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 \( A(x_0, x_k) : - B_1(x_0, x_1), B_2(x_1, x_2), \ldots, B_k(x_k-1, x_k) \), where relation \( B_i \) can be either an EDB or IDB predicate. CFL-R interprets the EDB relations as a labelled input graph, i.e., fact \( B_i(u, v) \) 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...
rule becomes a production rule \( A \rightarrow B_1 B_2 \ldots B_k \). This connection between Datalog and CFL-R has been introduced 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 \( a = \text{new Obj}() \) 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
eedge (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 gener-
isation 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 anal-
ysis framework by Reps [12]. A CFL-R problem instance requires a context-free language
L = (T, N, P, 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 Σ ⊆ T ∪ 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.
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-naive 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) $C \rightarrow \Delta_A$ -B
  (b) $C[f] \rightarrow !A \& \Delta_B [f]$

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 dependent 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 Cauliflower 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-naïve 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 $\Delta$ edges, are used to discover new paths, and each rule is duplicated such that at least one term in the rule contains a $\Delta$. 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 instead 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

²https://cauliflower-cflr.github.io/examples/
Cauliflower (CL) compared with the Datalog solvers. Since the Cauliflower tool is designed exclusively for CFL-R, we expect Cauliflower to outperform all benchmarks with a speedup of 1.89x on average and 1.34x weighted by problem size.

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

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
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 simultane-
ously (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 Gi-gascale 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


