Using Static Analysis to Improve the Efficiency of Program Analysis

Idriss Riouak, 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 make them suitable also for dataflow problems. However, we demonstrate that previous algorithms for on-demand circular RAG evaluation can be impractical and/or inefficient for dataflow analysis of real 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 results show a significant improvement in the performance of the generated code, with a median steady-state performance speedup of ~2.5x for a dead-assignment analysis and ~18x for a null-pointer dereference analysis.

1 INTRODUCTION

Static program analysis is a key part of modern compilers, optimizers, bug- and vulnerability detectors, and many other software tools. 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 [5]. 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 analysis that uses constant folding only to evaluate branch conditions or for interactive tools that use bug detectors only to check the visible part of a program.

Reference Attribute Grammars (RAGs) [10] are a high-level declarative formalism for specifying static program analyses in terms of *attributes*, i.e., properties associated with program nodes.

RAGs support the declarative specification of dataflow and similar fixed-point problems through the use of *circular attributes*, i.e., attributes that are allowed to depend (transitively) on themselves [16]. The use of references and circular attributes makes it possible to specify demand analyses on graphs that are themselves computed on demand.

A general efficient approach for solving a fixed-point equation system is to identify the strongly-connected components of the dependency graph, and iterate each component separately, visiting them in a data-driven topological order [13]. However, for RAGs (in contrast to AGs), the dependency graph is not known a priori (before evaluation starts), since many dependencies follow the graphs constructed from the computed reference attributes.

Circular attributes were originally proposed for Knuth attribute grammars by Farrow [9] and Jones et al. [14], and used data-driven algorithms. The original demand algorithm for circular attributes in RAGs was proposed by Magnusson et al. [16]. It could isolate certain strongly connected components to achieve fast fixed-point computation. However, this algorithm requires the developer to explicitly declare all attributes as circular that can be on a dependency cycle for some AST. This can be both impractical and inefficient for large-scale systems. Öqvist proposed a relaxed version of the algorithm to avoid these problems, requiring only one of the attributes on any cycle to be declared as circular [17]. While this algorithm can work well for some applications, it introduces a significant overhead for attributes that are not on a cycle.

In this paper, we propose a new evaluation algorithm for RAGs to overcome the drawbacks of the previous algorithms. Our algorithm combines ideas from the two earlier algorithms, and additionally provides a technique to statically identify attributes that are guaranteed to never be on a cycle.

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). Many recent compilers are built using such a query-based architecture, including Microsoft's Roslyn platform[1] and the `rustc` compiler for Rust[2]. We believe that our algorithms could also be useful for solving fixed-point problems in such systems.

We have implemented our new algorithm in the JastAdd metaprogramming system [11], a system that supports RAG specifications and generates implementations in the form of Java code. Our approach uses the call graph for the generated Java code to identify attribute declaration dependencies and to conservatively identify potential cycles. This information is then used by JastAdd to generate more efficient evaluation code.

We start by giving an overview of existing algorithms for demand-driven evaluation of RAGs (Section 2). We then present our contributions:

- We introduce our new attribute evaluation algorithm (Section 2).
- We propose a novel approach to conservatively identify dependencies in RAGs based on the call graph of the generated evaluation code and explain how this can be used by our new algorithm to speed up evaluation (Section 3).
- 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 4).

We then discuss related work (Section 5). Finally, we conclude the paper (Section 6).

Supervisors: Niklas Fors, Görel Hedin, and Christoph Reichenbach.
This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

[1]https://github.com/dotnet/roslyn
[2]https://rustc-dev-guide.rust-lang.org

2 CIRCULAR ATTRIBUTE ALGORITHMS

This section describes our on-demand algorithms, i.e., RelaxedStacked, for attribute evaluation in the presence of circular attributes. We start with some preliminaries, then we discuss the exising algorithms, and we conclude with a description of our new algorithm.

2.1 Preliminaries

For the purpose of attribute evaluation, we distinguish between three kinds of attribute declarations: Circular, NonCircular, and Agnostic:

Circular An instance of an attribute declared as Circular is allowed to be effectively circular. A Circular attribute instance will be evaluated by a fixed-point computation, and has an explicit bottom value.

NonCircular An instance of an attribute declared as NonCircular is not allowed to be effectively circular. If it is, trying to call it will give a runtime error.

Agnostic An instance of an attribute declared as Agnostic is allowed to be effectively circular, as long as 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, if the attribute is part of a fixed-point computation, its first approximation will be computed based on the approximations of its downstream attributes[3]. If it is on a cycle without any Circular attribute, attempting to evaluate it will result in a runtime error.

A Circular attribute instance and its downstream attributes, up to any NonCircular attribute, are said to belong to the same *fixed-point component*. This way, NonCircular attributes separate 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 fixed-point component, the first component will be stacked during the evaluation of the second component. Strongly connected components that are directly connected with an edge, or separated only by Agnostic attributes, will be evaluated as part of the same fixed-point component.

2.2 Evaluation Algorithms

We consider three different main algorithms for evaluation: BasicStacked, RelaxedMonolithic, and RelaxedStacked. BasicStacked corresponds to the original algorithm by Magnusson [16] and supports Circular and NonCircular attributes.

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 [17], 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.

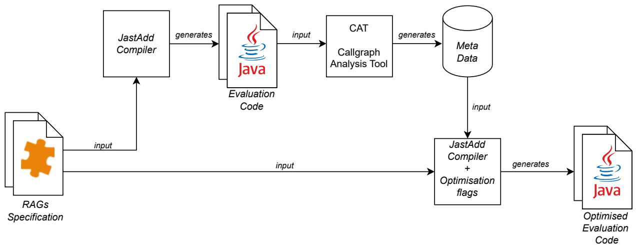

Figure 1: Approach overview: detection of non-circular attributes.

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. For the sake of brevity, we will not describe the algorithms in detail here, but refer the reader to the original papers [16, 17].

3 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, Agnostic attributes can in some cases be quite inefficient. These inefficiencies can be avoided by using 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[4], 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`.

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 1. Initially, the RAG specification is fed into the JastAdd metacompiler, which generates the corresponding evaluation code in Java. Then, CAT analyses the generated evaluation code and computes the corresponding call graph. In this call graph, method declarations

[3]If there is a path from an attribute instance a to an attribute instance b, we say that b is *downstream* from a.

[4]https://github.com/IdrissRio/cat

are nodes, and edges represent method calls. The CAT tool will use this call graph to identify what attributes can safely be declared as `NonCircular`, and output 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 information to generate optimized evaluation code where as many as possible of the unannotated attributes use the NonCircular evaluation algorithm instead of the Agnostic one.

3.1 Identifying Non-Circular Attributes

Since we are interested in how attributes call each other, we start by constructing a filtered call graph using CAT, including only the methods that correspond to attributes. Paths over other methods in the original call graph are projected to edges in the filtered graph. To identify which methods correspond to attributes, we use annotations generated by the JastAdd metacompiler.

To identify non-circular attributes, we employ *Tarjan's algorithm* on the filtered call graph to discover all strongly connected components (SCCs). A SCC constitutes a set of nodes in a directed graph where each node is reachable from every other node in the set.

Once the SCCs are computed, we can safely mark an attribute *n* as NonCircular if both of the following conditions hold:

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

4 EVALUATION

In this section, we present the evaluation performance of the three algorithms: BasicStacked, RelaxedMonolithic, and RelaxedStacked. The evaluation includes two distinct case studies: the construction of an LL(1) parser, and IntraJ, an extension of the ExtendJ [8] Java compiler frontend for data-flow analysis. For the latter, we do benchmarks both on a forward and a backward analysis. BasicStacked is only evaluated in the first case study as it requires all attributes on a cycle to be declared as circular, which is not practical for complex applications like 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 applications that were the driving forces behind the development of the BasicStacked and RelaxedMonolithic algorithms. The second case study is included to demonstrate the advantages of the RelaxedStacked algorithm for more complex applications and for analyses in on-demand settings.

4.1 Evaluation Setup

System Configuration. Our experiments were conducted on a machine with Intel Core i7-11700K CPU running at 3.60GHz and equipped with 128 GB RAM.

Evaluation Methodology. The measurements were conducted separately for start-up performance on a cold Java Virtual Machine (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. For steady-state, we introduced a 300 seconds timeout since RelaxedMonolithic took a long time to run for some benchmarks. The reported metrics include the median values and 95% confidence intervals.

Benchmark	LOC	#Methods	Version
`commons-cli`	6235	585	1.5
`commons-jxpath`	24320	2030	1.3
`pmd`	60749	5325	4.2.5
`struts`	81394	7023	2.3.22
`jfreechart`	95664	6980	1.0.0
`extendj`	147265	16025	11.0
`weka`	245719	14952	revision 7806

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

#T	#P	BasicStacked	RelaxedMonolithic	RelaxedStacked
155	332	$2.92_{\pm 0.05}$	$8.30_{\pm 0.12}$	$2.93_{\pm 0.03}$

Table 2: Startup performance results for the Java 1.2 grammar benchmark. Includes the number of terminals (#T) and productions (#P). The measurements are reported in milliseconds.

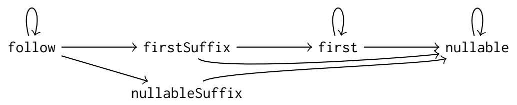

Figure 2: Simplified static call graph of the attributes defining the *nullable*, *first*, and *follow* sets.

Benchmarks. The Java benchmark projects used for the IntraJ case study are shown in Table 1. These benchmarks include various projects such as `pmd` and `jfreechart`, including the generated Java source code of ExtendJ itself.

4.2 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 [1]. Normally, these sets are computed by hand-written fixed-point algorithms. Magnusson et. al. [16] instead formulated the computation as circular attributes. We use the RAG specification from their artifact to evaluate our different algorithms.

Table 2 shows the startup performance results for computing *nullable*, *first*, and *follow* sets for a Java 1.2 grammar. From this table we can observe that the RelaxedStacked algorithm 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 this is that RelaxedStacked is able to compute *follow* in a separate fixed-point component than *first* and *nullable*.

Figure 2 shows the static call graph between a subset of the attributes involved. The attributes `follow`, `first`, `nullable` are declared as Circular. Our tool CAT infers that five attributes can be declared as NonCircular. Among these, we can find the attributes `firstSuffix` and `nullableSuffix`, which will break the circular evaluation when RelaxedStacked is used and when the attribute `follow` is called. Furthermore, CAT identifies that the attributes `nullable` and `first` for `Terminal`, despite being explicitly declared as Circular, are effectively NonCircular which allows the RelaxedStacked algorithm to avoid reevaluating them in each fixed-point iteration.

4.3 Case Study: IntraJ

IntraJ [18] is a dataflow analyser for Java built as an extension of ExtendJ. It currently supports detecting two kinds of dataflow bugs: null-pointer dereferences and and dead assignments. The dataflow information is propagated through the program using the *control-flow graph* (CFG), available with the functions *pred* (predecessors) and *succ* (successors).

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

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

The equations (1) and (2) are used to propagate information from the predecessors of a node n to n itself. On the other hand, the equations (3) and (4) are used to propagate information *backward* in the CFG. In IntraJ, *in*, *out*, *succ*, and *pred* are represented by attributes. Our tool CAT will detect that both *pred* and *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. Each analysis is done by querying an attribute in IntraJ that collects all problems (dead assignments or potential null-pointer dereferences) 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.

For dead assignment analysis the results show significant performance improvements for the RelaxedStacked algorithm compared to the RelaxedMonolithic algorithm. For startup, the speedup of RelaxedStacked is between ~1.3x to ~2.5x, and with a median of ~1.7x. For steady-state, the speedup is even more significant: between ~1.9x to ~2.8x, with a median of ~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 the results show an even more significant improvement for RelaxedStacked, with a speedup between ~4x to ~115x for startup performance, with a median of ~11.5x. For steady state performance, the speedup was between ~11x to ~35x, with a median of ~18.5x, disregarding 2 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.

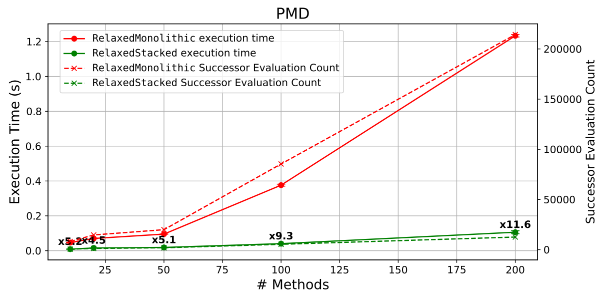

Figure 3: Steady-state performance of null-pointer dereference analysis for randomly selected sets of methods of the `pmd` benchmark. Solid lines represent execution time (left axis, seconds). Dashed lines represent successor attribute evaluations (right axis, count).

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 3 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.

5 RELATED WORK

Related work includes other work on circular attribute grammars, other work on declarative programming of fixed-point problems, like Datalog, and other approaches to demand-evaluation for program analysis problems.

Attribute grammars, as originally described by Knuth [15] were not allowed to have cyclic dependencies. Farrow [9] and Jones et al. [14] independently of each other introduced the notion of circular but well-defined attribute grammars. Farrow presented an algorithm based on a static analysis of the attribute grammar. Jones' algorithm uses a dynamic dependency graph to identify strongly connected components, and supports incremental evaluation. In contrast to our work, neither of these approaches supports reference attributes or demand evaluation.

Boyland implemented demand-driven evaluation for circular attributes in the presence of so called remote attributes (similar to reference attributes), but provided no explicit evaluation algorithm [3]. Hesamian recently implemented statically scheduled support for circular attributes in Boyland's remote attribute system APS [12]. However, this implementation is exhaustive and not demand-driven.

Another declarative approach to program analysis is to use the Datalog language. In this approach, initial facts are generated from

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	
`commons-cli`	$0.74_{\pm0.02}$	$0.56_{\pm0.01}$	× 1.31 ↑	$0.09_{\pm0.00}$	$0.05_{\pm0.00}$	× 1.89 ↑	$4.77_{\pm0.04}$	$1.08_{\pm0.02}$	× 4.44 ↑	$3.13_{\pm0.03}$	$0.19_{\pm0.01}$	× 16.10 ↑
`commons-jxpath`	$2.18_{\pm0.05}$	$1.41_{\pm0.03}$	× 1.55 ↑	$0.70_{\pm0.00}$	$0.29_{\pm0.00}$	× 2.46 ↑	$7.29_{\pm0.07}$	$1.77_{\pm0.04}$	× 4.12 ↑	$4.76_{\pm0.06}$	$0.41_{\pm0.00}$	× 11.50 ↑
`pmd`	$6.49_{\pm0.12}$	$3.48_{\pm0.05}$	× 1.86 ↑	$3.61_{\pm0.02}$	$1.39_{\pm0.02}$	× 2.60 ↑	$32.36_{\pm0.20}$	$4.46_{\pm0.09}$	× 7.26 ↑	$28.14_{\pm0.10}$	$1.73_{\pm0.01}$	× 16.24 ↑
`struts`	$9.32_{\pm0.18}$	$5.31_{\pm0.09}$	× 1.75 ↑	$5.18_{\pm0.07}$	$2.17_{\pm0.06}$	× 2.38 ↑	$66.97_{\pm0.85}$	$6.42_{\pm0.10}$	× 10.43 ↑	$61.74_{\pm0.74}$	$3.54_{\pm0.14}$	× 17.45 ↑
`jfreechart`	$14.13_{\pm0.13}$	$9.33_{\pm0.52}$	× 1.51 ↑	$11.28_{\pm0.20}$	$4.16_{\pm0.30}$	× 2.71 ↑	$202.15_{\pm1.62}$	$8.16_{\pm0.07}$	× 24.78 ↑	$205.43_{\pm2.08}$	$5.88_{\pm0.69}$	× 34.91 ↑
`extendj`	$40.88_{\pm0.74}$	$16.11_{\pm0.21}$	× 2.54 ↑	$37.16_{\pm0.66}$	$12.94_{\pm0.35}$	× 2.87 ↑	$1510.75_{\pm5.99}$	$13.04_{\pm0.08}$	× 115.87 ↑	≥ 300.00 ⏱	$9.55_{\pm0.08}$	≥ 31.42 ↑
`weka`	$28.12_{\pm0.09}$	$12.93_{\pm0.12}$	× 2.17 ↑	$23.50_{\pm0.20}$	$9.31_{\pm0.19}$	× 2.52 ↑	$475.89_{\pm2.66}$	$17.21_{\pm0.45}$	× 27.65 ↑	≥ 300.00 ⏱	$11.88_{\pm0.32}$	≥ 25.25 ↑

Table 3: Performance of *dead assignment analysis* and *null pointer dereference analysis* benchmarks, comparing the RelaxedMonolithic and RelaxedStacked algorithms in startup and steady state. The ⏱ symbol indicates that the analysis timed out.

the complete project code, and derived facts are computed using logic rules. This line of work includes commercial tools like the .QL system [6] system (now CodeQL). There are high performance toolboxes implemented using this approach, like the Doop framework [4] that supports points-to analysis of Java bytecode. However, most Datalog frameworks, including Doop, use data-driven rather than demand-driven evaluation, computing all derivable facts rather than only those needed for a particular query. Datalog programs can be rewritten to enforce on-demand evaluation [2], and some Datalog-based tools like Clog [7] make extensive use of such approaches to reduce the overhead of operations that are known to be slow. However, this optimization requires separate rulesets for each type of query. A general approach to demand-driven abstract interpretation was presented by Stein et al [19]. It supports cyclic computations over infinite-height domains, but requires an a priori computed control-flow graph, and was only evaluated on synthetic workloads.

6 CONCLUSION

In this paper, we have presented a new formulation of demand-driven evaluation of Reference Attribute Grammars with circular (fixed-point evaluated) attributes. Our formulation allows the coexistence of three important kinds of attributes: Circular, Agnostic, and NonCircular, resulting in the new RelaxedStacked algorithm. Previous work supported combining Circular with only NonCircular or Agnostic, but not both.

From our experiments, it is clear that using NonCircular attributes is key for efficient evaluation. However, manually specifying NonCircular instead of Agnostic can be error-prone, leading to runtime errors. Our approach uses call graph analysis on the RAG to automatically identify NonCircular attributes, ensuring both safety and efficiency.

We have done experiments to evaluate the effect of the new algorithm on LL(1) parser construction, and on two intraprocedural dataflow analyses for Java. In the parser construction case study, which has several nested circular attributes, RelaxedStacked performed as well as BasicStacked, and 2.8x better than RelaxedMonolithic. For the IntraJ case study, we compared only RelaxedStacked and RelaxedMonolithic, as it requires Agnostic attributes, and can therefore not be run with BasicStacked. For dataflow analyses applications we saw substantial speedups for RelaxedStacked over RelaxedMonolithic. For dead assignment analysis, we observed a median speedup of 1.7x in startup performance, and a median of 2.5x for steady-state. For null-pointer dereference analysis, we observed an even more significant improvement with a median of 11.5x for startup performance and a median of 18.5x for steady state.

REFERENCES

[1] A. W. Appel. *Modern compiler implementation in C.* Cambridge university, 2004.

[2] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, 1985.

[3] J. T. Boyland. *Descriptional Composition of Compiler Components.* PhD thesis, University of California, Berkeley, 1996.

[4] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of OOPSLA '09*, pages 243–262, New York, NY, USA, 2009. ACM.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'97 Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[6] O. De Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. .QL: Object-oriented queries made easy. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 78–133. Springer, 2007.

[7] A. Dura and C. Reichenbach. Clog: A declarative language for c static code checkers. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, CC 2024, page 186–197, 2024.

[8] T. Ekman and G. Hedin. The jastadd extensible java compiler. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 1–18. ACM, 2007.

[9] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the 1986 Symposium on Compiler Construction*, pages 85–98. ACM, 1986.

[10] G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[11] G. Hedin and E. Magnusson. Jastadd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[12] S. Hesamian. *Statically Scheduling Circular Remote Attribute Grammars.* PhD thesis, University of Wisconsin-Milwaukee, 2023. Theses and Dissertations. 3383.

[13] S. Horwitz, A. J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[14] L. G. Jones and J. Simon. Hierarchical VLSI design systems based on attribute grammars. In *POPL St. Petersburg Beach, Florida, USA, January 1986*, pages 58–69, 1986.

[15] D. E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[16] E. Magnusson and G. Hedin. Circular reference attributed grammars — their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, 2007. Special Issue on the ETAPS 2003 Workshop on Language Descriptions, Tools and Applications (LDTA '03).

[17] J. Öqvist and G. Hedin. Concurrent circular reference attribute grammars. In B. Combemale, M. Mernik, and B. Rumpe, editors, *Proceedings of the 10th ICSE, SLE 2017, Vancouver.* ACM, 2017.

[18] I. Riouak, C. Reichenbach, G. Hedin, and N. Fors. A precise framework for source-level control-flow analysis. In *SCAM 2021*, pages 1–11. IEEE, 2021.

[19] B. Stein, B. E. Chang, and M. Sridharan. Demanded abstract interpretation. In S. N. Freund and E. Yahav, editors, *PLDI '21, Virtual Event, Canada, June 20-25, 2021*, pages 282–295. ACM, 2021.