Context-Free Language Reachability via Skewed Tabulation

YUXIANG LEI[*], University of New South Wales, Australia
CAMILLE BOSSUT[*], Georgia Institute of Technology, USA
YULEI SUI, University of New South Wales, Australia
QIRUN ZHANG, Georgia Institute of Technology, USA

Context-free language reachability (CFL-reachability) is a prominent model for formulating program analysis problems. Almost all CFL-reachability algorithms are based on the Reps-Horwitz-Sagiv (RHS) tabulation. In essence, the RHS tabulation, based on normalized context-free grammars, is similar to the CYK algorithm for CFL-parsing. Consider a normalized rule $S ::= A\ B$ and a CFL-reachability problem instance of computing S-edges in the input graph. The RHS tabulation obtains all summary edges (*i.e.*, S-, A-, and B-edges) based on the grammar rules. However, many A- and B-edges are wasted because only a subset of those edges eventually contributes to generating S-edges in the input graph.

This paper proposes a new tabulation strategy for speeding up CFL-reachability by eliminating wasted and unnecessary summary edges. We particularly focus on recursive nonterminals. Our key technical insight is that the wasted edge generations and insertions caused by recursive nonterminals can be avoided by modifying the parse trees either statically (by transforming the grammar) or dynamically (using a specialized online CFL-reachability solver). For example, if a recursive nonterminal B, generated by a rule $B ::= B\ X$, appears on the right-hand side of a rule $S ::= A\ B$, we can make S recursive (by introducing a new rule $S ::= S\ X$) and eliminate the original recursive rule ($B ::= B\ X$). Due to the rule $S ::= S\ X$, the shapes of the parse trees associated with the left-hand-side nonterminal S become more “skewed”. Thus, we name our approach *skewed tabulation* for CFL-reachability.

Skewed tabulation can significantly improve the scalability of CFL-reachability by reducing wasted and unnecessary summary edges. We have implemented skewed tabulation and applied the corresponding CFL-reachability algorithm to an alias analysis, a value-flow analysis, and a taint analysis. Our extensive evaluation based on SPEC 2017 benchmarks yields promising results. For the three client analyses, CFL-reachability based on skewed tabulation can achieve 3.34×, 1.13× and 2.05× speedup over the state-of-the-art RHS-tabulation-based CFL-reachability solver and consume 60.05%, 20.38% and 63.06% less memory, respectively. Furthermore, the cost of grammar transformation for skewed tabulation is negligible, typically taking less than one second.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**.

Additional Key Words and Phrases: CFL-reachability, tabulation schemes, performance

ACM Reference Format:
Yuxiang Lei, Camille Bossut, Yulei Sui, and Qirun Zhang. 2024. Context-Free Language Reachability via Skewed Tabulation. *Proc. ACM Program. Lang.* 8, PLDI, Article 221 (June 2024), 24 pages. https://doi.org/10.1145/3656451

[*]Equal contribution.

Authors' addresses: Yuxiang Lei, University of New South Wales, Sydney, Australia, yuxiang.lei@unsw.edu.au; Camille Bossut, Georgia Institute of Technology, Atlanta, USA, cbossut21@gatech.edu; Yulei Sui, University of New South Wales, Sydney, Australia, y.sui@unsw.edu.au; Qirun Zhang, Georgia Institute of Technology, Atlanta, USA, qrzhang@gatech.edu.

ACM 2475-1421/2024/6-ART221
https://doi.org/10.1145/3656451

1 Introduction

Context-free language reachability (CFL-reachability) is a fundamental framework for program analysis [Reps 1998]. A CFL-reachability problem instance contains an edge-labeled digraph G and a context-free grammar CFG. Two nodes u and v in G are reachable iff there exists a path between them, and the string spelled out by the path is accepted by the CFG. A variety of program-analysis problems, such as interprocedural data-flow analysis [Reps et al. 1995], program slicing [Clarke et al. 1999], shape analysis [Rehof and Fähndrich 2001], taint analysis [Kodumal and Aiken 2004], type-based flow analysis [Naeem and Lhoták 2008], and pointer analysis [Zheng and Rugina 2008], have been formulated as CFL-reachability problems.[1]

The standard CFL-reachability algorithm [Melski and Reps 2000; Reps 1998] is based on tabulation due to Reps et al. [1995]. To facilitate the Reps-Horwitz-Sagiv (RHS) tabulation [Reps et al. 1995], a preliminary step in CFL-reachability is to convert input grammars to a normal form in which the right-hand side of each rule has at most two symbols [Melski and Reps 2000; Reps 1998]. Then, for each nonterminal A in the grammar, the RHS tabulation computes *summary edges* (A-edges) according to each normalized rule $A ::= B\ C$. Note that the RHS tabulation is a generalization of the CYK algorithm [Younger 1967], a bottom-up method for CFL-parsing [Chatterjee et al. 2018; Pavlogiannis 2022; Reps 1998]. When processing a normalized rule of the form $A ::= B\ C$, the RHS tabulation needs to compute all B- and C-edges to generate A-edges. If one of the right-hand side nonterminals B or C is recursive, those recursive summary edges are computed nonetheless, although, arguably, only some edges eventually contribute to the generation of A-edges.

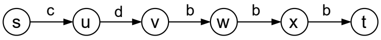
This paper proposes a general technique for speeding up CFL-reachability by improving the traditional RHS tabulation strategy. Our key idea is to analyze the input grammar and improve the summary edge generation statically and dynamically. *Statically*, we rearrange the recursions incurred in the RHS tabulation from a right-hand side nonterminal to a left-hand side nonterminal in the same grammar rule. Specifically, consider a rule of the form $S ::= A\ B$. Assume that B is recursive and can be generated by $B ::= B\ X$. The traditional RHS tabulation needs to compute all recursive B-edges in order to generate S-edges. In the CFL-parsing terminology, the parse tree of S is more "balanced" because it needs to combine the A-subtree and the B-subtree. Our approach rewrites the grammar by eliminating the recursive rule $B ::= B\ X$ and produces an equivalent grammar $S ::= A\ B$ and $S ::= S\ X$, *i.e.*, for $S ::= A\ B$, we promote the recursive behavior of B on the right-hand side to S on the left-hand side. Such promotion makes the parse tree of S more skewed. *Dynamically*, the tabulation algorithm can skip inserting some edges in the skewed grammar. For example, consider a nonterminal Y in a grammar after promoting recursive behaviors. If for all productions $X ::= Y\ Z$, whenever a Y-edge is generated, the Z-edges required for generating X-edges are already in the graph, then the Y-edge does not need to be inserted into the graph. It only needs to be added to the worklist for further processing. We name our approach *skewed tabulation* due to the skewed shape of the parse-tree structure that our approach induces.

The principal benefit of skewed tabulation is that it computes and inserts fewer summary edges than the traditional RHS tabulation. Thus, it reduces both CFL-reachability solving time and memory consumption. The challenge is to ensure that (1) in *static skewing*, the skewed grammar accepts the same language as the original grammar and (2) in *dynamic skewing*, the reachability algorithm correctly skips inserting unnecessary summary edges. To address these challenges, in static skewing, we propose a grammar transformation that utilizes the properties of recursive productions. We introduce a notation of *summary-degree* to measure the reduction of unnecessary recursive behaviors of non-S nonterminals while preserving grammar equivalence. In dynamic skewing, we analyze the productions of nonterminals and the solving process to identify *propagating*

[1] This paper focuses on *all-pairs* CFL-reachability problems.

$$\begin{aligned} S &::= A\ B \\ B &::= B\ B \mid b \\ A &::= c\ d \end{aligned}$$

(a) Input grammar *CFG*.



(b) Input graph G.

$$\begin{aligned} S &::= A\ B \mid S\ B \\ B &::= b \\ A &::= c\ d \end{aligned}$$

(c) CFG' for skewed tabulation.

Fig. 1. Motivating example of CFL-reachability. The start symbol in both *CFG* and CFG' is S, and CFL-reachability computes all S-edges in G.

edges, *i.e.*, edges not need to be inserted into the graph, and propose a new method for reachability propagation without inserting those propagating summary edges. By combining both static and dynamic skewing, we guarantee that skewed tabulation can reduce the number of summary edges in CFL-reachability.

We have implemented our skewed tabulation for CFL-reachability and applied it to three C/C++ practical client analyses, a field-sensitive alias analysis [Zheng and Rugina 2008], a context-sensitive value-flow analysis [Sui et al. 2014] and a taint analysis [Kodumal and Aiken 2004]. In addition, we compared skewed tabulation with a state-of-the-art RHS-tabulation-based CFL-reachability solver `POCR` [Lei et al. 2022b] and a state-of-the-art Datalog solver `Soufflé` [Scholz et al. 2016]. The empirical evaluation based on SPEC CPU 2017 benchmarks yields promising results. For the three client analyses, the skewed tabulation can achieve 3.34×, 1.13× and 2.05× speedup over the RHS tabulation and consume 60.05%, 20.38% and 63.06% less memory, respectively. Moreover, our CFL-reachability algorithm based on skewed tabulation is 3.16×, 1.85× and 2.28× faster than the *8-thread* `Soufflé` and consumes 61.45%, 16.47% and 56.56% less memory in the three client analyses. The running time for grammar transformation is less than one second, which is negligible.

To sum up, this paper makes the following contributions.

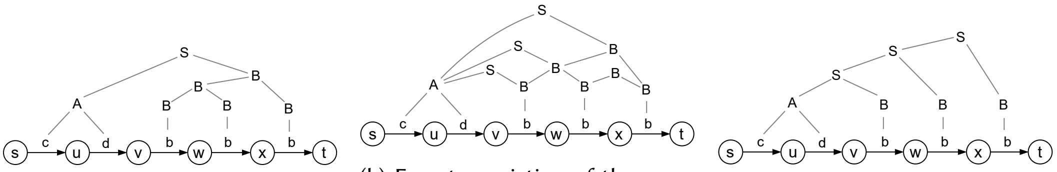
- We introduce a novel concept called skewed tabulation for improving the scalability of CFL-reachability algorithms. Skewed tabulation rearranges recursive grammar rules to more effectively reuse summary edges and gets rid of unusable summary edges.
- We present a formal analysis to demonstrate that skewed tabulation can reduce the number of summary edges obtained in the traditional RHS tabulation.
- We apply CFL-reachability via skewed tabulation to three client analyses [Sui et al. 2014; Zheng and Rugina 2008]. The evaluation based on SPEC 2017 benchmarks and real-world programs illustrates significant scalability advantages of the skewed tabulation.

The rest of the paper is structured as follows. Section 2 motivates skewed tabulation. Section 3 presents preliminaries. Section 4 discusses our skewed tabulation algorithm for CFL-reachability. Section 5 describes the evaluation setup and results. Finally, Section 6 surveys related work, and Section 7 concludes.

2 Motivating Example

This section motivates skewed tabulation for CFL-reachability using a concrete example. Consider a context-free grammar *CFG* and an input graph G in Figures 1a and 1b, respectively. Nonterminal S is the start symbol, and the CFL-reachability problem is to compute the S-reachability relations for all node pairs in G. The CFL-reachability tabulation is similar to the process of bottom-up CFL-parsing [Chatterjee et al. 2018; Pavlogiannis 2022; Reps 1998]. Therefore, we employ parse trees to demonstrate the tabulation results.

RHS tabulation. Traditional RHS tabulation [Reps et al. 1995] generates a new edge from two consecutive edges in the graph based on a *normalized* grammar, *i.e.*, every production contains at most two symbols on the right-hand side. Figure 2a gives a parse tree for generating $s \xrightarrow{S} t$ based



(a) Parse tree for $s \xrightarrow{S} t$ under RHS tabulation.

(b) Forest consisting of the parse trees for all inserted edges under RHS tabulation.

(c) Parse tree for $s \xrightarrow{S} t$ under skewed tabulation.

Fig. 2. Tabulation steps for computing the summary edge $s \xrightarrow{S} x$, which form parse trees for recognizing the path string "$c\,d\,b\,b\,b$" in Figure 1b.

on the grammar in Figure 1a and the graph in Figure 1b. Figure 2b gives all possible parse trees generated by RHS tabulation for the input graph.

Skewed tabulation and its benefits. Skewed tabulation skews the parse trees for nonterminals with recursive behaviors. By this means, it reduces unnecessary and unused summary edges in CFL-reachability. Figure 2c shows the parse tree for generating $s \xrightarrow{S} t$ (as well as $s \xrightarrow{S} w$ and $s \xrightarrow{S} x$) based on the skewed grammar in Figure 1c. The input grammar in Figure 1a was transformed such that the recursive behavior of non-S nonterminals, *i.e.*, $B ::= B\ B$, was promoted to be the recursive behavior of the start symbol as a new production $S ::= S\ B$. Comparing Figures 2a and 2c, we see that the parse tree in Figure 2c is more *skewed*, which means that it only computes the left sub-tree for $s \xrightarrow{A} v$, without the right sub-tree for $v \xrightarrow{B} t$ in Figure 2a, to obtain $s \xrightarrow{S} t$. We call the tabulation in Figure 2c *skewed tabulation*. Moreover, comparing Figures 2b and 2c, we see that skewed tabulation generates much fewer intermediate summary edges to compute the three S-summaries. Specifically, it reduces unnecessary summary edges $v \xrightarrow{B} x$, $w \xrightarrow{B} t$ and $v \xrightarrow{B} t$. To sum up, skewed tabulation achieves the reduction by "reusing" existing S-summaries.

Challenges. The essence of skewed tabulation includes a grammar transformation (to produce the CFG' in Figure 1c) and a tabulation algorithm (to skew the summary edge propagation). It is challenging to (1) preserve the grammar equivalence and (2) reduce the summary edges simultaneously. Our skewed tabulation utilizes *static skewing* and *dynamic skewing* to tackle the two challenges. In particular, static skewing promotes recursions in the input grammar by adding new equivalent grammar rules utilizing only nonterminals from the original grammar. In this example, we transfer the recursive behavior of B from $B ::= B\ B$ to S by adding a new rule $S ::= S\ B$. This restructure reduces some summary edges. While rewriting the grammar, static skewing also computes a set of *propagating nonterminals* that resemble nonterminals in a linear grammar. Dynamic skewing avoids inserting propagating nonterminals during the reachability propagation. In summary, static and dynamic skewing reduce summary edges by skewing the grammar and the tabulation, respectively.

3 Preliminaries

This section introduces the preliminaries of our work, including CFL-reachability in Section 3.1 and the correspondence between CFL-reachability and CFL-parsing in Section 3.2. Section 3.3 gives our problem formulation.

3.1 CFL-Reachability

A *context-free language (CFL)* is a set of strings derived via a *context-free grammar (CFG)*. In general, a context-free grammar $CFG = \langle \Sigma, N, T, P, S \rangle$ is comprised of five components where Σ is an alphabet that contains a set of symbols, $N \subset \Sigma$ is a set of nonterminal symbols (*i.e.*, nonterminals),

$T = \Sigma \backslash N$ is a set of terminal symbols (*i.e.*, terminals), P is a set of production rules (*i.e.*, productions), and $S \in N$ is the start symbol. In a CFG, each production of P shows how the nonterminal on the left-hand side derives the string on the right-hand side. A string $s \in T^*$ is accepted by *CFG* iff s can be derived from S via one or more productions in P. Such a string is also called an *S-string* of *CFG*. Given a context-free grammar *CFG* and an edge-labeled digraph $G = \langle V, E \rangle$, for a path $p = v_0 \xrightarrow{t_0} v_1 \xrightarrow{t_1} v_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{m-1}} v_m$ in G, the *path string* of p is the ordered concatenation of its edge labels, denoted by $R(p) = t_0 t_1 t_2 \cdots t_{m-1}$. Consider a nonterminal $X \in N$ and two nodes $v_i, v_j \in V$. We say there is an *X-edge* $u \xrightarrow{X} v$ iff there is a path p from v_i to v_j such that $R(p)$ can be derived from X. A context-free language reachability (CFL-reachability) problem is to determine in G the node pairs connected by a path whose edge labels form an S-string.

Definition 3.1 (CFL-reachability). Given a context-free grammar $CFG = \langle \Sigma, N, T, P, S \rangle$, and an edge-labeled digraph $G = \langle V, E \rangle$, a CFL-reachability problem is to determine all the node pairs $(v_i, v_j) \in V \times V$ such that there is an S-edge from v_i to v_j.

CFL-reachability via RHS tabulation. Solving CFL-reachability is a process that iteratively generates new summary edges from existing ones according to the productions in the *CFG*, and outputs the set of valid S-edges. The objective of the algorithm is to generate these S-edges. CFL-reachability can be solved by a dynamic-programming-style algorithm [Melski and Reps 2000; Reps 1998]. The algorithm is based on the Reps-Horwitz-Sagiv tabulation [Reps et al. 1995]. Given a production $X ::= Y\ Z \in P$, two consecutive edges $u \xrightarrow{Y} v$ and $v \xrightarrow{Z} w$ generate $u \xrightarrow{X} w$. Usually, $u \xrightarrow{X} w$ is inserted into the graph to make explicit the X-reachability relation from u to w if it was not already inserted. Algorithm 1 gives the CFL-reachability algorithm based on RHS tabulation. The algorithm requires the input CFG to be *normalized* so that the right-hand side of each production contains at most two symbols. The standard algorithm maintains a worklist W holding unsolved new edges. During reachability solving, the algorithm iteratively pops a worklist item, traverses the edges adjacent to the worklist item, and adds the newly generated edges to the graph and the worklist based on the grammar. Lines 6–13 show the processes of generating summary edges and ultimately S-edges, with graph/worklist operations. The worklist W also rejects edges already in the graph to avoid duplicate work and ensure the termination of the algorithm.

3.2 CFL-reachability and CFL-parsing

As noted by Reps [1998], the RHS tabulation is a generalization of the CYK algorithm [Younger 1967] for CFL-parsing. The technical description of our skewed tabulation is partially based on parse trees. Thus, we briefly review the connection between CFL-reachability and CFL-parsing.

Parse trees are a useful abstraction to visualize the effects of grammar structure on tabulation during CFL-reachability. A parse tree is a rooted tree that represents the syntactic structure of a string according to some context-free grammar. Specifically, given a context-free language *CFL* based on a grammar $CFG = \langle \Sigma, N, T, P, S \rangle$, a parse tree of *CFL* is a rooted tree such that

(1) each leaf is a terminal in T, each internal node is a nonterminal of N and the root is the start symbol S; and
(2) each internal node and its children form the left-hand side and the right-hand side, respectively, of a production belonging to P.

Given a parse tree constructed based on a context-free grammar *CFG*, concatenating the *leaves* from the leftmost to the rightmost always forms a string belonging to the language accepted by the *CFG*. Parse trees are closely related to CFL-reachability and CFL-parsing. Indeed, when traversing a parse tree from root to leaves, the tree represents the process of deriving a string from the start

Algorithm 1: CFL-reachability via RHS Tabulation.

```
Input  : Edge-labeled directed graph $G = \langle V, E \rangle$, normalized $CFG = \langle \Sigma, N, T, P, S \rangle$
Output : The set $\{ (u, v) \mid u \xrightarrow{S} v \in G \}$
$W \leftarrow E$
foreach $X ::= \varepsilon \in P$ do
    foreach $v \in V$ do add $v \xrightarrow{X} v$ to $E$ and to $W$
while $W \neq \emptyset$ do
    select and remove an edge $u \xrightarrow{Y} v$ from $W$
    foreach $X ::= Y \in P$ do
        if $u \xrightarrow{X} v \notin E$ then add $u \xrightarrow{X} v$ to $E$ and to $W$
    foreach $X ::= Y\ Z \in P$ do
        foreach $v \xrightarrow{Z} w \in E$ do
            if $u \xrightarrow{X} w \notin E$ then add $u \xrightarrow{X} w$ to $E$ and to $W$
    foreach $X ::= Z\ Y \in P$ do
        foreach $w \xrightarrow{Z} u \in E$ do
            if $w \xrightarrow{X} v \notin E$ then add $w \xrightarrow{X} v$ to $E$ and to $W$
```

symbol via a series of productions in *CFG*; when traversing from leaves to root, the tree represents the process of checking if a string is accepted by *CFG*.

Example 3.2. Let us revisit our motivating example to illustrate the relationship between parse trees and tabulation. Figure 2c shows a parse tree for "*cdbbb*" based on the grammar in Figure 1c. When we see the parse tree from a *top-down* perspective, its a process of deriving the string "*cdbbb*" from the start symbol S, in a process of "S" to "SB" via $S ::= S\ B$ then to "$ABBB$" via $S ::= A\ B$ and finally to "*cdbbb*" via $A ::= c\ d$ and $B ::= b$. When we see the parse tree from a *bottom-up* perspective, it is a process of determining whether the string "*cdbbb*" can be derived from S, which is exactly the process of tabulation. Thus, tabulation can be seen as the reverse process of string derivation. The difference is that in the bottom-up tabulation process, no matter whether a string belongs to the CFL, nonterminals corresponding to its sub-strings will always be generated first. This manifests in summary edges in the graph, like $s \xrightarrow{A} v$, $v \xrightarrow{B} x$, and $v \xrightarrow{B} t$ in Figure 2a.

Grammar structure and summary edges. Based on different grammar structures (Figure 1), the corresponding tabulation structures (Figure 2) can differ even for the same path string s. Given a grammar *CFG*, we introduce the notion of *summary-degree* of a string s to quantify the difference.

Definition 3.3 (Summary-degree). Consider a path p and its path string $s = R(p)$ in a graph G. Given a grammar *CFG*, we call the summary-count $SC_{CFG}(s)$ of the path string s the number of summary edges inserted into G by summarizing the path p. This number is 0 if no summary edges corresponding to p are inserted. The summary-degree is the sum of summary-counts for all the sub-strings of s, including s itself, denoted as $SD_{CFG}(s)$. Let the set of all sub-strings of s be $sub(s)$, we have $SD_{CFG}(s) = \sum_{s' \in sub(s)} SC_{CFG}(s')$.

Example 3.4. We discuss the summary-degree of the string $s = cdbbb$ for the grammars in Figures 1a and 1c, which we call CFG_1 and CFG_2 respectively. Figures 2b and 2c give all nontermals related to parsing s based on CFG_1 and CFG_2, respectively. This is a visual representation of their summary-degree, which we can compute: $SD_{CFG_1}(s) = 10$ and $SD_{CFG_2}(s) = 7$, respectively. Thus, $SD_{CFG_2}(s) < SD_{CFG_1}(s)$, indicating that in CFL-reachability the number of summary edges needed to compute s using CFG_2 is less than the number using CFG_1.

Summary-degree and skewing. We call the parsing structure of a grammar with a lower summary-degree for a string *more skewed* because of the shape of the parse trees it produces, as shown in the parse trees in Figure 2c compared to Figure 2a. In terms of the CFL-reachability algorithm, measuring the summary-degree $SD_{CFG}(s)$ is a way to measure how much work is done computing and/or inserting the corresponding summary edges. Note that the summary-degree also applies to all sub-strings of s.

Summary-degree and linear CFL-reachability. Linear grammar rules that contain at least one terminal on the right-hand side have an advantage: the input graph contains all its terminal edges from the beginning of the algorithm. If a summary edge labeled by X is generated in CFL-reachability, we can immediately check if any linear grammar rules containing X can ever be applied because all terminal edges must be in the input graph at the beginning of the algorithm. Thus, when using a linear grammar, we do not need to insert any summary edges, which reduces the summary-degree *dynamically*. The seminal work of Yannakakis [1990] gives a specialized CFL-reachability algorithm for linear grammars that runs in $O(mn)$ time. In linear CFL-reachability, the summary-degree is 0 for all strings, since no summary edges are inserted. We call this case *fully skewed* as it is the smallest possible summary-degree for all strings.

3.3 Problem Formulation

Section 3.2 defines the summary-degree $SD_{CFG}(s)$ as the number of summary edges that are inserted in a CFL-reachability algorithm. We call a grammar with a lower summary-degree over all strings *more skewed* than one with a higher summary-degree. Indeed, summary-degree can be reduced both statically (by changing the structure of the grammar CFG) and dynamically (by specializing the CFL-reachability algorithm itself to the grammar structure).

Ideally, we want to transform the input grammar so that all strings have an summary-degree of 0, which implies we can obtain an equivalent linear grammar and utilize the corresponding linear CFL-reachability algorithm that inserts no summary edges. However, not all context-free grammars have a linear form, and even checking if such a form exists is undecidable [Greibach 1966]. As a result, our approach *skewed tabulation* skews the grammar structure as much as possible through static transformation and dynamic reachability propagation. As the summary-degree is a visual measurement of skewedness, we state our problem formulation as follows.

> Given a grammar CFG and a graph G, skewed tabulation reduces the summary-degree $SD_{CFG}(s)$ of some path strings $s = R(p)$ in G.

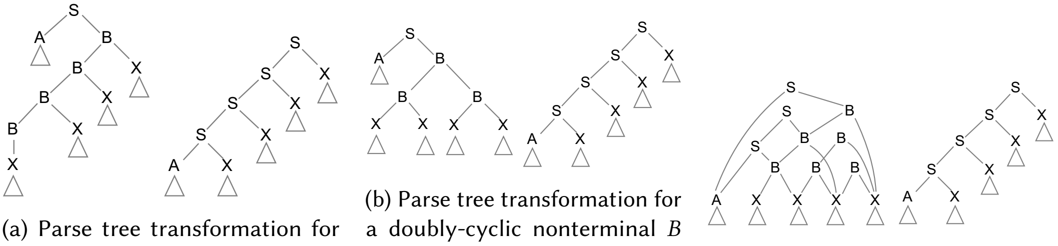
Section 4 presents the two steps of skewing: static skewing and dynamic skewing. Static skewing involves analyzing and transforming the input grammar, and dynamic skewing uses the statically skewed grammar to further exploit skewed parse trees at runtime.

4 CFL-Reachability via Skewed Tabulation

This section introduces skewed tabulation, which is comprised of a static and a dynamic part. Our key insight is that the parse trees of a given grammar can be skewed statically by equivalently rearranging their sub-trees through grammar transformation. Using the transformed grammar, dynamic skewing can be applied by adjusting the CFL-reachability algorithm Sections 4.2 and 4.3 introduce our static and dynamic skewing, respectively.

4.1 Basic Idea

As shown in Section 3.2, skewed tabulation reduces the summary-degree (Definition 3.3) of all strings under the input grammar. This section discusses two steps in skewed tabulation: (1) *static*



(a) Parse tree transformation for a left-cyclic nonterminal B with grammar rules $B ::= B\,X$, $S ::= A\,B$.

(b) Parse tree transformation for a doubly-cyclic nonterminal B with grammar rules $B ::= B\,B$, $B ::= X$, and $S ::= A\,B$.

(c) Parse forests for $AXXXX$ for the original and transformed grammars.

Fig. 3. Visual representation of the grammar transformations using cyclic nonterminals.

skewing, a grammar transformation in the preprocessing stage, and (2) *dynamic skewing*, a skewed CFL-reachability algorithm.

Static skewing. The motivating example shows that we can restructure the input grammar in Figure 1a to Figure 1c to reduce the summary-degree of a string, as shown by the summary edge reduction in Figure 2c compared to Figure 2a. Static skewing aims to reduce the summary-degree for all strings belonging to the language by restructuring the input grammar. We promote recursive behavior from non-S nonterminals up the parse tree, closer to the root node S. This transformation reuses previously valid sub-trees in other parse trees, reducing the number of distinct sub-trees needed, and therefore reducing the number of summary edges generated in CFL-reachability.

Dynamic skewing. When solving CFL-reachability based on a skewed grammar, there are often summary edges that do not need to be inserted into the graph to obtain the correct set of S-edges . We call such edges *propagating edges*. Specifically, a propagating edge p is always generated after all the relevant adjacent edges, used with p to generate new summary edges, are already in the graph. Thus, dynamic skewing treats propagating edges differently by only adding them to the worklist, not the graph. This reduces the summary-degrees of path strings dynamically.

4.2 Static-Skewing: Transforming Context-Free Grammar

Static skewing transforms the input context-free grammar, such that the transformed grammar has a lower summary-degree over all strings s under RHS tabulation.

Naively rewriting a grammar to reduce the summary-degree on a case-by-case basis does not necessarily lead to better performance for CFL-reachability overall; restructuring the grammar to improve summary-degree for one string may have negative repercussions on the summary-degree of other strings. To reduce the summary-degree of a grammar over all strings, our grammar transformation reuses sub-trees of parse trees derived from the original grammar to replace sub-trees of different parse trees also derived from the original grammar. The challenge is rewriting the grammar to do this while preserving equivalence with the original grammar *CFG*.

The static skewing transformation contains two phases: grammar transformation and grammar annotation. The grammar transformation alone produces parse trees with a lower summary-degree than the original, and grammar annotation collects information for the dynamic skewing approach presented in Section 4.3. A context-free grammar *CFG* in CNF has productions of the form $A ::= B\,C$ that fall into three categories: (1) non-recursive, where A, B, and C are distinct, (2) left- or right-recursive, where $A = B$ or C, meaning $A ::= A\,C$ or $A ::= B\,A$ respectively, and (3) doubly recursive a.k.a. transitive, where $A = B = C$. To reduce the summary-degree of the input CFG for all strings, we identify nonterminals that recursively generate or append a specific set of nonterminals where they are used. We call these *cyclic nonterminals*. For example, given a nonterminal N that satisfies

Algorithm 2: Skewed_Form(*CFG*).

```
Input  : Context-free grammar $CFG$, set of target nonterminals $T$
Output : Skewed grammar $SF(CFG) = \{CFG', PN\}$, where $PN$ is the set of propagating nonterminals
$CFG' \leftarrow CFG$
foreach $S' \in T$ do
    if $\exists S' ::= A\ B \in P$ then
        if isCyclic($B$, $CFG$) and $B \notin T$ and all rules for $S'$ are of the form $S' ::= X\ B$ then
            $CFG' \leftarrow S' ::= S'\ B$
            if $B ::= \epsilon \in P$ then
                remove $S' ::= X\ B$ from $CFG'$
                $CFG' \leftarrow S' ::= X$
        if isCyclic($A$, $CFG$) and $A \notin T$ and all rules for $S'$ are of the form $S' ::= A\ X$ then
            $CFG' \leftarrow S' ::= A\ S'$
            if $A ::= \epsilon \in P$ then
                remove $S' ::= A\ X$ from $CFG'$
                $CFG' \leftarrow S' ::= X$
foreach $S' ::= A\ S' \in P, S' \in T$ do
    if isCyclic($A$, $CFG'$) and $A \notin T$ and $A$ only used in singly-recursive rules then
        if $A ::= A\ A \in P$ then
            remove $A ::= A\ A$ from $CFG'$
        if $A ::= \epsilon \in P$ then
            remove $A ::= \epsilon$ from $CFG'$
        if $A ::= a \in P, a \in \Sigma$ then
            remove $A ::= a$ from $CFG'$
            $CFG' \leftarrow X ::= a\ X$ for all $X :: A\ X \in P$
            $CFG' \leftarrow X ::= X\ a$ for all $X :: X\ A \in P$
        if $A ::= X\ A \in P$ or $A ::= A\ X \in P$ then
            remove $A ::= X\ A$ or $A ::= A\ X$ from $CFG'$
            $CFG' \leftarrow S' ::= X\ S'$
foreach $S' ::= S'\ B \in P, S' \in T$ do
    // This procedure is similar to the above but handles left-recursion. It can be found in
       the supplemental material.
    HandleLeftRecursion($B$, $S'$, $CFG'$)
$PN \leftarrow \emptyset$
foreach $A \in N$ do
    if $\forall X ::= Y\ A \in P$ and $\forall X ::= A\ Y \in P$ we have $Y \in \Sigma$ and $X \neq A$ then
        $PN \leftarrow A$
return $(CFG', PN)$
```

this property, we can replace uses of $A ::= X\ N$ with two rules $A ::= X\ t$ and $A ::= A\ t$ for all ts as long as this does not change the language of nonterminal A.

Definition 4.1 (Cyclic Nonterminal). A cyclic nonterminal is a nonterminal that falls into exclusively one of the following categories:

- Left-cyclic: The nonterminal only has associated left-recursive rules.
- Right-cyclic: The nonterminal only has associated right-recursive rules.
- Doubly-cyclic: The nonterminal has one associated transitive rule, and all other associated rules are non-recursive.

Example 4.2. In Figure 3, we identify a cyclic nonterminal B in two different grammars. In both grammars, B recursively derives strings of X nonterminals of infinite length. In Figure 3a, B is a left-cyclic nonterminal. In Figure 3b, B is a doubly-cyclic nonterminal.

To compare the summary-degree for more than one string at a time under two grammars, we can compare the summary-degree of *sentential forms*. Recall that a string in sentential form contains terminals and nonterminals from a grammar [Aho et al. 2006]. Given two grammars G_1 and G_2 and a string s in sentential form, we assume inequality $SD_{G_1}(s) \leq SD_{G_2}(s)$ correctly holds for all the strings s represents. We formally explain how to compare summary-degree for sentential form in Lemma 4.8 with a proof in Section 4.4.

Example 4.3. In Figure 3a, B is a left-cyclic nonterminal, and in Figure 3b B is a doubly-cyclic nonterminal. We can promote the recursion of B up the parse tree to the nonterminal S as shown in Figures 3a and 3b. To compute the summary-degree, we treat A and X as terminal symbols in the sentential form string $AXXXX$. For both Figures 3a and 3b we have $SD_{CFG}(AXXXX) = 9$, and $SD_{CFG'}(AXXXX) = 5$; the parse forests with all nonterminals are shown in Figure 3c. Since the transformation didn't affect the nonterminal X, the summary-degree of all strings $AXXXX$ represents also decreased according to Lemma 4.8.

High-level transformation. Our objective is to reduce the summary-degree for some strings s without negatively affecting the summary-degree of other strings. At the parse-tree level, we identify cyclic-nonterminals, which reveal reusable sub-trees, and replace sub-trees derived from the original grammar with these. We derive new grammar rules to accomplish this sub-tree replacement using only nonterminals from the original grammar. We remove rules that become incorrect or redundant. Intuitively, this reduces the summary-degree of the transformed grammar for the affected parse trees because we have eliminated summary edges without affecting the reachability result (S-edges). We call the transformed grammar *skewed* because of the shape of the resulting parse trees.

Transformation steps. Algorithm 2 gives the static skewing transformation, which is a top-down procedure. It starts with the start symbol S and then operates on the remaining nonterminals. We call the nonterminals whose language cannot change *target nonterminals*, starting with S. We visit the right-hand sides for the symbol S of the form $S ::= A\ B$.

(1) Transform any cyclic behavior of A or B on the right-hand side into equivalent left- or right-recursive behavior for S, of the form $S ::= S\ B$ or $S ::= A\ S$, if the chosen nonterminal A or B is used in the same position for all right-hand sides of S. If an epsilon rule exists for A or B, rules of the form $S ::= A\ X$ or $S ::= X\ B$ are replaced with $S ::= X$ (lines 2-13).
(2) Locate a nonterminals A such that $S ::= A\ S$, A is cyclic, and A is only used in singly-recursive rules (lines 14-15).
(3) Given a doubly-cyclic nonterminal A, remove $A ::= A\ A$ (lines 16-17).
(4) Remove epsilon and unary rules for this nonterminal, since these are redundant and replaceable. Replace unary rule with additional linear rules where A is used (lines 18-23).
(5) For any recursive rules associated with A, propagate the recursion up to the S symbol considered (lines 24-26).
(6) Repeat steps (2)-(5) to handle the left-recursive case where A is such that $S ::= S\ A$ (lines 27-29).
(7) Iteratively apply this transformation to the remaining nonterminals, where S and the next nonterminal are labeled as target nonterminals.

Complexity of static skewing. We prove the correctness of this transformation and annotation and provide formal guarantees of its benefits in Section 4.4. For the transformation complexity, given M rules and N nonterminals, the complexity is $O(M^2 \times N)$. The transformation is applied N times, where each nonterminal is chosen as a target once (and S is always a target), and the for-each loops on lines 2, 14, 27, and 31 are bounded by the number of rules M. Within these for each loop, checking the *isCyclic* property along with other if conditions are also bounded by M. In practice, the transformation takes less than a second.

Annotation steps. We annotate nonterminals that are only used in linear-like rules in the skewed output grammar called *propagating nonterminals*. These are used in propagating edge recognition in the online CFL-reachability solving process, described in Section 4.3.

Definition 4.4 (Propagating Nonterminal). A nonterminal $Y \neq S$ is called a propagating nonterminal if all productions containing Y on the right-hand side are in the form of $X ::= Y$, $X ::= Y\ z$ or $X ::= z\ Y$ where z is a terminal.

4.3 Dynamic Skewing: Skewed Tabulation algorithm

Dynamic skewing focuses on identifying and handling edges that do not need to be inserted into the graph. We call such edges *propagating edges*. A typical example is the edges labeled by a propagating nonterminal (Definition 4.4). Consider a propagating nonterminal X and a newly generated X-edge that is pushed into the worklist. According to Definition 4.4, all productions containing X on the right-hand side are in the form of $Y ::= X$, $Y ::= X\ z$ or $Y ::= z\ X$, where z is a terminal. Namely, the z-edges are always in the initial graph. Thus, no matter whether the X-edges are in the graph or not, the Y-edges can be generated when the X-edge is popped from the worklist and processed. Therefore, the X-edges do not need to be inserted into the graph. In fact, besides the edges labeled by propagating nonterminals, which can be determined statically by the grammar transformation, propagating edges are also identified dynamically in the solving process. The following lists two types of propagating edges:

- *Static:* an edge $u \xrightarrow{X} v$ such that X is a propagating nonterminal. This type of propagating edge can be found in static skewing.
- *Dynamic:* an edge $u \xrightarrow{X} v$ that is generated by $X ::= X\ X$ and all the binary productions containing X on the right-hand side are recursive, *i.e.*, in the form of $A ::= X\ A$ or $A ::= A\ X$. This is because with the X-edges that were *not* generated by $X ::= X\ X$ inserted into the graph, those recursive productions can generate all the X- and A-edges without inserting $u \xrightarrow{X} v$ into the graph. This type of propagating edge can only be determined in the dynamic solving process.

The dynamic skewing approach leverages the result of static skewing. The skewed CFL-reachability algorithm takes the skewed context-free grammar (Section 4.2) as input and deals with propagating edges differently from the standard RHS-tabulation CFL-reachability algorithm. Algorithm 3 gives the skewed-tabulation CFL-reachability algorithm, where the treatment of propagating edges is displayed in lines 12–18. Our dynamic skewing reduces summary-degree by not inserting propagating edges. Specifically, once a generated edge is identified as a propagating edge, it is *never* inserted into the graph. To ensure termination, Algorithm 3 maintains an edge set *PE* to record propagating edges. Once a propagating edge is generated, it is inserted into *PE* and the worklist W if it is not already in *PE*. Otherwise, the propagating edge will be discarded.

The main structure of Algorithm 3 (lines 1–11) is similar to the RHS-tabulation-based standard CFL-reachability algorithm. The difference is dealing with propagating edges (line 12–18). Specifically, when generating an edge, *e.g.*, $v_i \xrightarrow{X} v_j$, it first identifies whether the edge is a propagating edge, using the two criteria proposed above. If the edge meets any of the two criteria, the algorithm adds it to the worklist but not into the graph, as seen in lines 13–16. Otherwise, the algorithm inserts it into the graph and the worklist.

Example 4.5. Let us consider a path $p = x \xrightarrow{b} y \xrightarrow{b} u \xrightarrow{b} v$ under a grammar containing $B ::= B\ B \mid b$, where B is not a cyclic nonterminal. In this case, $B ::= B\ B$ cannot be removed by static skewing (Algorithm 2). When processing p, $x \xrightarrow{B} y$, $y \xrightarrow{B} u$, and $u \xrightarrow{B} v$ are always generated and

Algorithm 3: CFL-Skewed: CFL-Reachability via Skewed Tabulation.

```
   Input  : Edge-labeled directed graph $G = \langle V, E \rangle$, normalized $CFG = \langle \Sigma, N, T, P, S \rangle$
   Output : The set $\{ (v_i, v_j) \mid v_i \xrightarrow{S} v_j \in E \cup PE \}$
1  $(CFG' = \langle \Sigma, N', T, P', S \rangle, PN) \longleftarrow$ Skewed_Form($CFG$)                         // Algorithm 2
2  $PE \leftarrow \emptyset$ ; $W \leftarrow E$
3  foreach $X ::= \varepsilon \in P'$ do
4      foreach $v \in V$ do add $v \xrightarrow{X} v$ to $E$ and to $W$
5  while $W \neq \emptyset$ do
6      select and remove an edge $v_i \xrightarrow{Y} v_j$ from $W$
7      foreach $X ::= Y \in P'$ do Update($v_i \xrightarrow{X} v_j$)                                       // line 12
8      foreach $X ::= Y\ Z \in P'$ do
9          foreach $v_j \xrightarrow{Z} v_k \in E$ do Update($v_i \xrightarrow{X} v_k$)                      // line 12
10     foreach $X ::= Z\ Y \in P'$ do
11         foreach $v_k \xrightarrow{Z} v_i \in E$ do Update($v_k \xrightarrow{X} v_j$)                      // line 12

12 Procedure Update($v_i \xrightarrow{X} v_j$)
13     if $X \in PN$ then
14         if $v_i \xrightarrow{X} v_j \notin PE$ then add $v_i \xrightarrow{X} v_j$ to $PE$ and to $W$     // $v_i \xrightarrow{X} v_j$ is a static propagating edge
15     else if $v_i \xrightarrow{X} v_j$ is generated by $X ::= X\ X$ and $(\forall A ::= X\ B \in P', \forall A ::= B\ X \in P' : A = B)$ then
16         if $v_i \xrightarrow{X} v_j \notin PE$ then add $v_i \xrightarrow{X} v_j$ to $PE$ and to $W$     // $v_i \xrightarrow{X} v_j$ is a dynamic propagating edge
17     else if $v_i \xrightarrow{X} v_j \notin E$ then
18         add $v_i \xrightarrow{X} v_j$ to $E$ and to $W$                                               // $v_i \xrightarrow{X} v_j$ is not a propagating edge
```

inserted into the graph. The difference between RHS tabulation and dynamic skewing is that RHS tabulation always inserts $x \xrightarrow{B} u$ (also $y \xrightarrow{B} v$ and $x \xrightarrow{B} v$) into the graph, whereas dynamic skewing does not insert them. Thus, $x \xrightarrow{B} v$ will only be generated once by dynamic skewing because it does not insert $x \xrightarrow{B} u$ into the graph. In this case, dynamic skewing reduces the summary-degree of the path string $R(p)$ by 3, which not only reduces memory consumption but also reduces the computations needed to generate summary edges.

As we can see in lines 12–18, our skewed solver guarantees that the summary-degrees of all the path strings in the dynamic solving process are always smaller than (or at least equal to when there are no propagating edges) the RHS-tabulation-based standard solver. Because the Algorithm 3 only differs from the standard algorithm by skipping the insertion step for certain edges and does not change the dynamic programming scheme, the time complexity of Algorithm 3 is the same as that of the standard algorithm, *i.e.*, $O(n^3)$. Section 4.4 discusses the correctness of Algorithm 3.

4.4 Formal Analysis of Skewed Tabulation

We first discuss the correctness of static skewing and dynamic skewing.

Lemma 4.6 (Correctness of static skewing). *Given an input grammar CFG and the transformed grammar CFG′, the language of CFG and CFG′ are the same.*

Proof. We introduce no new nonterminals in our grammar transformation. All new grammar rules use the properties of cyclic nonterminals (Definition 4.1) to introduce left- or right-recursion when this nonterminal is used on the right-hand side of a grammar rule. In particular, if C is cyclic, a use of C in a rule $A ::= B\ C$ can create a new rule $A ::= A\ C$, which is correctly provided that all rules for A have a C nonterminal in that position. As this is redundant with the cyclic property of the rule C, adding this rule does not change the language of CFG.

In terms of removing grammar rules, after adding rules as explained above, we remove the grammar rules that make C cyclic provided that *all right-hand side uses* of C have been turned into recursion. Since the cyclic nature of C was captured in the rules of the form $A ::= A\ C$, it is correct to remove the rule $C ::= C\ C$. If C is recursive like $C ::= C\ X$, we add a rules $A ::= A\ X$ to replace this rule and remove it. Again, we reiterate that this only occurs if all right-hand side uses of C are in recursive rules (for rules that do not have C on the left-hand side).

The remaining transformations replace C uses with equivalent terminals or epsilon when possible, which are trivially correct. Throughout this transformation, the nonterminal C cannot be a target nonterminal. S is always a target nonterminal. The only nonterminal language that changes throughout this iteration of the transformation is the language of C. Therefore, since $S \neq C$ the language of S has not changed, so the language of CFG and CFG' are the same. □

Lemma 4.7 (Correctness of Dynamic Skewing). *Given a skewed context-free grammar CFG', Algorithm 3 (without further skewing CFG' in line 1) and the standard algorithm (Algorithm 1) produce the same result from the same graph.*

Proof. We use a proof by contradiction. We want to show that the set of S edges in the output graph is the same using either algorithm. Thus, without loss of generality, we will prove the following property: the set of edges that are not labeled by propagating nonterminals are the same in the output graphs of Algorithm 1 and Algorithm 3. The only difference between Algorithm 1 and Algorithm 3 is that the first does not insert propagating nonterminals into the graph; these are only inserted into the worklist. Recall that a propagating nonterminal is a nonterminal that is only used on the right-hand side of linear rules (Definition 4.4). Consider a grammar rule $A := PQ$, where P is a propagating nonterminal, and let there be two graph edges: $v_k \xrightarrow{P} v_l, v_l \xrightarrow{Q} v_m$. Assume by contradiction that $v_k \xrightarrow{A} v_m$ is not generated. This means that when $v_k \xrightarrow{P} v_l$ arrived in the worklist, $v_l \xrightarrow{Q} v_m$ was not present in the graph. However, propagating nonterminals are only used in linear rules, meaning Q is actually a terminal symbol, and terminal edges are always present in the graph, so we have a contradiction. Therefore $v_k \xrightarrow{A} v_m$ is generated. Similarly, given a grammar rule $A := QP$ and edges $v_k \xrightarrow{Q} v_l, v_l \xrightarrow{P} v_m$, the edge A must be generated since Q has to be a terminal symbol. This covers all cases in which a summary edge may be generated based on a propagating nonterminal. Applying this recursively, all nonterminal edges that depend on a propagating nonterminal will eventually be generated since the rest of the algorithm is unchanged. Therefore, all edges that are not labeled by propagating nonterminals are the same in the output graphs of both Algorithms. As a result, the set of S-edges output by Algorithm 1 is the same as the set of S-edges output by Algorithm 3. □

Next, we discuss the summary edge reduction achieved by static skewing. Specifically, we want to show that the summary-degree of all strings s is the same or reduced under the transformed grammar CFG' compared to the original grammar CFG. First, we show this is true for strings in sentential form, which can contain nonterminals.

Lemma 4.8 (Comparing summary-degree of sentential forms). *Given a string s in sentential form containing nonterminal X and two grammars G_1, G_2, the inequality $SD_{G_2}(s) \leq SD_{G_1}(s)$ holds if (1) the strings X derives in G_2 are a subset of the strings X derives in G_1 and (2) the inequality $SD_{G_2}(s') \leq SD_{G_1}(s')$ holds for all s' in this subset of X-derived strings.*

Proof. Consider the inequality $SD_{G_2}(s) \leq SD_{G_1}(s)$, where s is in sentential form, and the definition of summary-degree (Definition 3.3). The summary-degree of a string s is reduced if the summary-degree for sub-strings s' are reduced since the summary-degree is defined as the sum of summary-count of the sub-strings of s. The summary-degree of the strings represented by the abstracted nonterminals X denoted s' are always reduced in G_2: for the strings derived from X in G_1 that are invalid in G_2 the summary-degree is now 0, and for the strings derived from X in both grammars, $SD_{G_2}(s') \leq SD_{G_1}(s')$. Thus, all the abstracted sub-strings s' have a reduced summary-degree. Therefore, all strings represented by this sentential form in using the original grammar have a reduced summary-degree using the new grammar. □

We use Lemma 4.8 to compare the summary-degree of all strings s under the grammar CFG'.

Theorem 4.9 (Summary-degree Reduction of static Skewing). *Given the summary-degree of any string s, under the transformed grammar CFG' and the original grammar CFG, we have $SD_{CFG'}(s) \leq SD_{CFG}(s)$.*

Proof. Nonterminals in CFG') fall into two categories: (1) they parse a subset of the strings they parsed in the original grammar, or (2) they parse the same strings as in the original grammar. This means the summary-degree of some sentential forms has decreased as the summary-count of some strings from the original grammar has become 0. Since the summary-degree of a sentential form can only decrease, applying Lemma 4.8 recursively implies that for any string s, the summary-degree of CFG' will either be the same or reduced compared to the original grammar. Therefore, we reduced the summary-degree of CFG' over all strings. □

Finally, we discuss the formal guarantees of skewed tabulation. Lemma 4.6 states that the static skewing preserves the language of the input grammar *CFG*, and Theorem 4.9 states that the transformed grammar only reduces the summary-degree, and therefore the number of summary edges, in CFL-reachability. Lemma 4.7 states that the dynamically skewed tabulation CFL-reachability algorithm preserves the same result as the RHS tabulation. Trivially, dynamic skewing can only reduce the number of summary edges inserted because its only difference with Algorithm 1 is skipping the insertion of some special edges. Thus, our overall approach preserves the output of RHS tabulation while reducing the number of summary edges inserted.

Theorem 4.10. *Given a grammar CFG and an edge-labeled graph G, Algorithms 1 and 3 produce the same reachability results.*

5 Experiments

CFL-reachability has been extensively discussed in the literature. Instead of building a CFL-reachability solver from scratch, we implemented our CFL-Skewed algorithm (CFL-Skewed) on a state-of-the-art general CFL-reachability solver POCR [Lei et al. 2022b]. The reachability propagation in POCR follows the traditional RHS tabulation. Therefore, we refer POCR as the baseline RHS-tabulation implementation (CFL-RHS). In this setting, both CFL-Skewed and CFL-RHS can benefit from POCR's ability to handle transitive rules. As a result, we can demonstrate the performance difference caused by different tabulation strategies. In the literature, Datalog has been a prominent framework for formulating program-analysis problems [Bravenboer and Smaragdakis 2009; Lu et al. 2013; Scholz et al. 2016; Whaley et al. 2005]. In particular, CFL-reachability can be encoded as a Datalog problem instance [Reps 1998]. Therefore, we adopt a state-of-the-art Datalog solver Soufflé [Scholz et al. 2016]. We perform extensive evaluations on three C/C++ client analyses: an alias analysis [Zheng and Rugina 2008], a value-flow analysis [Sui et al. 2014], and a polymorphic taint analysis [Kodumal and Aiken 2004]. Our experiments focus on three research questions.

RQ1: How much can CFL-Skewed reduce inserted summary edges?
RQ2: To what extent can CFL-Skewed accelerate CFL-reachability by reducing summary edges?
RQ3: Can CFL-Skewed outperform state-of-the-art solvers in memory overhead?

Table 1. Benchmark information, where PEG is used for alias analysis, and SVFG is used for value-flow analysis and taint analysis. #LOC(k) denotes the number of lines of codes, measured in thousands. #SumEdge by `CFL-RHS` denotes the number of summary edges in the final graph computed by `CFL-RHS`.

Bench.	#LOC(k)	PEG		SVFG		#SumEdge by `CFL-RHS`		
		#Node	#Edge	#Node	#Edge	Alias	Value-Flow	Taint
1.cactus	257	45,696	104,816	235,102	337,313	86,091,010	2,327,596	47,508,895
2.deepsjeng	10	7,588	16,564	9,813	19,962	4,459,735	1,113,521	48,254,292
3.imagick	259	43,801	116,706	344,796	539,533	835,383,009	74,655,258	-
4.lbm	1	783	1,616	995	1,391	29,643	11,410	47,003
5.leela	21	9,103	22,884	48,138	70,610	8,702,553	2,280,728	212,128,751
6.mcf	3	2,320	5,090	2,685	3,965	550,673	29,345	1,050,586
7.nab	24	4,337	9,950	32,037	41,617	458,650	3,854,605	26,013,979
8.omnetpp	134	75,418	172,436	468,794	1,613,922	279,194,458	39,426,888	-
9.parest	427	26,701	66,984	365,846	739,060	82,710,266	210,190,800	-
10.perlbench	362	40,078	115,890	613,180	1,507,152	-	1,026,629,696	-
11.povray	170	18,510	46,332	351,415	769,078	54,670,321	274,590,820	-
12.x264	96	19,835	48,678	140,517	256,521	13,202,928	35,521,661	-
13.xz	33	3,472	8,102	31,267	41,616	434,838	710,239	44,237,649

Summary. By reducing wasted summary edges, `CFL-Skewed` outperforms a state-of-the-art CFL-reachability solver `CFL-RHS` and a state-of-the-art Datalog solver `Soufflé` in all of the three clients. For alias analysis, `CFL-Skewed` is 3.34× and 3.16× faster than `CFL-RHS` and `Soufflé`, respectively. For value-flow analysis, `CFL-Skewed` is 1.13× and 1.85× faster than `CFL-RHS` and `Soufflé`, respectively. For taint analysis, `CFL-Skewed` is 2.05× and 2.28× faster than `CFL-RHS` and `Soufflé`, respectively. Moreover, for alias analysis, `CFL-Skewed` consumes 60.05% and 61.45% less memory than `CFL-RHS` and `Soufflé`, respectively. For value-flow analysis, `CFL-Skewed` consumes 20.38% and 16.47% less memory than `CFL-RHS` and `Soufflé`, respectively. For taint analysis, `CFL-Skewed` consumes 63.06% and 56.56% less memory than `CFL-RHS` and `Soufflé`, respectively.

5.1 Experimental Setup

All experiments were conducted on a platform with an eight-core 2.60 GHz Intel Xeon CPU and a 128GB RAM memory, running Ubuntu 20.04.

Benchmarks and graph collection. We evaluate the client analyses on the SPEC CPU 2017 suite. Three programs (*i.e.*, `xalancbmk`, `gcc` and `blender`) failed to be linked by `wllvm` and thus are not included in our evaluation. Our measurements take the average of three runtimes for each experiment. We found that the runtimes varied little in practice, so this number of runs was sufficient to present an accurate and fair comparison. Table 1 presents the statistics of the benchmarking graphs used in our evaluation. In particular, alias analysis is conducted on program expression graphs (PEGs) [Zheng and Rugina 2008], and value-flow analysis and taint analysis are conducted on sparse value-flow graphs (SVFGs) [Sui et al. 2014]. To obtain the graphs, we compile each program into a bitcode file using `Clang-14.0.0`, linked via `wllvm`[2] for whole-program analysis. Then, we extract the corresponding graphs using an open-source tool SVF [Sui and Xue 2016]. Existing graph simplification techniques, including cycle elimination [Tarjan 1972], offline variable substitution [Rountev and Chandra 2000], and non-contributing parenthesis-edge elimination [Li et al. 2020] are used to preprocess the input graphs.

Implementation. We implemented our grammar transformer (Algorithm 2) in Python 3. Our grammar transformation takes less than one second to process the three grammars used in our evaluation. Therefore, its running time is negligible. We implemented `CFL-Skewed` (Algorithm

[2] https://github.com/travitch/whole-program-llvm.

$$
\begin{array}{lcl}
M & ::= & DV\ d \\
DV & ::= & \overline{d}\ V \\
V & ::= & \overline{A}\ V \mid V\ A \mid FV_i\ f_i \mid M \mid \varepsilon \\
FV_i & ::= & \overline{f_i}\ V \\
A & ::= & A\ A \mid a\ M \mid a \mid \varepsilon \\
\overline{A} & ::= & \overline{A}\ \overline{A} \mid M\ \overline{a} \mid \overline{a} \mid \varepsilon
\end{array}
$$

(a) Normalized CFG for alias analysis.

$$
\begin{array}{lcl}
M & ::= & DV\ d \\
DV & ::= & \overline{d}\ V \\
V & ::= & \overline{A}\ V \mid \overline{a}\ V \mid V\ A \mid V\ a \mid FV_i\ f_i \mid M \mid \varepsilon \\
FV_i & ::= & \overline{f_i}\ V \\
A & ::= & a\ M \\
\overline{A} & ::= & M\ \overline{a} \\
PN & = & \{DV, FV_i, M\}
\end{array}
$$

(b) Skewed CFG for alias analysis.

Fig. 4. CFGs for alias analysis, where V is the start symbol.

$$
\begin{array}{lcl}
A & ::= & A\ B \mid A\ a \mid a \mid \varepsilon \\
B & ::= & CA_i\ ret_i \\
CA_i & ::= & call_i\ A
\end{array}
$$

(a) Normalized CFG for value-flow analysis.

$$
\begin{array}{lcl}
A & ::= & A\ B \mid A\ a \mid a \mid \varepsilon \\
B & ::= & CA_i\ ret_i \\
CA_i & ::= & call_i\ A \\
PN & = & \{CA_i\}
\end{array}
$$

(b) Skewed CFG for value-flow analysis.

Fig. 5. CFGs for value-flow analysis, where A is the start symbol.

$$
\begin{array}{lcl}
S & ::= & P\ N \\
P & ::= & A\ P \mid ret_i\ P \mid \varepsilon \\
N & ::= & A\ N \mid call_i\ N \mid \varepsilon \\
A & ::= & A\ A \mid CA_i\ ret_i \mid a \mid \varepsilon \\
CA_i & ::= & call_i\ A
\end{array}
$$

(a) Normalized CFG for taint analysis.

$$
\begin{array}{lcl}
S & ::= & P \mid S\ call_i \mid S\ A \\
P & ::= & A\ P \mid ret_i\ P \mid \varepsilon \\
A & ::= & A\ A \mid CA_i\ ret_i \mid a \mid \varepsilon \\
CA_i & ::= & call_i\ A \\
PN & = & \{CA_i\}
\end{array}
$$

(b) Skewed CFG for taint analysis.

Fig. 6. CFGs for taint analysis, where S is the start symbol.

3) in C++ based on LLVM-14.0.0. Our implementation uses `LLVM SparseBitVector` to construct the adjacency-list-based data structures for storing summary edges. Our dynamic skewing is implemented on top of the artifact of the paper [Lei et al. 2022a], an improved version of CFL-reachability that helps reduce redundant work caused by transitive rules. For `CFL-RHS`, we use the code released in the artifact of the paper [Lei et al. 2022a] and run the experiments with the "`cfl -pocr`" option.[3] For `Soufflé`, we use the stable release `2.3`. `Soufflé` synthesizes optimized Datalog solvers in C++ based on provided input relations (converted from the input grammars). Because `Soufflé` supports generating C++ programs with OpenMP annotations for parallel execution, our evaluation uses an 8-thread version. We compare with Soufflé in this way to demonstrate that our single-threaded `CFL-Skewed` is roughly comparable with an 8-threaded parallel program that does the same computation. For these experiments, we set a time limit of 48 hours and a memory limit of 128 GB for each run.

5.2 Evaluated Grammars

Alias analysis grammar. We adopt the field-sensitive version of the context-free grammar given by Zheng and Rugina [2008]. Figure 4a presents the context-free grammar in a normalized form, which is also used in existing works [Lei et al. 2022b; Wang et al. 2017; Zhang et al. 2014]. Figure 4b is the skewed version of the original grammar. In Figure 4a, there are three terminals a, d, f_i, denoting assignment, pointer dereference, and the address of the i-th field, respectively. The start symbol of the grammar is V, denoting value aliasing. The grammar works on bidirected graphs, *i.e.*, for each edge $u \xrightarrow{X} v$ in the graph where X is a symbol of the grammar, there always exists an inverse edge $v \xrightarrow{\overline{X}} u$ in the graph. Comparing 4b with Figure 4a, we see that for the non-start symbols A

[3] https://github.com/kisslune/POCR#usage.

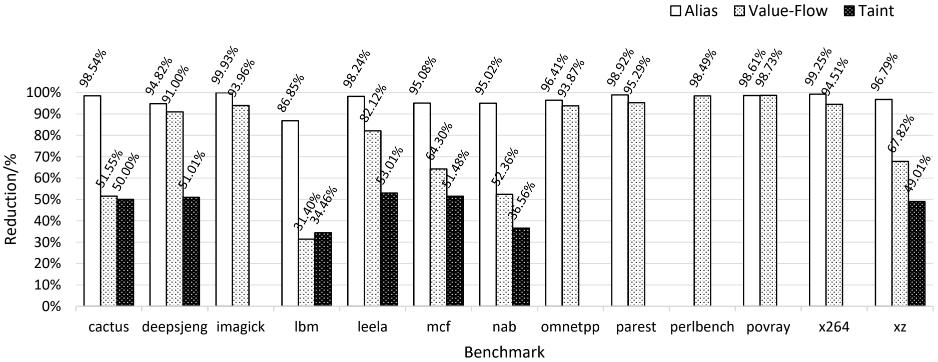


Fig. 7. Reduction rates of non-S summary edges by CFL-Skewed. The white bars denote alias analysis, the gray bars denote value-flow analysis and the black bars denote taint analysis.

and $\overline{A}$, our static skewing (Algorithm 2) eliminated the recursive parts of their productions. The removed recursions are promoted to the start symbol V, as the two new productions $V ::= \overline{a}\ V$ and $V ::= V\ a$. Our static skewing annotated three propagating nonterminals: DV, FV_i and M.

Value-flow analysis grammar. The context-sensitive value-flow analysis is used as a pre-analysis for memory leak detection formulated by Sui et al. [2014]. Figure 5a presents the context-free grammar in a normalized form. Figure 5b is the skewed version of the original grammar. In Figure 5a, there are three terminals a, $call_i$ and ret_i, denoting assignment, and calls and returns with a callsite index i, respectively. The start symbol is A, denoting value flow. Comparing Figure 5b with Figure 5a, we see that the static skewing did not change any production because there are no cyclic nonterminals (Definition 4.1) in the grammar. Our static skewing annotated a propagating nonterminal CA_i. Thus, the speedup is achieved by dynamic skewing, *i.e.*, the treatment of propagating edges.

Taint analysis grammar. We adopt the context-free grammar given in Section 5.1 of [Kodumal and Aiken 2004] for taint analysis. Figures 6a and 6b give the normalized grammar and the skewed grammar, respectively. In Figure 6a, there are three terminals a, $call_i$ and ret_i, denoting assignment, and calls and returns with a callsite index i, respectively. The start symbol is S, denoting interprocedural value-flow with unbalanced calls and returns. Comparing Figure 6b with Figure 6a, we see that the cyclic nonterminal N is removed in the skewed grammar, with all its recursive behavior promoted to the start symbol S, resulting in the two new productions $S ::= S\ call_i$ and $S ::= S\ A$. One propagating nonterminal CA_i is annotated.

5.3 RQ1: Reduction of Summary Edges

Figure 7 shows the reduction rates of *non-S* summary edges. In particular, non-S summary edges are the edges whose labels are not the start symbol of the CFG. Since a non-S edge does not represent any CFL-reachability solution, fewer non-S edges inserted during CFL-reachability solving is always better. According to the literature [Lei et al. 2022b], CFL-RHS does not reduce any inserted summary edges, so we use the data of CFL-RHS as the inserted summary edges of the standard RHS tabulation, as seen in the columns "#SumEdge by CFL-RHS" in Table 1. Specifically, the reduction rate of non-S edges of each program is calculated by $(\#\text{Edge}_{\text{RHS}} - \#\text{Edge}_{\text{Skewed}})/(\#\text{Edge}_{\text{RHS}} - \#\text{S})$, where $\#\text{Edge}_{\text{RHS}}$ and $\#\text{Edge}_{\text{Skewed}}$ denotes the number of summary edges inserted in RHS tabulation and CFL-Skewed, respectively; and #S denotes the number of S edges, which is consistent in both RHS and CFL-Skewed.

By observing Figure 7, we see that our CFL-Skewed drastically reduced the number of non-S edges. On average, for the programs successfully solved by both CFL-RHS and CFL-Skewed, our

CFL-Skewed reduced 96.54%, 78.11% and 46.50% non-S edges, respectively, for alias analysis, value-flow analysis and taint analysis. CFL-Skewed reduces non-S edges from two angles: (1) eliminating some recursive behaviors of non-S nonterminals, and (2) avoiding inserting propagating edges into the graph. By observing the grammars in Figures 4 and 6, alias and taint analyses can benefit from both angles. However, Figure 7 shows that the reduction rates of non-S edges of alias analysis are much larger than those of taint analysis. The reason behind this is that all non-S nonterminals of the alias grammar benefit from either of the two angles. As seen in Figure 4b, the recursive behaviors of A and $\overline{A}$ are totally removed, and DV, FV_i and M are annotated as propagating nonterminals. Therefore, CFL-Skewed is able to reduce almost all non-S summary edges. In the other case, for taint analysis, the recursive behaviors of both A and P are not eliminated, resulting in a large number of non-S edges inserted into the graph. The result of value-flow analysis also provides an interesting observation. As seen in Figure 5, the only difference of skewed grammar is annotating CA_i as a propagating nonterminal. However, Figure 7 shows that even without changing the grammar, CFL-Skewed still reduced the majority of non-S edges for value-flow analysis. This means that the propagating CA_i-edges takes a much larger proportion in non-S edges than B-edges.

5.4 RQ2: Runtime Speedups

Tables 2 presents the running time results, where we CFL-RHS, CFL-Skewed and Soufflé-8 to denote the results of CFL-RHS CFL-Skewed and 8-thread Soufflé, respectively. On average, for alias analysis, CFL-Skewed is 3.34× faster than CFL-RHS, and is 3.16× Soufflé-8. For value-flow analysis, CFL-Skewed is 1.13× faster than CFL-RHS, and is 1.85× faster than Soufflé-8. For taint analysis, CFL-Skewed is 2.05× faster than CFL-RHS, and is 2.28× faster than Soufflé-8.

We see that the speedups of CFL-Skewed in alias and taint analyses are much more significant than in value-flow analysis. The difference between value-flow analysis and the other two clients is that there is no removal of recursive behaviors of non-S nonterminals, and there are no dynamic propagating edges (Section 4.3) discovered. The experimental result indicates that reducing non-S recursive behaviors plays a more important role in accelerating CFL-reachability analysis. This is also supported by the results of the taint analysis. Considering Figure 7 and Table 2 together, we see that, although CFL-Skewed reduced a much smaller proportion of non-S edges in taint analysis than in value-flow analysis, the speedup of taint analysis is much more significant. A reasonable interpretation is that the reduced non-S recursive behaviors in taint analysis contribute to most of the wasted computations. This observation aligns with an existing work [Lei et al. 2022b], which claims that recursive behavior in CFL-reachability leads to tremendous redundancy.

5.5 RQ3: Reduction of Memory Consumption

Table 3 presents the memory consumption results. We measured memory consumption by recording the values VmRSS and VmSize at runtime reported by the process status file on our Ubuntu system ("/proc/self/status"). On average, for alias analysis, CFL-Skewed reduced 60.05% and 61.45% memory consumption for CFL-RHS and Soufflé-8, respectively. For value-flow analysis, CFL-Skewed reduced 20.38% and 16.47% memory consumption for CFL-RHS and Soufflé-8, respectively. For taint analysis, CFL-Skewed reduced 63.06% and 56.56% memory consumption for CFL-RHS and Soufflé-8, respectively.

Memory consumption reflects the number of summary edges inserted into the graph, *i.e.*, stored in memory. Comparing value-flow analysis with the other two clients, the reduced memory usage of value-flow analysis is much smaller than the other two clients. This implies that, based on the simpler grammar (Figure 5), the solving process of value-flow analysis involves much fewer non-S edges than other clients. Comparing taint analysis with alias analysis, although taint analysis reduced a much smaller proportion of non-S edges (Figure 7), it has memory reduction rates close

Table 2. Running time (in seconds) results on SPEC2017 benchmarks. The "-" mark indicates out-of-time.

Program	Alias Analysis			Value-Flow Analysis			Taint Analysis		
	CFL-RHS	CFL-Skewed	Soufflé-8	CFL-RHS	CFL-Skewed	Soufflé-8	CFL-RHS	CFL-Skewed	Soufflé-8
cactus	2,548.03	1,448.98	2,646.44	4.45	4.32	5.63	597.88	432.44	725.28
deepsjeng	10.05	5.90	6.19	1.48	1.26	2.26	1,518.75	651.87	1,499.59
imagick	30,514.60	8,312.89	23,125.35	245.84	229.15	373.22	-	-	-
lbm	0.03	0.02	0.29	0.01	0.01	0.02	0.06	0.04	0.05
leela	44.34	22.28	45.90	2.68	2.59	4.98	12,027.60	4,742.04	5,535.93
mcf	1.06	0.56	1.34	0.02	0.02	0.05	1.97	1.64	2.95
nab	1.24	0.48	0.77	3.41	3.41	6.52	262.92	74.84	483.47
omnetpp	146,438.00	21,188.20	33,860.40	186.22	135.82	293.98	-	-	-
parest	11,109.90	1,832.99	1,673.97	1,213.31	1,131.42	1,830.77	-	-	-
perlbench	-	-	-	6,514.56	5,956.88	7,787.65	-	-	-
povray	3,020.72	896.48	1,560.56	1,713.86	1,196.65	2,112.30	-	-	-
x264	127.54	22.42	34.73	67.79	62.68	91.24	-	-	-
xz	0.95	0.39	0.89	0.80	0.67	1.39	918.09	480.74	582.79

Table 3. Memory consumption (in MB) results on SPEC2017 benchmarks.

Program	Alias Analysis			Value-Flow Analysis			Taint Analysis		
	CFL-RHS	CFL-Skewed	Soufflé-8	CFL-RHS	CFL-Skewed	Soufflé-8	CFL-RHS	CFL-Skewed	Soufflé-8
cactus	2,085.70	1,070.47	2,691.38	266.44	244.06	291.86	2,340.73	1,081.74	1,108.75
deepsjeng	196.43	161.20	177.81	27.98	21.41	28.71	1,281.70	523.71	1,159.80
imagick	17,756.77	7,038.54	27,059.50	1,392.29	1,014.08	1,625.00	-	-	-
lbm	4.50	2.49	22.22	1.66	1.66	1.57	8.41	2.95	22.07
leela	262.09	86.07	295.05	78.65	63.24	77.74	3,775.13	1,307.32	4,998.11
mcf	26.07	13.77	22.21	3.46	2.95	2.54	36.64	18.46	45.54
nab	31.14	12.95	39.24	48.88	43.74	87.44	2,293.73	259.59	628.96
omnetpp	14,370.71	1,992.62	8,355.10	1,835.80	979.77	1,228.12	-	-	-
parest	3,705.35	736.63	2,581.16	2,928.81	2,522.41	3,083.87	-	-	-
perlbench	-	-	-	9,240.88	7,442.98	8,120.21	-	-	-
povray	2,289.31	508.88	1,720.26	4,132.14	2,980.36	3,808.22	-	-	-
x264	609.20	226.78	443.95	534.73	340.84	467.02	-	-	-
xz	27.05	8.28	21.94	38.23	31.83	29.56	1,112.15	445.50	1,113.14

to alias analysis where almost all non-S edges are reduced. This implies that non-S edges take a much larger proportion in taint analysis than in alias analysis. This is also reflected in the running time (Table 2), where the speedups of taint analysis are larger than others.

5.6 Discussions

Limitations. The grammar transformation does not work in the case where we cannot identify any cyclic nonterminals in the input grammar with the desired usage patterns. This is the case for the value-flow grammar shown in Figure 5. The grammar annotation alone can be fairly effective. However, as in the case of value-flow, all insertions of the nonterminal CA_i can be optimized out. In practice, there exist grammars that cannot be handled by both static and dynamic skewing. For those grammars, our grammar transformation can neither change the grammars nor produce the annotation set PN. In this case, the CFL-Skewed algorithm in Algorithm 3 is equivalent to the traditional CFL-reachability algorithm based on RHS tabulation. For more restrictive grammar, our approach may also be limited based on how skewed the input grammar is. For example, the well-known Dyck grammar[4] can only be skewed dynamically since there are no cyclic nonterminals other

[4] The Dyck language of k kinds of parentheses is defined by the grammar: $S ::= S\,S \mid (\,S\,) \mid \ldots \mid (_k\,S\,)_k \mid \varepsilon$.

than the start nonterminal S. In Algorithm 2, the static skewing algorithm, the start nonterminal S is part of the set of target nonterminals T which explicitly will not be modified by the transformation since the set of strings expressed by the start nonterminal cannot be changed in order to preserve the correctness of the output of CFL-Skewed. Similarly, a linear grammar can only be skewed dynamically since all its rules are linear by definition, and our grammar transformation only replaces and modifies nonlinear rules.

CFL-Skewed versus Soufflé. Soufflé is a Datalog solver, which is more general than CFL-reachability. Though Soufflé's partial evaluation can distinguish linear rules from non-linear ones based on extensional databases (EDBs) and intensional databases (IDBs), it does not have any knowledge about equivalent grammar forms. Our work introduces propagating nonterminals, a concept situated between EDB and IDB in Soufflé's terminology. Specifically, in static skewing, CFL-Skewed always tries to promote the recursive behavior of non-S nonterminals to S the start symbol to reduce the unnecessary rules and nonterminals and annotate propagating nonterminals—a notion that Soufflé does not have. In dynamic skewing, CFL-Skewed harnesses the propagating nonterminals to further reduce unnecessary summary edges. According to our experience, the performance of 8-thread Soufflé is significantly better than single-thread Soufflé when analyzing the benchmarks in our experiments. We chose 8 threads because the running time of Soufflé-8 is roughly similar to that of CFL-Skewed. The experimental result that our (single-thread) CFL-Skewed outperforms 8-thread Soufflé demonstrates that skewed tabulation outperforms the classic RHS tabulation, and it also implies that our CFL-Skewed can also significantly outperform single-thread Soufflé in solving the three clients.

Recursive productions and grammar rewriting. The work of POCR [Lei et al. 2022b] also studies recursive productions. It is interesting to note that the POCR only works when there is at least one "transitive" (*i.e.*, doubly recursive) production in the input context-free grammar. Another work [Shi et al. 2023] proposes PEARL, an approach that further targets these transitive productions by using batch-propagation of reachability information. In most cases, these require users to manually rewrite the grammar to expose such transitive productions. Moreover, they rely on the traditional RHS tabulation [Reps et al. 1995] and make memory trade-offs to achieve speedup. On the contrary, the grammar transformation (Algorithm 2) in our work can *automatic* rewrite the input grammar. Our approach directly improves the tabulation process and does not increase memory consumption. Moreover, our technique eliminates recursive productions by moving the recursion to the start symbol, as many as possible. However, it may not eliminate all transitive productions. In this case, skewed tabulation can be integrated into POCR and PEARL, which can process those transitive productions more effectively. In fact, our implementation is built on top of POCR, which demonstrates that the benefits of our approach can be observed in conjunction with the benefits of works that implement specialized optimizations for transitive rules.

6 Related Work

CFL-reachability is central to program analysis as many program properties can be specified as context-free grammars. The traditional CFL-reachability algorithm exhibits a cubic time complexity [Reps 1998]. Chaudhuri [2008] proposes a subcubic CFL-reachability algorithm, which improves the cubic complexity by a logarithmic factor. Chatterjee et al. [2018] establish a cubic conditional lower bound for CFL-reachability. Asymptotically fast algorithms exist only for special cases. For bidirected Dyck-reachability, Chatterjee et al. [2018] give an algorithm that runs in time $O(m + n \cdot \alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function. When restricted to graphs with bounded treewidth, Chatterjee et al. [2019] gave faster algorithms for solving demand-driven queries in the presence of graph changes. When restricted to directed acyclic graphs, Yannakakis [1990] noted that CFL-reachability could be solved in $O(n^{\omega})$ time. McAllester [2002] established a framework for determining the time complexity of static analysis. Kodumal and Aiken [2004] describe a specialized set constraint reduction for Dyck-reachability. Pavlogiannis [2022] surveys the recent algorithmic developments for improving CFL- and Dyck-reachability. Skewed tabulation does not improve the asymptotic complexity of CFL-reachability. Instead, our results demonstrate that a better grammar form can make CFL-reachability more scalable in practice.

In the seminal work, Yannakakis [1990] discusses an $O(mn)$-time algorithm for linear context-free language reachability problems. However, most existing work in program analysis follows the standard grammar normalization for CFL-reachability[Chaudhuri 2008; Melski and Reps 2000; Milanova 2020; Rehof and Fähndrich 2001; Reps 1998; Wang et al. 2017; Zheng and Rugina 2008; Zuo et al. 2021]. Hollingum and Scholz [2015] propose a general method for CFL-reachability based on a partial evaluation of a single input grammar, which does not require the normal form. Lange and Leiß [2009] discuss a different normal form for teaching the CYK parsing algorithm. Beyond CFL-reachability, Tang et al. [2015] require a similar normal form for tree-adjoining-language reachability. Zhang and Su [2017] propose linear-conjunctive language reachability where the input grammar is in the linear form. In general, linear context-free language is not expressive enough to be used in practical program analyses. Milanova [2020] observes that some linear constraints can simplify the underlying CFL-reachability formulation. Context-free path querying [Hellings 2014] in Database is similar to CFL-reachability, which also requires that the input grammar is in Chomsky normal form. A recent work [Lei et al. 2022b] improves the performance by modifying the online solver, which sacrifices memory consumption for higher speed. Another recent work [Shi et al. 2023] further improves performance by modifying the online solver using batch propagation of reachability information and transitivity-aware subgraphs. Both of these works [Lei et al. 2022b; Shi et al. 2023] present runtime optimizations that target and, therefore, require transitive rules in the input grammar. Our skewed tabulation can be seen as a more powerful grammar transformation combined with a specially designed solver, which works both offline and online and improves the performance of CFL-reachability for both time and memory consumption. Existing parser generators (*e.g.*, LL and LALR generators) only work for deterministic context-free languages. By contrast, our static skewing technique for general context-free grammars. It is an interesting future direction to further investigate the connections between parser generators and the grammar transformation for skewed tabulation.

Datalog is a well-known framework for formulating program-analysis problems [Bravenboer and Smaragdakis 2009; Lu et al. 2013; Scholz et al. 2016; Whaley et al. 2005]. CFL-reachability problem instances can be directly encoded as a Datalog specification, like for the state-of-the-art Datalog solver `Soufflé` [Scholz et al. 2016]. Problem instances can leverage advanced Datalog evaluation strategies such as semi-naïve evaluation [Ullman 1989], tabulation [Warren 1992], and magic sets [Bancilhon et al. 1986; Beeri and Ramakrishnan 1987]. Our work focuses on improving the tabulation [Melski and Reps 2000; Naeem et al. 2010; Reps 1998] for CFL-reachability, which does not rely on specialized evaluation strategies.

7 Conclusion

This paper has presented skewed tabulation for CFL-reachability. We skew the parse trees statically and dynamically by transforming the input grammar and adjusting the CFL-reachability algorithm. The key idea is to use grammar structure to make the CFL-reachability algorithm reuse summary edges. We prove that our approach can only reduce the number of summaries compared to the traditional CFL-reachability via RHS tabulation. Our experimental results show that this approach achieves significant speedups, reduces memory overhead, and reduces the number of non S-edges by a significant margin in practice.

Acknowledgments

The authors thank the anonymous reviewers for their feedback on earlier drafts of this paper. The work described in this paper was supported, in part, by Australian Research Grants DP210101348 and FT220100391; and by a generous Aspire Gift Grant from Google; and by the United States National Science Foundation (NSF) under grants No. 2114627 and No. 2237440; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

Data-Availability Statement

Materials for our evaluation are publicly available [Lei et al. 2024] and can be used to reproduce the data of our experiment.

References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*. 1–15.

Catriel Beeri and Raghu Ramakrishnan. 1987. On the Power of Magic. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems(PODS)*. 269–284.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*. 243–262.

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck reachability for data-dependence and alias analysis. *Proc. ACM Program. Lang.* 2, POPL (2018), 30:1–30:30.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Prateesh Goyal, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2019. Faster Algorithms for Dynamic Algebraic Queries in Basic RSMs with Constant Treewidth. *ACM Trans. Program. Lang. Syst.* 41, 4 (2019), 23:1–23:46.

Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 159–169.

Edmund M Clarke, Masahiro Fujita, Sreeranga P Rajan, T Reps, Subash Shankar, and Tim Teitelbaum. 1999. Program slicing of hardware description languages. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 298–313.

Sheila A. Greibach. 1966. The Unsolvability of the Recognition of Linear Context-Free Languages. *J. ACM* 13, 4 (1966). https://doi.org/10.1145/321356.321365

Jelle Hellings. 2014. Conjunctive Context-Free Path Queries. In *Proc. 17th International Conference on Database Theory (ICDT), Greece, March 24-28, 2014*. 119–130.

Nicholas Hollingum and Bernhard Scholz. 2015. Towards a Scalable Framework for Context-Free Language Reachability. In *Compiler Construction - 24th International Conference (CC 2015)*. 193–211.

John Kodumal and Alexander Aiken. 2004. The set constraint/CFL reachability connection in practice. In *PLDI*. 207–218.

Martin Lange and Hans Leiß. 2009. To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm. *Informatica Didact.* 8 (2009).

Yuxiang Lei, Camille Bossut, Yulei Sui, and Qirun Zhang. 2024. *Artifact of "Context-Free Language Reachability via Skewed Tabulation"*. Zenodo. https://doi.org/10.5281/zenodo.10892936

Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022a. *Artifact of "Taming Transitive Redundancy for Context-Free Language Reachability"*. Zenodo. https://doi.org/10.5281/zenodo.7066401

Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022b. Taming transitive redundancy for context-free language reachability. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1556–1582. https://doi.org/10.1145/3563343

Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast graph simplification for interleaved Dyck-reachability. In *PLDI '20: 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3385412.3386021

Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7791)*. Springer, 61–81.

David A. McAllester. 2002. On the complexity analysis of static analyses. *J. ACM* 49, 4 (2002), 512–537.

David Melski and Thomas W. Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theor. Comput. Sci.* 248, 1-2 (2000), 29–98.

Ana L. Milanova. 2020. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 178:1–178:29.

Nomair A Naeem and Ondrej Lhoták. 2008. Typestate-like analysis of multiple interacting objects. *ACM Sigplan Notices* 43, 10 (2008), 347–366. https://doi.org/10.1145/1449764.1449792

Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm. In *Compiler Construction, 19th International Conference, CC 2010*. 124–144.

Andreas Pavlogiannis. 2022. CFL/Dyck Reachability: An Algorithmic Perspective. *ACM SIGLOG News* 9, 4 (2022), 5–25.

Jakob Rehof and Manuel Fähndrich. 2001. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 54–66.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61. https://doi.org/10.1145/199448.199462

Thomas W. Reps. 1998. Program analysis via graph reachability. *Inf. Softw. Technol.* 40, 11-12 (1998), 701–726.

Atanas Rountev and Satish Chandra. 2000. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 47–56.

Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*. 196–206.

C. Shi, H. Li, Y. Sui, J. Lu, L. Li, and J. Xue. 2023. Two Birds with One Stone: Multi-Derivation for Fast Context-Free Language Reachability Analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 624–636. https://doi.org/10.1109/ASE56229.2023.00118

Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. ACM, 265–266.

Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. https://doi.org/10.1109/TSE.2014.2302311

Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. 2015. Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015,*. 83–95.

Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

Jeffrey D. Ullman. 1989. Bottom-Up Beats Top-Down for Datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 29-31, 1989, Philadelphia, Pennsylvania, USA*. ACM Press, 140–149.

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 389–404.

David Scott Warren. 1992. Memoing for Logic Programs. *Commun. ACM* 35, 3 (1992), 93–111.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Programming Languages and Systems, Third Asian Symposium (APLAS)*. 97–118.

Mihalis Yannakakis. 1990. Graph-Theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*. 230–242.

Daniel H. Younger. 1967. Recognition and Parsing of Context-Free Languages in Time n^3. *Inf. Control.* 10, 2 (1967), 189–208.

Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. 344–358.

Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient subcubic alias analysis for C. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*. 829–845.

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 197–208.

Zhiqiang Zuo, Kai Wang, Aftab Hussain, Ardalan Amiri Sani, Yiyu Zhang, Shenming Lu, Wensheng Dou, Linzhang Wang, Xuandong Li, Chenxi Wang, et al. 2021. Systemizing Interprocedural Static Analysis of Large-scale Systems Code with Graspan. *ACM Transactions on Computer Systems (TOCS)* 38, 1-2 (2021), 1–39.

Received 2023-11-16; accepted 2024-03-31