attribute instance and its downstream attributes, up to any NonCircular attribute, belong to the same *fixed-point component*. Thus, NonCircular attributes stratify different fixed-point components into an acyclic component graph. If evaluation starts in one fixed-point component and flows through a NonCircular attribute into another component, we suspend fixed-point computation for the first component until we have reached a fixed point for the second component (Section 3.5). When two strongly connected components are instead directly connected or separated only by Agnostic attributes, we evaluate them as one single fixed-point component.

We consider three different main algorithms for evaluation: BasicStacked, RelaxedMonolithic, and RelaxedStacked. BasicStacked corresponds to the original algorithm by Magnusson [22] and supports Circular and NonCircular attributes. Our version of BasicStacked is somewhat different from Magnusson's version: We keep track of in which fixed-point iteration each attribute was most recently evaluated. This allows for an important optimization where we avoid evaluating an attribute more than once if its value is used more than once in the same iteration. This also allows us to detect if an attribute that is (erroneously) classified as NonCircular is actually on a cycle at runtime. In the paper by Magnusson, the algorithm did not support such detection, but could instead compute the wrong result in case of a specification error like this. The paper only sketched a fix to this problem, and which relied on keeping track of sets of attribute instances for each fixed-point component, which would have slowed down the evaluation substantially.

A consequence of BasicStacked is that all attributes that may have an effectively circular instance, for some AST, must be declared as Circular. This can be impractical for larger systems, like compilers and program analyzers for real languages. For example, it may be the case that a common attribute, say a type attribute, can have instances that are on cycles only for particular language constructs, e.g., local type inference in lambda expressions. Requiring an attribute to be declared as Circular would then give an efficiency penalty when analyzing all other parts of the program where instances of the attribute are actually not on a cycle. To avoid this problem, Öqvist introduced an alternative algorithm that we call RelaxedMonolithic [25, 26], which supports Circular and Agnostic attributes. Agnostic attributes can be on a cycle, but if they are not, their evaluation can be more efficient than for Circular attributes.

When using RelaxedMonolithic, all attributes that are not explicitly declared as Circular are assumed to be Agnostic, so there are no NonCircular attributes that can separate strongly connected components. Therefore, when a Circular attribute is evaluated, all its downstream attributes, both Circular and Agnostic, will be evaluated as part of the same monolithic fixed-point component. As our evaluation will show, this can be very inefficient for demand analyses that start with querying a Circular attribute. To get the best of both BasicStacked and RelaxedMonolithic, we therefore propose a new algorithm, RelaxedStacked that supports all three kinds of attributes. In this algorithm, NonCircular attributes can be used to separate the evaluation into smaller fixed-point components. By using a static conservative analysis of the attribute specification, we can identify attributes that are guaranteed to never be on any cycle in any possible AST, and that can therefore safely be classified as NonCircular.

In the RAG specification, Circular attributes are the only attributes requiring an explicit annotation by the user, i.e., `circular`, like the `reachable` attribute in Figure 2. Attributes that are not declared as `circular`, such as `successors`, are referred to as *normal* attributes. Normal attributes are classified as either NonCircular or Agnostic, depending on the main algorithm used, and on results from the static analysis of the specification in the case of the RelaxedStacked algorithm.

3.3 Subalgorithms and Variables

There is one subalgorithm for each of the three attribute kinds: Circular, NonCircular, and Agnostic. We have formulated the subalgorithms so that all three main algorithms can use different combinations of exactly the same subalgorithms. Hence, BasicStacked corresponds to using the two subalgorithms Circular and NonCircular; RelaxedMonolithic corresponds to using Circular and Agnostic; RelaxedStacked corresponds to using all three subalgorithms. The subalgorithms are shown in Listings 1-3 and use the following key global variables.

`IN_CIRCLE` is a boolean global variable that is `true` when evaluation is ongoing inside a fixed-point component, and `false` otherwise. If an attribute is called when `IN_CIRCLE` is `false`, its final value is returned. If it is called when `IN_CIRCLE` is `true`, an approximation of it is returned.

`CHANGE` is a boolean global variable indicating if any value was changed in the current fixed-point iteration.

`CIRCLE_ITER` is a global object uniquely identifying the current fixed-point iteration.

Once an algorithm has computed the final value of some attribute *X.attr*, the algorithm memoizes the result in an instance variable `X.attr_value`. To keep track of whether the attribute is memoized or not, NonCircular attributes use a boolean instance variable `X.attr_computed`. For Circular and Agnostic attributes, the `X.attr_value` instance variable will hold the current approximation of the value. To monitor the status of `X.attr_value`, we use an instance variable `X.attr_iter` of type `Object`. Initially, this is set to `NOT_INITIALIZED` to indicate that no approximation has yet been computed. Then, in each fixed-point iteration when a new approximation is computed, the `X.attr_iter` is set to the object identifying that iteration. Finally, when it is deduced that the current approximation is the final value of the attribute, this is recorded by setting `X.attr_iter` to the constant object `FINAL_VALUE`.

Evaluation starts when the main program calls an attribute of some node of the AST. Since `IN_CIRCLE` is initially false, this call will return the final memoized value of the attribute. As an effect of this evaluation, other attributes may either be still unevaluated, or have an approximate value, or have their final memoized value. A later call to an attribute with an approximate value will continue its evaluation.

```
// Global variables
boolean IN_CIRCLE = false;
boolean CHANGE = false;
Object CIRCLE_ITER = new Object();

// Global constants
final Object FINAL_VALUE = new Object();
final Object NOT_INITIALIZED = new Object();

class X {
  // Instance variables
  Object attr_iter = NOT_INITIALIZED;
  T attr_value;

  T attr() {
    if (attr_iter == FINAL_VALUE) {
      return attr_value;
    }

    if (attr_iter == NOT_INITIALIZED) {
      attr_value = attr_bottom_value();
    }

    if (!IN_CIRCLE) {// DRIVES
      IN_CIRCLE = true;
      do {
        CHANGE = false;
        CIRCLE_ITER = new Object();
        attr_iter = CIRCLE_ITER;
        T v = attr_compute();
        if (!Objects.equals(attr_value, v)) {
          CHANGE = true;
        }
        attr_value = v;
      } while (CHANGE);
      attr_iter = FINAL_VALUE;
      IN_CIRCLE = false;
      return attr_value;
    } else if (attr_iter != CIRCLE_ITER) {// FOLLOWS
      attr_iter = CIRCLE_ITER;
      T v = attr_compute();
      if (!Objects.equals(attr_value, v)) {
        CHANGE = true;
      }
      attr_value = v;
      return attr_value;
    } else {// ALREADY HANDLED in this iteration
      return attr_value;
  } }

  T attr_compute() {
    // Locate and evaluate defining equation
} }
```

Listing 1. Evaluation of Circular attributes

```
// Global variables
Stack STACK = ...;

class X {
  // Instance variables
  boolean attr_computed = false;
  T attr_value;

  T attr() {
    if (attr_computed) {
      return attr_value;
    }
    if (!IN_CIRCLE) {// NORMAL
      attr_value = attr_compute();
      attr_computed = true;
      return attr_value;
    } else {
      push CHANGE, CIRCLE_ITER on STACK;
      IN_CIRCLE = false;
      attr_value = attr_compute();
      IN_CIRCLE = true;
      CHANGE, CIRCLE_ITER = pop from STACK;
      attr_computed = true;
      return attr_value;
  } }

  T attr_compute() {
    // Locate and evaluate defining equation
} }
```

Listing 2. Evaluation of NonCircular attributes

```
class X {
  // Instance variables
  Object attr_iter = NOT_INITIALIZED;
  T attr_value;

  T attr() {
    if (attr_iter == FINAL_VALUE) {
      return attr_value;
    }
    if (!IN_CIRCLE) {// NORMAL
      attr_value = attr_compute();
      attr_iter = FINAL_VALUE;
      return attr_value;
    } else {
      if (attr_iter != CIRCLE_ITER) {// FOLLOWS
        attr_value = attr_compute();
        attr_iter = CIRCLE_ITER;
        return attr_value;
      } else {// ALREADY HANDLED in this iteration
        return attr_value;
  } } }

  T attr_compute() {
    // Locate and evaluate defining equation
} }
```

Listing 3. Evaluation of Agnostic attributes

3.4 The Circular Subalgorithm

Listing 1 shows the subalgorithm for Circular attributes. The first Circular attribute that is called in a fixed-point component will take the role of *driver*, running the loop of fixed-point iterations. Other Circular attributes in the same component will be *followers*. The algorithm distinguishes three different situations when calling a Circular attribute:

DRIVES An attribute instance takes the role of driver and starts a fixed-point computation. It runs a loop and in each iteration, it calls its `compute` method to get a new approximation of its value, potentially (transitively) calling other Circular attributes in the component, as well as itself.

FOLLOWS An attribute instance other than the driver is called for the first time during the current fixed-point iteration.

$a = \{5\} \cup b \qquad b = \{42\} \cup c \cup d \qquad c = b \cup a \qquad d = c$

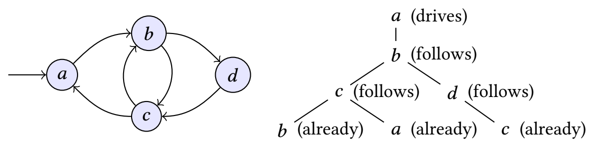


Figure 3. Equations for Circular attributes (top). Dynamic dependency graph (left). Call tree for one iteration (right).

It computes a new approximation by calling its `compute` method, again potentially (transitively) calling other Circular attributes in the component, as well as itself.

ALREADY HANDLED A driver or a follower is called during fixed-point iteration, but has either already been computed in that iteration or is in the process of being computed (i.e., another method invocation for the same attribute is on the call stack). Then it simply returns its current value.

To illustrate how the Circular evaluation works, consider the example in Figure 3 showing an equation system, the dynamic dependency graph, and the tree of method calls for one of the fixed-point iterations, given that a client demands the attribute a by calling `a()`. The attributes a, b, c, and d are all sets of integers, and we assume that they are all declared as Circular. Solving the equation system with a fixed-point iteration, starting out with the empty set as bottom, the solution will be $a = b = c = d = \{5, 42\}$.

Because the evaluation starts with a, this attribute becomes the driver, and will execute the **DRIVES** part of the code, with the fixed-point loop. In each iteration in the loop, it calls its `compute()` method which will in turn call `b()`. The attribute b becomes a follower, and will execute the **FOLLOWS** part of the code which calls its `compute` method that will first call `c()` and then `d()`. Both these attributes also become followers. The attribute c will similarly call `b()` and `a()`, but both these will execute the **ALREADY HANDLED** part of the code, since they are in the process of already being evaluated during the same iteration, and their current approximation is returned directly, without any call to `compute()`, ending the recursion. Similarly, when `d()` calls `c()`, then c has already computed a new approximate value in the same iteration, due to the previous call from `b()` to `c()`, again ending the recursion. We can see from this example that the recursion will terminate, and that each attribute that a depends on will update its value exactly once during an iteration.

Both the driver and the followers update the global variable `CHANGE` to keep track of whether any of the approximations were updated during the current iteration. The driver will loop until there is an iteration where no approximations are updated. The driver and all its followers will then have their final values and can memoize them. For simplicity, the algorithm in Listing 1 only memoizes the driver, i.e., a in the example in Figure 3. Using an optimization called *Last-Cycle* [22], the driver can memoize all the followers as well (b, c, and d in the example). This is accomplished by the driver calling its `compute()` method an extra time, with an extra global flag set to signal to followers that they should memoize their values. (The code for this is elided for brevity.) If this optimization is not used, and a previous follower is called at a later point in time, it will become a driver, and find after one iteration that it can be memoized.

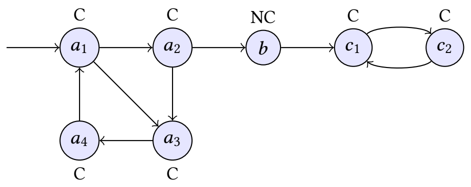


Figure 4. Attribute serving as a bridge between two circular components. C=Circular, NC=NonCircular.

3.5 The NonCircular Subalgorithm

NonCircular attributes use the subalgorithm in Listing 2. An instance of a NonCircular attribute is assumed to not be effectively circular. (If it actually is circular, a runtime error will be raised, see Appendix A.1.) The algorithm distinguishes between two different situations when the NonCircular attribute is called:

NORMAL In the normal case, the attribute is called when there is no ongoing fixed-point computation (i.e., `IN_CIRCLE == false`). It can then simply call its `compute()` method and memoize the result. This is so since when `IN_CIRCLE == false`, any attribute called by the `compute()` method will return its final value.

BRIDGE If the attribute is called during an ongoing fixed-point computation (i.e., `IN_CIRCLE == true`), any downstream Circular attribute will, by definition, belong to a separate fixed-point component. We say that the NonCircular attribute serves as a *bridge* between the upstream and any downstream components. A downstream component should run its own fixed-point computation for efficiency. This is accomplished by the NonCircular attribute stacking the state of the ongoing component (i.e., the variables `CHANGE` and `CIRCLE_ITER`), and setting `IN_CIRCLE` to `false` before calling its `compute()` method. If a Circular attribute is encountered during the `compute()` call, it will start its own fixed-point computation, and finish this computation before returning its value. After the `compute()` call, the stacked variables are restored, and `IN_CIRCLE` is set to `true` again.

Figure 4 shows an example. Here, the NonCircular attribute b serves as a bridge between the components $\{a_1, a_2, a_3, a_4\}$ and $\{c_1, c_2\}$. Suppose the evaluation starts by a call to a_1, which will become the driver of the $\{a_1, a_2, a_3, a_4\}$ component. When a_2 calls b in the first iteration, the component will be stacked. Then b calls c_1 which becomes the driver of a new fixed-point loop for $\{c_1, c_2\}$. This component will loop until c_1 is finalized and memoized, and then return control to b. Then b will compute and memoize its own value, pop the $\{a_1, a_2, a_3, a_4\}$ component, and return control to a_2. In the second iteration of $\{a_1, a_2, a_3, a_4\}$, calling b will directly return b's memoized value.

If the NonCircular b attribute did not stack the component state and set `IN_CIRCLE` to `false`, its call to `compute()` might yield only an approximation rather than a final value, so it would not be safe to memoize b's value here. Essentially, this would lead to the evaluation of all the attributes $\{a_1, a_2, a_3, a_4, b, c_1, c_2\}$ in a big monolithic fixed-point loop, driven by a_1. The NonCircular algorithm in Listing 2 thus has two advantages over a non-stacking variant of the algorithm: it will separate circular components, and it will avoid evaluating the NonCircular attribute more than once.

3.6 The Agnostic Subalgorithm

Agnostic attributes use the subalgorithm in Listing 3. They may be part of a cycle, but only as followers, never as drivers. We can thus assume that any cycle that contains an Agnostic attribute instance also contains a Circular attribute instance to act as the driver. (If an Agnostic attribute is on a cycle without any Circular attribute, the evaluation algorithm raises an error, cf. Appendix A.2.)

In the following, we explicitly note several differences to the previous algorithms that we discuss further in Sections 3.6.1, 3.6.2, and 3.6.3.

Note 1: Agnostic attributes have no bottom values. If they are in a cycle, we compute their first approximations from the bottom values of the Circular attributes on the cycle.

The algorithm distinguishes between three situations when an Agnostic attribute is called:

NORMAL This case is similar to the corresponding case for NonCircular attributes, i.e., there is no ongoing fixed-point computation (`IN_CIRCLE == false`). The attribute will call its `compute()` method and memoize its result.
Note 2: Since the Agnostic attribute may be on a cycle, it may be revisited by another downstream call.

FOLLOWS This case is similar to the corresponding case for Circular attributes. It occurs when there is an ongoing cycle (`IN_CIRCLE == true`), and its recorded iteration is different from the current one (`attr_iter != CIRCLE_ITER`). In this case, the attribute's `compute()` method is called to compute a new approximation.
Note 3: Unlike a Circular attribute, an Agnostic attribute does not compare its current value against the previous one and never updates the `CHANGE` flag.
Note 4: Unlike a Circular attribute, an Agnostic attribute updates its `attr_iter` *after* the call to `compute()`.

ALREADY HANDLED This case is similar to the corresponding one for Circular attributes. Here, there is an ongoing cycle (`IN_CIRCLE == true`), and the recorded `iter` is the same as the current one (`attr_iter == CIRCLE_ITER`). In this case, the attribute is already computed in the current iteration, and it simply returns its current approximation without computing a new one, thus ending the recursion.

The four properties that we noted above affect the performance and correctness of the algorithm, as we detail below.

3.6.1 An Agnostic Attribute Has No Explicit Bottom Value. Since Agnostic attributes have no explicit bottom value (see Note 1), they must not end up in the **ALREADY HANDLED** case without first having computed an approximate value. We can see that this cannot happen, because `attr_iter == CIRCLE_ITER` can happen only if the evaluation previously has passed through all of the **FOLLOWS** code, where `attr_iter` is set to `CIRCLE_ITER`, meaning that the value has been set. For this reason, it is important that `attr_iter` is set to `CIRCLE_ITER` *after* the call to `compute()`, and not before (see Note 4).

3.6.2 An Agnostic Attribute Does Not Set the `CHANGE` Flag. A new approximate value of an Agnostic attribute cannot depend on itself unless this dependency goes via one or more Circular attributes. The Agnostic attribute can only get a new approximation if at least one of these Circular attributes has a new value. But if it has, it will have set the `CHANGE` flag. Therefore, the Agnostic attribute does not need to set the `CHANGE` flag (see Note 3). This also means that if there is an iteration where the `CHANGE` flag was not set, i.e., the fixed-point computation has completed, also the Agnostic attribute will have its final value.

3.6.3 An Agnostic Attribute Executing the NORMAL Case May Be Revisited Downstream. If an Agnostic attribute is on a cycle, but called from outside of circular evaluation, a Circular attribute on the cycle will drive the fixed-point evaluation. The Agnostic attribute will then start by executing the **NORMAL** case, but as a part of this computation, be recursively called (see Note 2). When the fixed-point evaluation has started, the Agnostic attribute will enter the **FOLLOW** code in each iteration, and compute a new approximation. When the fixed-point evaluation terminates, the Agnostic attribute will also have been iterated to its final value, as explained in the previous Section 3.6.2. When the evaluation returns to the Agnostic call executing the **NORMAL** case, it is therefore safe to memoize the attribute.

4 Static Analysis to Identify NonCircular Attributes

For an attribute that is not declared as Circular, we have the option of either declaring it as Agnostic or as NonCircular. Using an Agnostic attribute has the advantage that it is safe to use, even if it is on a cycle (as long as there is some Circular attribute on the cycle).

However, an Agnostic attribute can be quite inefficient if it is called from an upstream cycle, but is not itself on a

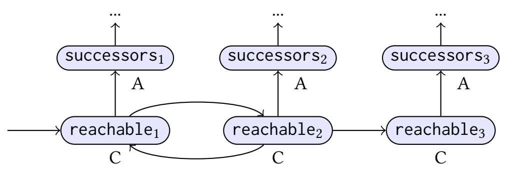


Figure 5. Dynamic dependency graph for the state machine example in Figure 1. C=Circular, A=Agnostic.

cycle. In this case, the upstream cycle will have a Circular attribute that drives a fixed-point loop, and the Agnostic attribute will be evaluated once for each iteration of this loop. Since the Agnostic attribute is not on any cycle in this particular case, each of these evaluations will result in the same value, resulting in unnecessary work.

As an example, this situation occurs for our state machine example from Figure 1. Suppose the `successors` attribute, as well as all its downstream attributes that compute the name analysis, are declared as Agnostic. In this case, we get the dependency graph shown in Figure 5. If the evaluation starts in $\texttt{reachable}_1$, each of the `successors` attributes, as well as all their downstream attributes, will be re-evaluated once per iteration during each of the n fixed-point iterations of $\texttt{reachable}_1$. This can potentially be very inefficient, leading to all downstream attributes being recomputed n times.

Another kind of inefficiency is due to different fixed-point components. If the two components in Figure 4 were separated by an Agnostic rather than by a `NonCircular` attribute, and evaluation starts in the upstream component, then the evaluation could not be done separately for the two components. Instead, all the attributes would be evaluated in a big monolithic component, which is less efficient.

Both these inefficiencies would be avoided if we used `NonCircular` attributes instead of Agnostic ones. However, we only want to use `NonCircular` attributes if we are sure that they will never be on any cycle, for any possible AST.

To solve this problem, we have implemented a tool CAT (https://github.com/idrissrio/cat), that we use to analyze the static call graph of a RAG. We use this analysis to identify attributes that can safely be declared as `NonCircular`.

4.1 Approach Overview

CAT is a general call graph analysis tool for Java, and we use it to analyze the Java code that is generated from a JastAdd RAG specification. An overview of our approach is shown in Figure 6. Initially, the RAG specification is fed into the JastAdd metacompiler, which generates the corresponding evaluation code in Java, using the subalgorithms described in Section 3. We use the Agnostic code as the default for attributes without annotations. Then, CAT analyses the generated evaluation code and computes the corresponding call graph. In this call graph, method declarations are nodes, and edges represent method calls. The CAT tool uses this call

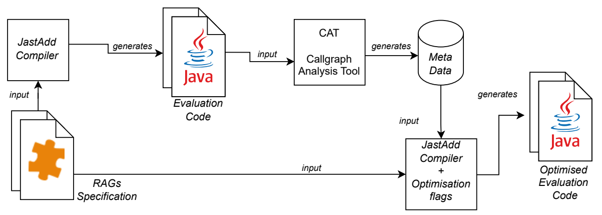


Figure 6. Overview over our approach. The JastAdd metacompiler generates Java code from a RAG specification. CAT determines which attributes may be NonCircular and feeds this information into a second run of JastAdd that can then generate more efficient evaluation code.

graph to identify what attributes can safely be declared as `NonCircular`, and outputs this information as a meta data file. Then JastAdd is run again, this time with the meta data as additional input and with some optimization flags enabled. JastAdd uses this extra information to generate optimized evaluation code where as many as possible of the unannotated attributes use the NonCircular subalgorithm instead of the Agnostic one.

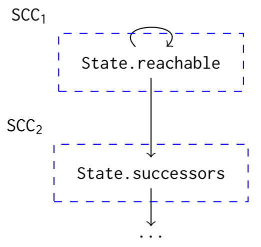
4.2 Call Graph Construction

A call graph is a directed graph that represents the calling relationships between methods in a program. We say that a call graph is *sound* if it contains all the possible method calls that can occur at runtime. One of the main challenges in constructing a sound call graph is effectively managing *dynamic dispatch*, which is the capability to dynamically choose the method to call based on the runtime type of the receiver object. CAT handles dynamic dispatch by using a technique called Class Hierarchy Analysis (CHA) [8]. CHA is a *context- and flow-insensitive* analysis, meaning that it does not consider the context of the method calls and it does not consider the order of the statements in the program. A key aspect of CHA is that given a method call on a receiver object of a certain type, it considers all the possible subclasses of that type, and includes all the methods in these subclasses in the call graph. This way, CHA ensures that all possible method calls are included in the call graph, even if the exact type of the receiver object is not known at compile time.

4.3 Identifying Non-Circular Attributes

Since we are interested in how attributes call each other, we start by constructing a filtered call graph that only includes methods that correspond to attributes. We first obtain this set of methods from annotations generated by the JastAdd metacompiler, and then project all paths from the original call graph onto the filtered one.

To identify non-circular attributes, we employ *Tarjan's algorithm* [37] on the filtered call graph to discover all strongly connected components (SCCs). A SCC constitutes a set of



```
/* − Generated Code − */
Set<State> reachable_compute(){
  Set<State> result =
    new HashSet<State>();
  for (State s : successors()) {
    result.add(s);
    result.addAll(s.reachable());
  }
  return result;
}
```

Figure 7. Call graph between attributes for the state machine language (left), and corresponding `compute` method (right). Dashed rectangles represent strongly connected components (SCC). Green methods are in the filtered call graph. Red methods are other methods in the original call graph.

nodes in a directed graph where each node is reachable from every other node in the set.

Given the SCCs, we can safely mark an attribute n as NonCircular if both of the following conditions hold:

1. n is in an SCC with only a single node, and
2. n does not directly call itself (no self-loop).

These attributes can never be circular for any AST. Attributes not marked as NonCircular by CAT, and not explicitly marked as Circular, are by default considered Agnostic.

To illustrate our approach, we revisit the state machine example from Section 2. Figure 7 shows a part of the call graph, and the `compute()` method for the `reachable` attribute that was used to generate it. The SCC analysis of the call graph identifies two distinct SCCs: SCC$_1$ and SCC$_2$. We see that the attribute `State.successors` can be declared as NonCircular, as both conditions 1 and 2 are met. Conversely, we see that the attribute `State.reachable` cannot be declared as NonCircular, since there is a self-loop in the graph, violating condition 2. This is expected since `State.reachable` is declared as Circular in the RAG, and instances of it are indeed on a cycle in the example in Figure 5.

4.4 Imprecision and Limitations

CAT is unsound on Java code that uses reflection, native calls, or dynamic class loading. Since CAT only analyzes code that JastAdd generates from RAG specifications, and since existing RAG specifications in JastAdd have not made use of these Java features, we currently expect that the practical significance of this limitation is minimal.

An important imprecision arises from attribute instances that recursively call other instances of the same attribute along the AST structure, but without being cyclic. An example would be when all calls between instances of the same attribute go downwards to children. The static approximation of the call graph will then be cyclic, whereas any dynamic instance of this part of the graph will be acyclic.

Since our approach can be adapted to any type of call graph, we expect that more precise call graphs can mitigate this imprecision, help identify more attributes as NonCircular, and thus further improve performance.

Another limitation of our approach is that the algorithm does not distinguish between different dynamic instances of the same static SCC. Generalizing the algorithm to detect dynamic SCCs at evaluation time, and investigating if this pays off in practice, is an interesting line of future research.

5 Evaluation

In this section, we present the evaluation performance of the three algorithms: BasicStacked, RelaxedMonolithic, and RelaxedStacked. We evaluated the RelaxedStacked algorithm across three distinct case studies: the construction of an LL(1) parser, the ExtendJ Java compiler, and IntraJ, an extension of the ExtendJ frontend for data-flow analysis. This section presents the findings for the LL(1) parser construction and IntraJ case studies, as the results for the ExtendJ case study, detailed in Appendix B, align with expectations and do not offer additional insights. For the IntraJ case study, we evaluate both a forward and a backward analysis. We exclude the BasicStacked algorithm from the second and third case studies, as it requires all attributes on a cycle to be declared Circular, which is impractical for complex applications like ExtendJ and IntraJ. When RelaxedStacked is evaluated, we use the CAT tool to automatically infer what attributes can be declared as NonCircular.

The first case study is included to demonstrate that the RelaxedStacked algorithm does not introduce any performance degradation for this application. On the other hand, with the IntraJ case study we demonstrate the advantages of the RelaxedStacked algorithm for more complex applications and for analyses in on-demand settings.

In all of these case studies, the specification includes a cache configuration, i.e., a specification of which attributes to memoize and which to reevaluate on each access. We used the cache configuration supplied by each respective tool, as the optimization of this aspect is a separate research challenge [1, 32].

5.1 Evaluation Setup

System Configuration. Our experiments were conducted on a machine with an Intel Core i7-11700K CPU running at 3.60GHz and equipped with 128 GB RAM. The machine ran Ubuntu 22.04.3 and the benchmarks were executed using OpenJDK Runtime Environment Zulu 8.50.0.53-CA-linux64, build 1.8.0_275-b01. Additionally, for all evaluations, we fixed the Java Virtual Machine (JVM) heap size to 8 GB.

Evaluation Methodology. The measurements were conducted separately for start-up performance on a cold JVM, involving a JVM restart for each run, and for steady-state performance, with a single measurement taken after 49 warmup runs. Each benchmark iteration was repeated 25 times, resulting in a total of 1250 runs for steady-state measurements.

Table 1. Evaluated Java benchmarks, including number of lines of code, number of methods, and version.

Benchmark Name	LOC	#Methods	Version
`antlr`	36525	2070	2.7.2
`pmd`	60749	5325	4.2.5
`struts`	81394	7023	2.3.22
`fop`	102746	8318	0.95
`extendj`	147265	16025	11.0
`castor`	235745	12643	1.3.3
`weka`	245719	14952	revision 7806
`poi`	329366	23816	3.11

Table 2. Startup performance for the Java 1.2 grammar.

BasicStacked$_{\text{old}}$	BasicStacked	RelaxedMonolithic	RelaxedStacked
$4.24_{\pm 0.12}$ ms	$2.92_{\pm 0.05}$ ms	$8.30_{\pm 0.12}$ ms	$2.93_{\pm 0.03}$ ms

For steady-state measurements, we introduced a 300-second timeout since RelaxedMonolithic took a long time to run for some benchmarks. If any of the 49 warmup runs exceeded 300 seconds, we terminated the evaluation process for that particular steady-state measurement, disregarding the remaining warmup runs. The reported metrics include the median values and 95% confidence intervals. We checked the correctness of all three case studies by comparing their results to those from the original tools.

Benchmarks. Table 1 shows the Java benchmark projects for the IntraJ and ExtendJ case studies. They include projects from the DaCapo [4] and Qualitas [38] suites, e.g., `antlr` and `jfreechart`, and projects that we selected to cover a wide range of applications, including the generated Java source code of ExtendJ itself.

The artifact for running all the experiments is available online [27].

5.2 Case Study: LL(1) Parser Construction

LL(1) parsers can be generated by computing the *nullable*, *first*, and *follow* sets for a context-free grammar [2]. Normally, these sets are computed by hand-written fixed-point algorithms. Magnusson et. al. [22] instead formulated the computation as circular attributes. We use the RAG specification from their artifact [23] to evaluate our different algorithms. For comparison, we also ran the original implementation from that artifact (BasicStacked$_{\text{old}}$).

Table 2 shows the startup performance results for computing *nullable*, *first*, and *follow* sets for a Java 1.2 grammar with 155 terminals and 332 productions. We can see that BasicStacked shows a performance improvement of $\frac{4.24}{2.92}$ =~1.45x over BasicStacked$_{\text{old}}$, confirming the efficacy of the improvements that we introduced for BasicStacked in Section 3.2. Furthermore, RelaxedStacked performs as well as BasicStacked, and is significantly faster than RelaxedMonolithic, with a speedup of $\frac{8.30}{2.93}$ =~2.8x. One reason for

$$\begin{tikzcd}
\texttt{CFGNode.in} \arrow[r] \arrow[d, bend left] & \texttt{CFGNode.pred} \arrow[d] \\
\texttt{CFGNode.out} \arrow[u, bend left] & \cdots
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{CFGNode.in} \arrow[d, bend left] & \cdots \\
\texttt{CFGNode.out} \arrow[u, bend left] \arrow[r] & \texttt{CFGNode.succ} \arrow[u]
\end{tikzcd}$$

Figure 8. Static call graph for forward (left) and backward (right) analysis.

this is that RelaxedStacked is able to compute *follow* in a separate fixed-point component than *first* and *nullable*.

5.3 Case Study: IntraJ

IntraJ [28] is a dataflow analyser for Java built as an extension of the ExtendJ Java compiler. It currently supports detecting two kinds of dataflow bugs: null-pointer dereferences and dead assignments. The analyses implemented in IntraJ are instances of the *Monotone frameworks* [24].

Monotone frameworks are a theoretical approach for reasoning about program dataflow properties. This approach provides a flexible and generic framework for expressing and solving dataflow equations, which can be used to reason about a wide range of dataflow properties, e.g., *reaching definitions* and *available expressions* analyses.

The dataflow information is propagated through the program using the *control-flow graph* (CFG), available with the functions *pred* (predecessors) and *succ* (successors). To propagate information from node *n* to its succeeding nodes (in the CFG) and to represent the effect of passing through a node we use the following equations:

$$\text{in}(n) = \bigsqcup_{p \in \text{pred}(n)} \text{out}(p) \quad (1) \qquad \text{out}(n) = \bigsqcup_{p \in \text{succ}(n)} \text{in}(p) \quad (3)$$

$$\text{out}(n) = f_{\text{tr}}(\text{in}(n), n) \quad (2) \qquad \text{in}(n) = f_{\text{tr}}(\text{out}(n), n) \quad (4)$$

Equations (1) and (2) are used to propagate information from the predecessors of a node *n* to *n* itself. Each instantiation or implementation of these equations corresponds to a *forward* analysis. Similarly, the equations (3) and (4) are used to propagate information *backward* in the CFG. The function f_{tr} is called the *transfer function* of the analysis and captures the effect of passing through a node in the CFG.

In IntraJ, *in*, *out*, *succ*, and *pred* are represented by attributes. Figure 8 shows the static call graphs for a forward and a backward analysis. In both graphs, the `CFGNode` class represents a node in the CFG. The attributes `CFGNode.in` and `CFGNode.out` are circular and require a fixed-point computation to compute their values. Our tool CAT will detect that both `CFGNode.pred` and `CFGNode.succ` can never be on a cycle and can thus be declared as NonCircular.

Performance. For IntraJ we conducted the evaluation on two dataflow analyses, namely the *null-pointer dereference* and the *dead assignment* analyses. The null-pointer dereference analysis detects expressions that may cause a null-pointer dereference. The dead assignment analysis detects assignments that are never used. Both the analyses are monotone frameworks, with the difference that the null-pointer

Table 3. Performance of *Dead Assignment Analysis* and *Null-Pointer Dereference Analysis*, comparing RelaxedMonolithic and RelaxedStacked in startup and steady state. The ⏱ symbol indicates that the analysis timed out after 300 seconds.

Benchmark	Dead Assignment Analysis						Null-Pointer Dereference Analysis					
	Start up			Steady State			Start up			Steady State		
	Relaxed-Monolithic	Relaxed-Stacked		Relaxed-Monolithic	Relaxed-Stacked		Relaxed-Monolithic	Relaxed-Stacked		Relaxed-Monolithic	Relaxed-Stacked	
	Time (s)	Time (s)	Speedup	Time (s)	Time (s)	Speedup	Time (s)	Time (s)	Speedup	Time (s)	Time (s)	Speedup
`antlr`	$3.16_{\pm 0.07}$	$1.83_{\pm 0.03}$	× 1.73 ↑	$1.42_{\pm 0.01}$	$0.51_{\pm 0.01}$	× 2.78 ↑	$28.09_{\pm 0.28}$	$2.48_{\pm 0.06}$	× 11.34 ↑	$26.74_{\pm 0.09}$	$0.73_{\pm 0.01}$	× 36.65 ↑
`pmd`	$6.49_{\pm 0.12}$	$3.48_{\pm 0.05}$	× 1.86 ↑	$3.61_{\pm 0.02}$	$1.39_{\pm 0.02}$	× 2.60 ↑	$32.36_{\pm 0.20}$	$4.46_{\pm 0.09}$	× 7.26 ↑	$28.14_{\pm 0.10}$	$1.73_{\pm 0.01}$	× 16.24 ↑
`struts`	$9.32_{\pm 0.18}$	$5.31_{\pm 0.09}$	× 1.75 ↑	$5.18_{\pm 0.07}$	$2.17_{\pm 0.06}$	× 2.38 ↑	$66.97_{\pm 0.85}$	$6.42_{\pm 0.10}$	× 10.43 ↑	$61.74_{\pm 0.74}$	$3.54_{\pm 0.14}$	× 17.45 ↑
`fop`	$8.38_{\pm 0.09}$	$4.74_{\pm 0.05}$	× 1.77 ↑	$5.57_{\pm 0.03}$	$2.08_{\pm 0.09}$	× 2.68 ↑	$83.52_{\pm 0.26}$	$6.04_{\pm 0.09}$	× 13.84 ↑	$79.12_{\pm 0.06}$	$2.99_{\pm 0.08}$	× 26.48 ↑
`extendj`	$40.88_{\pm 0.74}$	$16.11_{\pm 0.21}$	× 2.54 ↑	$37.16_{\pm 0.66}$	$12.94_{\pm 0.35}$	× 2.87 ↑	$1510.75_{\pm 5.99}$	$13.04_{\pm 0.08}$	× 115.87 ↑	≥ 300.00 ⏱	$9.55_{\pm 0.08}$	≥ 31.42 ↑
`castor`	$11.10_{\pm 0.25}$	$6.89_{\pm 0.14}$	× 1.61 ↑	$6.96_{\pm 0.04}$	$3.16_{\pm 0.15}$	× 2.21 ↑	$143.99_{\pm 4.06}$	$8.54_{\pm 0.27}$	× 16.85 ↑	$137.35_{\pm 1.85}$	$4.59_{\pm 0.13}$	× 29.93 ↑
`weka`	$28.12_{\pm 0.09}$	$12.93_{\pm 0.12}$	× 2.17 ↑	$23.50_{\pm 0.20}$	$9.31_{\pm 0.19}$	× 2.52 ↑	$475.89_{\pm 2.66}$	$17.21_{\pm 0.45}$	× 27.65 ↑	≥ 300.00 ⏱	$11.88_{\pm 0.32}$	≥ 25.25 ↑
`poi`	$34.63_{\pm 0.23}$	$16.17_{\pm 0.12}$	× 2.14 ↑	$27.85_{\pm 0.25}$	$11.14_{\pm 0.08}$	× 2.50 ↑	$571.56_{\pm 4.47}$	$21.14_{\pm 0.16}$	× 27.03 ↑	≥ 300.00 ⏱	$15.32_{\pm 0.08}$	≥ 19.59 ↑

dereference analysis is a forward analysis (see equations (1) and (2)), while the dead assignment analysis is a backward analysis (see equations (3) and (4)).

Each analysis is done by querying an attribute in IntraJ that collects all warnings in the benchmark program. This attribute will in turn demand the dataflow `in`/`out` attributes, which in turn demand the `pred`/`succ` attributes. These attributes may in turn demand name- and type analysis attributes as defined by the underlying compiler ExtendJ. Thus, in these analyses, many attributes will be demanded downstream from the circular dataflow attributes. It is therefore expected that RelaxedStacked will perform better than RelaxedMonolithic.

Table 3 shows the performance of the RelaxedMonolithic and RelaxedStacked algorithms for both the dead assignment and the null-pointer dereference analyses. The start up measurements include both parsing and analysis and the steady state measurements include only analysis. The results show significant performance improvements for the RelaxedStacked algorithm compared to the RelaxedMonolithic algorithm. For dead assignment analysis, the startup speedup of RelaxedStacked ranges from ~1.7x to ~2.5x, with a median speedup of around ~1.8x. In steady-state, the speedup becomes even more significant, ranging from ~2.2x to ~2.8x, with a median speedup of around ~2.5x. One reason for the speedup is that RelaxedMonolithic will compute the control-flow graph (`succ` and its downstream attributes) in each fixed-point iteration, whereas for RelaxedStacked `succ` will be classified as NonCircular, and will only be computed once.

For null-pointer dereference analysis, the results show an even more significant improvement for RelaxedStacked, with a speedup between ~7x and ~115x for startup performance, with a median of ~15.3x. For steady state performance, the speedup was between ~16x and ~35x, with a median of ~22x, disregarding 3 measurements that timed out for RelaxedMonolithic. The reason for the larger difference and variation is that this is a forward analysis which uses the `pred()` attribute which is defined as the reverse of the successor, leading to even more downstream attributes being unnecessarily reevaluated for the RelaxedMonolithic algorithm. We can observe an extremely high speedup for the `extendj` benchmark, where the RelaxedStacked algorithm is approximately 115 times faster than the RelaxedMonolithic algorithm. Upon closer inspection, we discovered that the `extendj` benchmark has a very large generated method for parsing Java source code, consisting of 6844 lines of code, and where the problems of RelaxedMonolithic become particularly pronounced.

The experiments in Table 3 analyze complete benchmark programs. To further demonstrate the on-demand nature of the algorithms, we ran the analyses on sets of randomly selected methods, querying an attribute summarizing the results for each of the selected methods. For each benchmark, we randomly selected 10, 20, 50, 100, and 200 methods to run the experiment, and report the steady-state performance of the analyses. We present the results exclusively for `pmd` as findings across other projects are similar. Figure 9 shows the results for the null-pointer dereference analysis. We report both the execution time and the number of times a `succ` attribute was evaluated, and it can be observed that these metrics correlate closely. We can also note that the speedups for RelaxedStacked are consistent with the earlier results in Table 3 running on the whole benchmark, approaching similar numbers as the number of methods increases. This experiment demonstrates the on-demand nature of the algorithms, resulting in very short response times when only a subset of the results are demanded, and with similar performance profiles as for the complete programs.

6 Related Work

Knuth's original attribute grammars [20] disallowed cyclic dependencies. Farrow [13] and Jones et al. [19] independently introduced circular but well-defined attribute grammars. Farrow's approach statically analyzes dependencies,

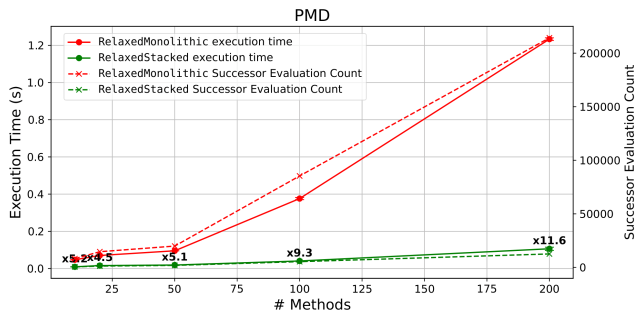


Figure 9. Steady-state performance of null-pointer dereference analysis for randomly selected sets of methods of the `pmd` benchmark. Solid lines represent execution time (left axis, seconds). Dashed lines represent successor attribute evaluations (right axis, count).

while Jones' relies on a dynamic dependency graph to identify strongly connected components and supports incremental evaluation. Sasaki and Sassa extend attribute grammars with remote links [29], a restricted form of reference attributes, and describe exhaustive circular evaluation over them. In contrast to our work, none of these approaches support demand evaluation, nor general reference attributes. Sasaki and Sassa's remote links must be set before evaluation, i.e., may not be computed by attributes.

Boyland describes demand-driven evaluation for circular attributes in the presence of so-called remote attributes (similar to reference attributes), but gives no explicit evaluation algorithm [5]. Hesamian recently added statically scheduled support for circular attributes to Boyland's remote attribute system APS [16]. However, this implementation is exhaustive (not demand-driven) and the experimental results are limited to comparatively small grammars and synthetic input. The largest grammar in this work is the nullable-first-follow grammar from Magnusson [22] that we discuss in Section 5.2.

Previous demand-driven algorithms for RAGs with circular attributes include BasicStacked$_{\text{old}}$ by Magnusson et al. [22] and RelaxedMonolithic by Öqvist et al. [25, 26]. Our algorithm generalizes both. Öqvist further presents a concurrent lock-free attribute evaluation algorithm based on RelaxedMonolithic, while Söderberg et al. [33] present an extension of BasicStacked to handle circular higher-order attributes. Both contributions are orthogonal to the ones presented here.

Kiama [30] and Silver [39] are two RAG systems that could benefit from using the RelaxedStacked algorithm. Kiama already supports circular attributes by implementing one of the basic algorithms described by Magnusson et al. [22].

Logic programming, especially in Datalog, is the basis for other declarative approaches to program analysis. Datalog-based analysis frameworks include Doop [6], which supports points-to analysis of Java bytecode, and the commercial .QL system [7]. These generally follow a two-phase process that first extracts program facts into a database and then evaluates logical rules until it reaches a fixed point, though Dura et al. describe how both phases can be integrated into a single declarative framework [11]. While these approaches are often limited to boolean lattices, Madsen et al. [21] demonstrate a Datalog variant with support for general complete finite-height lattices. Unlike our work, Datalog frameworks generally use exhaustive evaluation, though some Datalog-based tools use on-demand evaluation strategies based on logical rule rewriting to use so-called Magic Sets [3], or hybrid strategies, as in the Clog framework [10], which combines exhaustive evaluation with on-demand queries to a compiler frontend.

Stein et al. present a general approach to demand-driven abstract interpretation over a pre-computed CFG, with cyclic computations over infinite-height domains [36], though their experimental results are limited to synthetic workloads. Other demand-driven approaches range from frameworks for distributive interprocedural dataflow analysis [9, 18] to points-to analysis for full languages like Java [34, 35]. It is an area of future work to investigate how the RAG approach can be applied to similar problems.

7 Conclusion

We have presented a new formulation of demand-driven evaluation of Reference Attribute Grammars with circular (fixed-point evaluated) attributes. Our approach integrates three attribute kinds, Circular, Agnostic, and NonCircular, in our new RelaxedStacked algorithm, improving upon previous algorithms that only supported combining Circular with either NonCircular or Agnostic attributes.

Our experiments show that effective use of NonCircular attributes is crucial to efficient evaluation. Since manually selecting NonCircular attributes is challenging and error-prone, we perform a call graph analysis on the RAG to automatically identify NonCircular attributes, ensuring correctness and efficiency.

We have evaluated the new algorithm on LL(1) parser construction, on a Java compiler, and on two intraprocedural dataflow analyses for Java. In the parser case study, RelaxedStacked matched the performance of BasicStacked and was 2.8x faster than RelaxedMonolithic. For the Java compiler, RelaxedStacked matched the performance of RelaxedMonolithic, which was expected since this application contains few circular attributes. For the dataflow analyses, we observed substantial speedups for RelaxedStacked over RelaxedMonolithic: a 1.8x median improvement in startup and 2.5x in steady-state for dead assignment analysis, and 15.3x and 22x, respectively, for null-pointer dereference analysis. We also conducted experiments by sampling results from the benchmark programs, demonstrating that our algorithm works efficiently even when only a subset of the program's results are demanded.

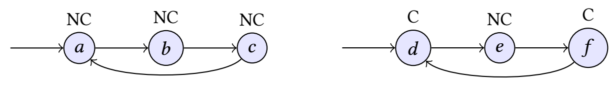


Figure 10. Attributes *a*, *b*, *c* and *e* are incorrectly specified as `NonCircular`.

A Safe Evaluation of Incorrectly Specified RAGs

This appendix details how a runtime error is raised if an Agnostic or NonCircular attribute is incorrectly specified.

A.1 Safe Evaluation of Incorrectly Specified NonCircular Attributes

It is important that the algorithms are safe in that they do not compute the wrong result even if the developer incorrectly declares an attribute as NonCircular, while for some AST, it turns out to be effectively circular. Rather, a runtime error should be raised in this case. In our algorithm, a NonCircular attribute on a cycle will lead to endless recursion, and therefore raise a stack overflow error. As an alternative solution, it would be straightforward to extend the algorithm to use one additional flag per NonCircular attribute instance to track and report such circular dependencies. (The code for this solution is elided for brevity.)

To see that the algorithm is safe in this respect, we can consider two cases, as shown in Figure 10. In the left example, all the attributes *a*, *b*, *c*, are (incorrectly) declared as NonCircular, although they are on a cycle. Suppose that evaluation starts by calling *a*. All the attributes will take the **`NORMAL`** branch in the algorithm, and just continue calling each other in an endless recursion, eventually leading to stack overflow.

In the example to the right, the attribute *e* is (incorrectly) declared as NonCircular and the others (correctly) as Circular. Suppose that evaluation starts in one of the Circular attributes, say *d*. It will become a driver, start a fixed-point evaluation, and start an iteration with a unique id, say 1. This id is saved in its `d_iter` instance variable. When the evaluation reaches *e*, it takes the **`BRIDGE`** branch, and stacks the current circular evaluation. The evaluation then reaches *f* which will become the driver of a new fixed-point evaluation, with a new unique iteration id, say 2. When the evaluation reaches *d* again, it will become a follower since circular evaluation is ongoing. It will call its compute method since its stored iteration id (1) differs from the current one (2). When the evaluation again reaches *e*, it again takes the **`BRIDGE`** branch, and stacks the current evaluation. The evaluation continues this way, stacking cyclic evaluation for every visit to the *f* attribute, leading to endless recursion and eventually stack overflow.

If the evaluation instead starts in the NonCircular *e*, it will first take the **`NORMAL`** branch, but at the next visit, it will take the **`BRIDGE`** branch, and lead to the same kind of endless recursion.

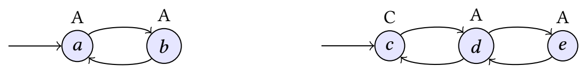


Figure 11. Attributes *a*, *b*, *d* and *e* are incorrectly specified as Agnostic.

A.2 Safe Evaluation of Incorrectly Specified Agnostic Attributes

If an Agnostic attribute is on a cycle without any Circular attribute, the algorithm must be safe in that it does not return an incorrect value, but instead raises a runtime error. As for the NonCircular attributes, we will use endless recursion, i.e., stack overflow, to identify such an error. (As for NonCircular attributes, an alternative solution using a boolean flag could be used, but elided here for brevity.)

To see that the algorithm is safe in this respect, we consider two cases, as shown in Figure 11. In the left example, an incorrectly specified Agnostic attribute *a* is called from outside any cyclic evaluation. It thus enters the **`NORMAL`** code, and calls *b*, which also enters its **`NORMAL`** code. Then *b* calls *a* which again enters the **`NORMAL`** code. We see that this leads to endless recursion and eventually stack overflow.

In the right example, an incorrectly specified Agnostic attribute *d* is called from inside a cyclic evaluation. Here, the evaluation starts with the Circular attribute *c* which becomes the driver. When *d* is reached, it will execute the **`FOLLOWS`** code, and call *e*. The *e* attribute is also Agnostic, and also executes the **`FOLLOWS`** code, and calls *d* again. Since the value of `attr_iter` for *d* is unchanged, and thus still different from `CIRCLE_ITER`, the *d* attribute will again execute the **`FOLLOWS`** code, and we have endless recursion, eventually leading to stack overflow. It is important that the `attr_iter` is not set until after the call to `compute` to get this behavior (relating to Note 4 in Section 3.6).

B Case Study: ExtendJ

ExtendJ [12] is a Java compiler supporting Java 11, built using the metacompilation system JastAdd [15]. ExtendJ uses the RelaxedMonolithic algorithm introduced by Öqvist [25, 26]. It cannot be run with the BasicStacked algorithm because it includes a number of attributes that are effectively circular only on rare occasions, and that are not declared as Circular. It can be run with our new RelaxedStacked algorithm, but we do not expect big performance differences. The reason is that ExtendJ has relatively few circular attributes, and these are typically downstream from the error checking and code generation attributes that drive the attribute evaluation.

In Table 4 we present performance results for the ExtendJ compiler, showing both startup and steady-state performance for both RelaxedMonolithic and RelaxedStacked. As expected, the results for the two algorithms are very similar.

Table 4. Performance of ExtendJ compilation (in seconds) of the benchmarks, comparing RelaxedMonolithic and RelaxedStacked in startup and steady state.

Benchmark	Start up			Steady State		
	Relaxed-Monolithic	Relaxed-Stacked		Relaxed-Monolithic	Relaxed-Stacked	
	Time (s)	Time (s)	Speedup	Time (s)	Time (s)	Speedup
antlr	$1.75_{\pm 0.03}$	$1.76_{\pm 0.05}$	≈	$0.42_{\pm 0.00}$	$0.42_{\pm 0.00}$	≈
pmd	$4.03_{\pm 0.07}$	$4.03_{\pm 0.08}$	≈	$1.31_{\pm 0.02}$	$1.30_{\pm 0.01}$	≈
struts	$5.32_{\pm 0.16}$	$5.14_{\pm 0.16}$	≈	$1.76_{\pm 0.05}$	$1.74_{\pm 0.04}$	≈
fop	$4.99_{\pm 0.19}$	$4.96_{\pm 0.15}$	≈	$1.72_{\pm 0.03}$	$1.71_{\pm 0.06}$	≈
extendj	$6.77_{\pm 0.14}$	$6.84_{\pm 0.12}$	≈	$3.92_{\pm 0.10}$	$3.90_{\pm 0.04}$	≈
castor	$8.17_{\pm 0.29}$	$7.98_{\pm 0.18}$	≈	$3.31_{\pm 0.09}$	$3.20_{\pm 0.03}$	≈
weka	$9.88_{\pm 0.16}$	$9.63_{\pm 0.17}$	≈	$4.77_{\pm 0.34}$	$4.63_{\pm 0.09}$	≈
poi	$14.18_{\pm 0.40}$	$14.54_{\pm 0.53}$	≈	$8.00_{\pm 0.10}$	$7.86_{\pm 0.06}$	≈

References

[1] M. Abadi, B. W. Lampson, and J. Lévy. 1996. Analysis and Caching of Dependencies *(ICFP'96, 6)*. 83–91.

[2] A. W. Appel. 1998. *Modern Compiler Implementation in C.* Cambridge University Press.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs *(PODS'86)*. 1–15.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis *(OOPSLA'06)*. 169–190.

[5] J. T. Boyland. 1996. *Descriptional Composition of Compiler Components.* Ph. D. Dissertation. University of California, Berkeley.

[6] M. Bravenboer and Y. Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses *(OOPSLA'09)*. 243–262.

[7] O. De Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. 2007. .QL: Object-oriented queries made easy *(GTTSE'07, LNCS 5235)*. 78–133.

[8] J. Dean, D. Grove, and C. Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis *(ECOOP'95)*. 77–101.

[9] E. Duesterwald, R. Gupta, and M. L. Soffa. 1995. Demand-driven Computation of Interprocedural Data Flow *(POPL'95)*. 37–48.

[10] A. Dura and C. Reichenbach. 2024. Clog: A Declarative Language for C Static Code Checkers *(CC'24)*. 186–197.

[11] A. Dura, C. Reichenbach, and E. Söderberg. 2021. JavaDL: Automatically Incrementalizing Java Bug Pattern Detection *(OOPSLA'21)*. 1–31.

[12] T. Ekman and G. Hedin. 2007. The JastAdd extensible Java compiler *(OOPSLA'07)*. 147–152.

[13] R. Farrow. 1986. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars *(CC'86)*. 85–98.

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

[15] G. Hedin and E. Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Sci. Comput. Program.* 47, 1 (2003), 37–58.

[16] S. Hesamian. 2023. *Statically Scheduling Circular Remote Attribute Grammars.* Ph. D. Dissertation. University of Wisconsin-Milwaukee. Theses and Dissertations. 3383.

[17] S. Horwitz, A. J. Demers, and T. Teitelbaum. 1987. An Efficient General Iterative Algorithm for Dataflow Analysis. *Acta Inf.* 24, 6 (1987), 679–694.

[18] S. Horwitz, T. W. Reps, and S. Sagiv. 1995. Demand Interprocedural Dataflow Analysis *(FSE'95)*. 104–115.

[19] L. G. Jones and J. Simon. 1986. Hierarchical VLSI Design Systems Based on Attribute Grammars *(POPL'86)*. 58–69.

[20] D. Knuth. 1968. Semantics of Context-Free Languages. *Math. Syst. Theory* 2, 2 (1968), 127–145.

[21] M. Madsen, M. Yee, and O. Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices *(PLDI'16)*. 194–208.

[22] E. Magnusson and G. Hedin. 2007. Circular reference attributed grammars — their evaluation and applications. *Sci. Comput. Program.* 68, 1 (2007), 21–37.

[23] E. Magnusson and G. Hedin. 2007. CRAG artifact. https://bitbucket.org/jastadd/crag-artifact. Accessed: 2024-09-12.

[24] F. Nielson, H. R. Nielson, and C. Hankin. 2010. *Principles of Program Analysis.* Springer.

[25] J. Öqvist. 2018. *Contributions to Declarative Implementation of Static Program Analysis.* Ph. D. Dissertation. Lund University, Sweden. http://lup.lub.lu.se/record/82b210fc-6d15-4f0a-82ff-24b024925d23

[26] J. Öqvist and G. Hedin. 2017. Concurrent circular reference attribute grammars *(SLE'17)*. 151–162.

[27] I. Riouak, N. Fors, J. Öqvist, G. Hedin, and C. Reichenbach. 2024. Efficient Demand Evaluation of Fixed-Point Attributes Using Static Analysis (Artifact). https://doi.org/10.5281/zenodo.13365896

[28] I. Riouak, C. Reichenbach, G. Hedin, and N. Fors. 2021. A Precise Framework for Source-Level Control-Flow Analysis *(SCAM'21)*. 1–11.

[29] A. Sasaki and M. Sassa. 2003. Circular Attribute Grammars with Remote Attribute References and their Evaluators. *New Generation Computing* 22, 1 (2003), 37–60.

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

[31] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. 2013. Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level. *Sci. Comput. Program.* 78, 10 (2013), 1809–1827.

[32] E. Söderberg and G. Hedin. 2010. Automated Selective Caching for Reference Attribute Grammars *(SLE'10, LNCS 6563)*. 2–21.

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

[34] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java *(ECOOP'16)*. 22:1–22:26.

[35] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. 2005. Demand-Driven Points-to Analysis for Java *(OOPSLA'05)*. 59–76.

[36] B. Stein, B. E. Chang, and M. Sridharan. 2021. Demanded abstract interpretation *(PLDI'21)*. 282–295.

[37] R. Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (June 1972), 146–160.

[38] E. Tempero, G. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies *(APSEC'10)*. 336–345.

[39] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. 2010. Silver: An extensible attribute grammar system. *Sci. Comput. Program.* 75, 1-2 (2010), 39–54.

[40] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. 1989. Higher Order Attribute Grammars *(PLDI'89)*. 131–145.