EPiC Series in Computing
Volume 46, 2017, Pages 171–180
LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Cauliflower: a Solver Generator Tool for Context-Free Language Reachability

Nicholas Hollingum and Bernhard Scholz

The University of Sydney, Camperdown, NSW, Australia
`nhol8058@uni.sydney.edu.au, bernhard.scholz@sydney.edu.au`

Abstract

Context-free language reachability (CFL-R) is a fundamental solving vehicle for computing essential compiler optimisations and static program analyses. Unfortunately, solvers for CFL-R encounter both inherently expensive problem formulations and frequent alterations to the underlying formalism. As such, tool designers are forced to create custom-tailored implementations with long development times and limited reusability. A better framework is crucial to facilitate research and development in CFL-R.

In this work we present Cauliflower, a CFL-R solver generator, that creates parallel executable C++ code from an input CFL-R rule-based specification. With Cauliflower, developers create working tools rapidly, avoiding lengthy and error-prone manual implementations. Cauliflower's domain-specific language provides semantic extension including reversal, branching, disconnection and templating. In practical experiments, Cauliflower achieves an average speedup of 1.8x compared with the best general purpose tools, and matches the performance of application-specific tools on many benchmarks.

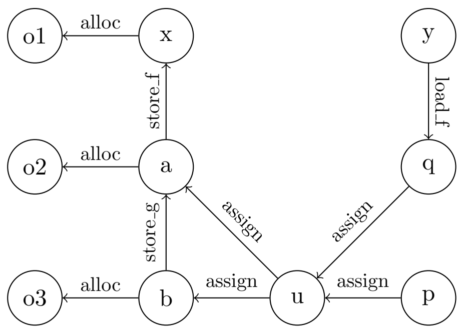
1 Introduction

The Context-free Language Reachability (CFL-R) problem [17] is fundamental to the study of program analysis, verification, and data analytics. CFL-R is used as a solving vehicle in many research areas, including: formal security verification [6], logic programming [17], data-flow [13], object-flow [18], control-flow [15], set-constraint solving [11], shape-analysis [12] and alias-analysis [14, 5]. Currently, tool designers employing CFL-R lack a unified research platform, and are left to craft their solvers by hand. Traditionally, solvers have been hand-specialised [19, 5], because generic solvers for CFL-R perform poorly in practice. Generic CFL-R solvers are often called a "cubic bottleneck" [7] for this reason. Hence, there is a need for CFL-R tools that overcome the engineering efforts of crafting CFL-R solvers.

CFL-R relates to the subset of Datalog [1] that consists of chain rules over binary relations. A chain rule is of the form $\texttt{A(x}_0\texttt{,x}_k\texttt{):-B}_1\texttt{(x}_0\texttt{,x}_1\texttt{),B}_2\texttt{(x}_1\texttt{,x}_2\texttt{),...,B}_k\texttt{(x}_{k-1}\texttt{,x}_k\texttt{)}$, where relation $\texttt{B}_i$ can be either an EDB or IDB predicate. CFL-R interpretes the EDB relations as a lablled input graph, i.e., fact $\texttt{B}_i\texttt{(u,v)}$ in the EDB relation creates an edge (u, v) with the label B_i. A chain rule defines a sequence of labelled edges, i.e. a path. A more succinct vehicle to express paths in the labelled input graph is a context-free grammar. For example, the chain

```
use(u){
 ...
 return u
}
main(){
 x = new Obj();
 a = new Obj();
 b = new Obj();
 a.f = x;
 b.g = a;
 p = use(a);
 q = use(b);
 y = q.f;
}
```

(a) Source code



(b) Pointer-assignment graph

```
alloc <- V.H;
assign <- V.V;
load_f <- V.V;
store_f <- V.V;
load_g <- V.V;
store_g <- V.V;
pt <- V.H;

pt -> alloc;
pt -> assign, pt;
pt -> load_f, pt, -pt,
      store_f, pt;
pt -> load_g, pt, -pt,
      store_g, pt;
```

(c) Points-to grammar

Figure 1: CFL-R points-to analysis for a Java-like language.

rule becomes a production rule $A \rightarrow B_1 B_2 \ldots B_k$. This connection between Datalog and CFL-R has been introdued by Yannakakis in his seminal work in [17].

This paper presents the Cauliflower tool, which is a solver generator for CFL-R problems. It alleviates the implementation efforts of tool designers, while granting them greater flexibility and expressibility for a diverse range of CFL-R applications. Cauliflower has a domain-specific language (DSL) that is simple and intuitive for rapid-prototyping a CFL-R application. Extensions to CFL-R and customisability are key for a CFL-R solver generator. For example, in [16] a CFL-R problem is used as a pre-analysis for points-to, and in [14] the analysis is iteratively re-evaluated. A standard CFL-R algorithm could not encode such features. Cauliflower meets these requirements by providing extensions in the DSL. Finally, Cauliflower meets scalability and performance demands by issuing efficient parallel C++ code from a CFL-R specification. As a result, Cauliflower-generated code rivals hand-crafted efforts.

The core contribution of this paper is the Cauliflower tool with its CFL-R extensions. Cauliflower is available as open-source software[1]. In Section 2 we describe a motivating problem, and how Cauliflower is used to solve it. We detail the design and features of Cauliflower, specifically the performance and usability advantages that it presents over other tools, in Section 3. In Section 4, we exemplify the usage of Cauliflower through two case studies, giving experimental validation to our performance and expressibility claims, before concluding in Section 5.

2 Motivating Example

We demonstrate Cauliflower and the use of CFL-R with a well known static program analysis called points-to analysis [2] for a Java-like language. Points-to analysis computes a memory invariant, i.e., a sound over-approximation of a program's memory behaviour at runtime. Optimising compilers and software tools rely on points-to analyses that are precise and scalable. Hence, points-to analysis is an active field of research.

Points-to analyses work by recording dataflow information from the input program's semantics, and deriving more complex points-to interactions inductively. For example, a statement like `a = new Obj()` implies that variable `a` "may point-to" the memory location allocated by this statement. Another statement like `b = a` implies a flow of pointer information from `a` to

[1]https://cauliflower-cflr.github.io/

`b`, hence that `b` may point-to the same objects as `a`. When two variables refer to the same memory location, we say that they **alias**. More intricate dataflow information is needed to model program semantics between loading and storing of fields. The statements `b.f=x` and `y=a.f` imply an indirect flow of information from variable `x` to variable `y` if variables `a` and `b` alias, i.e., they may point to the same object.

Source code for a small Java-like program is depicted in Figure 1a. In the next step, we convert the input program to a graph representation. The graph representation abstracts the control-flow of the input program. In this abstraction, the order of the statements is irrelevant. The graph contains nodes for variables and object allocations. The operations (e.g. allocations, assignments, field loads/stores) in the input program become labelled edges. For example, the edge $(x, o1)$ is created by the instruction on line 6, since the `x` variable allocates that object. Edges labelled *assign* are created by direct assignment, but also by assignment-like operations, such as the actual-formal parameter passing from lines 11 and 12, which create (u, a) and (u, b) respectively. Finally load/store operations, such as on line 13, create their edges, in this case (y, q). To distinguish between an object's fields, we use different labels in the load/store pair for each field, i.e. since line 13 loads the field `f`, its edge is labelled *load_f*.

The points-to analysis of the code would report, for example, that `x` points to the object allocated on line 6, `o1`, since it is assigned directly from the result of an allocation instruction, `new`. Points-to information is transferred between variables by different program instructions. For example, `u` may point to `o3`, since it receives the reference variable `b` from the call on line 12, and `b` points-to `o3`. Graphically, the points-to information is related to certain paths, i.e., if the labels on a path between a variable vertex and a heap-object vertex have a certain sequence, such as $assign^*\,alloc$, then that variable may point-to the heap object.

CFL-R allows for an elegant phrasing of such structured path problems. CFL-R is a generalisation of the context-free language recognition problem to graphs. It was originally devised by Yannakakis [17] for solving certain Datalog queries, and popularised as a program analysis framework by Reps [12]. A CFL-R problem instance requires a context-free language $\mathcal{L} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ of terminals, non-terminals, productions, and a start symbol, as well as an edge-labelled graph $G = (V, E)$, with labels drawn from the terminals and non-terminals of the language $\Sigma \subseteq \mathcal{T} \cup \mathcal{N}$. The solution to the CFL-R instance is the set of all pairs of vertices joined by a path whose concatenated edge-labels spell a word in the context-free language.

To generate a points-to solver, users need only specify the points-to language in Cauliflower's DSL. In Figure 1c, the canonical points-to analysis is encoded for Cauliflower. DSL directives are divided into two groups, **types** are identified by a left arrow `<-`, and **rules** by a right arrow `->`. Type declarations specify the terminals and non-terminals in the grammar, and constrain the endpoints of edges labelled by such symbols. In the example, `alloc` edges always connect a variable to a heap object, whilst `assign` edges connect two variables. Rule declarations define the productions which make up the context-free language. Rules are written in an adaptation of standard BNF grammar syntax [3], i.e. the rules on lines 9-10 model the regular language $assign^*\,alloc$. To fully capture the range of problems that CFL-R can encode, Cauliflower defines enhanced semantics for reversal, branching, disconnection and templating, which we detail in Section 3. Even the simple points-to analysis in Figure 1 makes use of Cauliflower's reversal semantics for the `-pt` terms; two variables alias if there is a points-to path from the first to a given heap object, and a points-to path **backwards** from that heap object to the second.

Further, Points-to analysis has many variations, which provide a tradeoff between precision and scalability. In a field-sensitive analysis, memory objects are partitioned into fields. In Figure 1a, variable `a` is stored in the `g` field of `b`, since `b` aliases `q`, we require field sensitivity so that the erroneous alias between `a` and `y` does not appear when field `f` is loaded on line 13.

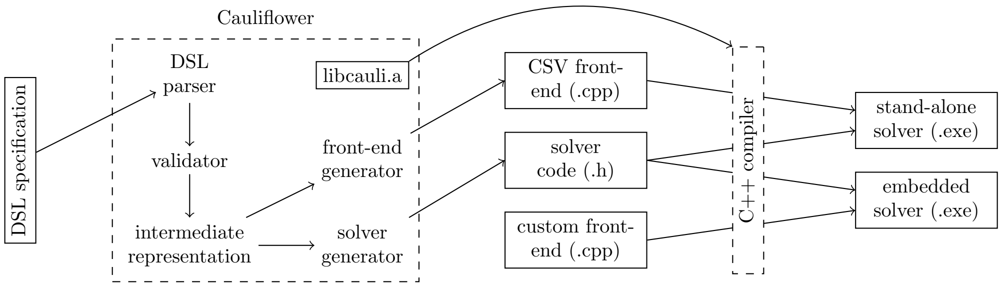


Figure 2: Schematic overview of Cauliflower's major components.

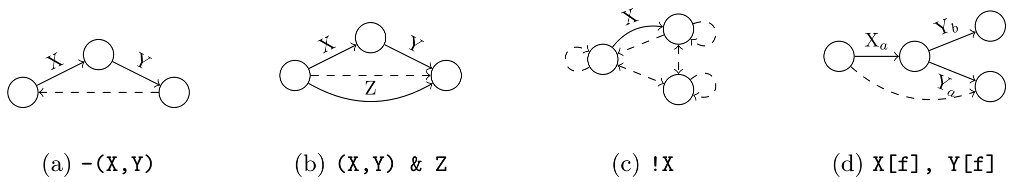


Figure 3: Cauliflower's semantic extensions. Dashed lines are the paths recognised in the language underneath each figure.

Context-sensitive analyses distinguish between variables which appear in different calling contexts. Without context sensitivity, Figure 1a exhibits a spurious alias between `q` and `a`, since they are assigned to and returned from `u` even though these occur in different calling contexts. Cauliflower's lightweight DSL allows users to easily modify their CFL-R specification to make a performance/precision tradeoff.

3 Design and Features

In practical CFL-R systems, the grammar is usually fixed while the graph instances change frequently. To capitalise on this, Cauliflower is designed as a **solver generator**, which synthesises the instance-solver based on an input specification. The major components of Cauliflower are illustrated in Figure 2. Given a specification, Cauliflower determines an efficient evaluation strategy and generates parallel C++ code to solve problem instances. In practice, this involves a partial evaluation of the CFL-R semi-naïve approach [8] for a single context-free language. To interface with the user, Cauliflower generates a solver, as a back-end library, as well as a simple front-end for processing problems given by a comma-separated value (CSV) formatted input graph. The user is free to replace the generated front-end with a custom front-end, such as a compiler or a bug-checker, which can improve performance by avoiding disk I/O. Finally the user compiles the generated C++ solver using a conventional compiler. Generating source code in this way allows Cauliflower to leverage powerful static optimisations available in most C compilers.

Cauliflower is designed to make CFL-R research and practice more feasible. To achieve this, we incorporate features which increase the flexibility and usability of the system, as well as features which enhance the tool's performance.

Enhanced Semantics. Cauliflower increases the scope of traditional CFL-R by offering enhanced semantics for recognising paths. Figure 3 demonstrates the kinds of paths that each

```
for all $A' \in \Delta_A$ in parallel do
    for all $(u, v) \in A'$ do
        for all $(w, v) \in B$ do
            $\Delta_C \leftarrow (u, w)$
```

(a) `C->`Δ_A` -B`

```
for all $f \in tmpl(B, C)$ in parallel do
    for all $(u, v) \in \Delta_{Bf}$ do
        if $(v, w) \notin A$ then
            $\Delta_{Cf} \leftarrow (u, v)$
```

(b) `C[f]->!A &` Δ_B`[f]`

Figure 4: Pseudocode generated by Cauliflower for different evaluation strategies.

new operation discovers. We developed these semantics in response to use cases presented by the research literature.

- CFL-R problems commonly require semantics to detect paths in **reverse** (3a). When searching for a reverse path, Cauliflower treats edges as though they point backwards, and matches grammar rules as though they were written backwards. Reversal is useful for encoding many CFL-R problems, including points-to analysis [14].
- Cauliflower provides a new CFL-R semantic operation for detecting **branching** paths (3b). Branching must be properly nested, i.e. two branches can only join if they were forked at the same point.Branching is a convenient way of expressing mutually dependant properties, such as context-sensitive analyses, or reasoning about dynamic dispatch mechanisms in Java/C++.
- The need for detecting **disconnected** paths (3c) is well established in the CFL-R literature [16]. Cauliflower provides a primitive operation to find the absence of a labelled edge in the graph, and allows CFL-R paths to be built across them. Since the disconnection semantics have performance implications, Cauliflower tries to rewrite rules that use negation to a more efficient form where possible (using De Morgan's laws).
- Finally, Cauliflower provides a **template** mechanism (3d) for deferring rule creation to runtime. This enhancement is a convenience prompted by the prevalence of Dyck rules (i.e. bracket-matching) in CFL-R analyses. Graph edges can be labelled with an arbitrary index, and the templated rule then searches over all potential indices for a match at runtime. The points-to problem from Figure 1 can replace the rules on lines 11 and 13 with a single templated rule.

Specialisation. To maintain high performance of the generated solvers, Cauliflower specialises the code it produces for an individual CFL-R problem. General-purpose solvers typically have performance penalties because they require some overhead to dynamically create an execution plan, which Caulilflower avoids by fixing the execution plan in advance. By preferring a static solver-generator, Cauliflower avoids the dynamic overheads, but also leverages optimisation opportunities that are provided by commodity compilers.

Code is generated according to the semi-naïïve strategy, as adapted for CFL-R [8]. In brief, a dependency graph is generated for each rule of the input language, where labels in a rule-s head are topologically greater or equal to all the labels in that rule's body. Information from each label is propagated for the rules that it appears in according to that evaluation order, with cyclic dependencies implying a loop-until-fixpoint constraint for those labels. Further, to avoid redundant calculations, only edges recently added to a label, called Δ edges, are used to discover new paths, and each rule is duplicated such that at least one term in the rule contains a Δ. Pseudo-code snippets for two rule evaluations are presented in Figure 4.

Static generation offers several advantages over a dynamic approach. The compiler is able to reason about future calculations, enabling data pre-fetching optimisations to be made in some cases. Further, as a byproduct of the evaluation order, Cauliflower-generated solvers maximise cache utility by ensuring frequently needed relations stay in cache.

Parallelism Cauliflower generates parallel code in order to ensure scalability on very large problems. We exploit opportunities for both coarse- and fine-grained parallelism, often within the same solver. The style of parallelism is chosen on a per-rule basis, based on a simple heuristic. In the presence of templated rules, Cauliflower adopts a coarse-grained approach. Every version of the rule (i.e. every combination of indices in the template) is evaluated separately in parallel. Cauliflower switches to a fine-grained parallel approach for normal (non-templated) rules. The code snippets from Figures 4a and 4b demonstrate how the fine- and coarse-grained mechanisms are implemented.

Parallel generation introduces several issues that the solver must account for, which sequentially generated code does not deal with. Cauliflower's data structures are designed for bi-modal synchronisation, meaning they can be read from or written to in a lock-free parallel setting, but not both. As a result, execution is staggered to prevent read/write conflicts from occurring. Further, some overhead is needed to partition data structures for rule evaluation. Note that parallelism does not imply the use of different data structures, so aside from the negligible thread-local variables, parallel solvers have the same memory requirements as sequential solvers.

4 Usage

This section details the usage of Cauliflower in a practical setting. Section 4.1 compares Cauliflower to other high-performance solvers, whilst Section 4.2 demonstrates the rapid-prototyping features and enhanced semantics. All experiments are run on a 2.1 GHz Intel® Xeon® CPU E5-2450 with 16 cores and 128 GB RAM under Linux. Our experimental results are recorded in Table 1. The DSL specifications are kept online[2].

4.1 Points-to Analysis for Java

We now compare the performance of Cauliflower's generated-code to a modern high-performance hand-optimised points-to analysis for Java called Gigascale. Developed by Dietrich et al. [5], Gigascale is a solver for the context- and flow- insensitive, field-sensitive points-to analysis problem for Java, particularly on large benchmarks. Cauliflower emulates the logic of Gigascale, but the evaluation strategy differs substantially, i.e. Gigascale contains optimisations that cannot be expressed natively in CFL-R. Notably, Gigascale uses a refinement/verification approach insctead of a totally bottom-up strategy, which is more performant in practice. Our results, depicted in Figure 5, show that Cauliflower is viable not merely as a prototyping tool, but as an optimised implementation.

Cauliflower's performance rivals or surpasses that of Gigascale on the DaCapo benchmarks. The most noticeable improvement occurs for sequential solvers, though with enough processors even parallel solvers outperform Gigascale. The parallel backend is necessary to scale to very large problems, though it is less viable than the sequential solver for small instances. On the OpenJDK benchmark, Gigascale outperforms Cauliflower with an 11x speedup. Given the Gigascale was designed for this benchmark, we consider Cauliflower's performance to be

[2] https://cauliflower-cflr.github.io/examples/

Benchmark	$\|V\|$	$\|pt\|$	GS-T (s)	GS-M (MB)	CL-T-S (s)	CL-T-1 (s)	CL-T-8 (s)	CL-M (MB)
lusearch	14,994	9,242	0.60	99	0.05	0.16	0.08	5
sunflow	15,957	16,354	0.59	96	0.06	0.15	0.09	16
luindex	17,375	9,677	0.77	120	0.06	0.21	0.10	33
avrora	25,196	21,532	0.88	131	0.09	0.29	0.13	36
eclipse	40,200	21,830	1.00	181	0.13	0.42	0.19	67
h2	56,683	92,038	1.24	207	0.27	0.71	0.29	82
pmd	59,329	60,518	1.18	206	0.27	1.35	0.45	92
xalan	62,758	52,382	1.21	224	0.27	1.43	0.47	103
batik	63,089	45,968	1.23	213	0.21	0.71	0.27	102
fop	83,016	76,615	1.79	391	0.33	1.19	0.45	138
tomcat	110,884	82,424	2.00	445	0.45	1.83	0.64	181
jython	260,034	561,720	2.89	708	1.55	5.20	1.92	332
tradebeans	466,969	696,316	5.97	1,533	2.94	12.50	3.98	763
tradesoap	468,263	698,567	5.97	1,529	2.97	11.99	4.05	758
openjdk	1,963,997	1,570,820,597	57.89	3,532	*	1,235.94	598.66	60,720

(a) Cauliflower (CL) compared with Gigascale (GS) over Da-Capo benchmarks and OpenJDK case. Cauliflower is run in sequential mode (-S) and in parallel mode with one (-1) and eight (-8) cores.

Benchmark	BD	Z3	LB	SF	CL
lusearch	2.32	0.12	1.76	0.12	0.05
sunflow	2.27	0.16	1.77	0.14	0.06
luindex	3.78	0.17	1.77	0.14	0.06
avrora	6.48	0.19	1.77	0.25	0.09
eclipse	9.10	0.26	1.81	0.27	0.13
h2	15.60	5.74	1.97	0.55	0.27
pmd	17.91	3.71	1.99	0.49	0.27
xalan	18.76	1.51	1.92	0.54	0.27
batik	17.58	0.60	1.89	0.44	0.21
fop	46.27	2.08	1.94	0.52	0.33
tomcat	68.14	2.84	2.05	0.61	0.45
jython	*	189.39	2.81	1.95	1.55
tradebeans	*	60.62	3.58	3.05	2.94
tradesoap	*	60.97	3.53	3.09	2.97

(b) Cauliflower (CL) compared with the Datalog solvers: Bddbddb (BD), Z3, LogicBlox (LB) and Soufflé (SF).

Benchmark	$\|V\|$	$\|pt\|$	$\|uninit\|$	time (s)
mcf	2,231	7,715	8	0.08
libquantum	5,453	10,244	12	0.14
astar	6,961	19,930	36	0.29
bzip2	14,211	1,914,709	4	198.75
sjeng	14,676	40,917	79	0.47
hmmer	48,945	8,751,195	48	2,827.90
omnetpp	55,596	430,226	153	16.15
h264ref	116,358	731,986	123	24.46

(c) Results of the prototype uninitialised-memory analysis for LLVM, over a subset of the Spec-int benchmarks (other benchmarks timed out).

Table 1: Experimental results. All times and memory sizes are an average over 3 runs.

reasonable, particularly as a prototyping tool. Cauliflower's comparative memory usage is similar to the execution time results.

When developing CFL-R analyses, it is common practice to adapt more expressive tools, such as Datalog solvers. Since the Cauliflower tool is designed exclusively for CFL-R, we expect it to have an inherent advantage over its Datalog competitors, which is shown in Figure 6. A timeout of 10 minutes is applied, which disqualifies the OpenJDK benchmark. Against its nearest competitor, Soufflé [10], Cauliflower outperforms all benchmarks with a speedup of 1.89x on average and 1.34x weighted by problem size.

4.2 Uninitialised Memory for LLVM

We develop a simple analysis in Cauliflower, which is used to identify uninitialised memory locations in LLVM bitcode. The purpose of this case study is to demonstrate the features and usage of Cauliflower. The uninitialised variables problem was also explored in CFL-R by

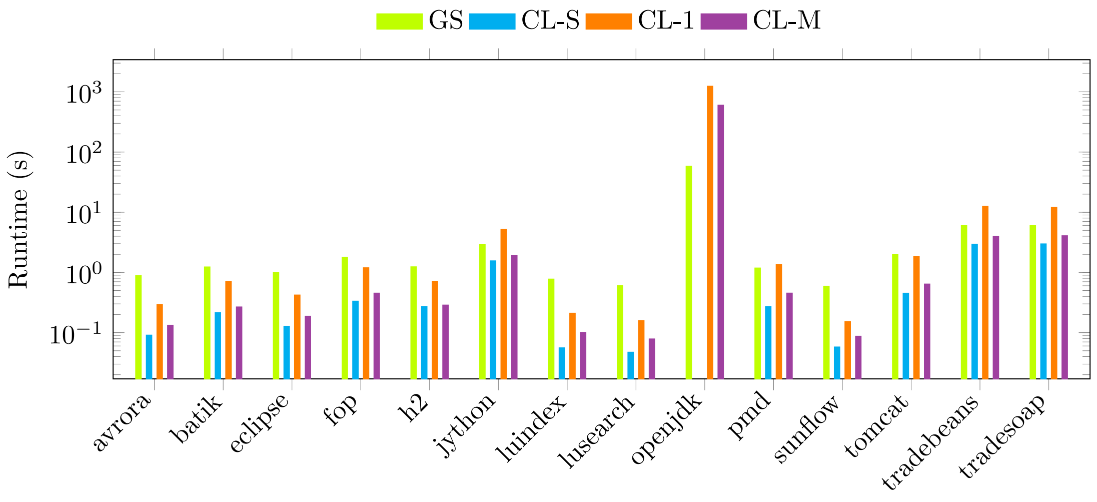


Figure 5: Runtimes of Gigascale and Cauliflower under sequential mode and parallel modes with 1 and 8 processors (logarithmic scale).

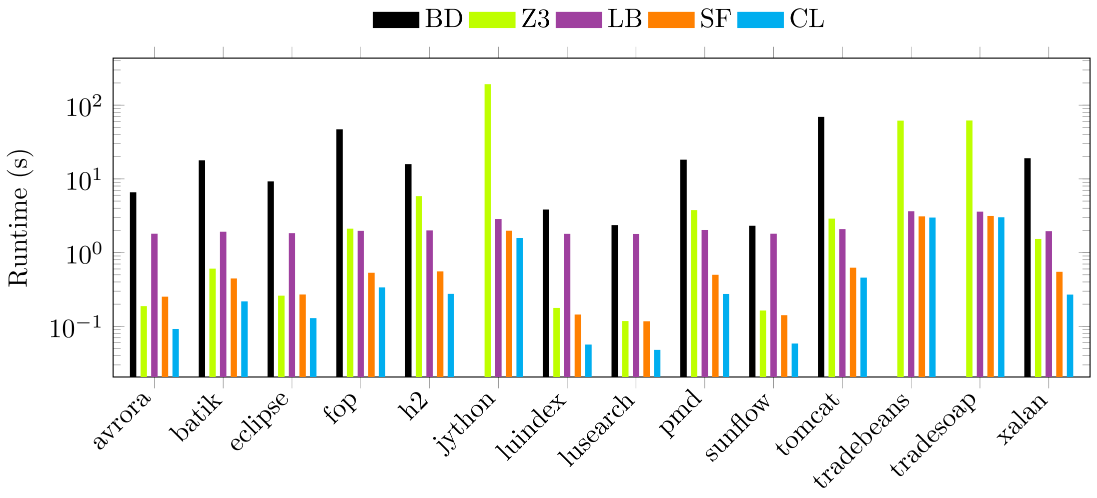


Figure 6: Runtimes of the Datalog solvers and Cauliflower's sequential mode (logarithmic scale).

Horwitz et al. [9], though our formulation differs from theirs. We use the Cclyzer tool [4] to generate program facts from LLVM bitcode. Though there are many formulations of uninitialised memory analysis, our formulation is entirely novel so, unlike the points-to analysis, no comparison with commodity tools is possible. The prototype analysis is powerful enough to run on a subset of the SPEC 2006 benchmarks, shown in Table 1c.

The analysis depends on all of Cauliflower's semantic extensions. Reversal is needed to phrase the underlying memory analysis, i.e. to compute points-to information similar to the example in Figure 1. We filter conditions like "a variable is used **and** a memory location is uninitialised" by using the branching semantics to ensure both relationships hold simultaneously (i.e. both paths exist). Disconnection is needed to exclude the variables which are **not** initialised, and templating is necessary to model some C-specific semantics, such as that call-site arguments are passed to formal parameters with the same index.

5 Conclusion

This work presents the Cauliflower tool, an efficient and expressive solver generator for CFL-R applications. Cauliflower provides an intuitive DSL for developing CFL-R applications quickly, and its code generator produces efficient parallel code for use in rapid prototypes and performance critical areas. By extending the traditional CFL-R formalism with reversal, branching, disconnection and templating semantics, Cauliflower is able to express and solve a wider range of problems than traditional CFL-R. Experimentally, Cauliflower has been shown to be an efficient solver-generator; its execution times were sufficiently close to the hand-optimised Gigascale points-to analysis, and its solvers outperformed more general Datalog tools. We believe, with continued research, that the advantages of a specialised CFL-R solver can be even greater. For example, the potential to improve runtime by dynamically optimising the execution plan, or to increase the flexibility of the DSL with more directives, were not explored, and are left as future work.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] Lars Ole Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Cophenhagen, 1994.

[3] John W Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *Proceedings of the International Comference on Information Processing, 1959*, 1959.

[4] George Balatsouras. Cclyzer version 1.1.2, https://github.com/plast-lab/cclyzer, February 2016.

[5] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 30th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, pages 535–551. ACM, 2015.

[6] D. Dolev, S. Even, and R.M. Karp. On the security of ping-pong protocols. *Information and Control*, 55(13):57 – 68, 1982.

[7] Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*, pages 342–351. IEEE, 1997.

[8] Nicholas Hollingum and Bernhard Scholz. Towards a scalable framework for context-free language reachability. In Björn Franke, editor, *Compiler Construction*, volume 9031 of *Lecture Notes in Computer Science*, pages 193–211. Springer Berlin Heidelberg, 2015.

[9] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 20(4):104–115, October 1995.

[10] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. *Soufflé: On Synthesis of Program Analyzers*, pages 422–430. Springer International Publishing, Cham, 2016.

[11] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(12):29 – 98, 2000. PEPM'97.

[12] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[13] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[14] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *ACM SIGPLAN Notices*, 41(6):387–400, 2006.

[15] Dimitrios Vardoulakis and Olin Shivers. Cfa2: A context-free approach to control-flow analysis. In AndrewD. Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 570–589. Springer Berlin Heidelberg, 2010.

[16] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings ECOOP'09*. Springer, 2009.

[17] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 230–242, New York, NY, USA, 1990. ACM.

[18] Hao Yuan and Patrick Eugster. An efficient algorithm for solving the dyck-cfl reachability problem on trees. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 175–189. Springer Berlin Heidelberg, 2009.

[19] Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 435–446, New York, NY, USA, 2013. ACM.