UNIVERSITY
OF TECHNOLOGY
SYDNEY

# Improving the Efficiency of Graph-Based Static Analysis

**by Yuxiang Lei**

Thesis submitted in fulfilment of the requirements for the degree of

**Doctor of Philosophy (Software Engineering)**

under the supervision of Dr Yulei Sui

University of Technology Sydney
Faculty of Engineering and Information Technology

12/2022

# Improving the Efficiency of Graph-Based Static Analysis

*A thesis submitted in fulfilment of the requirements*
*for the degree of*

Doctor of Philosophy
*in*
Software Engineering

*by*

**Yuxiang Lei**

*to*

School of Computer Science
Faculty of Engineering and Information Technology
University of Technology Sydney
NSW - 2007, Australia

December 2022

# Certificate of Original Authorship

I, *Yuxiang Lei* declare that this thesis is submitted in fulfilment of the requirements for the award of *Doctor of Philosophy*, in the *School of Computer Science, Faculty of Engineering and Information Technology* at the University of Technology Sydney. This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis. This document has not been submitted for qualifications at any other academic institution. This research is supported by the Australian Government Research Training Program.

Production Note:
Signature removed prior to publication.

SIGNATURE: ______________________________

DATE: 30th December, 2022

PLACE: Sydney, Australia

# Abstract

Generally speaking, static program analysis is to figure out whether a program can do whatever the program designers want it to do without actually executing the program. From different perspectives, static analysis studies various properties of a program, including correctness, robustness, liveness, safety and efficiency. As contemporary programs usually tend to be large and complex, developing efficient automatic program analysis techniques while maintaining soundness and precision is desirable.

Transitivity extensively manifests in the executions of programs, where controls and data are propagated and processed via flows. Taking data flow as an example, two assignment instructions `a = b` and `b = c` imply a result `a = c`, which means that the value of `c` flows into `a` via `b`. Static analyses inevitably include the analysis of flows, which is usually conducted in the form of solving dynamic transitive closure on the abstract graph of programs. The inefficiency arises from not only the high complexity of transitive closure itself but also the high redundancies of the analysis techniques.

This dissertation studies improving the efficiency of dynamic transitive closures on graph-based static analysis. Specifically, it focuses on improving the efficiencies of three popular static analysis frameworks: context-free language reachability, recursive state machine reachability and set constraint analysis. All the three frameworks are under the scope of graph analytics. Namely, all the analyses operate on an abstract graph of the target program.

In this dissertation, the methodologies focus more on eliminating redundancy rather than theoretically lowering complexity.

For transitive redundancy that arises from the massive re-computations and re-derivations during the analysis procedures, we design a hybrid data structure and apply it to context-free language reachability. Based on this, we propose a partially ordered algorithm, which significantly improves the scalability of context-free language reachability analysis by eliminating most re-computations and re-derivations.

For trivial nodes and edges in the abstract graphs of programs, which cause extra computations in the analysis procedure, we develop a graph folding technique to remove redundant nodes and edges in the preprocessing stage and apply it to recursive state machine reachability. The graph folding technique extends the applicability of some existing techniques from particular scenarios to general analysis as long as the recursive state machine is given and is well compatible with other preprocessing techniques.

For set constraint analyses where the graph contains weighted edges, we discover the derivation equivalence property and propose an approach that avoids the infinite iterations caused by weighted cycles during constraint solving. The derivation equivalence based constraint solving is highly efficient while maintaining the precision.

Notably, the three dynamic transitive closure based program analysis frameworks, i.e., context-free language reachability, recursive state machine reachability and set constraint solving, are generally recognized as interconvertible. Accordingly, the three techniques proposed in this dissertation are mutually compatible. The empirical study on real-world clients, including value-flow analysis, alias analysis and pointer analysis, shows that our approaches are practical and effective.

# Acknowledgments

First, I would like to express my sincere gratitude to my supervisor, Prof. Yulei Sui, for presenting me with a chance to study at the University of Technology Sydney as a sponsored student. He told me how to improve my communication, writing and programming skills. His wisdom, dedication, scrupulousness and patience guided me throughout my PhD candidature and continuously illuminated my future research.

Besides, I would like to thank Prof. Qirun Zhang and Prof. Shinhwei Tan. Their deep insights and sweet suggestions inspired me when I got lost in my research topics.

Moreover, I would like to thank my colleagues Dr. Pei Xu, Dr. Guanqin Zhang and Dr. Xiao Cheng, who helped me with my school issues when I was on leave of absence and not in Sydney due to COVID-19. I would also like to thank Dr. Chandranath Adak, who provided this beautiful thesis template.

My greatest thankfulness is owed to my parents. They brought me up, sharing knowledge with me and supporting me to pursue a higher education degree from material and spiritual tiers. Without them, I would never have had a chance to study and meet friendly people at UTS. When I was frustrated, their encouragements were always the most powerful motivation to me.

# TABLE OF CONTENTS

# LIST OF PUBLICATIONS

The following is a list of publications that are included in this thesis.

**Chapter 3:**

**Yuxiang Lei**, Yulei Sui, Shuo Ding, and Qirun Zhang. "Taming transitive redundancy for context-free language reachability." *Proceedings of the ACM on Programming Languages 6. OOPSLA2 (2022): 1556-1582.* (SIPLAN 2022 Distinguished Artifact Award)

**Chapter 4:**

**Yuxiang Lei**, Yulei Sui, Qirun Zhang, and Shinhwei Tan. "Recursive State Machine Guided Graph Folding for Context-Free Language Reachability." *Proceedings of the ACM on Programming Languages 7. PLDI (2023).*

**Chapter 5:**

**Yuxiang Lei** and Yulei Sui. "Fast and Precise Handling of Positive Weight Cycles for Field-sensitive Pointer Analysis." *Static Analysis: 26th International Symposium, SAS (2019), Proceedings 26 (pp. 27-47).* (Radhia Cousot Young Researcher Best Paper Award)

# LIST OF FIGURES

**FIGURE** **Page**

# List of Tables

**Table** **Page**

# CHAPTER 1

# INTRODUCTION

This dissertation, from the perspective of static analysis, provides three techniques to optimize the dynamic transitive closure based program analysis. In this chapter, we give a brief introduction about the background, research topics, contribution and organization of this dissertation.

## 1.1 Background

Generally speaking, program analysis is to figure out whether a program can do whatever the program designers want it to do. From different perspectives and in different ways, program analysis studies various properties of a program, including correctness, robustness, liveness, safety and efficiency.

Static analysis, relative to dynamic analysis, operates without actually executing a program. When developing real-world projects, the nature of static analysis allows developers to use it before deployment and permits weaknesses to be found earlier in the development life cycle. By detecting defects at an early stage, the cost for rework can be reduced and more efficient development can be achieved. In recent years, a large variety of (commercial/educational) static analysis tools were proposed and performing important roles in better program understanding, vulnerability detection, compiler optimization, etc. For example, for C/C++ analysis, there are Clang [83], CPAchecker [19], UTaipan [35], SVF [124], etc; for Java analysis, there are PMD [98], Checkstyle [28], Findbugs [39], etc.

Static analysis is usually performed based on the user's requirements or design of code. To satisfy different requirements, static analysis uses formal methods to carry out semantics analysis, syntax analysis, control flow analysis, data flow analysis, etc, among which a large proportion aims to determine specific relations among the elements (e.g., variables/functions/classes) of a program and is carried out in the form of solving dynamic transitive closures [10, 27, 105, 110, 143].

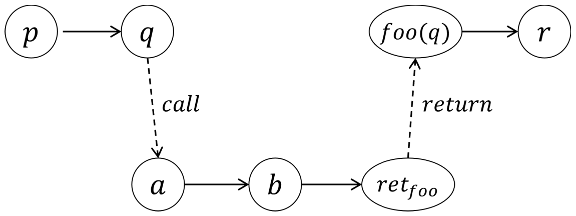
Dynamic transitive closure based program analysis usually operates on an abstract graph, which is obtained from the source code, intermediate representation (after pre-compilation) or bitcode (after compilation) by particular tools [19, 52, 73, 82, 124]. In the abstract graph, the nodes represent the program elements, like functions in a call graph [47, 48], variables in a value-flow graph [61, 126] and pointers in an Andersen's constraint graph [10, 58], and the edges describe the relations among the elements, which are usually formal representations of the program instructions.

Transitivity is ubiquitous in static analysis. Consider a simple and common example, which studies value flows among variables: given two assignment instructions `b = a` and `c = b`, the transitivity of value flows indicates that `c = a`, meaning that the value of the variable `a` flows into the variable `c` via the variable `b` through two assignment instructions. In ordinary directed (or undirected) graphs where edges are not classified into different types, transitive closure can be solved efficiently using ordered graph traversal, e.g., depths-first traversal or width-first traversal.

The difficulty is that, in a real-world program, there are multiple types of relations manifesting in different kinds of edges in the abstract graph, among which some are not explicit until others are solved. The computation in such scenarios is terms as "dynamic

```
int foo(a) {
    int b = a;
    return b;
}
int p,q,r;
q = p;
r = foo(q);
```

(a) Code fragment.



(b) Abstract graph.

**Figure 1.1.** A example of value-flow analysis, where the value-flow from `p` to `r` is unknown unless the value-flow relations of the variables in the function `foo` is determined.

transitive closure" problem [10, 27, 105, 110, 143]. The naive graph-traversal approaches can hardly solve the dynamic transitive closure of the abstract graphs of programs. Consider the example in Figure 1.1. Whether the value of `q` flows to `p` is unknown unless we make clear the value-flow relations of the variables in the function `foo`.

Formal methods for solving dynamic transitive closure based program analysis includes formal languages [21, 46, 69, 141, 146], automata [6–8, 88, 115], constraint analysis [10, 49, 122], etc. This dissertation involves three fundamental frameworks of them, i.e., context-free language reachability [141], recursive state machine [6] and set constraint analysis [10].

### 1.1.1 Context-Free Language Reachability (CFL-Reachability)

A context-free language (CFL) [11, 71] is a set of strings (words) which are comprised of symbols (letters) belonging to an alphabet and are accepted by a context-free grammar (CFG). The CFG contains a series of production rules describing how a symbol of the alphabet generates a string comprised of multiple symbols. Judging whether a string is accepted by the CFG is to determine whether the string can be derived from a start symbol (a particular symbol of the alphabet) of the CFG via one or more production rules.

Given an edge-labeled directed graph, a context-free language reachability (CFL-reachability) problem [141] determines whether specific source-sink pairs in the graph are connected by a reachable path, i.e., a path whose edge labels form (in sequence) a string accepted by the given context-free grammar. CFL-reachability is solved using a dynamic programming algorithm [86], which exhaustively computes and records all reachability relations of nodes in the input graph via a scheme called summarization [105, 141], which summarizes new edges from existing ones and adds them into the graph to represent the detected reachability relations. The mechanism of the algorithm can be briefly described as following steps: (1) keep a worklist to hold all the unprocessed edges; (2) iteratively select two adjacent edges, of which at least one is removed from the worklist, to generate a new edge according to the production rules; (3) add the generated edge to the graph and the worklist; (4) repeat steps (2)–(3) until the worklist is empty, which means that there is not any new edge that can be added to the graph.

CFL-reachability extends ordinary graph reachability from unlabeled graphs to edge-labeled graphs and allows reachability relations of nodes to be determined by particular rules that describes semantics of specific applications. It has been widely applied to a large variety of static analyses including shape analysis [106], polymorphic flow analysis [104], data flow analysis [100, 105, 141, 146], typestate analysis [89, 132] and pointer analysis [117, 147, 149].

### 1.1.2 Recursive Static Machine (RSM)

In real applications, context-free languages are also expressed using automata [6, 11]. As an alternative form of context-free languages, recursive state machines (RSMs) [6, 8, 27, 149] are also widely used in program analysis. Recursive state machines enhance the power of ordinary finite state machines by allowing them to recursively invoke each other. An RSM is comprised of a series of component finite state machines with each component containing one or more entrances and exits to accept the invocation from other components. In an RSM constructed upon an alphabet, a state is reached from another via a symbol of the alphabet according to a transition function. The state transitions can either entering or exiting boxes, which naturally model the calls and returns of interprocedural control flows.

By formulating CFL-reachability into RSM-reachability, special algorithms for particular clients are available. For example, based on the equivalent RSM of the context-free grammar of alias analysis, Zheng and Rugina [149] proposed an algorithm for demand-driven alias analysis for C. Zhang et al. [147] fine tuned the RSM and proposed another algorithm to efficiently solve all-pair alias analysis for C. A large variety of CFL-reachability problems are solved with the auxiliary of recursive state machines [84, 105, 114, 116, 117, 132].

### 1.1.3 Set Constraint Analysis

A set constraint problem [49] is comprised of a graph and a series of rules describing how the constraint is solved. In the graph, each node possesses a set holding the elements subordinate to the node, and each edge denotes a set constraint relation between two vertices, including union, intersection, complementation, equivalence, etc.

Solving set constraint analysis is to iteratively propagate the elements in the set of nodes according to the descriptive rules until all the constraints in the graph are satisfied. In general, set constraint is solved using a dynamic programming algorithm [10, 51] which maintains a worklist holding the nodes whose set elements need to be propagated to the set of other nodes along the edges, which is caused by unsatisfied constraints. The algorithm terminates when the worklist is empty, which means that all the constraint relations among the sets (of nodes) are satisfied.

The most popular application of set constraint analysis in static analysis is Andersen's pointer analysis [10, 51, 97, 117], where each node represents a variable (a pointer or a memory object), and each edge is a constraint converted from instructions involving pointer operations, such as address taken, reference/dereference, field access [96], etc.

Each node is assigned a point-to set, holding memory objects that may be pointed to by the variable represented by the node during the execution of the program. By solving the constraints, the result will reveal the potential pointee for all the pointers.

The three frameworks were claimed and proved to have equivalent expressiveness in the literature [6, 86]. They can be used alternatively to solve the same problem from different angles.

## 1.2 Research Topics

The efficiency of program analysis has been attracting researchers' interest. In particular, for dynamic transitive closure based program analysis, the large (cubic) complexity and redundancy limit its scalability for analyzing large programs. Researchers have tried to scale dynamic transitive closure based program analysis from two perspectives: lowing complexity and reducing redundancy.

Due to the difficulty in implementation, existing subcubic dynamic transitive closure algorithms [22, 75, 136, 138] were not widely applied in practice. Relatively, a large variety of techniques were proposed to reduce redundancy in the preprocessing stage [41, 51, 52, 57, 80, 111] or in the solving procedure [1, 65, 68, 90, 95, 133, 145, 147, 150] for real-world program analysis clients.

This dissertation, from a more practical perspective, studies reducing redundancy for the three interchangeable fundamental frameworks of dynamic transitive closure based program analysis from three aspects: (1) reducing redundant computation on the fly, (2) simplifying the input graph in the preprocessing stage, and (3) optimizing the solving of weighted constraints.

### 1.2.1 Eliminating Transitive Redundancy On-The-Fly

Our empirical study (Section 3.4) shows that, in the solving procedure of context-free language reachability (CFL-reachability), redundant computation is majorly caused by *transitive relations*. Transitive relations can manifest in various forms, whereas the essence is that new transitive relations can be derived from existing ones via concatenation. Specifically, with $A$ denoting a type of transitive relations, for any three nodes $v_i$, $v_j$ and $v_k$ in a graph, $v_i \xrightarrow{A} v_j \xrightarrow{A} v_k$ indicates $v_i \xrightarrow{A} v_k$. Contemporary program analyses extensively deal with transitive relations. For example, control/data/value flows [105] are transitive. Besides, the widely applicable Dyck-relations [25, 114, 144, 145] are also transitive.

Let us consider the example in Figure 1.2, for a path $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4$, there are two ways to derive the edge $v_1 \xrightarrow{A} v_4$, whereas $v_1 \xrightarrow{A} v_4$ only needs to be derived once to make explicit the $A$-relation from $v_1$ to $v_4$. Intuitively, we can summarize $v_1 \xrightarrow{A} v_4$ from the head, i.e., $v_1$, to the tail, i.e., $v_4$, so as to avoid the repeated derivation of $v_1 \xrightarrow{A} v_4$ caused by Figure 1.2(b). However, the standard, also most widely used, CFL-reachability algorithm [86] does not have the ability to find a proper order to derive edges and avoid repeated computations. Conversely, all ways to derive an edge will be triggered in most cases, leading to the so-called *transitive redundancy*. And such transitive redundancy is further amplified for longer paths with more transitive edges derived during the dynamic CFL-reachability solving.

Unfortunately, transitive redundancy cannot be eliminated by simply contracting transitive edge during solving because arbitrary contraction can lead to incorrect/imprecise results, especially in the dynamically changing graph of CFL-reachability. For example, given three edges $v_i \xrightarrow{A} v_j$, $v_i \xrightarrow{A} v_k$ and $v_l \xrightarrow{A} v_j$, merging $v_i$ and $v_j$ causes $v_l$ to reach $v_k$, which is incorrect.

The challenge for reducing transitive redundancy of whole-program (a.k.a., all-pair) CFL-reachability analysis is to find the best possible edge derivation order. Obviously,

$$\dfrac{\dfrac{v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3}{v_1 \xrightarrow{A} v_3} \qquad v_3 \xrightarrow{A} v_4}{v_1 \xrightarrow{A} v_4}$$

**(a)**

$$\dfrac{v_1 \xrightarrow{A} v_2 \qquad \dfrac{v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4}{v_2 \xrightarrow{A} v_4}}{v_1 \xrightarrow{A} v_4}$$

**(b)**

**Figure 1.2.** Transitive redundancy caused by two ways to derive $v_1 \xrightarrow{A} v_4$ from $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4$.

getting the best derivation order to avoid redundancy on a simple path (e.g., Figure 1.2) is trivial. However, a node can reside in multiple paths and/or in cycles on a dynamically updated graph. Identifying the derivation order in such dynamic graphs is non-trivial. Intuitively, retrieving the topological order by traversing the graph can help determine the order, so as to reduce redundancy. The problem is that, to maintain the precision and correctness in the presence of resolving dynamic transitive closure, the analysis requires repeatedly computing the topological order, which significantly increases the overheads and defeats the purpose of improving the scalability of CFL-reachability.

In this dissertation, we address this challenge by introducing a hybrid graph representation combining spanning trees and adjacency lists, together with a fast yet effective dynamic construction algorithm, to on-the-fly infer the derivation order during CFL-reachability solving. The acyclic property of spanning trees makes the traversals for determining derivation orders efficient. Based on this representation, we propose a partially ordered CFL-reachability algorithm POCR, which quickly solves all-pairs reachability analysis by reducing the transitive redundancy. Compared to the standard algorithm, POCR computes the same solution and is much more efficient. Our empirical results show that POCR is over 20× faster than the standard algorithm in real-world clients which need to solve transitive relations.

### 1.2.2 Simplifying the Input Graph in the Proprocessing Stage

Besides the solving procedure, the input graph of CFL-reachability also contains much redundant information. Considering that the complexity of CFL-reachability analysis is cubic with respect of the number of nodes in the input graph [86, 141], a smaller and cleaner input graph is important to improve the efficiency.

In the literature [38, 51, 95, 97, 126], the technique of cycle elimination has been widely applied to simplify the graph of particular clients. In such clients, nodes in cycle comprised of particular types of edges can be easily identified as equivalent and hence be merged to reduce the computation for the further solving procedure. However, equivalent nodes do not necessarily form a cycle. In fact, there are also a large amount of equivalent nodes not residing in the same cycle [52, 111]. Moreover, from the perspective of CFL-reachability, simplifying the input graph is not restricted to merge equivalent nodes [80]. A graph simplification is correct as long as the simplified graph and the original graph have identical/equivalent CFL-reachability solutions.

In this dissertation, we propose a preprocessing technique called *graph folding*, which does not require the merged items to be equivalent. Different from cycle elimination, we try to simplify the input graph by *folding* pairs of adjacent nodes, i.e., merging the two nodes and removing all the edges joining them. The foldability of the graphs of CFL-reachability originates from the existence of trivial edges, i.e., edges that do not contribute to the CFL-reachability solution. If two nodes are joined by trivial edges, they can be folded. Considering that the graphs of CFL-reachability are multigraphs, where there can be multiple edges between two nodes, our graph folding technique surpasses existing edge contraction technique [111] as it avoids introducing new self-cycles to the multigraphs.

The challenge lies in identifying foldable node pairs. Essentially, after running a whole-program CFL-reachability analysis, the foldable node pairs are naturally explicit. However, this obviously defeats the purpose of improving scalability. We address this challenge using an alternative form of CFL called recursive state machine (RSM). By formulating CFL-reachability into RSM-reachability, the correspondences between paths on the graph and state transition in the RSM reveals the essence of foldable node pairs. In particular, we consider a node pair $(x, y)$ as foldable if the corresponding transition chains of each path are equivalent in the graphs before and after folding $(x, y)$. By exploiting the dependency of global state transitions on local ones, we propose a graph folding principle for deterministic RSMs , which is able to determine whether two nodes are foldable by examining only their incoming and outgoing edges.

On top of the graph folding principle, we establish a graph folding algorithm GF, which has a linear time complexity with respect to the number of nodes in the input graph. Working in the preprocessing stage, GF is generally applicable to CFL-reachability problems whose CFLs can be formulate into deterministic RSMs, and is especially effective when the size of the graph is far larger than the size of the RSM. By reducing the input graph size, graph folding significantly improves the scalability of CFL-reachability for both time and space.

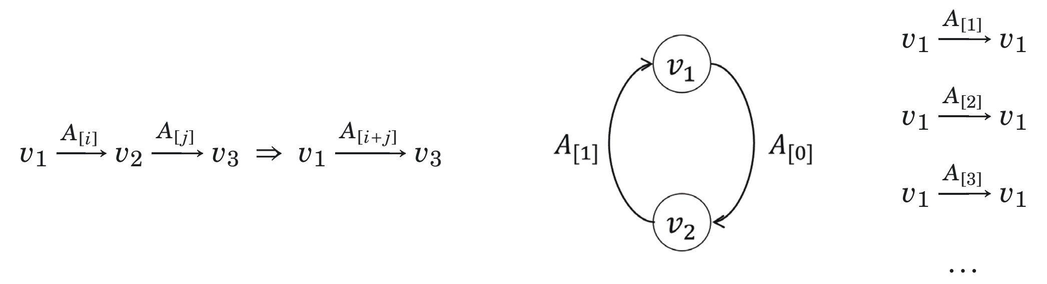
### 1.2.3 Precise and Efficient Handling of Positive Weight Cycles

In dynamic transitive closure analysis, weighted constraints are usually involved with offset arithmetic [96, 112, 135, 148]. Such offset usually manifests in weighted relations (e.g., in Datalog [112, 135]) or weighted set elements (e.g., in set constraint problems [15, 96]).

Cycles containing weighted edges often lead to severe extra computation. A typical instance is called *positive weight cycle*, which causes infinite loops and limits the precision and scalability of program analysis [76, 96]. Figures 1.3 and 1.4 illustrates positive weight cycles in CFL-reachability [1] and set constraint analysis [2], where we use $X_{[i]}$ to denote a

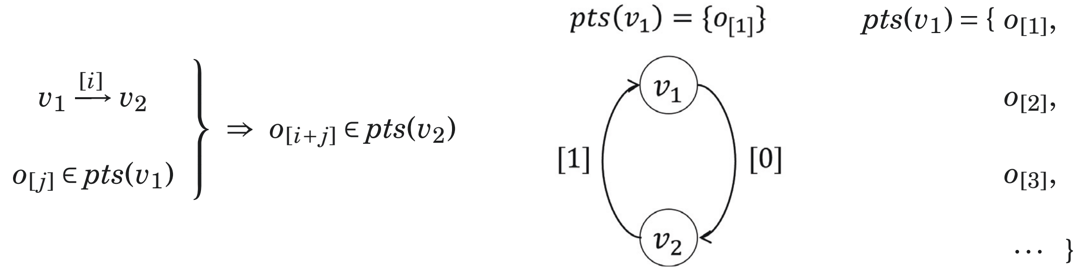
[1] Section 3.1 details the definitions of CFL-reachability

[2] Section 5.1 details the definitions of set constraint analysis


$v_1 \xrightarrow{A_{[i]}} v_2 \xrightarrow{A_{[j]}} v_3 \Rightarrow v_1 \xrightarrow{A_{[i+j]}} v_3$
$v_1$
$v_2$
$A_{[1]}$
$A_{[0]}$
$v_1 \xrightarrow{A_{[1]}} v_1$
$v_1 \xrightarrow{A_{[2]}} v_1$
$v_1 \xrightarrow{A_{[3]}} v_1$
...

**(a)** Rule for solving weighted edges. **(b)** Positive weight cycle. **(c)** Results.

**Figure 1.3.** Positive weight cycles and infinite derivations in CFL-reachability.



**(a)** Rule for solving weighted edges. **(b)** Positive weight cycle. **(c)** Results.

**Figure 1.4.** Positive weight cycles and infinite derivations in set constraint analysis.

symbol $X$ with an offset $i$. We can see that *infinite derivations* of edges (in Figure 1.3) or set elements (in Figure 1.4) occur when iteratively solving the positive weight cycles.

Notably, cycle elimination [38, 51, 92, 97, 127] is not applicable to collapse positive weight cycles because the nodes in such cycles are not equivalent items, thus merging a positive weight cycle into a node will cause incorrect/imprecise results.

To avoid infinite derivations, existing works [96, 121] manually set an upper bound of the number of derivations for each weighted constraint. This trades off precision with practicability and has two defects. First, if the upper bound is small, solving positive weight cycles will be highly imprecise although it can possibly be quicker. Second, if the upper bound is set to be large enough, solving positive weight cycles will be more precise but very inefficient.

We observed that the derivations caused by positive weight cycles follow particular *derivation equivalence* patterns. Briefly, when iteratively processing the cycles, the items are derived using a particular initial value with some constant strides, which depends on the weights of the weighted edges. For example, the offsets of both results in Figures 1.3 and 1.4, i.e., edge labels in Figure 1.3(c) and set elements in Figure 1.4(c), follow an initial value 1 with a constant stride 1. This property allows us to use a representative item, which captures the initial value and the constant strides of the items derived from the same positive weight cycle, to avoid infinite derivation while preserving the precision.

In this dissertation, we use field-sensitive constraint-based pointer analysis [10, 12, 15, 74, 87, 93, 96] as a client to show how positive weight cycles limits the precision and scalability of program analysis and demonstrate our strategy. Specifically, the weighted edges are the abstractions of field accesses in a program, with the weight of each edge denoting the index of an accessed field, and cycles containing such weighted edges causes infinite abstract field derivations. Correspondingly, we propose a new *stride-based field representation* to capture the derivation equivalence patterns of fields derived by the same positive weight cycles, which avoids infinite derivations while preserving the precision for constraint solving. On top of the stride-based field representation, we provide a *derivation equivalence algorithm*, for precise and efficient field-sensitive pointer analysis. By capturing derivation equivalence, our approach avoids redundant field derivations with greatly reduced overhead during points-to propagation, making constraint solving converge more quickly.

As set constraint analysis and CFL-reachability are intervertable, although our derivation equivalence algorithm is proposed in the form of set constraint, it can be converted into a version suitable to solve CFL-reachability using the method in [86].

## 1.3 Contributions

This dissertation improves the scalability of dynamic transitive closure based program analysis from three aspects, as depicted in Figure 1.5.

### 1.3.1 Eliminating Transitive Redundancy On-The-Fly

- We offer a perspective that reduces transitive redundancy via ordered derivations, and introduce a hybrid graph representation to efficiently infer the derivation order during on-the-fly all-pairs CFL-Reachability analysis.
- We present POCR, a partially ordered CFL-reachability algorithm, which significantly accelerates the solving of CFL-reachability problems where transitive redundancy dominates.
- We apply our technique to a context-sensitive value-flow analysis [149] and a field-sensitive alias analysis for C/C++ [126]. The empirical results show that: (1) POCR eliminates almost all redundant derivations. On average, the reduction rates of value-flow analysis and alias analysis are 98.50% and 97.26%, respectively, and (2) By eliminating redundant derivations, POCR achieves speedups of 21.48× and 19.57× over the standard algorithm [86], respectively, for value-flow analysis and alias analysis. POCR is also over 3× faster than the two recent CFL-reachability solvers Graspan [133] and Soufflé [68] in the two clients.

### 1.3.2 Recursive State Machine Guided Graph Folding

- We offer a perspective that utilizes recursive state machine to guide the simplification of the input graph of CFL-reachability.
- For the CFL-reachability problems where the CFL can be expressed as a deterministic RSM, we propose a graph folding principle that is able to identify whether two

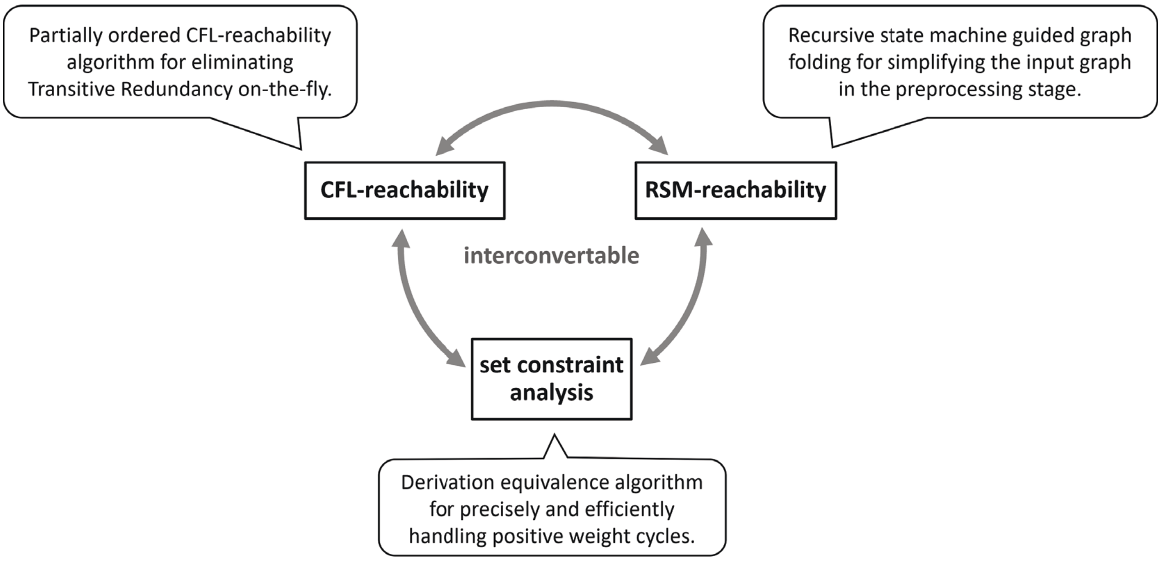

Partially ordered CFL-reachability algorithm for eliminating Transitive Redundancy on-the-fly.
Recursive state machine guided graph folding for simplifying the input graph in the preprocessing stage.
CFL-reachability
RSM-reachability
interconvertable
set constraint analysis
Derivation equivalence algorithm for precisely and efficiently handling positive weight cycles.

**Figure 1.5.** Main contributions of this thesis.

adjacent nodes are foldable by examining only their incoming and outgoing edges.

- We provide a graph folding algorithm GF, whose time complexity is linear with respect to the number of nodes of the input graph.
- We apply GF to context-sensitive value-flow analysis [126] and field-sensitive alias analysis [149] of 10 open-source C/C++ programs to evaluate the performance of GF. By reducing 60.96% nodes and 42.67% edges of the input graphs, GF accelerates context-sensitive value-flow analysis by 3.16× with a memory reduction rate of 14.74%; by reducing 38.93% nodes and 35.61% edges, GF accelerates field-sensitive alias analysis by 3.68× with a memory reduction rate of 16.93%.

### 1.3.3 Fast and Precise Handling of Positive Weight Cycles

- We propose a new stride-based model to capture the initial value and strides of the derivation equivalent items and avoid infinite derivation.
- Using field-sensitive Andersen's pointer analysis as a client, we present a fast and precise handling of positive weight cycles to significantly boost the performance by capturing derivation equivalence when solving positive weight cycles.

- We apply DEA to 11 real-world large C/C++ programs. The results show that DEA on average is 7.1× faster than existing field-sensitive analysis with the best speedup of 11.0×.

## 1.4 Thesis Organization

This thesis is organized as follows:

- Chapter 2 is a literature review, including existing techniques for improving dynamic transitive closure based program analysis.
- Chapter 3 details our partially ordered algorithm for taming transitive redundancy in the dynamic solving procedure of CFL-reachability.
- Chapter 4 details our recursive state machine guided graph folding technique, which is to simplify the input graph in the preprocessing stage.
- Chapter 5 details our stride-based model and derivation equivalence algorithm for precisely and efficiently handle positive weight cycles.
- Chapter 6 concludes this thesis.

# Chapter 2

# Literature Review

Dynamic transitive closure based program analysis has been studied for many years [3, 11, 13, 29, 36, 45, 59, 60, 66, 78, 81, 91, 103, 128]. In this chapter, we briefly review the literature which is most relevant to the works of this thesis.

## 2.1 Context-Free Language and CFL-reachability

Context-free language (CFL), as a power automatic sentence-generating device [30], has been extensively studied and applied for many years [11, 18, 43, 71, 101]. Theoretically studies on context-free language include its algebraic theory [30], semantics [71], expressiveness [11], relation to natural language [101], etc. Context-free languages have been applied to many areas, including compiler design [2, 73], pattern recognition [31, 44, 85], sentence generation [102, 140], program analysis [61, 106], etc.

In 1990, Yannakakis [141] formulated the context-free language reachability (CFL-reachability) framework, which incorporates CFLs to extend the expressiveness of ordinary graph reachability [63, 130], to handle Datalog chain query evaluation. It soon became a fundamental technique for program analysis and was applied to a large variety of clients [105]. For example, Reps et al. [105] formulated interprocedural dataflow analysis into CFL-reachability and applied it to finding possibly uninitialized variables for C programs. Rehof and Fähndrich [104] used CFL-reachability to solve type-based polymorphic flow analysis, which supports context-sensitive, global flow summarization and includes polymorphic recursion. Pratikakis et al. [100] used CFL-reachability to conduct existential label flow inference. Sridharan et al. [116, 117] performed demand-driven context-sensitive pointer analysis for Java via CFL-reachability. Zheng and Rugina [149] proposed the context-free grammar for field-insensitive and field-sensitive alias analysis and applied it to demand-driven alias analysis for C. Xu et al. [139] improved the scalability of CFL-reachability based pointer analysis using context-sensitive must-not-alise analysis. Shang et al. [114] improved the precision of on-demand dynamic summary-based points-to analysis by performing partial point-to analysis. Wang et al. [132] proposed a typestate-guided fuzzer for discovering use-after-free vulnerabilities based on CFL-reachability framework. Zhang and Su [146] proposed linear conjunctive language (LCL) reachability based on CFL-reachability, and improved the precision and scalability of context-sensitive data-dependence analysis.

## 2.2 Pushdown Automata and Recursive State Machines

Context-free languages are also studied in alternative forms like pushdown automata [11, 40, 43, 88, 113] and recursive state machines [6]. Schützenberger et al. [113] and Muller et al. [88] demonstrated the expressiveness of pushdown automata and the relation between pushdown automata and context-free languages. Ginsburg et al [43] studied the determinacy of context-free languages from the perspective of pushdown automata and formulated deterministic context-free languages, i.e., the languages accepted by deterministic pushdown automata. Finkel et al. [40] proposed a simple and direct algorithm for computing the always regular set of reachable states of a pushdown system. Pushdown systems are widely applied to model checking and context-sensitive analysis

[20, 37, 107, 115, 131]. For example, Bouajjani et al. [20] used pushdown systems to model multi-thread programs. bouajjani2005regular Reps et al. [108] used weighted pushdown system to handle interprocedural dataflow analysis. Alur and Madhusudan [7] proposed the class of visibly pushdown languages as embeddings of context-free languages that is rich enough to model program analysis questions and yet is tractable and robust like the class of regular languages. Kumar et al. [72] studied visibly pushdown automata (VPA) for processing XML documents. Späth et al. [115] performed control-, flow- and field-sensitive dataflow analysis using synchronized pushdown systems.

Alur et al. [6] proposed recursive state machines, which naturally model the control flow in sequential imperative programs containing recursive procedure calls. In the same work, they also illuminated the equivalent expressiveness of recursive state machines and context-free languages. Chaudhuri [27] proposed a subcubic algorithms for recursive state machines. Benerecetti et al. [16] extended recursive state machines to timed recursive state machines. A large variety of CFL-reachability analysis are solved with the auxiliary of recursive state machines [84, 89, 104, 105, 114, 116, 117, 132, 147, 149]. Recursive state machines are also applied to semantic parsing for speech understanding [99].

## 2.3 Set Constraint Analysis

Set constraint analysis has also been studied and applied for many years. The earlist example can be tracked back to 1969, when Reynolds [109] proposed an analysis of Lisp programs based on the resolution of inclusion constraints. Heintze and Jaffar [55] first coined the term “set constraints” and recognized and formalized set constraints in their full generality. An algorithm [4] were soon proposed to solve a subclass of set constraints which excludes projections but includes all other operations. Bachmair et al. [14] showed that set constraints without projections are equivalent to the monadic class of predicate logic. There were several works proposed to show increasingly powerful systems of constraints to be decidable [14, 23, 42]. Charatonik and Pacholski [24] finally showed that the full set constraint language is decidable. Melski and Reps [86] showed that set constraint and CFL-reachability are interconvertible, and proposed algorithms to convert a set constraint problem to an equivalent CFL-reachability and convert a CFL-reachability problem to an equivalent set constraint problem.

Set constraint analysis is widely applied to program analysis such as dataflow analysis [70], abstract interpretation [34], model checking [32], invariant generation [33], etc. One prominent application of set constraint analysis in recent years is pointer analysis [10, 51, 76, 77, 96, 97, 111, 119]. Andersen first formulated pointer analysis into set constraints in his PhD thesis [10]. Then the constraint-based pointer analysis was extensively studied and applied. Su et al. [119] proposed a project merging technique to reduce path redundancy in constraint solving. Heintze and Tardieu [56] introduced a new algorithm for computing the dynamic transitive closure for constraint-based pointer analysis. Berndl et al. [17] described a field-sensitive inclusion-based pointer analysis for Java, which uses BDDs to represent both the constraint graph and the points-to solution. Considering the difference of C from Java that it permits the addresses of fields to be taken, Pearce et al. [96] used weighted constraints to model field accesses and proposed an algorithm for solving field-sensitive pointer analysis for C. Lhoták and Chung [77] proposed an efficient flow-sensitive pointer analysis algorithm, which focuses on handling the strong updates of singletons. Sui et al. [126] detected memory leak using sparse value-flow analysis, which is based on constraint-based pointer analysis.

## 2.4 Efficiency Study

In view of the wide application and the high (cubic) complexity of CFL-reachability and its alternative forms, improving efficiency has been a important research topic and was extensively studied. Existing works improve the efficiency of CFL-reachability from two aspects: lowering the complexity of the algorithm [9, 26, 75, 118, 138, 145] and reducing the redundant computation via preprocessing or on-the-fly [26, 51, 57, 80, 90, 111, 133].

Complexity study of CFL-reachability algorithm is an interesting topic and has attracted researchers for many years. A series of works focusing on the $O(n^{3-\varepsilon})$ upper bound of matrix multiplication has reduced the theoretical time complexity of CFL-reachability from $O(n^{2.81})$ to $O(n^{2.373})$ [9, 75, 118, 136–138]. Lowering the complexity of the alternative forms of CFL-reachability has also been studied. For example, Chaudhuri [27] proposed subcubic algorithms for recursive state machines, which solve all-pairs reachability in bounded-stack RSMs in $O(n^3/\log^2 n)$ time. Efficient algorithms for particular subclass of CFL-reachability have also been proposed. Zhang et al. [145] proposed an algorithm that solves Dyck-reachability in $O(n + m \times \log m)$ time, where $n$ and $m$ denotes the number of nodes and edge. Chatterjee et al. [25] further reduce the time complexity to $O(m \times \log n)$.

The defect of the aforementioned lower complexity techniques is that they are either difficult to implement or restricted to particular aspects. Thus, when solving real-world problems, researchers also scales CFL-reachability from a more practical perspective: reducing redundancy. For reducing redundant computations on-the-fly, Zhang et al. [147] designed an efficient algorithm to accelerate all-pair alias analysis based on the context-free grammar. Chatterjee et al. [26] proposed an algorithms for algebraic path properties in recursive state machines with constant treewidth, which quickly answers multiple-query problems. Nappa et al. [90] designed a parallel union-find data structure for scalable equivalence relations. Recently, Zuo et al. [133, 150] proposed a generally applicable edge-versioning algorithm for CFL-reachability, which reduces edge duplication during the solving procedure. It is interesting to note that, the above techniques are either have strong performance but restricted to particular languages [145], or have relatively weak performance because of not modifying the unordered solving process of CFL-reachability. In this dissertation, we propose a partially ordered CFL-reachability algorithm (Chapter 3), which reduces over 90% of redundant computation in the solving process of CFL-reachability.

Redundancy of CFL-reachability is also reduced in preprocessing stage. One perspective is to simplify the input context-free grammar or its alternative representations [5, 41, 57, 62]. Heizmann et al. [57] proposed a technique to minimize pushdown automata using partial Max-SAT. Such technique can effectively improve the efficiency of CFL-reachability when the input pushdown automata is improperly designed. Another choice is to simplify the input graph [38, 51, 80, 95, 97, 111] based on the input grammar. Such techniques are usually client-specific. Recently, Li et al [80] proposed an algorithm for eliminating the "non-Dyck-contributing" edges for interleaved Dyck reachability, and improved the speed and precision of context- and field-sensitive taint analysis. In the literature, graph simplification is also widely applied to pointer analysis, in the form of set constraint analysis [38, 51, 95, 97, 111]. Rountev and Chandra [111] proposed an edge contraction technique based on equivalent variable substitution. And there are several offline [51] and online [38, 51, 95, 97] cycle elimination techniques which collapse equivalent strongly-connected components [92, 127]. It is interesting to note that, all the above techniques are restricted to particular languages [80] or particular real-world clients [51, 95, 97, 111]. In this dissertation, we proposed a general graph folding technique for CFL-reachability (Chapter 4), which reduces the size of the input graphs in the preprocessing stage under the guidance of recursive state machines.

CHAPTER 3

# Taming Transitive Redundancy for Context-Free Language Reachability

Section 1.2.1 has briefly introduced this topic, in this chapter, we start with the formal definitions of context-free language reachability and transitive redundancy, and then show our solution.

## 3.1 Problem Formulation

### 3.1.1 CFL-Reachability

A context-free language reachability (CFL-reachability) instance $\mathit{Reach}\langle \mathit{CFG}, G\rangle$ is comprised of a context-free grammar $\mathit{CFG} = \langle \Sigma, N, T, P, N_{start}\rangle$ and an edge-labeled graph $G = \langle V, E\rangle$. In *CFG*,

$\Sigma = N \cup T$ is an *alphabet* containing two kinds of symbols: $N$ a set of *non-terminals* and $T$ a set of *terminals*;

$P$ is a set of *production rules*, with each rule describing how the non-terminal on the left side produces the symbols on the right side, which can be terminals or non-terminals;

$N_{start} \in N$ is the start symbol of the language.

The graph $G$ is dynamically incremental, which means that edges are added into the graph during CFL-reachability solving. In the graph, each edge $v_i \xrightarrow{X} v_j \in E$ labeled by a symbol $X \in \Sigma$. In general, the initial $G$ contains only edges labeled by terminals.

Solving CFL-reachability is to summarize new edges from existing ones according to the production rules, and make explicit reachability relations by adding the new edges into the graph. In the graph $G$, a path $v_0 \xrightarrow{Y_1} v_1 \xrightarrow{Y_2} \dots \xrightarrow{Y_n} v_n$ indicates an *$X$-relation* from $v_0$ to $v_n$, if there is a non-terminal $X \in N$ that can produce the string $Y_1Y_2\cdots Y_n \in \Sigma^n$, which is formed by the sequence of edge labels of the path, via one or more production rules. For any two nodes $v_i, v_j \in V$, if there is a path from $v_i$ to $v_j$ implying an $X$-relation, $v_j$ is said to be *$X$-reachable* from $v_i$.

Reachability relations are made explicit in two steps – deriving new edges from existing paths and adding the edges to the graph. Specifically,

- *Edge Derivation*: An $X$-edge $v_i \xrightarrow{X} v_j$ can be derived from a path $v_i \xrightarrow{Y_1} \cdots \xrightarrow{Y_k} v_j \in G$ if there is a production rule $X ::= Y_1 \cdots Y_k \in P$.
- *Edge Addition*: A newly derived $X$-edge $v_i \xrightarrow{X} v_j$ should be added to the graph to make explicit the $X$-relation from $v_i$ to $v_j$ *if it is not already in the graph*.

CFL-reachability is solved using a standard dynamic programming algorithm [86]. The algorithm maintains a worklist $W$ to hold all the unprocessed edges. Initially, all the edges in the graph $G$ and all underlying self-cycles caused by empty rules are considered as unprocessed. The algorithm iteratively processes and removes the edges in the worklist, generating new edges by consulting the edges and its neighbors based on the grammar *CFG*. Any new edge that is not already in the graph will be added to the graph and to the worklist, to make explicit a new discovered reachability relation. The algorithm keeps solving until the worklist is empty, which means that there is not any new edge that can be added to the graph, i.e., all the reachability relations are explicit. The pseudocode of the algorithm is presented in Algorithm 1 for further

**Algorithm 1:** Standard CFL-reachability algorithm.

```
   $P$: the set of production rules in $CFG$.
1  Function Reach($CFG$,$G$)
2      init();                                                    /* Lines 11-15 */
3      while $W \neq \emptyset$ do
4          select and remove an edge $v_i \xrightarrow{Y} v_j$ from $W$;
5          for each production $X ::= Y \in P$ do
6              if $v_i \xrightarrow{X} v_j \notin E$ then add $v_i \xrightarrow{X} v_j$ to $E$ and to $W$;
7          for each production $X ::= Y\ Z \in P$ do
8              CheckSucc($X$,$Z$,$v_i$,$v_j$);                       /* Lines 19-21 */
9          for each production $X ::= Z\ Y \in P$ do
10             CheckPred($X$,$Z$,$v_i$,$v_j$);                       /* Lines 16-18 */
11 Procedure init()
12     add all edges of $E$ to $W$;
13     for each production $X ::= \varepsilon \in P$ do
14         for each node $v_i \in V$ do
15             if $v_i \xrightarrow{X} v_i \notin E$ then add $v_i \xrightarrow{X} v_i$ to $E$ and to $W$;
16 Procedure CheckPred($X$,$Z$,$v_i$,$v_j$);
17     for each edge $v_k \xrightarrow{Z} v_i \in G$ do
18         if $v_k \xrightarrow{X} v_j \notin E$ then add $v_k \xrightarrow{X} v_j$ to $E$ and to $W$;
19 Procedure CheckSucc($X$,$Z$,$v_i$,$v_j$)
20     for each edge $v_j \xrightarrow{Z} v_k \in G$ do
21         if $v_i \xrightarrow{X} v_k \notin E$ then add $v_i \xrightarrow{X} v_k$ to $E$ and to $W$;
```

discussion. Notably, the algorithm accepts a *normalized* CFG, i.e., the right-hand side of each production has at most two symbols. Lines 5–6 derive edges from paths containing only one edge. And the two procedures CheckPred (Lines 16–18) and CheckSucc (Lines 19–21) derive edges from paths consisting of two edges.

### 3.1.2 Redundant Derivations and Transitive Redundancy

Identical to most existing techniques, we conservatively use *adjacency lists* to collect the edges in the graph of CFL-reachability. Specifically, with respect to a label $X$ and a node $v_i$, the $X$-predecessors and the $X$-successors of $v_i$ are stored in two sets $pred(X,v_i) \subseteq V$ and $succ(X,v_i) \subseteq V$ such that

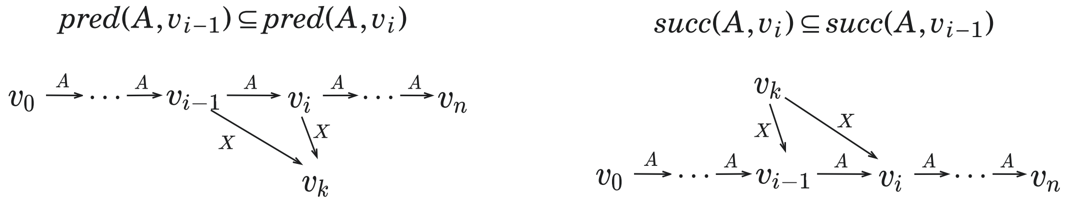
$$\begin{cases} pred(X,v_i) = \{v'_i \mid v'_i \xrightarrow{X} v_i \in E\} \\ succ(X,v_i) = \{v'_i \mid v_i \xrightarrow{X} v'_i \in E\}. \end{cases}$$

The adjacency lists are growing when solving CFL-reachability, e.g., adding an edge $v_i \xrightarrow{X} v_j$ to $E$ means to add $v_i$ to $pred(X,v_j)$ and $v_j$ to $succ(X,v_i)$.

**Redundant Derivation.** In Algorithm 1, redundant derivations mainly occur in `CheckPred` and `CheckSucc`, where edge derivations are implied in the traversal of predecessors/successors of nodes. For example, `CheckPred`$(X,Z,v_i,v_j)$ naturally derives $|pred(Z,v_i)|$ $X$-edges and each derived edge is checked to determine whether it is already in the graph before addition. Thus, the shared predecessors or successors of nodes is a major contributing factor in redundant derivations.

*Example* 3.1. While processing an edge $v_i \xrightarrow{Y} v_j$, for any production rule $X ::= Z\ Y \in P$, `CheckPred`$(X,Z,v_i,v_j)$ traverses $pred(Z,v_i)$ to determine whether the derived $v_k \xrightarrow{X} v_j$ is in the graph for all $v_k \in pred(Z,v_i)$. Similarly, while processing another edge $v'_i \xrightarrow{Y} v_j$, `CheckPred`$(X,Z,v'_i,v_j)$ traverses $pred(Z,v'_i)$ to determine whether the derived $v'_k \xrightarrow{X} v_j$ is in the graph for all $v'_k \in pred(Z,v'_i)$. The nodes in $pred(Z,v_i) \cap pred(Z,v'_i)$ are repeatedly visited by `CheckPred`$(X,Z,v_i,v_j)$ and `CheckPred`$(X,Z,v'_i,v_j)$. Namely, there are at least $|pred(Z,v_i) \cap pred(Z,v'_i)|$ repeated edge derivations.

**Transitive Redundancy.** In CFL-reachability, the transitivity of a relation $A$ can either manifest in a doubly recursive rule (Definition 3.1) $A ::= A\ A$ or be implied in other production rules such as $A ::= A^*$, $A ::= A^+$, etc. Any edge $v_i \xrightarrow{A} v_j \in G$ denoting



**(a)** Processing $v_{i-1} \xrightarrow{X} v_k$ and $v_i \xrightarrow{X} v_k$ based on $X ::= A\ X$ leads to $|pred(A,v_{i-1})|$ repeated computations.

**(b)** Processing $v_k \xrightarrow{X} v_{i-1}$ and $v_k \xrightarrow{X} v_i$ based on $X ::= X\ A$ leads to $|succ(A,v_i)|$ repeated computations.

**Figure 3.1.** Redundant checks in partial transitive derivations, where $A$ is a transitive relation.

a transitive relation implies a series of reachability relations $v_k \xrightarrow{A} v'_k$ for all $(v_k, v'_k) \in pred(A,v_i) \times succ(A,v_j)$.

Besides transitive relations, CFL-reachability also handles partial transitive relations whose edges are derived via singly recursive rules (Definition 3.2). For example, the production rule $F_i ::= F_i\ A$ in our motivating example is left-recursive. Different from transitive relations, partial transitive relations do not benefit from cycle elimination since cycles consisting of partial transitive edges cannot be merged.

**Definition 3.1** (*Doubly Recursive Rules*)**.** A doubly recursive rule, also called a left-right-recursive rule, is in form of $A ::= A\ A$.

**Definition 3.2** (*Singly Recursive Rules*)**.** The *left-recursive* rules, in the form of $X ::= X\ A$, and the *right-recursive* rules, in the form of $X ::= A\ X$, are collectively called singly recursive rules.

In CFL-reachability, both transitive and partial transitive relations suffer from transitive redundancy. The property of transitive relations implies that for any path $v_0 \xrightarrow{A} v_1 \xrightarrow{A} \cdots \xrightarrow{A} v_n$ where $A$ is transitive, there are $pred(A,v_{i-1}) \subseteq pred(A,v_i)$ and $succ(A,v_i) \subseteq succ(A,v_{i-1})$ for all $i \in \{1,\ldots,n\}$. Therefore, for any $X ::= A\ X \in P$, processing $v_i \xrightarrow{X} v_k$ and $v_j \xrightarrow{X} v_k$ where $i,j \in \{0,\ldots,n\}$ and $i < j$ leads to $|pred(A,v_i)|$ repeated derivation. Similarly, for any $X ::= X\ A \in P$, processing $v_k \xrightarrow{X} v_i$ and $v_k \xrightarrow{X} v_j$ where $i,j \in \{0,\ldots,n\}$

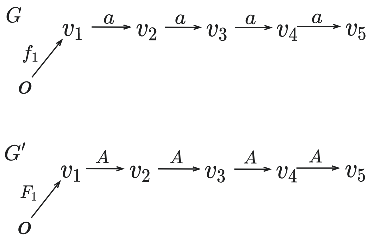
$$A ::= A\ A \mid a$$

$$F_i ::= F_i\ A \mid f_i$$

**(a)** A context-free grammar.

```
struct T{int f1; int f2;}
struct T o; ...
int v1,v2,v3,v4,v5;
v1 = o.f1;
v5 = v4 = v3 = v2 = v1;
```

**(b)** C code fragment.



**(c)** $G$ is the graph abstracted from (a). $G'$ is transformed from $G$ via $A ::= a$ and $F_i ::= f_i$.

$$o \xrightarrow{F_1} v_5 \Leftarrow \begin{cases} o \xrightarrow{F_1} v_1 \xrightarrow{A} v_5 \\ o \xrightarrow{F_1} v_2 \xrightarrow{A} v_5 \\ o \xrightarrow{F_1} v_3 \xrightarrow{A} v_5 \\ o \xrightarrow{F_1} v_4 \xrightarrow{A} v_5 \text{ /* POCR */} \end{cases}$$

**(d)** $o \xrightarrow{F_1} v_5$ can be derived in four ways by the standard algorithm whereas our approach reduces the four ways to one (in this example, it is the last one).

**Figure 3.2.** An example.

and $i < j$ leads to $|succ(A, v_j)|$ repeated derivation, as illustrated in Figure 3.1. Such property greatly increase redundancy when the path to be derived is long.

### 3.1.3 Research Problem

We use an example to show transitive redundancy in real-world static analysis and illustrate the key idea of our solution.

*Example* 3.2. The CFG in Figure 3.2(a) is often appears in field-sensitive analysis [149]. In the grammar, $A$ denotes a value flow, $F_i$ denotes the propagation of the value of the $i$-th field, $a$ denotes an assignment, and $f_i$ denotes the address of the $i$-the field. The two production rules mean that an $A$-edge can be generated from an $a$-edge or two connected $A$-edges in an edge labeled graph; and an $F_i$-edge can be generated from an $f_i$-edge or a path comprised of an $F_i$-edge and an $A$-edge. Obviously, $A$ is a transitive relation, and $F_i$ is partially transitive because it can be transited via $A$ relations. In Figure 3.2(c), graph $G$ is abstracted from the code fragment of Figure 3.2(b) and $G'$ is transformed from $G$ by applying $A ::= a$ and $F_i ::= f_i$.

The standard CFL-reachability algorithm (i.e., Algorithm 1) derives new edges from paths consisting of at most two edges, where redundant derivations can be triggered in various ways. Here we illustrate one case. According to the productions $A ::= A\ A$ and $F_i ::= F_i\ A$ in Figure 3.2(a), there will be $v_k \xrightarrow{A} v_5$ for all $k \in \{1,\ldots,4\}$ and $o \xrightarrow{F_1} v_j$ for all $j \in \{1,\ldots,5\}$ in the graph after solving reachability. Then we focus on how $o \xrightarrow{F_1} v_5$ is derived. Due to $F_i ::= F_i\ A$, $o \xrightarrow{F_1} v_5$ can be derived from $o \xrightarrow{F_1} v_j$ and $v_k \xrightarrow{A} v_5$ whenever $j = k$. There are four paths (consisting of two edges) in total that can generate $o \xrightarrow{F_1} v_5$, as shown in Figure 3.2(d). The traditional algorithm computes edges in an arbitrary order. To ensure a correct reachability solution, it derives edges from each of the four paths at least once. Thus, there are at least three redundant derivations of $o \xrightarrow{F_1} v_5$.

**Our Insight and Goal.** Our approach determines the computation order of the $F_1$-edges based on the order of the nodes on the path $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4 \xrightarrow{A} v_5$ in $G'$ of Figure 3.2(c). Specifically, we only derive $F_1$-edges $o \xrightarrow{F_1} v_{j+1}$ from $o \xrightarrow{F_1} v_j \xrightarrow{A} v_{j+1}$, where $j \in \{1,\ldots,4\}$, and always derive $o \xrightarrow{F_1} v_{j+1}$ immediately after adding $o \xrightarrow{F_1} v_j$ to the graph. This is a "head-to-tail" derivation order, and it only derives $o \xrightarrow{F_1} v_5$ by the last line in Figure 3.2(d), avoiding the first three ways causing redundant derivations. Similarly, the redundant derivations of $o \xrightarrow{F_1} v_2$, $o \xrightarrow{F_1} v_3$ and $o \xrightarrow{F_1} v_4$ are eliminated by our approach. Moreover, if $o \xrightarrow{F_1} v_1$ is added to the graph again based on they other ways described in Figure3.2(d), our approach can avoid the repeated derivation of $o \xrightarrow{F_1} v_k$ where $k \in \{2,\ldots,5\}$ because we know that such edges have already been added to the graph along with the first time adding $o \xrightarrow{F_1} v_1$. Compared to existing techniques, cycle elimination has no effect in this example because there is no cycle in the graph. The existing edge duplication reduction technique (Section 4.2 in [133]) which does not exploit ordered derivations still introduces redundancy that $o \xrightarrow{F_1} v_5$ can be derived from $o \xrightarrow{F_1} v_1 \xrightarrow{A} v_5$ and $o \xrightarrow{F_1} v_2 \xrightarrow{A} v_5$. Our experiment also confirms that the technique still suffers from a lot of redundant derivations during CFL-reachability solving.

Our insight shows that our technique is to benefit the CFL-reachability problems containing transitive relations, which manifest in doubly recursive rules, and partial transitive relations, which manifest in left- or right-recursive rules. Similar to the standard CFL-reachability algorithm, our technique also works on normalized context-free grammars. Specifically, it requires the transitive relations to be explicit in the form of doubly recursive rules, i.e., $A ::= A\ A$.

We formulate the research problem as follows:

> Given a CFL-reachability instance containing transitive relations, use partially ordered derivation to eliminate the redundant derivations caused by singly recursive or doubly recursive rules.

**Challenges.** In the example of Figure 3.2, we can easily obtain the computation order along the path $v_1 \xrightarrow{A} v_2 \xrightarrow{A} v_3 \xrightarrow{A} v_4 \xrightarrow{A} v_5$. However, the graphs of real-world CFL-reachability problems are large and complex, where multiple paths and/or cycles may share vertices. Moreover, adding edges when solving CFL-reachability may also change the transitive closure of the graph. Hence, the best derivation order is also changed dynamically. These require an effective representation to maintain and infer the computation order efficiently, and the representation can be updated accordingly with the dynamic graph during CFL-reachability solving.

## 3.2 Our Solution

Example 3.2 shows that ordering computations based on the property of transitive relations can effectively reduce redundant derivations. The challenge lies in constructing a proper representation of the transitive relations on top of the dynamically changed graph and correctly updating the representation when solving CFL-reachability. This section details our solution. Section 3.2.1 introduces a hybrid graph representation to reduce redundant derivations. Section 3.2.2 provides a dynamic construction algorithm to efficiently update the spanning-tree model in our hybrid graph representation. Section 3.2.3 proposes the overall solution: a partially ordered CFL-reachability algorithm POCR for all-pairs CFL-reachability analysis.

### 3.2.1 Hybrid Graph Representation for Reducing Redundant Derivations

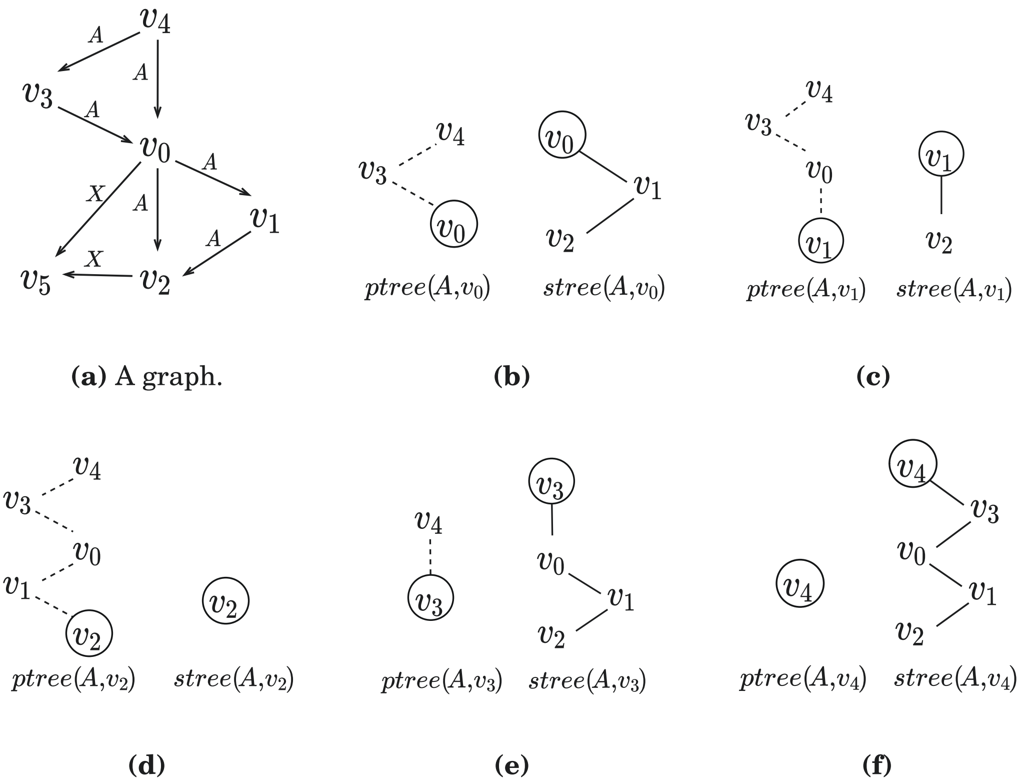
For any node $v_i \in V$, we can always construct a spanning tree rooted at $v_i$ to represent its predecessors/successors associated with a transitive relation $A$ [65]. Before introducing our spanning-tree model, we first study the following property: In CFL-reachability, a transitive $A$-edge can be created not only by using a doubly recursive rule $A ::= A\ A$ but also by other rules such as $A ::= a$ or $A ::= B\ C$, etc. We classify the $A$-edges created via different production rules into two categories: *primary edges* (Definition 3.3) and *secondary edges* (Definition 3.4).

**Definition 3.3** (*Primary Edges*)**.** For a transitive relation $A$, a primary $A$-edge is created via a production rule that is not in the form of $A ::= A\ A$.

**Definition 3.4** (*Secondary Edges*)**.** For a transitive relation $A$, a secondary $A$-edge is created via the production rule $A ::= A\ A$.

We chose to define the notion of secondary edge because such edges derived by $A := A\ A$ do not change the set of reachable nodes. Conversely, primary edges may add to the set of reachable nodes. Figure 3.2(c) in Example 3.2 illustrates this property: adding the $A$-edges created via $A ::= a$ increases the number of nodes that can be visited by traversing along the $A$-edges, whereas adding the $A$-edges created via $A ::= A\ A$ does not. Thus, while constructing spanning trees, we do not consider the secondary edges.

**Predecessor Trees and Successor Trees.** We use the primary edges to construct spanning trees to determine the computation order. Corresponding to the adjacency lists



**(a)** A graph. **(b)** **(c)**

**(d)** **(e)** **(f)**

**Figure 3.3.** Predecessor trees and successor trees. The root of each spanning tree is circled.

(Section 3.1.2), for each transitive relation $A$, we assign to each node $v_i$ a *predecessor tree* *ptree*$(A,v_i)$ and a *successor tree stree*$(A,v_i)$. Both *ptree*$(A,v_i)$ and *stree*$(A,v_i)$ are rooted at $v_i$ with the following properties:

*ptree*$(A,v_i)$: (1) for any node $v_j \neq v_i$, $v_j \in$ *ptree*$(A,v_i)$ iff $v_j \in$ *pred*$(A,v_i)$;

(2) for any two nodes $v_k, v_l$ such that $v_l$ is a child of $v_k$ in *ptree*$(A,v_i)$, there is a primary $A$-edge $v_l \xrightarrow{A} v_k \in E$.

*stree*$(A,v_i)$: (1) for any node $v_j \neq v_i$, $v_j \in$ *stree*$(A,v_i)$ iff $v_j \in$ *succ*$(A,v_i)$;

(2) for any two nodes $v_k, v_l$ such that $v_l$ is a child of $v_k$ in *stree*$(A,v_i)$, there is a primary $A$-edge $v_k \xrightarrow{A} v_l \in E$.

The above properties provide an efficient tree traversal to determine the computation order of partial transitive relations. The benefit of the above properties is two-fold. On one hand, the first property of *ptree*$(A,v_i)$ and *stree*$(A,v_i)$ ensures that the traversal can touch all the $A$-predecessors and $A$-successors of $v_i$, which ensures complete edge additions. On the other hand, the tree structure provides an efficient traversal to determine computation order. Specifically, traversing the $A$-predecessors (resp. $A$-successors) of a node can be done in $O(|pred(A,v_i)|)$ (resp. $O(|succ(A,v_i)|)$) time, as the number of edges in a tree is always equal to the number of nodes minus one.

*Example* 3.3 (Predecessor Trees and Successor Trees). Figure 3.3(a) is a graph where $A$ is transitive and (b)–(f) display the predecessor trees and successor trees of nodes $v_0,\ldots,v_4$. The root of each tree is marked by a circle, and the edges in the predecessor trees are marked by dashed edges. Note that the two secondary edges $v_4 \xrightarrow{A} v_0$ and $v_0 \xrightarrow{A} v_2$ are not included in any of the spanning trees.

**Hybrid Graph Representation.** We embed the spanning-tree model into the standard adjacency-list graph representation, constructing a *hybrid graph representation*. In the hybrid graph representation, for a transitive relation $A$ and a node $v_i$, each element $v_j \in pred(A,v_i)$ is maintained as a pointer pointing to the node $v_j \in ptree(A,v_i)$, and each element $v_k \in succ(A,v_i)$ is maintained as a pointer pointing to the node $v_k \in stree(A,v_k)$. This keeps good time efficiency for both lookups and traversals. Specifically, we perform lookups of nodes in adjacency lists[1], and traversals of predecessors and successors of nodes in predecessor trees and successor trees, respectively.

**Efficient Singly Recursive Edge Creations.** Algorithm 2 performs efficient derivations based on singly recursive rules. Like `CheckPred` and `CheckSucc` in Algorithm 1, `CheckPtree` (resp. `CheckStree`) create edges based on $X ::= A\ Y \in P$ (resp. $X ::= Y\ A \in P$) for all transitive relations $A$. The difference is that `CheckPtree` traverses a predecessor tree $ptree(A,v_t)$ (whose root $v_t$ is input as parameter), instead of traversing the adjacency list $pred(A,v_x)$. Similarly, `CheckStree` traverses $stree(A,v_t)$ instead of traversing $succ(A,v_t)$.

[1] Implementing the adjacency lists by hash tables can reduce the time complexity of lookups to $O(1)$.

**Algorithm 2:** Singly recursive edge derivations.

```
1  Function CheckPtree($X, A, v_x, v_y, v_t$)
2      for each $v_z$ child of $v_x$ in $ptree(A, v_t)$ do
3          if $v_z \xrightarrow{X} v_y \notin E$ then
4              add $v_z \xrightarrow{X} v_y$ to $E$ and to $W$;
5              CheckPtree($X, A, v_z, v_y, v_t$);
6  Function CheckStree($X, A, v_x, v_y, v_t$)
7      for each $v_z$ child of $v_y$ in $stree(A, v_t)$ do
8          if $v_x \xrightarrow{X} v_z \notin E$ then
9              add $v_x \xrightarrow{X} v_z$ to $E$ and to $W$;
10             CheckStree($X, A, v_x, v_z, v_t$);
```

While processing an edge $v_x \xrightarrow{X} v_y$, for each $X ::= A\ X$ where $A$ is transitive, `CheckPtree` $(X, A, v_x, v_y, v_x)$ is called (with $v_t$ specified as $v_x$) to traverse *ptree*$(A, v_x)$ from the root to the leaves to create and add the $X$-edges $v_z \xrightarrow{X} v_y$ to the graph, where $v_z \in pred(A, v_x)$. Similarly, for each $X ::= X\ A$, `CheckStree`$(X, A, v_x, v_y, v_y)$ is called to traverse *stree*$(A, v_y)$ to create and add the $X$-edges $v_x \xrightarrow{X} v'_z$ to the graph, where $v'_z \in succ(A, v_y)$.

**Soundness of Algorithm 2.** In Algorithm 1, while processing $v_i \xrightarrow{X} v_j$, for all transitive $A$, replacing `CheckPred`$(X, A, v_i, v_j)$ (resp. `CheckSucc`$(X, A, v_i, v_j)$) by `CheckPtree` $(X, A, v_i, v_j, v_i)$ (resp. `CheckStree`$(X, A, v_i, v_j, v_j)$) yields identical results with respect to the standard CFL-reachability solution (Algorithm 1).

Algorithm 2 preserves the above property from two aspects. Here we only discuss `CheckPtree` as `CheckStree` is similar. On one hand, with respect to `CheckPtree`$(X, A, v_x, v_y, v_x)$ which processes $v_x \xrightarrow{X} v_y$, if the traversal on *ptree*$(A, v_x)$ is never truncated by Line 3, this means that all the nodes $v_z \in ptree(A, v_x)$ are visited and the edges $v_z \xrightarrow{X} v_y$ are added to the graph and to the worklist. This is equivalent to what `CheckPred`$(X, A, v_x,$

$v_y$) does. On the other hand, if the traversal on $ptree(A,v_x)$ is truncated by Line 3, this means that the edge $v_z \xrightarrow{X} v_y$ has already been added to the graph. The edge must also have been previously added to the worklist and has been processed (or will be processed) by other calls of `CheckPtree`, which add all $v'_z \xrightarrow{X} v_y$ to the graph and to the worklist, where $v_z$ is an element of the subtree rooted at $v_z \in ptree(A,v_x)$. Therefore, the final result of `CheckPtree`$(X,A,v_x,v_y,v_x)$ is also equivalent to what `CheckPred`$(X,A,v_x,v_y)$ do (the detailed proof can be seen in Appendix A.1).

Algorithm 2 stops the redundant derivations at the first stage via the termination criteria at Line 3 and Line 8. Specifically, with respect to `CheckPtree`, once visiting a node $v_z \in ptree(A,v_t)$ such that the associated edge $v_z \xrightarrow{X} v_y$ is already in the graph, the algorithm stops traversing the subtree rooted at $v_z$. This avoids the redundant derivations of $v'_z \xrightarrow{X} v_y$, where $v'_z$ is an element of the subtree. Such traversals stop similarly in $stree(A,v_t)$.

*Example* 3.4 (*Singly Recursive Edge Creations*). In Figure 3.3(a), given $X ::= A\ X \in P$, while processing $v_0 \xrightarrow{X} v_5$, `CheckPtree`$(X,A,v_0,v_5,v_0)$ traverses $ptree(A,v_0)$ (Figure 3.3(b)) and adds $v_3 \xrightarrow{X} v_5$ and $v_4 \xrightarrow{X} v_5$ to the graph. While processing $v_2 \xrightarrow{X} v_5$, `CheckPtree`$(X,A,v_2,v_5,v_2)$ traverses $ptree(A,v_2)$ (Figure 3.3(d)). While visiting $v_1$, it adds $v_1 \xrightarrow{X} v_5$ to the graph. Then it goes to $v_0$ child of $v_1$ in $ptree(A,v_2)$ and stops the traversal. This is because $v_0 \xrightarrow{X} v_5$ is already in the graph (Line 3, Algorithm 2). Thus, the redundant derivations of two edges $v_3 \xrightarrow{X} v_5$ and $v_4 \xrightarrow{X} v_5$ is avoided.

### 3.2.2 Dynamic Construction of Spanning Trees

Algorithm 2 utilizes the two properties of the spanning-tree model (presented in Section 3.2.1) to reduce the redundant derivations caused by singly recursive rules. The key step in our CFL-reachability algorithm is to update the spanning trees representing the transitive relations among nodes and maintain the two properties.

Adding an edge $v_i \xrightarrow{A} v_j$, where $A$ is transitive, to the graph changes $succ(A,v_k)$

and $pred(A,v'_k)$ for all $k \in pred(A,v_i) \cup \{v_i\}$ and $k' \in succ(A,v_j) \cup \{v_j\}$ via $succ(A,v_k) = succ(A,v_k) \cup succ(A,v_j)$ and $pred(A,v'_k) = pred(A,v'_k) \cup pred(A,v_i)$. Note that $pred(A,v_i)$ and $succ(A,v_j)$ take constant time during the edge addition. Thus, the updates of the spanning trees are similar to handling singly recursive rules.

We propose Algorithm 3, a dynamic construction algorithm, to update the predecessor and successor trees when solving CFL-reachability. Algorithm 3 consists of a main procedure `NewTrEdge` and three subprocedures `TravPtree`, `TravStree` and `Update`. Given an edge $v_{pt} \xrightarrow{A} v_{st}$ where $A$ is transitive, `TravPtree` and `TravStree` perform a nested traversal. Namely, `TravPtree` that traverses $ptree(A,v_{pt})$ is invoked from `TravStree` that traverses $stree(A,v_{st})$. During the nested traversal, `Update` updates the spanning trees and the adjacency lists of the visited nodes simultaneously, ensuring that for all $(v_k,v'_k) \in ptree(A,v_{pt}) \times stree(A,v_{st})$ : (1) $v_k \in ptree(A,v'_k)$ and $v'_k \in stree(A,v_k)$, and (2) $v_k \in pred(A,v'_k)$ and $v'_k \in succ(A,v_k)$.

We first illustrate Algorithm 3 using the following example:

*Example* 3.5 (*Dynamic Construction of Spanning Trees*). Adding a primary edge $v_2 \xrightarrow{A} v_3$ to Figure 3.3(a) means that for all $(v_k,v'_k) \in pred(A,v_2) \times succ(A,v_3)$, there will be $v_k \in ptree(A,v'_k)$ and $v'_k \in stree(A,v_k)$. `NewTrEdge` $(A,v_2,v_3)$ realizes this by traversing $ptree(A,v_2)$ and $stree(A,v_3)$, and updating the spanning trees (and adjacency lists simultaneously) as shown in Figure 3.4(a)–(d).

Initially, the traversals of $ptree(A,v_2)$ and $stree(A,v_3)$ start at their roots $v_2$ and $v_3$ respectively, as marked in red in Figure 3.4(a). In this step, $v_2$ is added to $ptree(A,v_3)$ as a child of $v_3$, and $v_3$ is added to $stree(A,v_2)$ as a child of $v_2$, as marked in blue in Figure 3.4(a). After this, the inner traversal of $ptree(A,v_2)$ visits $v_1$, a child of $v_2$, and the outer traversal of $stree(A,v_3)$ stays at $v_3$. This step adds $v_1$ to $ptree(A,v_3)$ as a child of $v_2$ and adds $v_3$ to $stree(A,v_1)$ as child of $v_2$, as shown in Figure 3.4(b). This can be viewed as copying the edge $v_2$ to $v_1$ from $ptree(A,v_2)$ to $ptree(A,v_3)$. Then the depth-first traversal of $ptree(A,v_2)$ continues, as shown in Figure 3.4(c), updating the spanning trees rooted

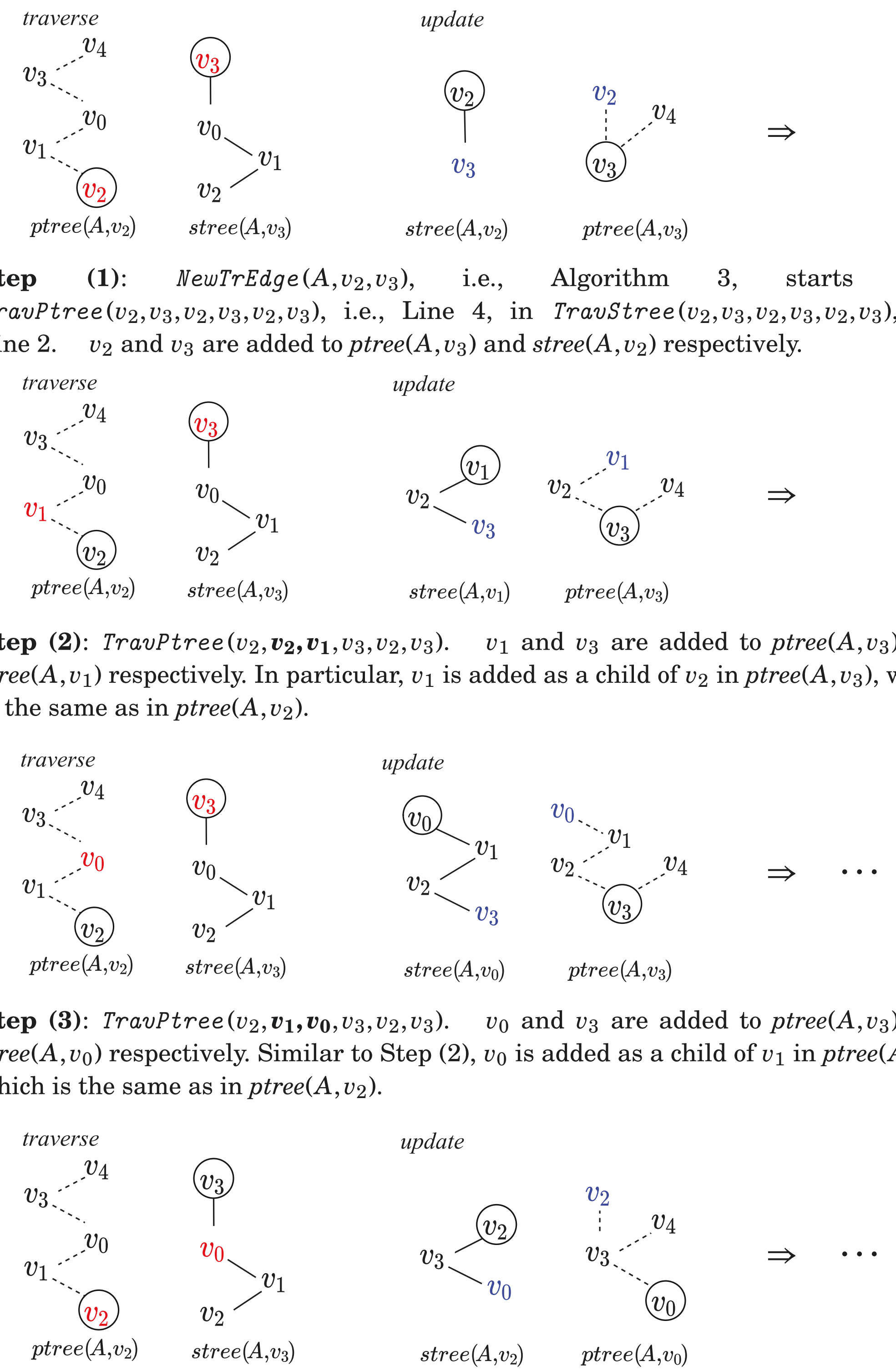
**Algorithm 3:** Dynamic construction of the spanning-tree model.

```
Function NewTrEdge($A, v_{pt}, v_{st}$)
    TravStree($A, v_{pt}, v_{st}, v_{pt}, v_{st}, v_{pt}, v_{st}$);
Procedure TravStree($A, v_{pt}, v_{px}, v_{py}, v_{st}, v_{sx}, v_{sy}$)
    TravPtree($A, v_{pt}, v_{px}, v_{py}, v_{st}, v_{sx}, v_{sy}$);
    for each $v_{sz}$ child of $v_{sy}$ in stree($A, v_{st}$) do
        if $v_{py} \xrightarrow{A} v_{sz} \notin E$ then
            TravStree($A, v_{pt}, v_{px}, v_{py}, v_{st}, v_{sy}, v_{sz}$);
Procedure TravPtree($A, v_{pt}, v_{px}, v_{py}, v_{st}, v_{sx}, v_{sy}$)
    Update($A, v_{px}, v_{py}, v_{sx}, v_{sy}$);
    for each $v_{pz}$ child of $v_{py}$ in ptree($A, v_{pt}$) do
        if $v_{pz} \xrightarrow{A} v_{sy} \notin E$ then
            TravPtree($A, v_{pt}, v_{py}, v_{pz}, v_{st}, v_{sx}, v_{sy}$);
Procedure Update($A, v_{px}, v_{py}, v_{sx}, v_{sy}$)
    add $v_{py} \xrightarrow{A} v_{sy}$ as a secondary edge to $E$ and to $W$;
    if $v_{py} \neq v_{sy}$ and $v_{py} \notin ptree(A, v_{sy})$ then
        add a new node $v_{py}$ pointed to by $v_{py} \in pred(A, v_{sy})$ to $ptree(A, v_{sy})$ as a
          child of $v_{px}$;
        add a new node $v_{sy}$ pointed to by $v_{sy} \in succ(A, v_{py})$ to $stree(A, v_{py})$ as a
          child of $v_{sx}$;
```

at the visited nodes until the traversal finishes.

After finishing the inner traversal of *ptree*$(A, v_2)$, the outer traversal of *stree*$(A, v_3)$ visits $v_0$, a child of $v_3$, and starts another inner traversal of *ptree*$(A, v_2)$, as Figure 3.4(d). The nested traversal terminates when for all $(v_k, v'_k) \in ptree(A, v_2) \times stree(A, v_3)$, $v_k \in ptree(A, v'_k)$ and $v'_k \in stree(A, v_k)$.

The nested traversals and edge updates of Algorithm 3 work as follows:



**Step (1):** $\texttt{NewTrEdge}(A,v_2,v_3)$, i.e., Algorithm 3, starts with $\texttt{TravPtree}(v_2,v_3,v_2,v_3,v_2,v_3)$, i.e., Line 4, in $\texttt{TravStree}(v_2,v_3,v_2,v_3,v_2,v_3)$, i.e., Line 2. $v_2$ and $v_3$ are added to $\mathit{ptree}(A,v_3)$ and $\mathit{stree}(A,v_2)$ respectively.

**Step (2):** $\texttt{TravPtree}(v_2,\boldsymbol{v_2},\boldsymbol{v_1},v_3,v_2,v_3)$. $v_1$ and $v_3$ are added to $\mathit{ptree}(A,v_3)$ and $\mathit{stree}(A,v_1)$ respectively. In particular, $v_1$ is added as a child of $v_2$ in $\mathit{ptree}(A,v_3)$, which is the same as in $\mathit{ptree}(A,v_2)$.

**Step (3):** $\texttt{TravPtree}(v_2,\boldsymbol{v_1},\boldsymbol{v_0},v_3,v_2,v_3)$. $v_0$ and $v_3$ are added to $\mathit{ptree}(A,v_3)$ and $\mathit{stree}(A,v_0)$ respectively. Similar to Step (2), $v_0$ is added as a child of $v_1$ in $\mathit{ptree}(A,v_3)$, which is the same as in $\mathit{ptree}(A,v_2)$.

**Step (6):** $\texttt{TravPtree}(v_2,v_3,v_2,v_3,\boldsymbol{v_3},\boldsymbol{v_0})$. After finishing the inner traversal of $\mathit{ptree}(A,v_2)$, the outer traversal of $\mathit{stree}(A,v_3)$ visits $v_0$. Similarly, $v_0$ is added as a child of $v_3$ in $\mathit{stree}(A,v_2)$, which is the same as in $\mathit{stree}(A,v_3)$.

**Figure 3.4.** Processing a new edge $v_2 \xrightarrow{A} v_3$ added to Figure 3.3(a) by $\texttt{NewTrEdge}(A,v_2,v_3)$, which traverses $\mathit{ptree}(A,v_2)$ and $\mathit{stree}(A,v_3)$, and updates the *ptrees* and *strees* of the visited nodes. In each step, nodes being visited and newly added are marked in red and blue, respectively.

**Nested depth-first traversal**. `TravPtree` and `TravStree` accept seven parameters:

$A$ - the label of the transitive edges;

$v_{pt}$ - the root of $ptree(A, v_{pt})$, i.e., the predecessor tree to be traversed;

$v_{st}$ - the root of $stree(A, v_{st})$, i.e., the successor tree to be traversed;

$v_{py}$ - the node currently visited in $ptree(A, v_{pt})$;

$v_{px}$ - the parent of $v_{py}$ in $ptree(A, v_{pt})$;

$v_{sy}$ - the node currently visited in $stree(A, v_{st})$;

$v_{sx}$ - the parent of $v_{sy}$ in $stree(A, v_{st})$.

With a new primary edge $v_{pt} \xrightarrow{A} v_{st}$ added to the graph, the traversal of $ptree(A, v_{pt})$ in `TravPtree` starts at its root $v_{pt}$ which does not have an actual parent. So the pseudo parent of $v_{pt}$ is set as $v_{st}$. Similarly, the traversal of $stree(A, v_{st})$ in `TravStree` starts at $v_{st}$ and the pseudo parent of $v_{st}$ is set as $v_{pt}$, as shown in Line 2.

The traversal of $ptree(A, v_{pt})$ is nested in the traversal of $stree(A, v_{st})$, as shown in Line 4. During the traversal of $ptree(A, v_{pt})$, nodes $v_{sx}$ and $v_{sy}$ relevant to the outer traversal remain constant. For each $v_{pz}$ child of $v_{py}$ in $ptree(A, v_{pt})$, the traversal steps from $v_{py}$ to $v_{pz}$ only if $v_{pz} \xrightarrow{A} v_{sy}$ is not in the graph, as shown in Lines 10–12. After finishing the inner traversal of $ptree(A, v_{pt})$, the outer traversal of $stree(A, v_{st})$ steps to $v_{sz}$ a child of $v_{sy}$ in $stree(A, v_{st})$ only if $v_{py} \xrightarrow{A} v_{sz}$ is not in the graph (Lines 5–7), and starts another inner traversal of $ptree(A, v_{pt})$ as shown in Line 4.

The nested depth-first traversal of Algorithm 3 is terminated when any one of the following two constraints is satisfied:

- all the nodes pairs $(v_k, v'_k) \in ptree(A, v_{pt}) \times stree(A, v_{st})$ are visited;
- all the attempts of visiting new tree nodes are stopped by Line 6 or Line 11.

**Updating spanning trees and adjacency lists**. When creating a new edge $v_{py} \xrightarrow{A} v_{sy}$ not in the graph, Algorithm 3 calls `Update`$(A, v_{px}, v_{py}, v_{sx}, v_{sy})$ at Line 9 to update the spanning trees and the adjacency lists simultaneously. `Update` first adds $v_{py} \xrightarrow{A} v_{sy}$ to the graph at Line 14. In particular, the edge $v_{py} \xrightarrow{A} v_{sy}$ is marked as a "secondary" edge so that it will not be processed by the future calls of $\mathit{NewTrEdge}$. The updates of $\mathit{ptree}(A, v_{sy})$ and $\mathit{stree}(A, v_{py})$ by Lines 15–17 can be viewed as follows: (1) copying the edge from $v_{px}$ to $v_{py}$ in $\mathit{ptree}(A, v_{pt})$ and attaching the copy to the node $v_{px}$ in $\mathit{ptree}(A, v_{sy})$, and (2) copying the edge from $v_{sx}$ to $v_{sy}$ in $\mathit{stree}(A, v_{st})$ and attaching the copy to the node $v_{sx}$ in $\mathit{stree}(A, v_{py})$. To avoid redundant tree edges, the above two steps are performed only if $v_{py} \neq v_{sy}$ and $v_{py} \notin \mathit{ptree}(A, v_{sy})$. The simultaneous updates of spanning trees and adjacency lists of $\mathit{Update}$ maintains the two properties of the spanning-tree model in our hybrid graph representation (Section 3.2.1).

Executing Algorithm 3 can be viewed as updating $\mathit{stree}(A, v_k)$ and $\mathit{ptree}(A, v'_k)$ for all $(v_k, v'_k) \in \mathit{pred}(A, v_{pt}) \times \mathit{succ}(A, v_{st})$ using the following two steps: (1) pruning a copy of $\mathit{stree}(A, v_{st})$ (resp. $\mathit{ptree}(A, v_{pt})$) by eliminating the nodes already in $\mathit{stree}(A, v_k)$ (resp. $\mathit{ptree}(A, v'_k)$); and (2) attaching the pruned copy to the node $v_{pt}$ (resp. $v_{st}$) in $\mathit{stree}(A, v_k)$ (resp. $\mathit{ptree}(A, v'_k)$). It is almost obvious that using Algorithm 3 to process the primary $A$-edges maintains the transitive closure of the relation $A$. Similar to Algorithm 2, we have the following property for Algorithm 3:

**Soundness of Algorithm 3.** In Algorithm 1, replacing $\mathit{CheckPred}(A, A, v_i, v_j)$ and $\mathit{CheckSucc}(A, A, v_i, v_j)$ by $\mathit{NewTrEdge}(A, v_i, v_j)$ for all primary edges $v_i \xrightarrow{A} v_j$ and omitting $\mathit{CheckPred}(A, A, v_i, v_j)$ and $\mathit{CheckSucc}(A, A, v_i, v_j)$ for all secondary edges $v_i \xrightarrow{A} v_j$ maintains the same CFL-reachability results with respect to the original algorithm.

Notably, $\mathit{CheckPred}(A, v_i, v_j)$ and $\mathit{CheckSucc}(A, v_i, v_j)$ can be omitted while processing a secondary edge $v_i \xrightarrow{A} v_j$ because all the secondary edges are processed in the calls of $\mathit{NewTrEdge}$ for processing primary edges.

Algorithm 3 avoids redundancy by the termination criteria at Line 6 and Line 11. In Line 6, $v_{py} \xrightarrow{A} v_{sz} \in E$ avoids the repeated traversal of $v'_{sz}$ descendants of $v_{sz}$ in $\mathit{stree}(A, v_{st})$. Similarly, in Line 11, $v_{pz} \xrightarrow{A} v_{sy} \in E$ avoids the repeated traversal of $v'_{pz}$

descendants of $v_{pz}$ in $ptree(A, v_{pt})$.

### 3.2.3 POCR: A Fast Partially Ordered CFL-Reachability Algorithm for All-Pairs Analyses

With Algorithms 2 and 3 running on top of the hybrid graph representation discussed in Section 3.2.1, we propose POCR, a fast partially ordered CFL-reachability algorithm for all-pairs analysis, as given in Algorithm 4.

POCR consists of two parts: initialization (Lines 2–5) and solving reachability (Lines 6–20). Initialization first follows the initialization scheme of the standard algorithm (Lines 11–15, Algorithm 1) to initialize the graph and the worklist. Then, it initializes the predecessor tree and successor tree for each node $v_i \in V$ (Lines 3–5). Solving reachability also follows the strategy of the standard algorithm, i.e., iteratively solving the edges in the worklist and adding new edges to the graph and to the worklist until no new edges can be added to the graph.

POCR differs from the standard algorithm in handling edge derivations based on singly or doubly recursive rules. It uses *NewTrEdge* to deal with edge derivations based on doubly recursive rules (Line 9) and replaces *CheckPred* (resp. *CheckSucc*) by *TravPtree* (resp. *TravStree*) to create edges based on singly recursive rules when the derivation needs to traverse the predecessors or successors of nodes associated with transitive relations (Lines 15 and 19). In particular, all transitive edges created during CFL-reachability solving are marked "**primary**" by default except for those created in *NewTrEdge* (Line 14, Algorithm 3), and *NewTrEdge* only accepts primary edges. The secondary edges created in *NewTrEdge* are further used to create new edges based on non-doubly recursive rules in the subsequent procedures (Lines 10–20, Algorithm 4).

In summary, when processing an edge $v_i \xrightarrow{Y} v_j$, POCR only differs from the standard algorithm in the following two aspects:

**Algorithm 4:** POCR: partially ordered CFL-reachability algorithm.

```
Function POCR(CFG,G)
    init();                                              /* Lines 11-15, Algorithm 1 */
    for each transitive relation $A$ do
        for each node $v_i \in V$ do
            set both $ptree(A,v_i)$ and $stree(A,v_i)$ a tree containing a single node $v_i$ as the
              root.
    while $W \neq \emptyset$ do
        select and remove an element $v_i \xrightarrow{Y} v_j$ from $W$;
        if $v_i \xrightarrow{Y} v_j$ is a primary edge then
            NewTrEdge($Y,v_i,v_j$);                                     /* Algorithm 3 */
        else
            for each production $X ::= Y \in P$ do
                if $v_i \xrightarrow{X} v_j \notin E$ then  add $v_i \xrightarrow{X} v_j$ to $E$ and to $W$;
            for each $X ::= Y\ Z \in P$ and $\neg(X = Y = Z)$ do
                if $Z$ is transitive and $X = Y$ then
                    CheckStree($X,Z,v_i,v_j,v_j$);                      /* Algorithm 2 */
                else CheckSucc($X,Z,v_i,v_j$);           /* Lines 19-21, Algorithm 1 */
            for each $X ::= Z\ Y \in P$ and $\neg(X = Y = Z)$ do
                if $Z$ is transitive and $X = Y$ then
                    CheckPtree($X,Z,v_i,v_j,v_i$);                      /* Algorithm 2 */
                else CheckPred($X,Z,v_i,v_j$);           /* Lines 16-18, Algorithm 1 */
```

1. replacing $\mathtt{CheckPred}(X,A,v_i,v_j)$ (resp. $\mathtt{CheckSucc}(X,A,v_i,v_j)$) with $\mathtt{CheckPtree}(X, A,v_i,v_j,v_i)$ (resp. $\mathtt{CheckStree}(X,A,v_i,v_j,v_j)$) when $A$ is transitive and $X = Y$;

2. replacing $\mathtt{CheckPred}(Y,Y,v_i,v_j)$ (resp. $\mathtt{CheckSucc}(Y,Y,v_i,v_j)$) with $\mathtt{NewTrEdge}(Y,v_i, v_j)$ if $v_i \xrightarrow{Y} v_j$ is primary and omitting $\mathtt{CheckPred}(Y,Y,v_i,v_j)$ (resp. $\mathtt{CheckSucc}(Y,Y,v_i, v_j)$) if $v_i \xrightarrow{Y} v_j$ is secondary.

According to the properties of Algorithms 2 and 3, these replacements do not change the CFL-reachability solution. Namely, for a CFL-reachability problem, POCR produces an identical solution to the standard algorithm.

## 3.3 Discussion: Effectiveness of POCR

As demonstrated in Section 3.2, POCR uses ordered derivations based on the spanning trees to reduce redundant derivations for singly and doubly recursive rules. It is called "partially ordered" because it does not order the edge derivations through the production rules where the right-hand side has no transitive relation. Hence, the redundant derivations caused by such production rules are not eliminated by POCR. In particular, given $X ::= Y\ Z \in P$, two paths $v_1 \xrightarrow{Y} v_2 \xrightarrow{Z} v_3$ and $v_1 \xrightarrow{Y} v_4 \xrightarrow{Z} v_3$ cause one redundant derivation of $v_1 \xrightarrow{X} v_3$, which is not transitive redundancy and is not handled by POCR. The effectiveness of POCR's redundancy elimination is also related to the CFG of a particular CFL-reachability problem. We further discuss two aspects of this issue as below.

### 3.3.1 Grammars Benefiting from POCR

A comparison of Algorithm 4 and Algorithm 1 shows that for a CFL-reachability problem where there is no transitive relation, the part of POCR which solves reachability is identical to that of Algorithm 1, because Lines 8–9, 14–15 and 18–19 of Algorithm 4 will never be executed. Therefore, POCR does not benefit the problems where there are no transitive relations in the CFG.

However, in some CFL-reachability problems, the transitive relations are implicit. We can rewrite the input grammars in CFL-reachability to utilize POCR. For example, the following grammar

$$A ::= a\ B \mid \varepsilon$$
$$B ::= b\ A$$

can be rewritten as $A ::= A\ A \mid a\ b \mid \varepsilon$ to be used by POCR. Conversely, for a CFL-reachability problem that has no transitive relation, i.e., its CFG cannot be rewritten to obtain any doubly recursive rule, we assume that it does not have transitive redundancy.

Moreover, some grammars containing doubly recursive rules can be rewritten using only left- or right-recursive rules. For example, the CFG in Figure 3.2(a) can be rewritten into

$$F_i ::= F_i\ a \mid f_i.$$

Given this modified grammar, there will be no $A$-edge in the graph, and we will not have the redundant derivations caused by the four cases in Figure 3.2(d). However, transitive relations are prevalent in many popular real-world CFL-reachability problems, and the input graphs are typically large. Existing techniques [86, 133] still suffer from a large number of redundant derivations while solving with such modified grammars because they do not exploit an effective computation order. This is also confirmed in our experiment (Section 3.4.4).

### 3.3.2 Grammar-Driven Redundancy Reduction

Here we provide an optimization of POCR to further improve its efficiency given any grammar with the following three properties:

1. the grammar has one or more transitive relations;
2. for any production rule $X ::= Y\ A \in P$ and $X ::= A\ Y \in P$ where $A$ is transitive, $X = Y$;
3. there is no $X ::= A \in P$ where $A$ is transitive and $X \neq A$.

For such type of grammars, while processing a primary edge $v_i \xrightarrow{A} v_j$ (Line 9, Algorithm 4), we do not add the secondary edges created in *NewTrEdge* to the worklist as in Line 14 of Algorithm 3 (we still add such edges to the graph). Instead, we only add $v_i \xrightarrow{A} v_j$ itself as a secondary edge to the worklist. The insight is to use *CheckPtree* and *CheckStree* to deal with derivations based on as many singly recursive rules as possible.

We briefly demonstrate the feasibility of the optimized POCR in handling grammars with the aforementioned three properties. For singly recursive rules, we only discuss $X ::= X\ A$ because $X ::= A\ X$ is similar. In a CFL-reachability instance where $X ::= X\ A \in P$ and $A$ is transitive, adding a primary $A$-edge $v_i \xrightarrow{A} v_j$ to the graph results in $v_l \xrightarrow{X} v_k \in E$ for all $(v_l,v_k) \in \mathit{pred}(X,v_i) \times \mathit{stree}(A,v_j)$. In the original POCR, *NewTrEdge*$(A,v_i,v_j)$ is first called to create and add all the new secondary edges $v'_k \xrightarrow{A} v_k$ to the graph, where $(v'_k,v_k) \in \mathit{ptree}(A,v_i) \times \mathit{stree}(A,v_j)$. Then *CheckPred*$(X,X,v'_k,v_k)$ is called to process each $v'_k \xrightarrow{A} v_k$ in the worklist (Line 20, Algorithm 4) so that for all $v_l \in \mathit{pred}(X,v'_k)$, $v_l \xrightarrow{X} v_k \in E$. Because $X ::= X\ A$, for all $v_l \in \mathit{pred}(X,v'_k)$, $v_l \in \mathit{pred}(X,v_i)$. Hence, all $v_l \xrightarrow{X} v_k$ such that $(v_l,v_k) \in \mathit{pred}(X,v_i) \times \mathit{stree}(A,v_j)$ are added to the graph.

The optimized POCR handles this in a different way. First, *NewTrEdge*$(Y,v_i,v_j)$ only adds $v_i \xrightarrow{A} v_j$ as a secondary edge to the graph and to the worklist. Then for each $X ::= X\ A \in P$, *CheckPred*$(X,X,v_i,v_j)$ processes $v_i \xrightarrow{A} v_j$, adding all the new $v_l \xrightarrow{X} v_j$ such that $v_l \in \mathit{pred}(X,v_i)$ to the graph and to the worklist. Processing all the $v_l \xrightarrow{X} v_j$ in the worklist via *CheckStree*$(X,A,v_l,v_j,v_j)$ results in $v_l \xrightarrow{X} v_k \in E$ for all $(v_l,v_k) \in \mathit{pred}(X,v_i) \times \mathit{stree}(A,v_j)$.

The original POCR and the optimized POCR obtain the same results when dealing with $X ::= X\ A$. Similarly, they also obtain the same results when dealing with $X ::= A\ X$. Furthermore, the three properties listed at the beginning of the subsection mean that there is no need to process $X ::= A$, $X ::= Y\ A$ and $X ::= A\ Y$ where $X \neq Y$ for all transitive $A$. Therefore, the optimized POCR is applicable to CFL-reachability whose grammar has the three properties.

Unlike the original POCR, *CheckPred* and *CheckSucc* in the optimized POCR are only called to process the input primary edges of *NewTrEdge*. Most of the derivations based on singly recursive rules are done by *CheckPtree* and *CheckStree*, instead of *CheckPred* and *CheckSucc*. Note that *CheckPtree* and *CheckStree* reduce redundant derivations whereas *CheckPred* and *CheckSucc* do not. So the optimized POCR further improves the efficiency of solving CFL-reachability. It is also interesting to note that there is a large variety of real-world static analyses that can be formulated into CFL-reachability problems whose grammars have the aforementioned three properties, e.g., dataflow/valueflow analysis [105], typestate analysis [132], alias analysis [149], etc. Thus, this grammar-driven optimization is worth incorporating.

## 3.4 Experimental Evaluation

In this section, we evaluate the performance of POCR by applying it to two popular static analyses for C/C++: context-sensitive value-flow analysis [126] and field-sensitive alias analysis [149], where transitive redundancy dominates. In our experiment, we use cycle elimination [92] and variable substitution [111] for offline processing of the graphs abstracts from the benchmark programs. The baseline of our experiment is the standard CFL-reachability algorithm [86] accepting the preprocessed input graphs. We also compare our approach with two open-source CFL-reachability/Datalog tools, Graspan [133] (a multi-thread disk-based CFL-reachability solver which uses old-new edge sets to avoid a proportion of redundant computation) and Soufflé [68] (a Datalog analyzer which is able to automatically generate a CFL-reachability solver and compute the result when the grammar and graph are provided.). We choose to compare with the two tools because they are the most recent and integrate the state-of-the-art techniques for CFL-reachability. We perform all-pairs CFL-reachability analysis in POCR and all the baselines for both clients.

Our experiments aim to answer the following research questions:

*RQ* 1. How many redundant derivations can POCR reduce in real-world CFL-reachability problems based on the two popular clients?

*RQ* 2. How is the performance of CFL-reachability improved by eliminating transitive redundancy via POCR?

*RQ* 3. How about the performance of POCR when comparing it with the grammar rewriting method which removes doubly recursive rules from the grammar?

### 3.4.1 Experimental Setup

We have conducted our experiment on a platform consisting of an eight-core 2.60GHz Intel Xeon CPU with 128 GB memory, running Ubuntu 18.04.

**Value-flow analysis** Our context-sensitive value-flow analysis is conducted on the sparse value-flow graphs (SVFG) [126]. We use the context-free grammar (CFG) in Figure 3.5 for the value-flow analysis, where "$call_i$" and "$ret_i$" denote, respectively, a call and a return with a callsite index $i$, "$a$" denotes an assignment instruction, and "$A$" denotes a value flow. Note that the grammar in Figure 3.5 only considers context-sensitivity. However, each field object is represented as a single node in the field-sensitive SVFG, so the analysis is also field-sensitive.

**Alias analysis** The CFG for C/C++ field-sensitive alias analysis is listed in Figure 3.6, which is from [149]. In the grammar, $a$ denotes an assignment, $d$ denotes a pointer dereference, $f_i$ denotes the address of the $i$-th field, $A$ denotes a value flow, $M$ denotes memory aliasing, and $V$ denotes value aliasing. The alias analysis is performed on the program expression graph (PEG), which is bi-directed, i.e., for each edge $v_i \xrightarrow{X} v_j \in E$, there is a reverse edge $v_j \xrightarrow{\overline{X}} v_i \in E$.

$$A ::= A\ A \mid call_i\ A\ ret_i \mid a \mid \varepsilon$$

**(a)** Context-free grammar.

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

**(b)** Normalized grammar.

**Figure 3.5.** CFG for context-sensitive value-flow analysis.

$$\begin{aligned} M &::= \overline{d}\ V\ d \\ V &::= \overline{A}\ V\ A \mid \overline{f_i}\ V\ f_i \mid M \mid \varepsilon \\ A &::= A\ A \mid a\ M? \mid \varepsilon \\ \overline{A} &::= \overline{A}\ \overline{A} \mid M?\ \overline{a} \mid \varepsilon \end{aligned}$$

**(a)** Context-free grammar.

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

**(b)** Normalized grammar.

**Figure 3.6.** CFG for field-sensitive alias analysis.

**Setup and Benchmarks** The SVFG and PEG of each program are constructed from the bitcode files compiled by Clang-12.0.0 and linked via `WLLVM` [129] for whole-program all-pairs CFL-reachability analysis. The SVFG and PEG are preprocessed by cycle elimination [127] which merges cycles comprised of $a$-edges and variable substitution [111] which contracts particular $a$-edges. The preprocessing is to make sure the input graph is compacted after applying the existing offline approach to make sure the input does not favor our online approach undesirably. We used 10 SPEC 2017 C/C++ programs for our evaluation. We did not include three small programs (`lbm`, `mcf` and `deepsjeng` whose sizes are less than 1 MB). The other three C/C++ programs in SPEC 2017, i.e., `xalancbmk`, `gcc` and `blender`, failed to be linked by `wllvm`. Table 3.1 lists the size and graph statistics of each program.

**Implementation** We have implemented our POCR on top of the LLVM compiler, and the sub-project SVF [124]. To study the relationship between the improvement of performance and the reduction of redundant derivations, we also implement the standard algorithm [86] and the edge-reduction technique described in the paper of Graspan (Section 4.2 in [133]) on top of SVF. Tables 3.2 and 3.3 list the main results of our experiments, where "Base" and "GSA" denote, respectively, the standard algorithm and the edge-reduction algorithm of Graspan. "POCR" denotes our approach with optimization

**Table 3.1.** Benchmark info. #Node and #Edge respectively denote the number of nodes and edges in the initial graphs.

| **Benchmark** | **Size(MB)** | **SVFG** | | **PEG** | | **Description** |
|---|---|---|---|---|---|---|
| | | **#Node** | **#Edge** | **#Node** | **#Edge** | |
| xz | 1.24 | 51666 | 65235 | 12425 | 26468 | General data compression |
| nab | 1.41 | 59253 | 76105 | 16261 | 34676 | Molecular dynamics |
| leela | 2.93 | 68250 | 92865 | 22186 | 49748 | Monte Carlo tree search (Go) |
| x264 | 4.68 | 213943 | 347142 | 60956 | 136352 | Video compression |
| cactus | 5.88 | 563208 | 1026726 | 93557 | 212478 | Physics: relativity |
| povray | 7.38 | 555807 | 1059724 | 76405 | 174258 | Ray tracing |
| imagick | 13.68 | 601687 | 870107 | 119314 | 301846 | Image manipulation |
| parest | 16.20 | 325592 | 433217 | 117500 | 251436 | Biomedical imaging |
| perlbench | 18.69 | 1031348 | 2203010 | 156664 | 388564 | Perl interpreter |
| omnetpp | 21.81 | 703952 | 1897474 | 241916 | 509166 | Discrete Event simulation |

(Section 3.3.2) enabled. The columns “Graspan” and “Soufflé” list the results of the open-source tools Graspan [134] and Soufflé [67], running with their default configurations.[2] In particular, when running the mult-thread solver Graspan, 8 threads were used, which aligns with [133]. The cases which failed to obtain results due to the time constraint (24 hours) are annotated with “–”.

### 3.4.2 RQ 1: Reduction of Redundant Derivations

The numbers of edges added to the graph and the numbers of edges created during CFL-reachabilty solving of each benchmark are listed in Column 1 and Columns 2–4 of Tables 3.2 and 3.3. The number of redundant derivations is obtained via (#Deriv - #Add). The proportion of redundant derivations is computed via (#Deriv - #Add) / #Deriv.

In general, with regard to the standard CFL-reachability algorithm (listed in Col-

[2] We refer to [133] and `https://souffle-lang.github.io/build#cmake-configuration-options` for the default configurations of Graspan and Soufflé, respectively.

umn 2 of Tables 3.2 and 3.3), redundant derivations are prevalent in both value-flow analysis and alias analysis. The average proportions for redundant derivations out of all derivations are 98.03% and 97.89%, respectively, in value-flow analysis and alias analysis when using the standard algorithm.

A comparison of Columns #Deriv(k) in Tables 3.2 and 3.3 shows that both GSA (the edge-reduction technique of Graspan) and our POCR reduce redundant derivations but POCR is more effective. The reduction rates of the redundant derivations are listed in Columns Reductions(%) of the two tables, showing that, on average, POCR reduces 98.50% and 97.26% of redundant derivations respectively for value-flow analysis and alias analysis, which is much more than GSA. We also observe that the redundant derivations of POCR are only 4.67% and 9.68% that of GSA respectively for value-flow analysis and alias analysis. Namely, the redundant derivations of POCR are much fewer than GSA. Interestingly, the set of redundant derivations captured by POCR is not always a superset of those captured by Graspan because Graspan can also cover some non-transitive relations. However, we can infer from the experiment results that the redundancy caused by non-transitive relations has only a marginal impact on performance. Our approach, which exploits a proper derivation order based on the property of transitive relations, is much more effective in reducing redundant derivations than Graspan. We did not include the numbers of derivations of Soufflé in the tables because it does not provide an option to collect those numbers directly.

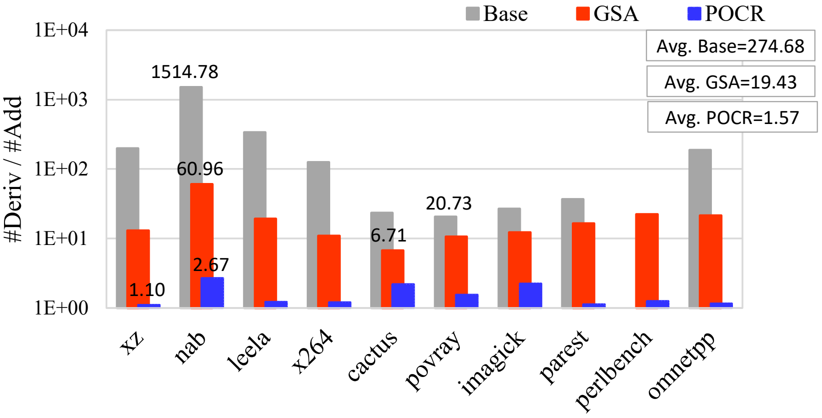
We show the computational redundancy of the three approaches in Figure 3.7, which is based on the numbers in Columns #Add and #Deriv of Tables 3.2 and 3.3. Specifically, the redundancy of each approach is represented by (#Deriv / #Add). The data shows how many derivations are needed to actually add an edge to the graph on average. From Figure 3.7(a) and Figure 3.7(b), we can see that there is a substantial amount of redundancy in the standard algorithm in both value-flow analysis and alias analysis. The redundancy in the value-flow analysis is more significant as most of its reachability

**Table 3.2.** Result of context-sensitive value-flow analysis. #Add(k) and #Deriv(k) denotes the number of edges added to the graph and created when solving CFL-reachability, measured in thousands. Reduction(%) denotes the reduction rate of redundant derivations of GSA and POCR. Time(s) denotes the runtime of each approach, measured in seconds. The baselines of both Reduction(%) and speedup are the columns "Base".

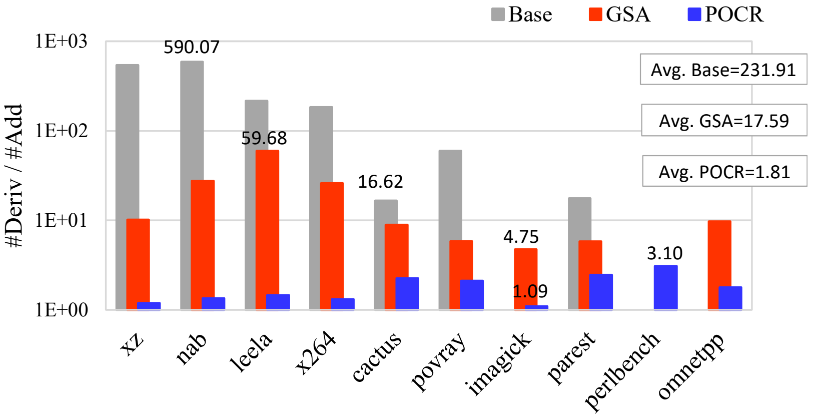
| **Benchmark** | **#Add(k)** | **#Deriv(k)** | | | **Reduction(%)** | | **Time(s)** | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **Base** | **GSA** | **POCR** | **GSA** | **POCR** | **Base** | **Graspan** | **Soufflé** | **POCR** |
| xz | 732 | 145140 | 9544 | 807 | 93.90 | 99.95 | 11.86 | 2.43 | 4.31 | 0.88 |
| nab | 1341 | 2031555 | 81757 | 3582 | 96.04 | 99.89 | 662.73 | 69.92 | 71.78 | 39.60 |
| leela | 1518 | 513825 | 29330 | 1846 | 94.57 | 99.94 | 30.22 | 6.62 | 12.58 | 2.04 |
| x264 | 60441 | 7537094 | 657053 | 72668 | 92.02 | 99.84 | 4915.96 | 656.69 | 1495.35 | 299.29 |
| cactus | 105114 | 2465484 | 705243 | 229347 | 74.57 | 94.74 | 42121.28 | 6285.94 | 6714.28 | 1754.60 |
| povray | 182537 | 3783718 | 1943433 | 281611 | 51.10 | 97.25 | 77651.60 | 9149.52 | 12796.00 | 3893.08 |
| imagick | 55561 | 1493419 | 683175 | 123044 | 56.35 | 95.31 | 31210.60 | 3645.31 | 5634.66 | 1043.20 |
| parest | 29749 | 1090633 | 491326 | 33441 | 56.49 | 99.65 | 14419.60 | 3746.12 | 2265.00 | 433.20 |
| perlbench | 834251 | - | 18662423 | 1042244 | - | - | - | 63737.24 | - | 13962.88 |
| omnetpp | 262480 | 49378518 | 5648505 | 302842 | 89.03 | 99.92 | 58454.40 | 9630.63 | 5496.90 | 2367.20 |
| *Mean* | | | | | 78.23 | 98.50 | | | *Speedup* | 21.48× |

**Table 3.3.** Result of field-sensitive alias analysis. #Add(k) and #Deriv(k) denotes the number of edges added to the graph and created when solving CFL-reachability, measured in thousands. Reduction(%) denotes the reduction rate of redundant derivations of GSA and POCR. Time(s) denotes the runtime of each approach, measured in seconds. The baselines of both Reduction(%) and speedup are the columns "Base".

| **Benchmark** | **#Add(k)** | **#Deriv(k)** | | | **Reduction(%)** | | **Time(s)** | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | **Base** | **GSA** | **POCR** | **GSA** | **POCR** | **Base** | **Graspan** | **Soufflé** | **POCR** |
| xz | 427 | 230356 | 4332 | 507 | 98.30 | 99.97 | 2.50 | 1.71 | 2.04 | 0.24 |
| nab | 472 | 278736 | 13017 | 635 | 95.49 | 99.94 | 3.40 | 1.47 | 3.00 | 0.38 |
| leela | 8797 | 1904077 | 524949 | 12875 | 72.77 | 99.78 | 340.52 | 38.26 | 41.72 | 13.96 |
| x264 | 13167 | 2413368 | 341681 | 17290 | 86.31 | 99.83 | 537.85 | 140.11 | 112.12 | 25.65 |
| cactus | 81832 | 1360390 | 727621 | 184821 | 49.49 | 91.94 | 11156.20 | 1479.63 | 1039.05 | 591.39 |
| povray | 53698 | 3214220 | 313985 | 112872 | 91.76 | 98.13 | 17601.30 | 1393.13 | 1238.86 | 631.86 |
| imagick | 422916 | - | 2007956 | 462378 | - | - | - | 5733.14 | 4076.94 | 1309.51 |
| parest | 83800 | 1472666 | 485888 | 205702 | 71.05 | 91.22 | 15337.50 | 943.52 | 1713.89 | 604.04 |
| perlbench | 1226586 | - | 24686471 | 3797919 | - | - | - | 29548.05 | 15536.13 | 5400.19 |
| omnetpp | 485066 | - | 4721535 | 866541 | - | - | - | 14660.59 | 11235.29 | 1842.43 |
| *Mean* | | | | | 80.74 | 97.26 | | | *Speedup* | 19.57× |



**(a)** Value-flow analysis.



**(b)** Alias analysis.

**Figure 3.7.** The computational redundancy of the three approaches in solving the two clients. The value is computed by (#Deriv / #Add). The vertical axis is logarithmic. The peak, valley and average values of each approach are marked in the charts.

relations are transitive (i.e., the $A$-relations in the grammar of Figure 3.5).

A comparison between POCR and GSA in Figure 3.7 demonstrates that: (1) POCR is much more effective than GSA in reducing computational redundancy. The average values of the redundancy are only 1.57 and 1.81, respectively, in value-flow analysis and alias analysis, which is much smaller than that of GSA. (2) Even in the worst cases, POCR keeps the computational redundancy in very low values, with the largest ones being only 1.54 and 3.10, respectively, in value-flow analysis and alias analysis. A comparison of Figure 3.7(a) and Figure 3.7(b) shows that the performance of POCR is slightly better for the value-flow analysis than the alias analysis. This is because there are more non-transitive relations in alias analysis than those in value-flow analysis.

Another interesting observation is the significance of the grammar-based optimization in POCR. We also run the original POCR on the two clients and compare the results with the optimized ones in Tables 3.2 and 3.3. We find that compared with the original POCR, the optimized POCR further reduces 78.72% and 83.56% redundant derivations, respectively, for the two clients on average. Therefore, when handling real-world problems, establishing a proper optimization (e.g. Section 3.3.2) to exploit the benefit of spanning trees as much as possible is important to boost the performance of POCR.

### 3.4.3 RQ 2: Speedups Over Baselines

Columns 7–10 of Tables 3.2 and 3.3 list the runtime of the four approaches for the two clients. We first focus on columns 7 and 10. A comparison between Base and POCR shows that POCR significantly accelerates the analyses for both value-flow analysis and alias analysis by eliminating transitive redundancy. The largest speedups of POCR over the standard algorithm occur in `parest` and `povray`, which accords with relatively large reduction rates of computational redundancy in Figure 3.7. The largest speedups do not occur in the programs with the largest reduction rates of derivations because the preprocessing time (e.g., constructing graphs and performing graph simplification) is also included in the total runtime of POCR. Such preprocessing time can take a large percent of runtime when handing small programs where the largest reduction rates of derivations occur.

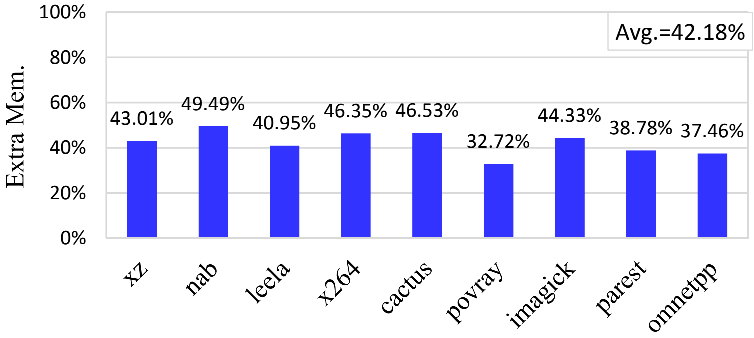
By reducing a large portion of redundant derivations, POCR successfully solves the value-flow analysis for `perlbench` and the alias analyses for `imagick`, `perlbench` and `omnetpp`, where the standard algorithm fails to solve within the time limit (24 hours). Therefore, taming redundant derivations plays an important role in scaling

CFL-reachability analysis.

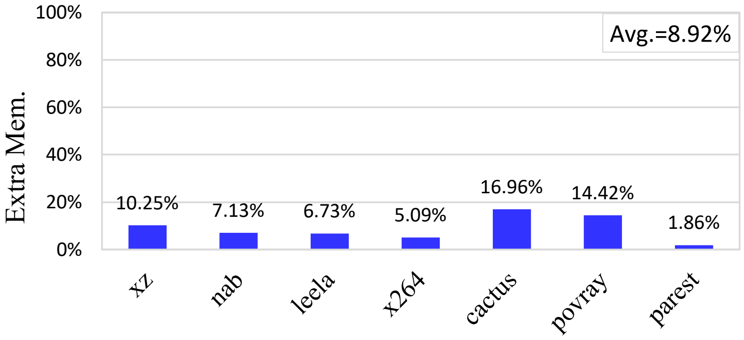
It is also interesting to note that the time consumed by an analysis depends not only on the program size but also on the features of the graph abstracted from the program. We found that `perlbench`, though not the largest graph, has a complex graph structure, including extremely long value-flow chains and dynamically/incrementally formed large transitive cycles. For example, in the PEG of `perlbench` after CFL solving, 83% of the total nodes are in cycles consisting of "A"-edges, with the largest cycle containing over 43k nodes and the longest simple path (without any cycles) containing 3k nodes, which are larger than those of other programs. This feature makes other CFL-reachability solvers incur much more transitive redundancy while solving, which makes `perlbench` arguably the most challenging program to solve.

To compare the analysis time, we further perform the experiments by running the open-source Graspan and Soufflé tools using their default configurations. Their results are listed in Columns 8 and 9 of Table 3.2 and Table 3.3. A comparison of Columns 8–10 of Tables 3.2 and 3.3 shows that Graspan and Soufflé can effectively accelerate the two clients. Moreover, POCR is much more efficient. On average, POCR is 3.67× and 4.10× faster than Graspan and Soufflé for value-flow analysis, and is 3.73× and 4.19× faster than Graspan and Soufflé for alias analysis, respectively.

As POCR uses spanning trees to store transitive edges, we study the extra memory overhead of POCR over the standard algorithm as shown in Figure 3.8. A comparison between Figure 3.8(a) and Figure 3.8(b) shows that the extra memory overheads of POCR in value-flow analysis are much larger than those in alias analysis. This reflects the characteristics of the two clients: in value-flow analysis, most reachability relations are transitive relations (the $A$-relations in Figure 3.5). In alias analysis, there is a smaller proportion of transitive relations (the $A$-relations in Figure 3.6), whereas the alias relations (the $V$-relations in Figure 3.6) are more prevalent.



**(a)** Value-flow analysis.



**(b)** Alias analysis.

**Figure 3.8.** Extra memory overhead of POCR over the standard algorithm. Only the benchmarks successfully solved by the standard algorithm are considered.

### 3.4.4 RQ 3: POCR vs. Grammar Rewriting

As addressed in Section 3.3.1, some grammars can be rewritten to eliminate transitive relations while maintaining correct solutions. The grammars in Figure 3.5 and Figure 3.6 of our example can be rewritten into Figure 3.9(a) and Figure 3.9(b) with the doubly-recursive rules $A ::= A\ A$ and $\overline{A} ::= \overline{A}\ \overline{A}$ removed. The modified grammar still has the ability to compute the required relations (i.e., $A$ for value-flow analysis and $V$ for alias analysis).

We study the performance impact of such grammar rewriting and compare it with POCR. The experimental results of the modified grammars are shown in Figures 3.10 and 3.11, in which the reduction rate and speedup of each approach are computed based

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

**(a)** Modified grammar for context-sensitive value-flow analysis.

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

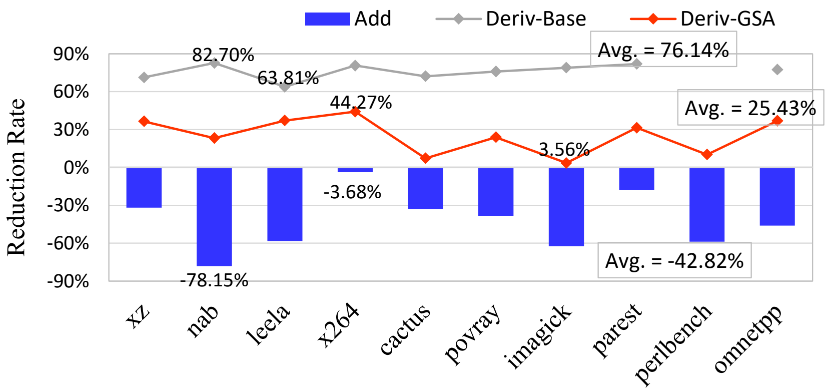
**(b)** Modified grammar for field-sensitive alias analysis.

**Figure 3.9.** CFG modified from Figure 3.6 by removing the doubly recursive rules.

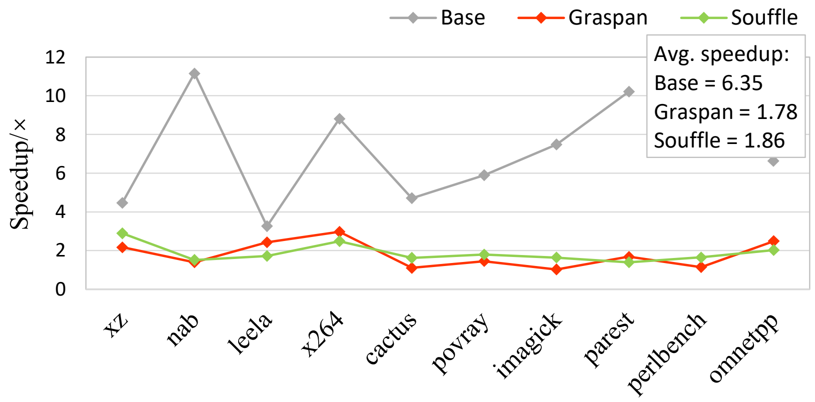
on the result obtained from the original grammar. POCR is not taken into consideration as it does not benefit the modified grammar.

With respect to value-flow analysis (Figure 3.10), the values of the reduction rate of the added edges are negative. This is because the modified grammar in Figure 3.9(a) introduces an extra non-terminal $B$, leading to extra edges added to the graphs. However, the reduction rate of derivations of the standard algorithm is large because of the removed doubly-recursive rules. In contrast, the reduction rate of derivations of GSA is low because it has already reduced redundant derivations for the original grammar. Accordingly, the standard algorithm is accelerated much more than the other two techniques by grammar rewriting as shown in Figure 3.10(b). Regarding alias analysis (Figure 3.11), the total numbers of added edges are reduced because the modified grammar in Figure 3.9(b) removes doubly recursive rules and does not introduce any extra non-terminal. However, the reduction rates of derivations of both the standard algorithm and GSA are low. This confirms with the earlier observation that $A$- and $\overline{A}$-edges only take a small proportion in alias analysis. A comparison of Figure 3.10(b) and Figure 3.11(b) shows that the accelerations brought by modifying the grammar are much smaller for all three techniques in alias analysis.

Additionally, it can be computed from Figure 3.10(b) and 3.11(b) that all three accelerated techniques through the modified grammars are still slower than POCR. We



**(a)** Reduction rates of #Add and #Deriv.



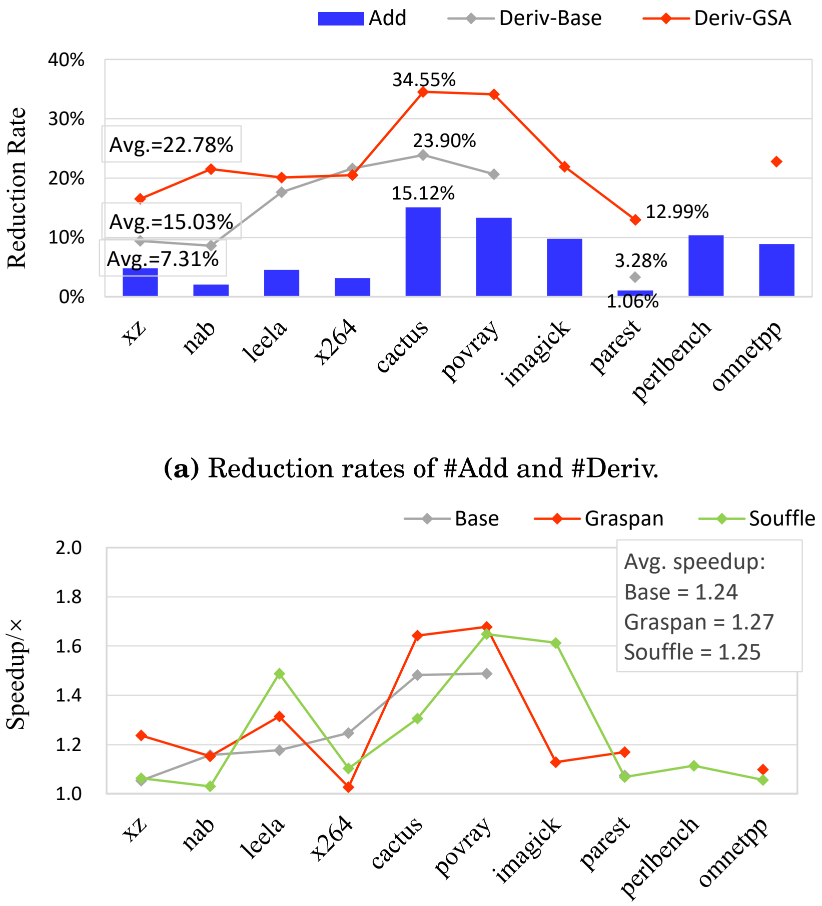
**(b)** Speedups comparing to the original grammar.

**Figure 3.10.** Results of context-sensitive value-flow analysis using the modified grammar in Figure 3.9(a).

analyze the reason as follows. Although the modified grammars appear to have the "head-to-tail", as illustrated in Section 3.1.3, both the standard worklist algorithm and GSA do not strictly follow this derivation order. Hence, POCR is faster than the three techniques in the presence of grammar rewriting.

### 3.4.5 Summary

We summarize our experimental results as follows: (1) POCR is highly effective in taming transitive redundancy with 98.50% and 97.26% of redundant derivations being



**(a)** Reduction rates of #Add and #Deriv.

**(b)** Speedups comparing to the original grammar.

**Figure 3.11.** Results of field-sensitive alias analysis using the modified grammar in Figure 3.9(b).

eliminated for context-sensitive value-flow analysis and field-sensitive alias analysis, respectively. (2) By eliminating redundant derivations, POCR significantly accelerates the standard algorithm by 21.48× and 19.57× respectively for the value-flow and alias analyses. (3) Though grammar rewriting can reduce some redundancy by removing doubly recursive rules, POCR is still much more effective in reducing redundant derivations than grammar rewriting.

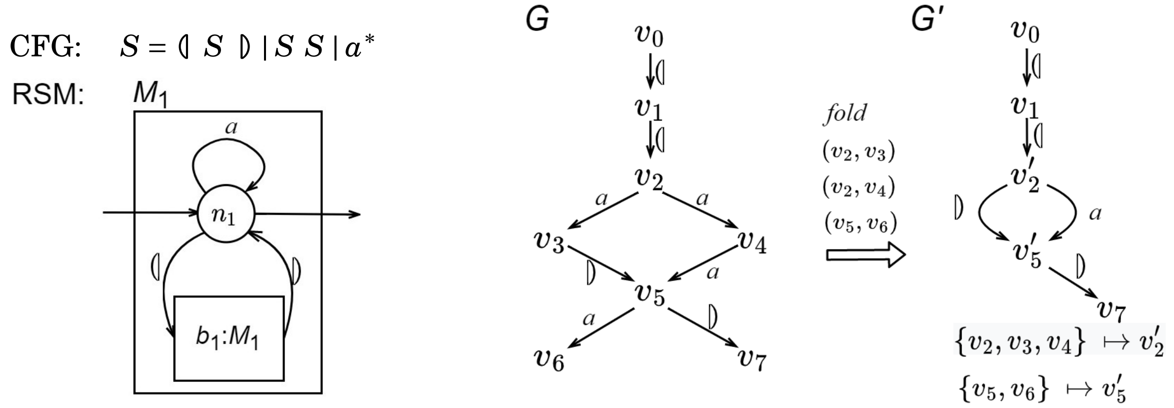
CHAPTER 4

# Recursive State Machine Guided Graph Folding

Popular static analyses, such as context-sensitive value-flow analysis and field-sensitive alias analysis, are formulated using Context-Free Language Reachability (CFL-Reachability), as expressed using Recursive State Machines (RSMs) [117, 149]. The input graphs that are automatically generated by compilers contain redundant nodes and edges with respect to particular RSMs. This chapter presents a novel graph folding algorithm GF that takes a graph $G$ and an RSM $R$ and produces a substantially smaller graph $G'$ which is equivalent to $G$ with respect to reachability for $R$ yet is solvable much more efficiently.

## 4.1 Problem Formulation

In this Chapter, we describe graph folding in the context of RSM-reachability. We first introduce the background and formulate the problem using an example in Figure 4.1, where the grammar accepts a Dyck+linear language, which is widely used in context-sensitive program analyses [50, 54, 105, 146]. In Figure 4.1, the RSM-reachability aims to determine nodes that are reachable from $v_0$ or $v_1$ via a path whose edge labels contain matched parentheses and an arbitrary number of "$a$".



**(a)** A context-free grammar and its equivalent recursive state machine.

**(b)** $G$ is the original graph. $G'$ is transformed from $G$ via graph folding.

**Figure 4.1.** An example of RSM-reachabililty and graph folding.

### 4.1.1 Recursive Static Machine

A recursive state machine over a finite alphabet $\Sigma$ is defined as a tuple $R = \langle M_1, \cdots, M_t\rangle$ comprised of $t$ component finite state machines, where each component $M_i = \langle N_i, B_i, Y_i, En_i, Ex_i, \delta_i\rangle$ is a finite state machine consisting of the following:

$N_i$ – a finite set of local states.

$B_i$ – a finite set of boxes in $M_i$, with each of which mapped to a component state machine.

$Y_i$ $: B_i \mapsto \{1, \cdots, t\}$ – a mapping function assigning each box of $M_i$ an index of one of the components $M_1, \cdots, M_t$;

$En_i$ $\subseteq N_i$ – a set holding the entries of $M_i$;

$Ex_i$ $\subseteq N_i$ – a set holding the exits of $M_i$;

$\delta_i$ $: \big(N_i \cup \bigcup_{b \in B_i} Ex_{Y_i(b)}\big) \times \Sigma \to N_i \cup \bigcup_{b \in B_i} En_{Y_i(b)}$ – a local transition function that maps specific states and labels to specific target states.

Specifically, for a local transition in $\delta_i$, denoted by $n_{i_1} \xrightarrow{\ell} n_{i_2}$, (1) the source $n_{i_1}$ must be either a local state or an exit of a box belonging in $M_i$, (2) the label $\ell$ is an element of $\Sigma$, and (3) the target $n_{i_2}$ must be either a local state or an entry of a box in $M_i$.

*Example* 4.1. Figure 4.1(a) gives an example RSM over an alphabet $\Sigma = \{\llparenthesis\ , a,\ \rrparenthesis\}$. There is only one component $M_1$ calling itself recursively. $M_1$ is comprised of: (1) a local state $n_1$, which is also the entry and the exit of $M_1$; (2) a box $b_1$ which is mapped to $M_1$; and (3) three transition rules $n_1 \xrightarrow{\llparenthesis} \langle b_1, n_1 \rangle$, $n_1 \xrightarrow{a} n_1$ and $\langle b_1, n_1 \rangle \xrightarrow{\rrparenthesis} n_1$ where $\langle b_1, n_1 \rangle$ denotes the entry (also the exit) of the box $b_1$, as the labeled edges in Figure 4.1(a).

The semantics of RSM is given by global states and transitions:

*Global states.* A global state $s \in S$, where $S = B^* \times N$, $B = \bigcup_i B_i$, $N = \bigcup_i N_i$, can be viewed as a local state nested in layers of boxes. Consider a sequence of boxes $b_1, \cdots, b_k$ such that $b_i \in B_{j_i}$ and $B_{j_i}, B_{j_{i+1}} \in B$ for all $i \in \{1, \cdots, k\}$. A global state $s = \langle b_1, \cdots, b_k, n \rangle$ follows (1) $Y_{j_i}(b_i) = j_{i+1}$ for all $i \in \{1, \cdots, k\}$, i.e., $b_{i+1}$ is nested in $b_i$, and (2) $n \in N_{j_{k+1}}$, i.e., $n$ is nested in $b_k$.

Intuitively, a global state denotes the current position of the initial state (usually not nested in any box) after entering a series of boxes.

*Global transitions* $\Delta$. The global transition function $\Delta : S \times \Sigma \to S$ maps specific global states and labels to specific target global states. For a global transition rule $s_1 \xrightarrow{\ell} s_2 \in \Delta$, $s_1, s_2 \in S$ and $\ell \in \Sigma$. Global transitions are restricted by the local transition rules. Given two global states $s_1, s_2 \in S$ and $s_1 = \langle b_1, \cdots, b_{k-1}, b_k, n_1 \rangle$ where $b_k \in B_{j_k}$, $Y_{j_k}(b_k) = j_{k+1}$ and $n_1 \in N_{j_{k+1}}$, $s_1 \xrightarrow{\ell} s_2 \in R$ iff one of the following four points holds:

1. $s_2 = \langle b_1, \cdots, b_{k-1}, b_k, n_2 \rangle$ and $n_1 \xrightarrow{\ell} n_2 \in \delta_{j_{k+1}}$ (local transition);
2. $s_2 = \langle b_1, \cdots, b_{k-1}, b_k, b_{k+1}, n_2 \rangle$, $b_{k+1} \in B_{k+1}$, $n_2 \in En_{Y_{j_{k+1}}(b_{k+1})}$ and $n_1 \xrightarrow{\ell} \langle b_{k+1}, n_2 \rangle \in \delta_{j_{k+1}}$ (entering a box);
3. $s_2 = \langle b_1, \cdots, b_{k-1}, n_2 \rangle$, $n_1 \in Ex_{j_{k+1}}$ and $\langle b_k, n_1 \rangle \xrightarrow{\ell} n_2 \in \delta_{j_k}$ (exiting from a box);
4. $s_2 = \langle b_1, \cdots, b_{k-1}, b'_k, n_2 \rangle$, $n_1 \in Ex_{j_{k+1}}$, $b'_k \in B_{j_k}$, $n_2 \in En_{Y_{j_k}(b'_k)}$ and $\langle b_k, n_1 \rangle \xrightarrow{\ell} \langle b'_k, n_2 \rangle \in \delta_{j_k}$ (transiting across two boxes).

**Property 4.1** (*Dependency of Global Transitions on Local Transitions*)**.** A global transition can only change the innermost box (by entering/exiting) and the innermost local state (by the local transition function of the component state machine, to which the innermost box maps) of a global state.

For brevity, in the remainder of this paper, we refer to all the states and transitions as the global ones unless specified.

*Deterministic RSMs.* In a deterministic RSM, given a state $s_i$ and a label $t$, if $\exists s_i \xrightarrow{\ell} s_j \in R$, $s_j$ is unique. In other words, for the transition function of a deterministic RSM, when the input state and label are specified, we can always determine the output state. In the remainder of this paper, all the demonstrations and conclusions are based on deterministic RSMs.

*Transition Chains.* In an RSM $R$, a transition chain $p_R \in R$ from a source state $s_0$ to a target state $s_k$ is a sequence of global transitions $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_k} s_k$ such that $s_{i-1} \xrightarrow{\ell_i} s_i \in \Delta$ for all $i \in \{1, \cdots, k\}$. In a deterministic RSM, given a source state $s_0$ and a sequence of label $\ell_1, \cdots, \ell_k$, the transition chain $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_k} s_k \in R$ is unique if it exists.

*Acceptable Strings.* Given an RSM $R$ with a specified initial state $s_{init} \in S$ and a set of accepting states $F \subseteq S$, a string $w \in \Sigma^*$ is accepted by $R$ iff it takes a transition chain from the initial state to one of the accepting states, i.e., $\exists\ s_{init} \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_k} s_k \in R$ such that $s_k \in F$ and $\ell_1 \cdots \ell_k = w$.

### 4.1.2 RSM-Reachability

In general, RSM-reachability is to check particular source-sink pairs whether the sink is reachable from the source by a path whose edge labels forms, in sequence, an acceptable string of the RSM.

*Reachable Paths.* Given an RSM-reachability instance $Reach\langle R, G\rangle$ where $R$ is an RSM and $G = \langle V, E\rangle$ is an edge-labeled directed graph, a path $p_G = v_i \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_k} v_j \in G$ is a reachable path iff the string $\ell_1 \cdots \ell_k$ formed by the sequence of the edge labels of $p_G$ is accepted by $R$.

*Reachable Pairs.* In an RSM-reachability instance, a reachable pair $(v_i, v_j)_{reach} \in V \times V$ is a node pair in $G$ such that there exists at least one reachable path from $v_i$ to $v_j$.

*RSM-reachability.* Formally, given an RSM $R$ with specified initial state and accepting states and a graph $G$ with specified sources $V_{src} \subseteq V$ and sinks $V_{snk} \subseteq V$, an RSM-reachability problem aims to determine for each source-sink pair $(v_i, v_j) \in V_{src} \times V_{snk}$ whether it is a reachable pair.

*Example* 4.2. We formulate the example (Figure 4.1) as an RSM-reachability problem. In this instance, $R = \langle M_1 \rangle$, $s_{init} = \langle n_1 \rangle$, $F = \{\langle n_1 \rangle\}$, $V_{src} = \{v_0, v_1\}$, $V_{snk} = \{v_0, \cdots, v_7\}$. For the path $v_1 \xrightarrow{\llparenthesis} v_2 \xrightarrow{a} v_3 \xrightarrow{\rrparenthesis} v_5 \in G$, the sequence of edge labels forms a string "$\llparenthesis\ a\ \rrparenthesis$" which is accepted by $R$ because there is a transition chain $\langle n_1 \rangle \xrightarrow{\llparenthesis} \langle b_1, n_1 \rangle \xrightarrow{a} \langle b_1, n_1 \rangle \xrightarrow{\rrparenthesis} \langle n_1 \rangle \in R$. Therefore, $(v_1, v_5)$ is a reachable pair. Similarly, $(v_0, v_7), (v_1, v_6)$ and $(v_1, v_7)$ are also reachable pairs.

### 4.1.3 Research Problem

Given a directed multigraph $G$ and two adjacent nodes $(x, y) \in V \times V$, an *xy-folded graph* $G'$ is constructed by removing all the edges joining $x$ and $y$ and collapsing $(x, y)$ into a *representative node*, e.g., $z$, such that $Rep(x) = Rep(y) = z$. Here we use $Rep(v_i)$ to denote the representative node of $v_i$. Namely, if $v_i$ is merged into another node $z$ after graph folding, $Rep(v_i) = z$, Otherwise, $Rep(v_i) = v_i$.

Let us revisit the motivating example in Figure 4.1(b), where $G'$ is the graph folded from the original version $G$. Specifically, the node pair $(v_2, v_3)$ is folded into a representative node $v'_2$ with $v_2 \xrightarrow{a} v_3$ removed. Likewise, $(v_2, v_4)$ and $(v_5, v_6)$ are folded into $v'_2$ and $v'_5$, respectively. The solutions on $G$ and $G'$ are displayed in Table 4.1. Compare the solutions, we can find that by expanding $v'_5$ back into $v_5$ and $v_6$, we get two reachable pairs $(v_1, v_5)$ and $(v_1, v_6)$ from $(v_1, v'_5)$. In this way, we obtain identical solutions from $G'$ and $G$. This means that $G$ and $G'$ are equivalent with respect to this CFL-reachability problem. In view of the (sub)cubic complexity of CFL/RSM-reachabililty solving, reducing

| | Reachable pair | Reachable path | String |
|---|---|---|---|
| $G$ | $(v_0,v_7)$ | $v_0 \xrightarrow{\llparenthesis} v_1 \xrightarrow{\llparenthesis} v_2 \xrightarrow{a} v_3 \xrightarrow{\rrparenthesis} v_5 \xrightarrow{\rrparenthesis} v_7$ | $\llparenthesis\ \llparenthesis\ a\ \rrparenthesis\ \rrparenthesis$ |
| | $(v_1,v_5)$ | $v_1 \xrightarrow{\llparenthesis} v_2 \xrightarrow{a} v_3 \xrightarrow{\rrparenthesis} v_5$ | $\llparenthesis\ a\ \rrparenthesis$ |
| | $(v_1,v_6)$ | $v_1 \xrightarrow{\llparenthesis} v_2 \xrightarrow{a} v_3 \xrightarrow{\rrparenthesis} v_5 \xrightarrow{a} v_6$ | $\llparenthesis\ a\ \rrparenthesis a$ |
| | $(v_1,v_7)$ | $v_1 \xrightarrow{\llparenthesis} v_2 \xrightarrow{a} v_4 \xrightarrow{a} v_5 \xrightarrow{\rrparenthesis} v_7$ | $\llparenthesis\ aa\ \rrparenthesis$ |
| $G'$ | $(v_0,v_7)$ | $v_0 \xrightarrow{\llparenthesis} v_1 \xrightarrow{\llparenthesis} v_2' \xrightarrow{\rrparenthesis} v_5' \xrightarrow{\rrparenthesis} v_7$ | $\llparenthesis\ \llparenthesis\ \rrparenthesis\ \rrparenthesis$ |
| | $(v_1,v_5')$ | $v_1 \xrightarrow{\llparenthesis} v_2' \xrightarrow{\rrparenthesis} v_5'$ | $\llparenthesis\ \rrparenthesis$ |
| | $(v_1,v_7)$ | $v_1 \xrightarrow{\llparenthesis} v_2' \xrightarrow{a} v_5' \xrightarrow{\rrparenthesis} v_7$ | $\llparenthesis\ a\ \rrparenthesis$ |

**Table 4.1.** The solutions of the RSM-reachability problem in $G$ and $G'$ of Figure 4.1(b).

the size of the graph in the preprocessing stage can significantly boost the performance of reachability solving.

Intuitively, an aggressive graph folding approach should fold as many node pairs as possible. However, arbitrarily folding node pairs may change reachable pairs due to removals of the original reachable paths or additions of new ones, resulting in an inconsistent RSM-reachability solution. A correct graph folding approach ensures the equivalence of reachable pairs in the original graph and the folded graph, as Definition 4.1.

**Definition 4.1** (*Reachability Equivalence*)**.** Let $G = \langle V,E\rangle$ be the original graph with $V_{src}$ and $V_{snk}$ specified, and let $G' = \langle V',E'\rangle$ be the folded graph, $G$ and $G'$ are reachability equivalent iff $\forall(v_i,v_j) \in V_{src} \times V_{snk},\ (v_i,v_j)_{reach} \in G \Leftrightarrow (Rep(v_i),Rep(v_j))_{reach} \in G'$.

Definition 4.1 implies that a correct graph folding (1) preserves the information of all the original reachable pairs, and (2) does not introduce any redundant reachable pair. In another word, two graphs being reachability equivalent yields equivalent RSM-reachability solutions. If folding a node pair $(x,y)$ preserves reachability equivalence, we say that $(x, y)$ is *foldable*.

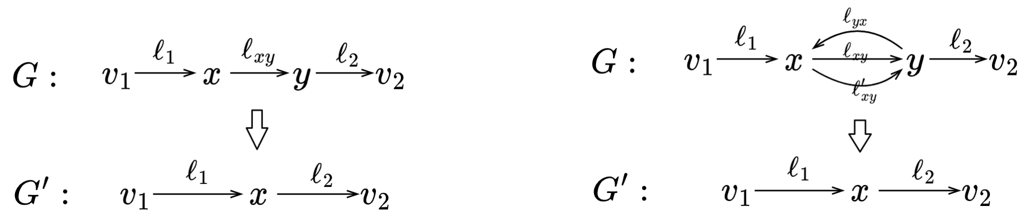
*Example* 4.3. In Figure 4.1(b), after folding the node pair $(v_2, v_3) \in G$ into $v_2' \in G'$, the reachable path $v_1 \xrightarrow{\llparenthesis} v_2 \xrightarrow{a} v_3 \xrightarrow{\rrparenthesis} v_5$ becomes $v_1 \xrightarrow{\llparenthesis} v_2' \xrightarrow{\rrparenthesis} v_5$, whose edge labels also form an acceptable string of the RSM. Thus, the reachable pair $(v_1, v_5)$ is preserved after the folding. It can be computed that other reachable pairs in the original graph are also preserved. Therefore, $(v_2, v_3)$ is foldable. In contrary, $(v_3, v_5)$ is not foldable. If we were to fold $(v_3, v_5)$ into $v_3'$, the path $v_1 \xrightarrow{\llparenthesis} v_2 \xrightarrow{a} v_3 \xrightarrow{\rrparenthesis} v_5$ would become $v_1 \xrightarrow{\llparenthesis} v_2 \xrightarrow{a} v_3'$ where $v_3'$ represents $v_3$ and $v_5$, which is no longer a reachable path. Moreover, as there is no other reachable path from $v_1$ to $v_3'$ in the folded graph, $(v_1, v_3')$ is not a reachable pair, indicating that the original reachable pair $(v_1, v_5)$ in $G$ is lost in the folded graph.

We formulate graph folding problem as follows:

> Given an RSM-reachability instance $Reach\langle R, G\rangle$, generate a smaller graph $G'$ by folding node pairs in $G$ and ensures that $G$ and $G'$ are reachability equivalent.

## 4.2 Principle for Graph Folding

Identifying foldable node pairs based on Definition 4.1 requires figuring out all reachable paths in the original graph $G$ and the folded graph $G'$, which violates our purpose to improve scalability. To effectively identify foldable node, we first study the correspondences between the original graph and the folded graph and between the paths on the graph and transition chains in the RSM, leading to a practical criteria for reachability equivalence. Then, we further exploit the subsumption relations of states in RSMs, using it to formulate our folding principle, which is able to identify whether a node pair is foldable by examining only its incoming and outgoing edges.



**(a)** When $x$ and $y$ are joined by only one edge, each path passing through $x$ and $y$ will be folded into a distinct path after folding $(x,y)$.

**(b)** When $x$ and $y$ are joined by multiple edges, there will be multiple paths folded into the same path after folding $(x,y)$.

**Figure 4.2.** Correspondences between paths before and after folding. Without loss of generality, we assume that $y$ is merged into $x$ in the $xy$-folded graph.

### 4.2.1 Correspondences in Graph Folding and RSM-Reachability

**Folding-Equivalent Class and Criteria for Reachability Equivalence.** In Figure 4.2(a), the path $p_{G'}$ is obtained from $p_G$ by folding $(x,y)$, i.e., contracting the edge $x \xrightarrow{\ell_{xy}} y$. In fact, the graph of RSM-reachability can be a multigraph, meaning that there can be multiple edges between $x$ and $y$, as shown in Figure 4.2(b). As folding $(x,y)$ will remove all the edges joining $x$ and $y$, there can be multiple original paths folded into the same *xy-folded path*. Such paths in the original graph $G$ constitute folding-equivalent classes based on their endpoints:

**Definition 4.2** (*xy-Folding-Equivalent (xy-FEQ) Classes*, $\boldsymbol{P}_{xy}$)**.** Two paths $p_{G_1}, p_{G_2} \in G$ are of the same $xy$-folding-equivalent class $\boldsymbol{P}_{xy}$ iff they have identical endpoints and are folded into the same path $p_{G'}$ in the $xy$-folded graph $G'$. In particular, a path is $xy$-folding-equivalent to itself.

In Figure 4.2(b), all the paths starting with $v_1$ and ending with $v_2$ are of the same $xy$-FEQ class. Based on $xy$-folded paths and their corresponding $xy$-FEQ classes, we can refine the impractical reachable-pair oriented criteria for reachability equivalence to practical reachable-path oriented criteria as Definition 4.3, which is sufficient to guarantee reachability equivalence:

**Definition 4.3** (*Sufficient Condition for Reachability Equivalence*)**.** A $xy$-folded graph $G'$ is reachability equivalent to its original graph $G$ if the following conditions are satisfied:

*Cond. 1.* Each source-sink path in $G'$ has a corresponding $xy$-FEQ class in $G$ (exclusiveness).

*Cond. 2.* For each $xy$-FEQ class $\boldsymbol{P}_{xy}$ in $G$, its corresponding $xy$-folded path in $G'$ is a reachable path iff $\boldsymbol{P}_{xy}$ contains a reachable path in $G$ (consistency).

Satisfying *Cond. 1* is simple. For example, we can avoid introducing extra source-sink paths to $G'$ by not merging $(x,y)$ when (1) there is no edge from $y$ to $x$ and (2) $y$ is a source or $y$ has incoming edges not from $x$. To satisfy *Cond. 2*, we need to check the corresponding transition chains of reachable paths. In Section 4.3, we will provide an algorithm that identifies some foldable pairs that meet these conditions

**Corresponding Transition Chains and Corresponding States.** As addressed in Section 4.1.1, given a specific initial state $s_{init}$ and a string of labels $\ell_1 \cdots \ell_k$, a deterministic RSM has at most one transition chain leading $s_{init}$ through $\ell_1 \cdots \ell_k$ to a deterministic target state, e.g., $s_k$. Notably, the corresponding transition chain is defined below:

**Definition 4.4** (*Corresponding Transition Chain*)**.** Given a RSM-reachability instance $Reach\langle R, G\rangle$ where $R$ is deterministic, consider a path $p_G = v_0 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_k} v_k \in G$ where $\ell_1 \cdots \ell_k$ forms a string $w \in \Sigma^k$. If there exists $p_R = s_{init} \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_k} s_k \in R$ such that $\ell_1 \cdots \ell_k = w$, we say that $p_R$ is the *corresponding transition chain* of $p_G$. When the RSM $R$ is deterministic, a path $p_G \in G$ has zero or one corresponding transition chain.

Figure 4.3(a) shows a path $p_G$ and its corresponding transition chain $p_R$. Similarly, Figure 4.3(b) depicts an $xy$-folded path $p_{G'} \in G'$ which has zero or one corresponding transition chain $p'_R \in R$.

As depicted in Figure 4.3(a), with respect to the corresponding transition chain, each node $v_i$ of a path $p_G \in G$ is mapped to exactly one state $s_i$ in the RSM. We call $s_i$ the

Before folding $(x,y)$

$$p_G \in G : v_0 \xrightarrow{\ell_0} \cdots \xrightarrow{\ell_1} x \xrightarrow{\ell_{xy}} y \xrightarrow{\ell_2} v_2 \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_k} v_k$$

$$p_R \in R : \; s_{init} \xrightarrow{\ell_0} \cdots \xrightarrow{\ell_1} s_x \xrightarrow{\ell_{xy}} s_y \xrightarrow{\ell_2} s_2 \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_k} s_k$$

**(a)** A path $p_G$ and its corresponding transition chain $p_R$

After folding $(x,y)$

$$p_{G'} \in G' : v_0 \xrightarrow{\ell_0} \cdots \xrightarrow{\ell_1} x \xrightarrow{\ell_2} v_2 \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_k} v_k$$

$$p'_R \in R : \; s_{init} \xrightarrow{\ell_0} \cdots \xrightarrow{\ell_1} s_x \xrightarrow{\ell_2} s'_2 \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_k} s'_k$$

**(b)** The $xy$-folded path $p_{G'}$ of $p_G$ and its corresponding transition chain $p'_R$.

**Figure 4.3.** A path $p_G$ and its corresponding transition chain $p_R$ before and after folding $(x,y)$. Without loss of generality, we assume that $y$ is merged into $x$ in the $xy$-folded graph.

corresponding state of $v_i$ in $p_G$. In a graph, a node $v_i$ may belong to multiple paths corresponding to different transition chains, leading to the notion of *corresponding states*:

**Definition 4.5** ($Q_{v_i}$ *– the Corresponding States of* $v_i$)**.** For a node $v_i \in V$, $Q_{v_i} \subseteq S$ is the set holding the target states of all the corresponding transition chains of paths ending at $v_i$. Taking empty paths into consideration, $s_{init} \in Q_{v_i}$.

Similarly, a path $p_G \in G$ can be a subpath of other paths, which means that $p_G$ can also correspond to one or more *sub-transition chains* which do not start with $s_{init}$. Nevertheless, the starting states of the sub-transition chains are limited:

**Remark** (*Sub-Transition Chain*)**.** While considering a path $p_G$ starting with $v_i$ as a subpath of other paths, each corresponding sub-transition chain of $p_G$ must start with a state belonging to $Q_{v_i}$.

*Example* 4.4 (Corresponding States and Reachability Equivalence). Consider an instance in Figure 4.3 where $v_2, \cdots, v_k$ do not contain any of $x$ or $y$. According to Definition 4.5, $s_x \in Q_x$. By comparing Figures 4.3(a) and 4.3(b), we can see that $s_2$ is obtained from $s_x$ by two transitions $s_x \xrightarrow{\ell_{xy}} s_y \xrightarrow{\ell_2} s_2$, while $s'_2$ is obtained from $s_x$ by one transition $s_x \xrightarrow{\ell_2} s'_2$.

**Table 4.2.** Edge label notations for discussing RSM-reachability.

| Notation | Description |
|---|---|
| $L_{xy} \;= \{\ell \mid x \xrightarrow{\ell} y \in E\}$ | the set of labels of the edges from $x$ to $y$. |
| $L_{x\_} \;= \{\ell \mid x \xrightarrow{\ell} v_i \in E, v_i \in V\}$ | the set of labels of the outgoing edges of $x$. |
| $L_{\_x} \;= \{\ell \mid v_i \xrightarrow{\ell} x \in E, v_i \in V\}$ | the set of labels of the incoming edges of $x$. |
| $L_{\not{y}x} \;= \{\ell \mid v_i \xrightarrow{\ell} x \in E, v_i \in V, v_i \neq y\}$ | the set of labels of the edges ending with $x$ and not starting with $y$. |
| $L_{x\not{y}} \;= \{\ell \mid x \xrightarrow{\ell} v_i \in E, v_i \in V, v_i \neq y\}$ | the set of labels of the edges starting with $x$ and not ending with $y$. |

If $s_2 = s'_2$, then $s_k = s'_k$ because the sequence of edge labels $\ell_2, \ell_3, \cdots, \ell_k$ is not changed after folding $(x, y)$. Moreover, if for any $s_x \in Q_x$, $s_2 = s'_2$, then for any $p_G$ (Figure 4.3(a)) and its $xy$-folded $p_{G'}$ (Figure 4.3(b)), their corresponding transition chains end at the same state. This means that folding $(x, y)$ keeps the paths passing through $x \xrightarrow{\ell_{xy}} y$ and their $xy$-folded paths consistent with respect to whether they are reachable paths.

The above example shows that, with determined $Q_x$ and $Q_y$, we are able to identify whether folding $(x, y)$ preserves reachability equivalence by computing *at most two transitions* for each state of $Q_x$ and $Q_y$.

### 4.2.2 Folding Principle

Precisely computing $Q_x$ and $Q_y$ is equivalent to solving RSM-reachability, which is too expensive for graph folding. This section formulates our graph folding principle using an alternative overapproximation of $Q_x$, utilizing RSM properties, *i.e.*, *subsumption and equivalence relations* of states.

**Overapproximating $Q_x$.** With determined labels of incoming edges of $x$, the local states of the states in $Q_x$ are determined. To facilitate the discussion, Table 4.2 lists the notations about the edge labels involving $x$ and $y$. With all incoming edge labels $L_{\_x}$ of a node $x$ in $G$, we collect a set of RSM local states $n'$ with transitions $n \xrightarrow{\ell} n'$ and $\ell \in L_{\_x}$. We define the set $Nr(L_{\_x})$ as

$$Nr(L_{\_x}) = \{n' \mid\ n \xrightarrow{\ell} n' \in \bigcup_i \delta_i,\ \ell \in L_{\_x}\}. \tag{4.1}$$

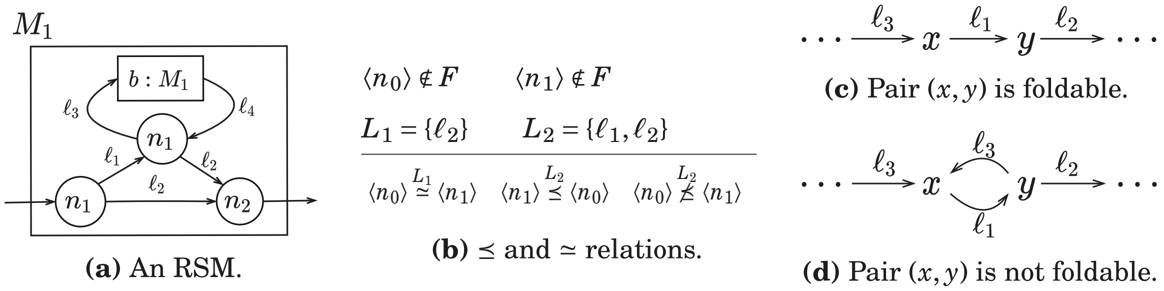
Intuitively, $Nr(L_{\_x})$ holds the target RSM states whose transition labels belong to the incoming edge label set $L_{\_x}$ of $x$. In particular, if $x \in V_{src}$, we let $s_{init} \in Nr(L_{\_x})$. Obviously, the local states of the states in $Q_x$ belong to $Nr(L_{\_x})$. Computing $Nr(L_{\_x})$ is very inexpensive because given $L_{\_x}$, $Nr(L_{\_x})$ can be directly determined by checking the local transition functions of the RSM. As global states are local states wrapped by layers of boxes, we have:

$$Q_x \subseteq B^* \times Nr(L_{\_x}). \tag{4.2}$$

*Example* 4.5 (Computing $Nr(L_{\_x})$). Given an RSM in Figure 4.4(a), Figure 4.4(c) is a graph segment of an RSM-reachability instance running on the RSM. In the graph segment, $L_{\_x} = \{\ell_3\}$. In the RSM, there is only one transition $n_1 \xrightarrow{\ell_3} \langle b, n_0\rangle$ labeled by $\ell_3$. Thus, in Figure 4.4(c), $Nr(L_{\_x}) = \{\langle b, n_0\rangle\}$. Moreover, there is only one box in the RSM, i.e., $B = \{b\}$. Thus, $Q_x \subseteq \{b\}^* \times \{\langle b, n_0\rangle\}$.

In Section 4.2.1, we use Example 4.4 to show that we can determine whether folding $(x, y)$ preserves reachability equivalence by computing at most two transitions for each state of $Q_x$ and $Q_y$. According to Property 4.1, each transition can exit at most one box. Namely, for $Q_x$ and $Q_y$, examining the states with two layers of boxes is sufficient to cover all states, as outer boxes are never affected by the two transitions. Correspondingly, while replacing $Q_x$ by $B^* \times Nr(L_{\_x})$ and $B^* \times Nr(L_{\_y})$, examining the states of $B^\alpha \times Nr(L_{\_x})$ and $B^\alpha \times Nr(L_{\_y})$ where $\alpha \le 2$ is enough.

**Rules for Consistency.** In Section 4.2.1, we show that ensuring reachability equivalence is to satisfy the two conditions of Definition 4.3, among which satisfying the exclusiveness condition (*Cond. 1*) is simple and addressed immediately after Definition


$M_1$
$b : M_1$
$\ell_3$
$\ell_4$
$n_1$
$\ell_1$
$\ell_2$
$\ell_2$
$n_1$
$n_2$
$\langle n_0\rangle \notin F$
$\langle n_1\rangle \notin F$
$L_1 = \{\ell_2\}$
$L_2 = \{\ell_1, \ell_2\}$
$\langle n_0\rangle \overset{L_1}{\simeq} \langle n_1\rangle$
$\langle n_1\rangle \overset{L_2}{\preceq} \langle n_0\rangle$
$\langle n_0\rangle \overset{L_2}{\not\preceq} \langle n_1\rangle$
$\cdots \xrightarrow{\ell_3} x \xrightarrow{\ell_1} y \xrightarrow{\ell_2} \cdots$
$\cdots \xrightarrow{\ell_3} x$
$\ell_3$
$\ell_1$
$y \xrightarrow{\ell_2} \cdots$

**(a)** An RSM.

**(b)** $\preceq$ and $\simeq$ relations.

**(c)** Pair $(x, y)$ is foldable.

**(d)** Pair $(x, y)$ is not foldable.

**Figure 4.4.** Example of subsumption and equivalence relations of states and instances of foldable pairs $(x, y)$.

*Rule [x-x]:* $\forall s_x \xrightarrow{\ell_{xy}} s_y \xrightarrow{\ell_{yx}} s'_x \in R \;\; s.t. \;\; s_x \in B^\alpha \times Nr(L_{\_x}) \wedge \ell_{xy} \in L_{xy} \wedge \ell_{yx} \in L_{yx}\,, \quad s'_x \overset{L_{x\_}}{\preceq} s_x\,.$

*Rule [y-y]:* $\forall s_y \xrightarrow{\ell_{yx}} s_x \xrightarrow{\ell_{xy}} s'_y \in R \;\; s.t. \;\; s_y \in B^\alpha \times Nr(L_{\_y}) \wedge \ell_{xy} \in L_{xy} \wedge \ell_{yx} \in L_{yx}\,, \quad s'_y \overset{L_{y\_}}{\preceq} s_y\,.$

*Rule [x-y]:* $\forall s_x \in B^\alpha \times Nr(L_{\_x}) \wedge \forall \ell_{xy} \in L_{xy}\,, \quad \exists s_x \xrightarrow{\ell_{xy}} s_y \in R \quad s.t. \;\; s_x \overset{L_{y\not{x}}}{\simeq} s_y\,.$

*Rule [y-x]:* $\forall s_y \in B^\alpha \times Nr(L_{\_y}) \wedge \forall \ell_{yx} \in L_{yx}\,, \quad \exists s_y \xrightarrow{\ell_{yx}} s_x \in R \quad s.t. \;\; s_y \overset{L_{x\not{y}}}{\simeq} s_x\,.$

**Figure 4.5.** Rules for *Cond. 2*, where $\alpha \in \{0,1,2\}$, $L_{xy}$, $L_{x\_}$, $L_{y\not{x}}$, and $Nr(L_{\_x})$ are defined in Table 4.2 and Eq. 4.1.

4.3. Here, we provide four rules in Figure 4.5 for satisfying the consistency condition (*Cond. 2*). The four rules exploit subsumption and equivalence relations of states (Definition 4.6). Basically, the two relations are used to measure and compare the capabilities of states being transited by edge labels involving $x$ and $y$.

**Definition 4.6** (*Subsumption $\preceq$ and Equivalence $\simeq$ Relations of States*)**.** Given a set of labels $L \subseteq \Sigma$ and two states $s_i, s_j \in S$, $s_i$ is subsumed by $s_j$ with respect to $L$, denoted by $s_i \overset{L}{\preceq} s_j$, iff

$$(1)\ \forall \ell \in L,\ \forall s_k \in S,\ s_i \xrightarrow{\ell} s_k \in \Delta \Rightarrow s_j \xrightarrow{\ell} s_k \in \Delta, \quad \text{and} \quad (2)\ s_i \in F \Rightarrow s_j \in F.$$

Specifically, $s_i$ is equivalent to $s_j$ with respect to $L$, denoted by $s_i \overset{L}{\simeq} s_j$, iff $s_i \overset{L}{\preceq} s_j \wedge s_j \overset{L}{\preceq} s_i$.

*Example* 4.6. Figure 4.4(b) gives an example of subsumption and equivalence relations of states in the RSM of Figure 4.4(a). $\langle n_0\rangle \overset{L_1}{\simeq} \langle n_1\rangle$ because both $\langle n_0\rangle$ and $\langle n_1\rangle$ can transit to $\langle n_2\rangle$ via $\ell_2$. $\langle n_1\rangle \overset{L_2}{\preceq} \langle n_0\rangle$ because $\langle n_1\rangle$ can transit to $\langle n_2\rangle$ via $\ell_2$ and $\langle n_0\rangle$ can not only transit to $\langle n_2\rangle$ via $\ell_2$ but also transit to $\langle n_1\rangle$ via $\ell_1$. In contrast, $\langle n_0\rangle \overset{L_2}{\not\preceq} \langle n_1\rangle$ because $\langle n_0\rangle$ can transit to $\langle n_1\rangle$ via $\ell_1$ whereas $\langle n_1\rangle$ cannot.

Based on the two relations in Definition 4.6, we briefly discuss the rules in Figure 4.5. By examining the states of $B^{\alpha} \times Nr(L_{\_x})$ and the labels of incoming and outgoing edges of $x$ and $y$, Rule [x-x] ensures the consistency of the corresponding transition chains of paths passing through $x$ and $y$ by starting and ending both with $x$. Rule [x-y] ensures the consistency of the corresponding transition chains of paths passing through an edge from $x$ to $y$. Rules [y-y] and [y-x] are symmetric to Rules [x-x] and [x-y] with respect to $x$ and $y$.

**Graph Folding Principle.** We provide our principle for identifying foldable $(x,y)$ in Theorem 4.1, where Principles ① and ② are used to satisfy *Cond. 1* and *Cond. 2* of Definition 4.3, respectively. Notably, in our folding principle, each rule in Figure 4.5 contains no more than two transitions, and the value of $\alpha$, *i.e.*, the layers of boxes, is no more than two.

**Theorem 4.1** (*Graph Folding Principle*). *Consider an RSM-reachability instance Reach*$\langle R,G\rangle$*. Without loss of generality, assume that there is always at least one edge from node x to y in G. The node pair* $(x,y) \in G$ *is foldable if both ① and ② hold:*

① *When there is no edge from y to x,* $y \notin V_{src}$ *and all the incoming edges of y starts at x;*

② *The four rules in Figure 4.5 hold for all* $\alpha \in \{0,1,2\}$.

*Example* 4.7 (Identifying Foldable Node Pairs). Figure 4.4(a) is an RSM, and Figures 4.4 (c) and (d) are two graph segments based on the RSM. Assume that neither $x$ nor $y$ is a source. In Figure 4.4(c), there is no edge from $y$ to $x$, and the only incoming edge of $y$ starts at $x$, so ① is satisfied. And for ②, we only need to consider Rule [x-y]. It can be observed that $Nr(L_{\_x}) = \{\langle b,n_0\rangle\}$, $L_{xy} = \{\ell_1\}$ and $L_{y\not x} = \{\ell_2\}$. We have

$\langle b,n_0\rangle \xrightarrow{\ell_{xy}=\ell_1} \langle b,n_1\rangle \in R$ and $\langle b,n_0\rangle \overset{L_{y\not{x}}}{\simeq} \langle b,n_1\rangle$. Therefore, the node pair $(x,y)$ in Figure 4.4(c) is foldable.

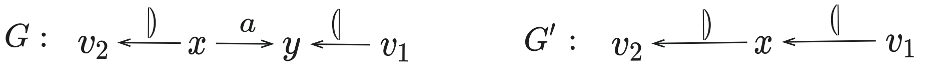
In Figure 4.4(d), $Nr(L_{\_x}) = \{\langle b,n_0\rangle\}$, $L_{xy} = \{\ell_1\}$, $L_{yx} = \{\ell_3\}$ and $L_{y\not{x}} = \{\ell_2\}$. We have $\langle b,n_0\rangle \xrightarrow{\ell_{xy}=\ell_1} \langle b,n_1\rangle \xrightarrow{\ell_{yx}=\ell_3} \langle b,b,n_0\rangle \in R$ and $\langle b,b,n_0\rangle \overset{L_{\_x}}{\npreceq} \langle b,n_0\rangle$. This means that Rule [x-x] of ② is not satisfied. Therefore, the node pair $(x,y)$ in Figure 4.4(d) is not foldable.

The soundness of our graph folding principle manifests in that folding any node pair preserves reachability equivalence. Besides, since the states of $Nr(L_{yx})$ in Rule [x-y] are doubly checked by Rules [x-x] and [y-x], $Nr(L_{\_x})$ in Rule [x-y] can be replaced by $Nr(L_{\not{y}x})$ for simplicity. Similarly, in Rule [y-x], $Nr(L_{\_y})$ can be replaced by $Nr(L_{\not{x}y})$. The principle decides foldability via only local information (*i.e.*, incoming and outgoing edges of a node pair in graph $G$). Therefore, it does not exhaustively detect all foldable node pairs. For example, in Figure 4.1(b), if we only consider $v_0$ as the source and $v_7$ as the sink, folding $(v_1,v_2)$ does not affect CFL-reachability result, whereas we do not consider it as foldable because it violates Rule [x-y] in Figure 4.5.

### 4.2.3 Correctness of Folding Principle

We demonstrate the correctness of our graph folding by showing that Principles ① and ② in Theorem 4.1 satisfies *Cond. 1* and *Cond. 2* in Definition 4.3, respectively. The proof is separated into proving Lemma 4.1 and Lemma 4.2, where the former is simple and intuitive. To prove Lemma 4.2, we first categorize the paths involving $x$ and $y$ into four basic types, as in Figure 4.7, and show how each rule in Figure 4.5 ensures the consistency of each type, respectively. This gives rise to two properties, *i.e.*, Property 4.2 and Property 4.3, which are further used to prove Lemma 4.2.

**Lemma 4.1.** *Principle ① in Theorem 4.1 implies the exclusiveness condition (Cond. 1) of Definition 4.3.*



**(a)** Original graph.

**(b)** Incorrectly folded $(x, y)$.

**Figure 4.6.** For the problem running on the RSM in Figure 4.1(a), if $v_1 \in V_{src}$ and $v_2 \in V_{snk}$, folding $(x, y)$ introduces an additional reachable pair $(v_1, v_2)$ in $G'$ via $v_1 \xrightarrow{\llparenthesis} x \xrightarrow{\rrparenthesis} v_2$, which violates reachability equivalence.

*Proof.* A source-sink path in $G'$ that does not have an $xy$-FEQ class in $G$ can only be introduced when there is no edge from $y$ to $x$, as shown in Figure 4.6. In this case, when $y \in V_{src}$ or $y$ has an incoming edge from a node other than $x$, folding $(x, y)$ may introduce new source-sink paths from $y$ or the predecessors of $y$ to $x$ or the successors of $x$. Such new paths do not have any $xy$-FEQ class in $G$ and can lead to spurious reachable pairs. Principle ① avoids such incorrect foldings. □

**Lemma 4.2.** *Principle ② in Theorem 4.1 implies the exclusiveness condition (Cond. 2) of Definition 4.3.*

We first illustrate the objectives to which the four rules take effect. Figure 4.7 categorizes all paths containing at most one *xy-subpath*[1] into four basic types of $xy$-FEQ classes, and shows the corresponding rules for ensuring consistency for each type. Briefly, Rule [x-x] ensures the consistency of the corresponding transition chains of paths where the $xy$-subpath starts and ends both with $x$ (Type 1). Rule [x-x] and Rule [x-y] together ensure the consistency of the corresponding transition chains of paths where the $xy$-subpath starts with $x$ and ends with $y$ (Type 2). Correspondingly, [y-y] and [y-x] ensure the consistency of Type 3, where $xy$-subpath starts and ends both with $y$, and Type 4, where $xy$-subpath starts with $y$ and ends with $x$.

Principle ② preserves the consistency for the four basic types as it has Property 4.2 and Property 4.3, among which Property 4.2 is the realization of Property 4.1 in the corresponding states of nodes, and it indicates that $B^{\alpha} \times Nr(L_{\_x})$ and $B^{\alpha} \times Nr(L_{\_y})$ where

[1] Given two nodes $x$ and $y$ in a graph, an $xy$-path is a path comprised of edges joining $x$ and $y$.

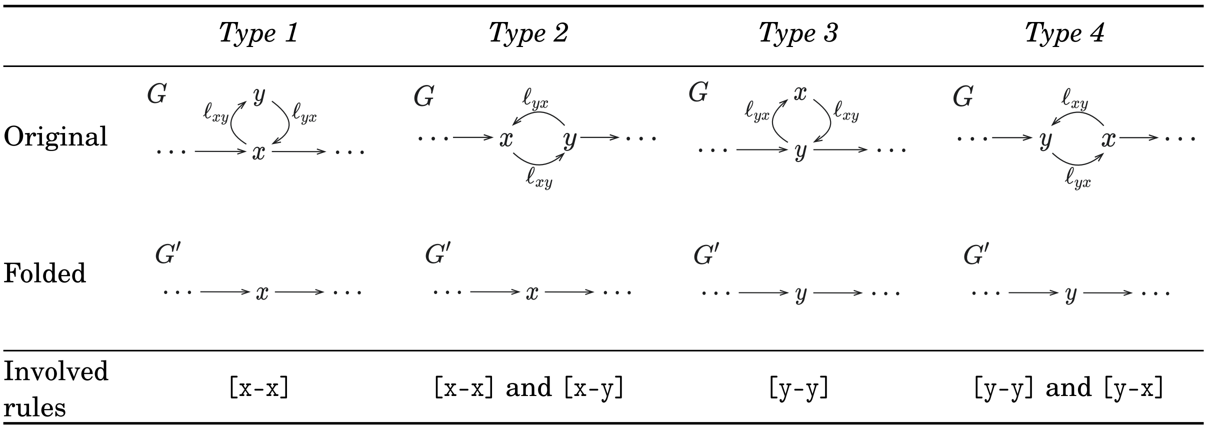


**Figure 4.7.** Four basic types of $xy$-FEQ classes, where each type contains at most one $xy$-subpath. For simplicity, we only draw one edge from $x$ to $y$ and one edge from $y$ to $x$.

$\alpha = 0,1,2$ is enough to cover all cases (states) in $Q_x$ and $Q_y$. Property 4.3 guarantees the consistency of the endpoints of transition chains corresponding to the four basic types of $xy$-FEQ classes, which further ensures the satisfaction of *Cond. 2*. The detailed proofs of Property 4.2 and Property 4.3 are provided in our supplementary material.

**Property 4.2.** If the four rules in Figure 4.5 hold when $\alpha \le 2$, they also hold for all $\alpha > 2$.

**Property 4.3.** With $\boldsymbol{P}_{xy}$ denoting an $xy$-FEQ class belonging to the four basic types of Figure 4.7 and $p_{G'}$ denoting the $xy$-folded path of $\boldsymbol{P}_{xy}$, if the four rules in Figure 4.5 hold, then (i) $p_{G'}$ is a reachable path iff $\boldsymbol{P}_{xy}$ contains a reachable path, and (ii) when the paths of $\boldsymbol{P}_{xy}$ do not end with $x$ or $y$, $p_{G'}$ corresponds to a sub-transition chain from $s_0$ to $s_k$ iff $\boldsymbol{P}_{xy}$ also contains a path corresponding to a sub-transition chain from $s_0$ to $s_k$.

*Proof of Lemma 4.2.* We use the four basic types and the two properties to prove Lemma 4.2. In the original graph $G$, any path $p_G$ can be seen as the concatenation of subpaths $p_{G1}p_{G2}\cdots p_{Gk}$ such that (1) for each $i \in \{1,\cdots,k\}$, $p_{Gi}$ is either a path belonging to the four basic types of Figure 4.7 or a path not containing any $xy$-subpath, and (2) for all $i < k$, $p_{Gi}$ does not end with either $x$ or $y$. Discussing $k = 1$ is trivial as it either belongs to the four basic types, which is covered by Property 4.3, or is a path not containing any $xy$-subpath, which is never changed by folding $(x, y)$.

Next, we consider $k > 1$ and start from $p_{G1}$. For the case that $p_{G1}$ does not belong to the four basic types, the corresponding transition chain is never changed by folding $(x, y)$, *i.e.*, the corresponding states of nodes are not changed. For the case that $p_{G1}$ belongs to $\boldsymbol{P}_{xy_1}$ one of the four basic types in Figure 4.7, according to Definition 4.2, the $xy$-folded path $p_{G1'}$ is exactly $\boldsymbol{P}_{xy}$. Property 4.3 ensures that $p_{G1'}$ corresponds to a transition chain from $s_{init}$ to $s_1$ iff $\boldsymbol{P}_{xy_1}$ also contains a path corresponding to a transition chain from $s_{init}$ to $s_1$.

Analogously, for the concatenation $p_{G1}p_{G2}\cdots p_{Gk-1}$ that belongs to $\boldsymbol{P}_{xy_{k-1}}$, Property 4.3 ensures that the $xy$-folded path of $p_{G1}p_{G2}\cdots p_{Gk-1}$ (*i.e.*, of $\boldsymbol{P}_{xy_{k-1}}$) corresponds to a transition chain from $s_{init}$ to $s_{k-1}$ iff $\boldsymbol{P}_{xy_{k-1}}$ also contains a path corresponding to a transition chain from $s_{init}$ to $s_{k-1}$.

Finally, we consider the whole path $p_G$, which belongs to $\boldsymbol{P}_{xy}$ and is folded into $p_{G'}$. (1) For the case that $p_{G_k}$ does not end with $x$ nor $y$, it can be inferred that $p_{G'}$ corresponds to a transition chain from $s_{init}$ to $s_k$ iff $\boldsymbol{P}_{xy}$ also contains a path corresponding to a transition chain from $s_{init}$ to $s_k$. (2) For the case that $p_G$ ends with $x$ or $y$, Property 4.3 also ensures that $p_{G'}$ corresponds to a transition chain from $s_{init}$ to $s_k$ such that $s_k \in F$ iff $\boldsymbol{P}_{xy}$ also contains a path corresponding to a transition chain from $s_{init}$ to $s'_k$ such that $s'_k \in F$. Therefore, for any $xy$-FEQ class $\boldsymbol{P}_{xy}$ in $G$ and its $xy$-folded path $p_{G'}$ in $G'$, the four rules of Figure 4.5 ensures that $p_{G'}$ is a reachable path iff $\boldsymbol{P}_{xy}$ contains a reachable path, which indicates the satisfaction of *Cond. 2*. □

Putting Lemmas 4.1, 4.2 and Definition 4.3 together, folding a node pair $(x, y)$ satisfying ① and ② in Theorem 4.1 preserves reachability equivalence. Namely, Theorem 4.1 is correct.

## 4.3 Graph-Folding Algorithm

We then give an efficient graph-folding algorithm GF that traverses and folds the input graphs for CFL-reachability. Our algorithm implements the graph folding principle (Theorem 4.1). GF has a linear time complexity with respect to the number of nodes in input graphs.

### 4.3.1 Identifying Foldable Node Pairs

#### 4.3.1.1 Implementation of Graph Folding Principle

Algorithm 5 describes a practical realization of the folding principle in Theorem 4.1. The characteristic is that it only needs to compute the transitions involving the incoming and outgoing edges of a pair of adjacent nodes $x$ and $y$. Specifically, in lines 2–3, we set the values of $Q_x$ and $Q_y$ as their overapproximate supersets according to Principle ② in Theorem 4.1. lines 4–5 and lines 6–7 verify the foldability of the input node pair based on Principles ① and ②, respectively.

**Time Complexity.** Algorithm 5 has a time complexity of $O(|B|^2 \times |N| \times |\Sigma|^3)$, where $N = \cup_{i \in \{1,\cdots,t\}} N_i$ is the collection of all local states of the RSM. In Algorithm 5, the time complexity of lines 2–5 can be regarded as $O(1)$ as they can be considered as lookups of hash tables. Then, the time complexity of Algorithm 5 depends on the subprocedure `Check`. We can first assume that a state transition $s_i \xrightarrow{t} s_j \in R$ can be performed in $O(1)$ time. According to Definition 4.6, given a set of label $L$ and two states $s_1$ and $s_2$, checking $s_1 \overset{L}{\simeq} s_2$ and $s_1 \overset{L}{\preceq} s_2$ needs to perform $O(|L|)$ transitions. Thus, the loops in lines 10–16 cost $(|Q_{v_1}| \times |L_{v_1v_2}| \times |L_{v_2\not{v}_1}| + |Q_{v_1}| \times |L_{v_1v_2}| \times |L_{v_2v_1}| \times |L_{v_1\_}|)$ time, where the value of $Q_{v_1}$ is given in line 2 or line 3. We can find that $O(|Q_{v_1}|) = O(|B|^2 \times |N|)$, $|L_{v_1v_2}| \leq |\Sigma|$, $|L_{v_2\not{v}_1}| \leq |\Sigma|$, $|L_{v_2v_1}| \leq |\Sigma|$ and $|L_{v_1\_}| \leq |\Sigma|$. Therefore, the time complexity of Algorithm 5 is $O(|B|^2 \times |N| \times |\Sigma|^3)$.

**Algorithm 5:** Implementation of Graph Folding Principle

```
Function Identify($x, y$)
    $Q_x := Nr(L_{\_x}) \cup \big(B \times Nr(L_{\_x})\big) \cup \big(B^2 \times Nr(L_{\_x})\big)$;
    $Q_y := Nr(L_{\_y}) \cup \big(B \times Nr(L_{\_y})\big) \cup \big(B^2 \times Nr(L_{\_y})\big)$;
    if $L_{yx} = \emptyset$ and $\big(y \in V_{src}$ or $L_{\not{x}y} \neq \emptyset\big)$ then        /* Theorem 4.1: ① */
        return false;
    if Check($x, y$) and Check($y, x$) then        /* Theorem 4.1: ② */
        return true;
    return false;
Procedure Check($v_1, v_2$)
    for each $s_{v_1} \in Q_{v_1}$ do
        for each $\ell_1 \in L_{v_1 v_2}$ do
            if not exists $\big(s_{v_1} \xrightarrow{\ell_1} s_{v_2} \in R$ s.t. $s_{v_1} \overset{L_{v_2 \not{v}_1}}{\simeq} s_{v_2}\big)$ then
                return false;        /* Rules [x-y] and [y-x] of Figure 4.5 */
            for each $\ell_2 \in L_{v_2 v_1}$ do
                if exists $\big(s_{v_1} \xrightarrow{\ell_1} s_{v_2} \xrightarrow{\ell_2} s'_{v_1} \in R$ s.t. $\neg(s'_{v_1} \overset{L_{v_1 \_}}{\preceq} s_{v_1})\big)$ then
                    return false;        /* Rules [x-x] and [y-y] of Figure 4.5 */
    return true;
```

#### 4.3.1.2 Efficient Identification for Foldable Node Pairs

Algorithm 5 runs fast for small RSMs, but the overall runtime increases when the RSMs become more complex. For real-world problems, we need a more efficient identification strategy. In fact, real-world CFL-reachability problems usually have an important trait—many node pairs share the same "pattern" of incoming and outgoing edges. Specifically, we define the "pattern" of a node pair $(x, y)$ as a tuple:

$$Pattern_{(x,y)} = \langle isSrc(x),\ isSrc(y),\ L_{\not{y}x},\ L_{\not{x}y},\ L_{xy},\ L_{yx},\ L_{x\not{y}},\ L_{y\not{x}} \rangle \tag{4.3}$$

**Algorithm 6:** Efficient Identification of Foldable Node Pairs

```
Ω⊕: a set holding the patterns of foldable node pairs.
Ω⊖: a set holding the patterns of non-foldable node pairs.
Function IsFoldable(x, y)
    consult the incoming and outgoing edges to determine $Pattern_{(x,y)}$;
    if $Pattern_{(x,y)} \in \Omega_\oplus$ then return true ;
    if $Pattern_{(x,y)} \in \Omega_\ominus$ then return false ;
    if Identify(x, y) then                              /* Algorithm 5 */
        $\Omega_\oplus := \Omega_\oplus \cup \{Pattern_{(x,y)}\}$
        return true;
    $\Omega_\ominus := \Omega_\ominus \cup \{Pattern_{(x,y)}\}$
    return false;
```

where $isSrc(x)$ and $isSrc(y)$ are two boolean variables denoting whether $x \in V_{src}$ and whether $y \in V_{src}$. The value of $Pattern_{(x,y)}$ can be predefined or determined by consulting $x$ and $y$ and their incoming and outgoing edges. Obviously, for two node pairs $(x_1, y_1)$ and $(x_2, y_2)$, checking whether $Pattern_{(x_1,y_1)} = Pattern_{(x_2,y_2)}$ is much faster than calling Algorithm 5 twice.

For the node pairs sharing the same pattern $Pattern_{(x,y)}$, we do not need to repeatedly invoke Algorithm 5 to check whether they are foldable. We use a set denoted by $\Omega_\oplus$ to collect the patterns of node pairs that are already identified as foldable by Algorithm 5. When $\Omega_\oplus$ is fully filled, we can identify whether a node pair is foldable by checking whether the pattern of the node pair is already in $\Omega_\oplus$. $\Omega_\oplus$ can be filled by verifying each possible pattern through Algorithm 5, where the number of invocation depends only on the size of the alphabet $\Sigma$. Besides, we provide another strategy in Algorithm 6, which dynamically constructs $\Omega_\oplus$ during the process of graph folding with a set $\Omega_\ominus$ holding patterns of non-foldable node pairs.

**Algorithm 7:** Graph-Folding Algorithm GF

```
    $V_{visited}$: a set holding visited nodes.
    $E_{visited}$: a set holding visited edges.
 1  Function GF(G,R)
 2      set $\Omega_\oplus$, $\Omega_\ominus$, $V_{visited}$ and $E_{visited}$ as $\emptyset$;
 3      for each $x \in V$ do
 4          if $x \notin V_{visited}$ then Visit($x$); ;
 5      return $G$;
 6  Procedure Visit(x)
 7      $V_{visited} := V_{visited} \cup \{x\}$
 8      for each $x \xrightarrow{\ell} y \in E$ s.t. $y \notin V_{visited}$ and $x \xrightarrow{\ell} y \notin E_{visited}$ do
 9          add all edges joining $x$ and $y$ into $E_{visited}$;
10          if IsFoldable($x,y$) then
11              Fold($x,y$);
12          Visit($y$)
13  Procedure Fold(x,y)
14      Remove all edges joining $x$ and $y$;
15      for each $z \in V$ s.t. $Rep(z) = y$ do          /* Update representative nodes */
16          $Rep(z) := x$;
17      Remove node $y$;                                  /* Merge y into x */
```

### 4.3.2 Overall Algorithm

Algorithm 7 describes the overall graph-folding algorithm GF. In addition to $\Omega_\oplus$ and $\Omega_\ominus$ in Algorithm 6, Algorithm 7 maintains two sets $V_{visited}$ and $E_{visited}$ to collect the visited nodes and edges. The procedures *GF* and *Visit* scan the input graph via a depth-first traversal. When visiting a node $x$, the algorithm calls the identification procedure *isFoldable* for each unvisited direct successor $y$ of $x$ to check whether $(x,y)$ is foldable. If $(x,y)$ is foldable, the algorithm uses *Fold*$(x,y)$ in lines 13–17 to fold $(x,y)$ and update representative nodes.

**Complexity.** Algorithm 7 has a linear time complexity with respect to the number of nodes in the input graph. For real-world problems where the RSM is far smaller than the graph, the cost for identifying and folding a node pair (lines 10–11) can be considered as $O(1)$ time. The ordinary depth-first traversal costs $O(|V|+|E|)$ time. However, different from the ordinary depth-first traversal, line 9 ensures that each node should not be visited twice through different edges. Lines 8–12 show that the number of method invocations for $\mathtt{IsFoldable}(x,y)$ and $\mathtt{Fold}(x,y)$ does not exceed the number of visited nodes. In Algorithm 7, each node is visited once, hence the time complexity of Algorithm 7 is $O(k|V|)$ where $k$ is a constant representing the time for identifying and folding a node pair. Namely, GF has a linear time complexity with respect to the number of nodes of the input graph.

## 4.4 Experiment

We evaluate our graph folding technique by applying it to two popular clients on C/C++ static analysis: value-flow analysis and alias analysis. In particular, we study the performance of GF from two aspects: (1) the performance of GF in reducing the input graph sizes and (2) the speedups and memory overhead reductions of CFL-reachability solving, with the input graphs simplified by GF.

We use a recent algorithm proposed in Graspan [133] as the CFL-reachability solver. We compare the effectiveness of GF with the baseline and two state-of-the-art graph simplification techniques: (1) "Base" (a baseline approach that solves CFL-reachability on the graph without any simplification), (2) "SCC" (cycle elimination [92, 127]), and (3) "InterDyck" (InterDyck graph simplification [80]).

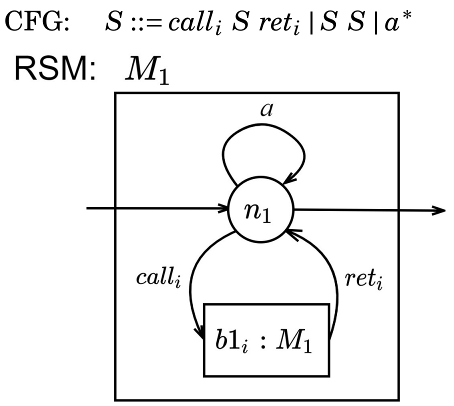


**Figure 4.8.** The CFG and RSM for C/C++ context-sensitive value-flow analysis.

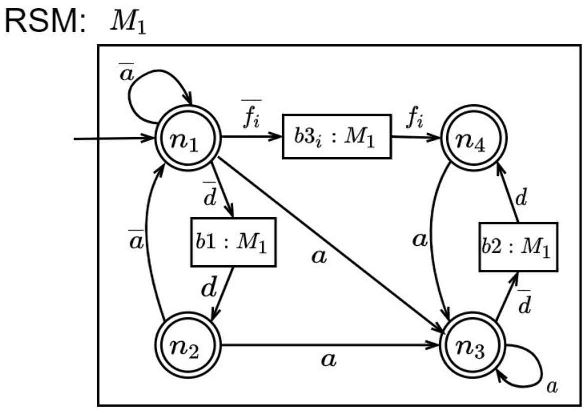


**Figure 4.9.** The RSM for C/C++ field-sensitive alias analysis, where the nodes in double circle denote the exits of the box.

### 4.4.1 Experimental Setup

We have implemented GF on top of LLVM-12.0.0, and conducted our experiment on a platform consisting of an eight-core 2.60GHz Intel Xeon CPU with 128 GB memory, running Ubuntu 18.0.4. In our experiment, the patterns of foldable node pairs, i.e., $\Omega_\oplus$ in Section 4.3.1, are precomputed.

**Value-flow analysis.** We conduct context-sensitive value-flow analyses on sparse value-flow graphs (SVFGs) [126] abstracted of our benchmarking programs. The CFG and the RSM for context-sensitive value-flow analysis are displayed in Figure 4.8, where "$call_i$" and "$ret_i$" denote call and return of a callsite, whose index is $i$; and "$a$" denotes an assignment instruction. In the RSM, the initial state and the final state are both $\langle n_1 \rangle$, and the unique box $b$ has a subscript $i$ matching the index of $call_i$ and $ret_i$. In this problem, nodes denoting the allocation/deallocation sites are marked as sources/sinks. $call_i$-$ret_i$ forms matched parentheses. It is worthwhile to point out that although the CFG in Figure 4.8 only focuses on context-sensitivity, since each field object in SVFG is already represented as a distinct node, the analysis is also field-sensitive.

**Alias analysis.** We conduct all-pair field-sensitive alias analyses based on the RSM in Figure 4.9, which is constructed from the CFG proposed in [149]. $a$ denotes assignment, $d$ denotes dereference and $f_i$ denotes the address of the $i$-th field. The RSM contains three boxes $b1$, $b2$ and $b3$, which are all mapped to $M_1$. $b3$ has a subscript $i$ matching the field index of $\overline{f_i}$ and $f_i$. The RSM has an initial state $\langle n_1 \rangle$ and four accepting states $\langle n_1 \rangle$, $\langle n_2 \rangle$, $\langle n_3 \rangle$ and $\langle n_4 \rangle$. The alias analysis is conducted on a program expression graph (PEG), which is bi-directed, i.e., for each $v_i \xrightarrow{t} v_j \in E$ where $t \in \Sigma$, there is a reverse $v_j \xrightarrow{\bar{t}} v_i \in E$. $\overline{d}$-$d$ and $\overline{f_i}$-$f_i$ form matched parentheses.

**Benchmarks.** We use 10 popular GitHub open-source C/C++ programs to benchmark our analysis as listed in Table 4.3. We select these programs because they are diverse in terms of functionalities as they include: development (`astyle`, `nvim`), version control (`git-checkout`), compiler (`janet`, `mruby`), database (`psql`, `redis_cli`), computer vision (`opencv_test_video`), window manager (`i3`) and terminal multiplexer (`tmux`). The SVFG and PEG of each program are generated by the open-source tool SVF [125] from the bitcode files compiled by Clang-12.0.0 from source code with the `-O3` flag and integrated by Whole Program LLVM[2]. For each input graph, cycle elimination detects and collapses cycles comprised of $a$-edges, and the InterDyck algorithm detects and eliminates the non-Dyck-contributing [80] parenthesis edges. Table 4.3 displays the graph statistics and the results of the baseline approach.

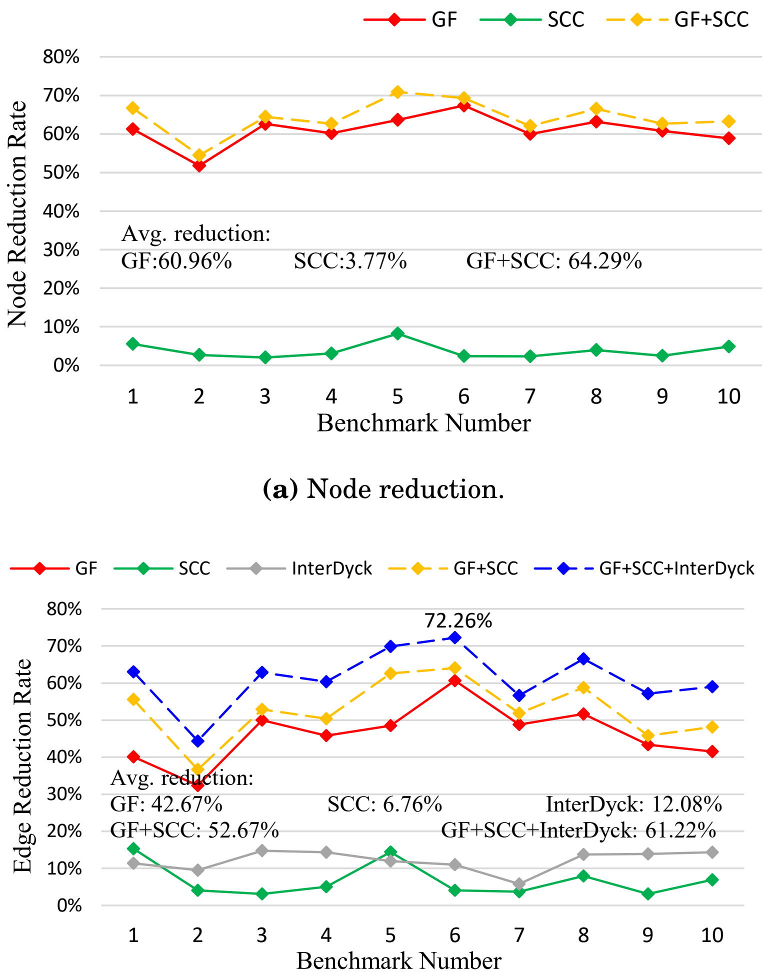
### 4.4.2 Performance in Reducing Graph Sizes

Figures 4.10 and 4.11 depict the reduction rates of nodes and edges in the input graphs of value-flow analysis and alias analysis, respectively. As the InterDyck algorithm never removes nodes from a graph, the node reduction charts (Figure 4.10(a) and Figure

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

**Table 4.3.** Benchmark info and results of the baseline. #Node and #Edge denote the number of nodes and edges of each input graph. P-Edge% denotes the percentage of parenthesis edges out of total edges of each input graph. Time/s and Mem./GB denote the runtime and memory overhead of the baseline for analyzing each program, measured in seconds and gigabytes, respectively.

| **Bench.** | ***Value-flow analysis*** | | | | | ***Alias analysis*** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | #Node | #Edge | P-Edge% | Time/s | Mem./GB | #Node | #Edge | P-Edge% | Time/s | Mem./GB |
| 1.astyle | 227140 | 394839 | 16.58% | 14287 | 17.82 | 70906 | 150090 | 45.84% | 6327 | 7.08 |
| 2.git_checkout | 488092 | 919390 | 37.65% | 48807 | 26.60 | 78801 | 184734 | 42.05% | 51024 | 10.26 |
| 3.i3 | 145259 | 212761 | 29.96% | 809 | 3.41 | 39894 | 95620 | 43.57% | 8383 | 7.50 |
| 4.janet | 227081 | 359552 | 33.10% | 7879 | 11.19 | 43837 | 99292 | 43.17% | 19085 | 37.14 |
| 5.mruby | 226528 | 340374 | 19.18% | 26938 | 23.63 | 71265 | 176888 | 43.17% | 26368 | 27.93 |
| 6.nvim | 332733 | 402753 | 21.75% | 880 | 3.73 | 99826 | 224170 | 43.18% | 49600 | 20.17 |
| 7.opencv_test_video | 385060 | 509439 | 10.47% | 785 | 1.67 | 132068 | 279528 | 40.13% | 24638 | 8.27 |
| 8.psql | 157014 | 228604 | 27.43% | 2486 | 8.40 | 40145 | 99164 | 44.59% | 8808 | 12.54 |
| 9.redis-cli | 231372 | 367291 | 34.59% | 1802 | 5.59 | 55250 | 129528 | 45.45% | 20459 | 17.70 |
| 10.tmux | 243828 | 390084 | 29.11% | 15218 | 8.77 | 76522 | 186808 | 45.63% | 61168 | 25.52 |

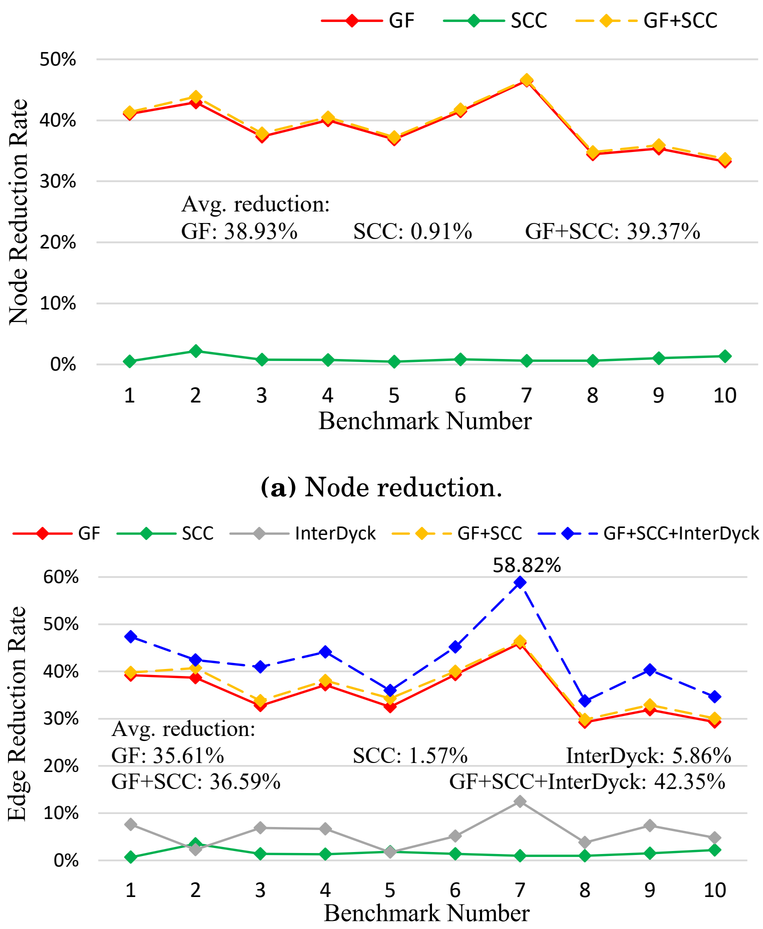


**(a)** Node reduction.

**(b)** Edge reduction.

**Figure 4.10.** Reduction rates of nodes and edges in the input graphs of value-flow analysis.

4.11(a)) do not include the information of InterDyck. A comparison of SCC, InterDyck and GF shows that GF is more powerful than SCC and InterDyck in reducing the size of the input graphs for both clients. Specifically, by comparing GF with SCC, we find that cycle elimination reduces only a small number of nodes and edges in the preprocessing stage. In contrast, prior work shows that cycle elimination behaves well in particular clients, e.g., constraint-based pointer analysis because it detects and collapses cycles dynamically in the solving procedure [51, 97]. This is beyond the topic of this paper, and hence is not discussed. By comparing GF with InterDyck, we find that InterDyck can reduce the number of parenthesis edges. But since the proportions taken by parenthesis



**(a)** Node reduction.

**(b)** Edge reduction.

**Figure 4.11.** Reduction rates of nodes and edges in the input graphs of alias analysis, where GF denotes graph folding, SCC denotes cycle elimination and InterDyck denotes the InterDyck graph simplification.

edges are not large (see Columns 4 and 9 of Table 4.3) in these two real-world clients, whereby the effectiveness of the InterDyck algorithm is limited. Besides, a comparison between node reduction rates and edge reduction rates of GF shows that folding tends to increase the density (#Edge/#Node) of the graph.

We also study the combinations GF+SCC and GF+SCC+InterDyck, which are depicted as the dashed lines in Figures 4.10 and 4.11. The result shows that GF complements other approaches (e.g., SCC and InterDyck) well (i.e., they can be used together to further improve the performance of CFL-reachability). Table 4.4 shows the runtime of GF, SCC

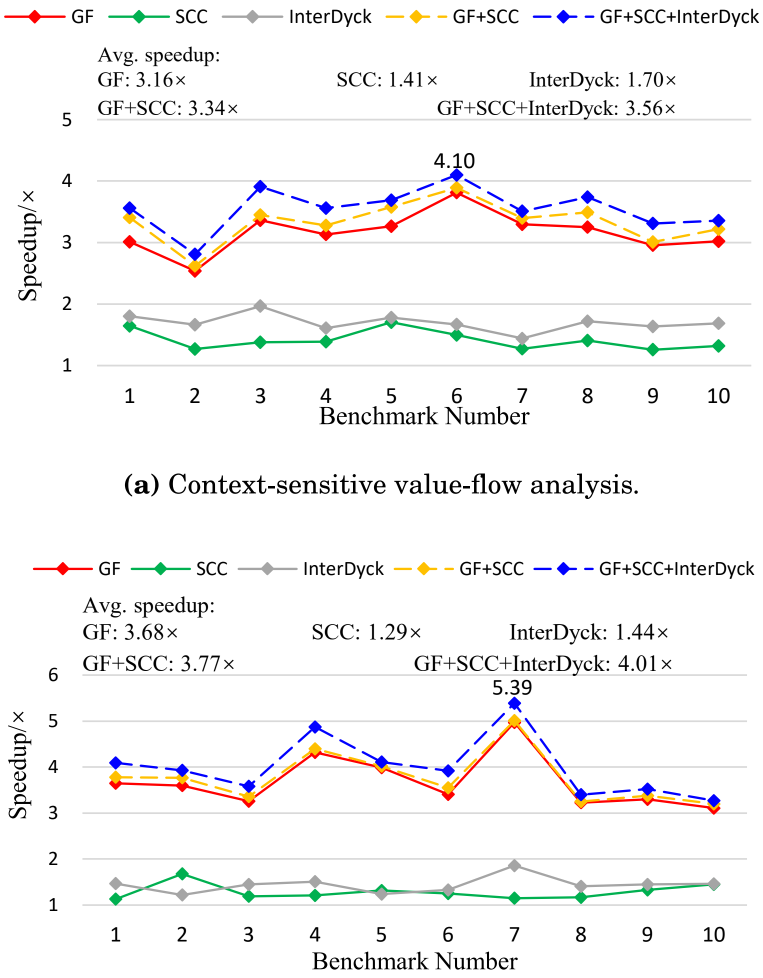
**Table 4.4.** Runtime of GF, SCC and InterDyck, measured in seconds.

| **Bench.** | ***SVFG*** | | | ***PEG*** | | |
|---|---:|---:|---:|---:|---:|---:|
| | **GF** | **SCC** | **InterDyck** | **GF** | **SCC** | **InterDyck** |
| 1.astyle | 3.70 | 0.69 | 273.04 | 0.29 | 0.13 | 19.82 |
| 2.git_checkout | 15.29 | 2.09 | 1625.84 | 0.40 | 0.21 | 337.87 |
| 3.i3 | 2.63 | 0.27 | 177.74 | 0.54 | 0.05 | 45.96 |
| 4.janet | 4.01 | 0.80 | 464.37 | 0.19 | 0.07 | 27.41 |
| 5.mruby | 15.74 | 0.82 | 718.08 | 0.44 | 0.15 | 257.75 |
| 6.nvim | 16.72 | 0.94 | 152.02 | 0.44 | 0.31 | 122.89 |
| 7.opencv_test_video | 57.80 | 2.35 | 541.36 | 0.55 | 0.47 | 22.54 |
| 8.psql | 2.39 | 0.30 | 130.31 | 0.29 | 0.05 | 37.56 |
| 9.redis-cli | 7.34 | 0.64 | 143.18 | 0.21 | 0.14 | 85.16 |
| 10.tmux | 10.13 | 0.75 | 1040.18 | 0.32 | 0.16 | 216.08 |

and InterDyck when they are applied separately to the graphs. The results show that SCC is the fastest, followed by GF, and InterDyck is much slower. This is because implementing InterDyck in these two clients needs to (1) detect and contract all non-parenthesis edges, (2) use the FastDyck algorithm [145] to find Dyck-contributing edges and mark the "anchor" nodes and (3) detect and remove non-Dyck-contributing edges, where (1) costs $O(|V|^2)$ time, (2) costs $O(|V|+|E|\log|E|)$ time and (3) costs $O(|E|\log|E|)$ time. Hence, to effectively combine the three graph simplification techniques, we suggest running SCC first, then GF and finally InterDyck, using the simplified graph to reduce the runtime of InterDyck.

### 4.4.3 Speedup and Memory Overhead

Figure 4.12 shows the speedups of CFL-reachability in the two clients by GF, SCC, InterDyck and their combinations. Taking the edge reduction rate (Figure 4.10(b) and Figure 4.11(b)) into consideration and comparing the speedups among different graph simplifi-



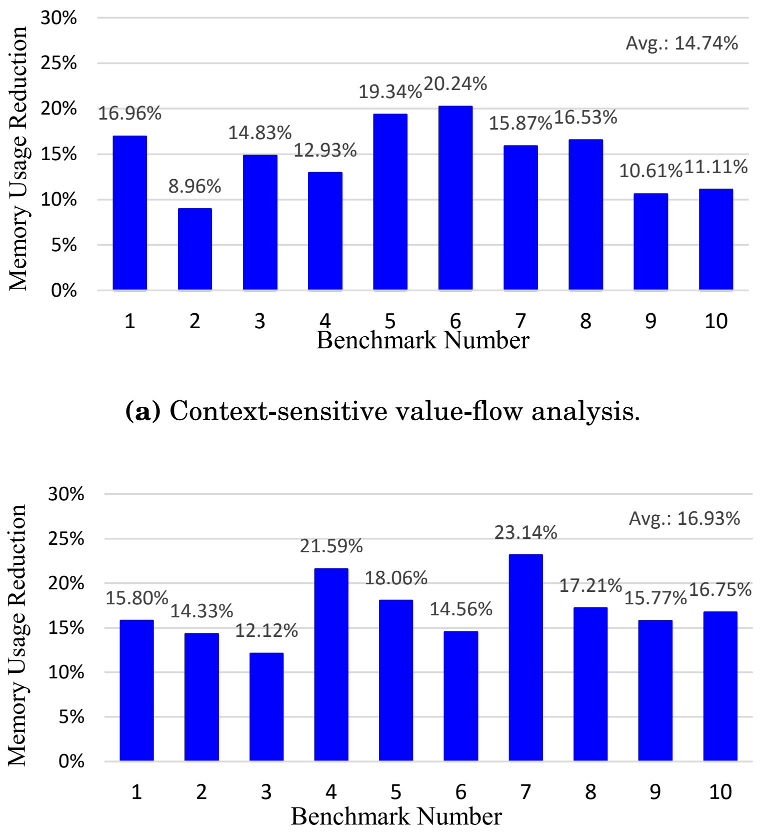
**(a)** Context-sensitive value-flow analysis.

**(b)** Field-sensitive alias analysis.

**Figure 4.12.** Speedups of CFL-reachability by GF, SCC, InterDyck and their combinations.

cation approaches, we observe that the reduction of more edges from the graphs helps to accelerate CFL-reachability solving. Comparing the speedups of a graph simplification approach among different programs, we see that larger edge reduction rates usually, but not always, result in larger speedups. This is because the runtime of CFL-reachability solving depends not only on the size of the input graph but also on other graph traits such as density, edge types, etc.

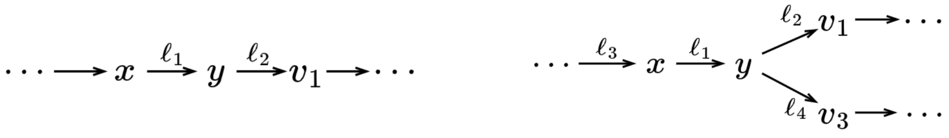
Figure 4.13 shows the reduction rates of memory overheads by GF in the two clients. We see that by reducing the size of the input graphs, GF effectively reduces the memory overhead of CFL-reachability analysis for both clients. Comparing Figure 4.12 and



**(a)** Context-sensitive value-flow analysis.

**(b)** Field-sensitive alias analysis.

**Figure 4.13.** Reduction rates of memory overhead by GF.


$\cdots \longrightarrow x \xrightarrow{\ell_1} y \xrightarrow{\ell_2} v_1 \longrightarrow \cdots$
$\cdots \xrightarrow{\ell_3} x \xrightarrow{\ell_1} y$
$\ell_2$
$v_1 \longrightarrow \cdots$
$\ell_4$
$v_3 \longrightarrow \cdots$

**(a)** Instance 1.

**(b)** Instance 2.

**Figure 4.14.** Two instances of foldable node pair $(x, y)$ in the problem running upon the RSM of Figure 4.4.

Figure 4.13, we see that the largest/smallest speedups correspond to the largest/smallest reduction rates of memory overhead. What lies behind this correlation is that solving CFL-reachability is a procedure that uses existing edges to create and add new edges to the graph. Such edges are stored in memory. For solving CFL-reachability, a larger speedup means a larger reduction rate of edges that need to be created and stored, hence means a larger reduction rate of memory overhead.

### 4.4.4 Discussions

**Graph Folding and Domain-Specific Edge Contraction** Perhaps the best-known domain-specific edge contraction technique in static analysis is the offline variable substitution (OVS) for pointer analysis [111]. Specifically, OVS can contract a *copy*-edge from a node $x$ to another node $y$ if $y$ does not have its address taken and $y$ does not have any other incoming *copy*-edge. In our evaluated alias analyses, GF folds a node pair $(x, y)$ where there is an $a$-edge from $x$ to $y$ if $y$ does not have any incoming $d$-edge and $f_i$-edge, and does not have any incoming $a$-edge not from $x$. The intrinsic meaning of the folding condition of GF is equivalent to that of OVS.[3] Thus, OVS can be viewed as an instantiation of graph folding in pointer/alias analysis.

**Foldability of Node Pairs v.s. Self-Loops in the RSM** By observing the RSMs of the two clients of our experiments (Figures 4.8 and 4.9), a common feature is that both RSMs contain self-loops, *e.g.*, $n_1 \xrightarrow{a} n_1$ in the RSM of Figure 4.8. It is interesting to note that a foldable node pair does not necessarily need to be joined by an edge that corresponds to a self-loop in the RSMs. Consider the node pair $(x, y)$ in Figure 4.4(c), whose foldability is discussed in Example 4.7. In the example, the edge $x \xrightarrow{\ell_1} y$ does not correspond to a self-loop in the RSM of Figure 4.4(a). In fact, the RSM in this example does not contain any self-loop.

### 4.4.5 Summary

The results from the empirical evaluation of GF are promising. On average, by reducing node counts by 60.96% and edge counts by 42.67% on benchmark input graphs, GF accelerates context-sensitive value-flow analysis by 3.16× with a memory usage reduction of 14.74%. By reducing node counts by 38.93% and edge counts by 35.61% on benchmark input graphs, GF accelerates field-sensitive alias analysis by 3.68× with a

[3] In particular, an $f_i$-edge is regarded as a *copy*-edge with an offset $i$ in field-sensitive analysis [96].

memory usage reduction of 16.93%. GF is also compatible with existing techniques. The combination of GF with SCC and InterDyck reduces up to 72.26% and 58.85% edges from the input graph of context-sensitive value-flow analysis and field-sensitive alias analysis, respectively, empirically accelerate those analyses by up to 4.10× and 5.39×. Furthermore, GF subsumes some existing client-based edge contraction techniques and is more expressive than SCC and InterDyck.

CHAPTER 5

# Derivation Equivalence Based Set Constraint Solving

In this chapter, we demonstrate our derivation equivalent algorithm in the form of set constraint analysis and on the client of field-sensitive pointer analysis [10, 96], where positive weight cycles arose and extensively studied [76, 96, 97, 121].

## 5.1 Problem Formulation

### 5.1.1 Pointer Analysis in Set Constraints

Andersen first formulated pointer analysis as the form of set constraint problem in his PhD thesis [10]. So constraint-based pointer analysis is also called Andersen's pointer analysis. In the constraint graph $G = \langle V,E \rangle$ of Andersen's pointer analysis, pointers and abstract memory objects are modeled as nodes with each node $v_i \in V$ assigned with a point-to set $pts(v_i)$ holding the possible point-to targets (pointees) of $v_i$ at runtime. The program instructions involving pointers are formulated as constraints among the point-to sets of the nodes, with each constraint represented by an edge. The early version of Andersen's pointer analysis contains four types of constraints [51, 56], as shown in

**Table 5.1.** Program instructions, constraints and edges.

| | Instruction | Constraint | Edge Type | Meaning |
|---|---|---|---|---|
| Base | `p = &o` | $\{o\} \subseteq p$ | $o \xrightarrow{\texttt{Addr}} p$ | $o \in pts(p)$ |
| Direct | `q = p` | $p \subseteq q$ | $p \xrightarrow{\texttt{Copy}} q$ | $pts(p) \subseteq pts(q)$ |
| | `q = &p` → $\texttt{f}_i$ | $p + i \subseteq q$ | $p \xrightarrow{\texttt{Field}_i} q$ | $\forall o \in pts(p) : o_{[i]} \in pts(q)$ |
| Indirect | `*q = p` | $p \subseteq {}^*q$ | $p \xrightarrow{\texttt{Store}} q$ | $\forall o \in pts(q) : pts(p) \subseteq pts(o)$ |
| | `q = *p` | ${}^*p \subseteq q$ | $p \xrightarrow{\texttt{Load}} q$ | $\forall o \in pts(p) : pts(o) \subseteq pts(q)$ |

Table 5.1 except for the third one $(p+i) \subseteq q$.

Among the four types of constraints, Addr directly put a node into the point-to set of another; Copy is a direct constraint as it directly propagate the point-to set from a node to another; Store and Load are indirect constraints as they involve the propagation of the point-to sets of the pointees of nodes and such constraints are usually solved by adding direct edges, i.e., Copy edges, into the graph. Specifically, $p \xrightarrow{\texttt{Store}} q$ is handled by adding $p \xrightarrow{\texttt{Copy}} o$ to the graph for all $o \in pts(q)$; $p \xrightarrow{\texttt{Load}} q$ is handled by adding $o \xrightarrow{\texttt{Copy}} q$ to the graph for all $o \in pts(p)$.

Complex program instructions are transformed into the four basic ones in Table 5.1 using auxiliary variables. For example, `*q = *p` is transformed into $\texttt{v}_1$ `= *p` and `*q =` $\texttt{v}_1$ and then be represented by a Load a Store edge in the constraint graph.

Constraint-based pointer analysis is solving using a dynamic transitive closure algorithm [17, 38, 56, 95]. The algorithm maintains a worklist $W$ holding nodes which need to propagate their point-to set via their outgoing direct edges. The algorithm iteratively processes and removes the nodes in the worklist, propagating point-to sets among nodes and adding edges to the graph based on the rules for solving constraints, until the worklist is empty, which means that there is not any new point-to set propagation to do, i.e., all the constraints in the graph are satisfied. Figure 5.1 is a flowchart of the dynamic transitive closure algorithm.

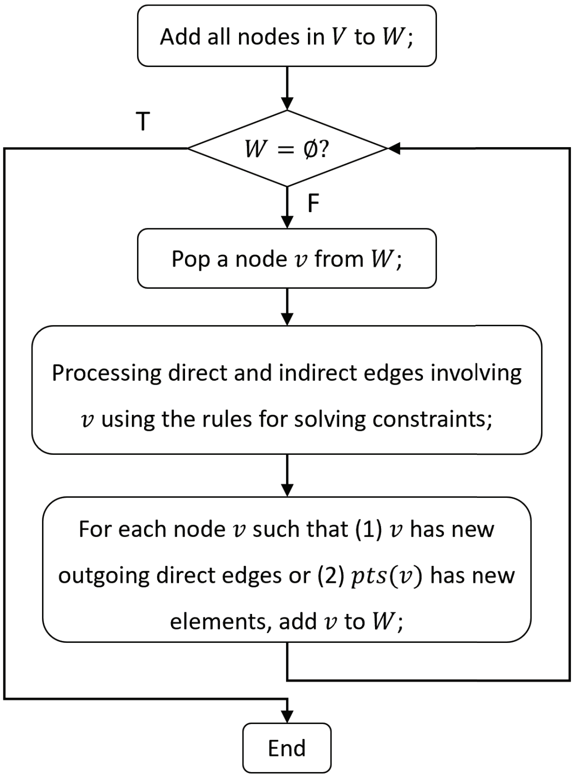

Add all nodes in $V$ to $W$;
$W = \emptyset$?
T
F
Pop a node $v$ from $W$;
Processing direct and indirect edges involving $v$ using the rules for solving constraints;
For each node $v$ such that (1) $v$ has new outgoing direct edges or (2) $pts(v)$ has new elements, add $v$ to $W$;
End

**Figure 5.1.** Flowchart of dynamic transitive closure algorithm for pointer analysis.

### 5.1.2 Field-Sensitivity and Positive Weight Cycles

The prominent constraint-based field-sensitive pointer analysis was proposed by Pearce et al. [96], who used weighted constraints to model field accesses in C/C++. In the literature, the instruction `q = &p → f`$_i$ – where `f`$_i$ denotes the $i$-th field of the point-to object of p – is formulated into a constraint $(p+i) \subseteq q$ and is represented by an edge $p \xrightarrow{\texttt{Field}_i} q$ in the constraint graph.

In field-sensitive analysis, the constraint $(p+i) \subseteq q$ means that

$$\forall o \in pts(p) : o_{[i]} \in pts(q),$$

where $o_{[i]}$ denotes the base location $o$ with an offset $i$.

In the *classical field modeling* by Pearce et al. [95], $o_{[i]}$ represents the $i$-th field of $o$. Besides, in the ANSI-compliant [64] field modeling, the fields of nested structs flattened, i.e., $(o_{[i]})_{[j]}$ is represented by $o_{[i+j]}$. Figure 5.2 displays the five rules for

$$\texttt{[Addr]}\ \frac{o \xrightarrow{\texttt{Addr}} p \in E}{o \in pts(p)} \qquad \texttt{[Copy]}\ \frac{p \xrightarrow{\texttt{Copy}} q \in E}{pts(p) \subseteq pts(q)}$$

$$\texttt{[Store]}\ \frac{p \xrightarrow{\texttt{Store}} q \in E \quad o \in pts(q)}{p \xrightarrow{\texttt{Copy}} o \in E} \qquad \texttt{[Load]}\ \frac{p \xrightarrow{\texttt{Load}} q \in E \quad o \in pts(p)}{o \xrightarrow{\texttt{Copy}} q \in E}$$

$$\texttt{[Field-1]}\ \frac{p \xrightarrow{\texttt{Field}_i} q \in E \quad o \in pts(p)}{o_{[i]} \in pts(q)} \qquad \texttt{[Field-2]}\ \frac{p \xrightarrow{\texttt{Field}_i} q \in E \quad o_{[j]} \in pts(p)}{o_{[i+j]} \in pts(q)}$$

**Figure 5.2.** Rules for field-sensitive pointer analysis.

solving constraints in field-sensitive pointer analysis, where the rule for processing `Field` is written in two parts to explicitly handle pointees without offsets and with offsets, respectively.

Algorithm 8 is the dynamic transitive closure algorithm for solving field-sensitive constraint-based pointer analysis. It differs from the original field-insensitive version by adding Lines 17–21 to process `Field` constraints. In general, `Field` edges are regarded as direct edges as processing them do not add any new edge to the graph. It is notable that processing `Field` generates new nodes from existing pointees to represent abstract fields (Line 19). This causes *infinite derivations* when `Field` edges are in the cycles of direct edges, which makes *positive weight cycles*, as seen in Definition 5.1.

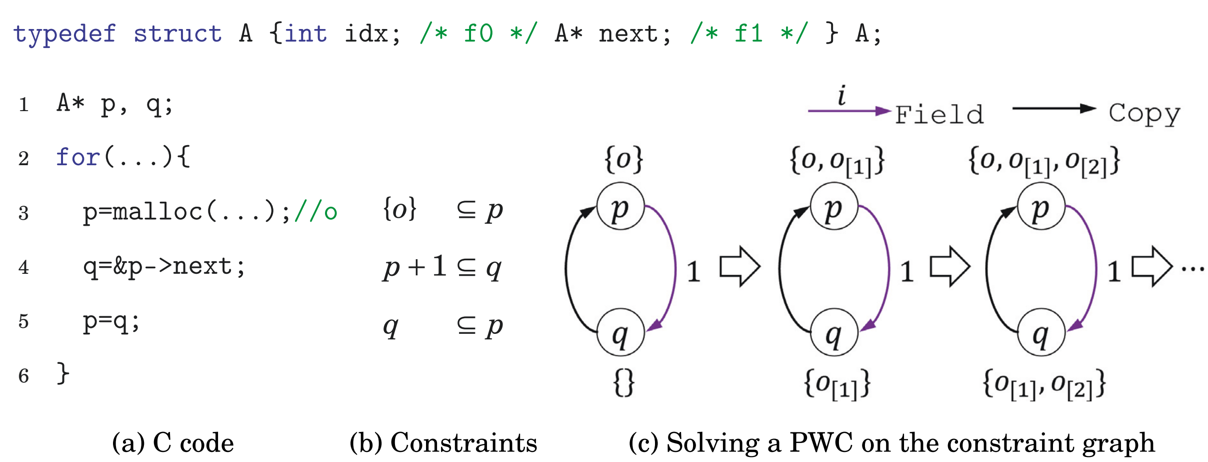
**Definition 5.1** (*Positive Weight Cycle (PWC)*)**.** In field-sensitive constraint-based pointer analysis, a positive weight cycle is a cycle comprised of `Copy` and `Field` edges.

Figure 5.3 is an example illustrating a PWC that incurs infinite derivations during constraint solving. Figure 5.3(b) gives the constraints transformed from the code according to Pearce et al.'s modeling [96]. Figure 5.3(c) shows its corresponding constraint graph with a PWC containing a positive weighted edge $p \xrightarrow{\texttt{Field}_1} q$ (representing the constraint $p+1 \subseteq q$) and a simple copy edge from $q \xrightarrow{\texttt{Copy}} p$ (representing the constraint $q \subseteq p$). An abstract object $o$ allocated at line 3 is initially added to $p$'s points-to set. Each

**Algorithm 8:** Field-sensitive Andersen's pointer analysis

```
Function PTA($G = \langle V,E \rangle$)
    for each $o \xrightarrow{\texttt{Addr}} p \in E$ do
        add $o$ to $pts(p)$; add $p$ to $W$;
    while $W \neq \emptyset$ do
        select and remove an node $p$ from $W$;
        for each $o \in pts(p)$ do                         /* Process indirect edges */
            for each $p \xrightarrow{\texttt{Load}} q \in E$ do
                if $o \xrightarrow{\texttt{Copy}} q \notin E$ then
                    add $o \xrightarrow{\texttt{Copy}} q$ to $E$ and add $o$ to $W$;
            for each $q \xrightarrow{\texttt{Store}} p \in E$ do
                if $q \xrightarrow{\texttt{Copy}} o \notin E$ then
                    add $q \xrightarrow{\texttt{Copy}} o$ to $E$ and add $q$ to $W$;
        for each $p \xrightarrow{\texttt{Copy}} q \in E$ do    /* Process direct edges */
            $pts(q) \leftarrow pts(p) \cup pts(q)$;
            if $pts(q)$ is changed then
                add $q$ to $W$;
        for each $p \xrightarrow{\texttt{Field}_i} q \in E$ do
            for each $o \in pts(p)$ do
                add $o_{[i]}$ to $pts(q)$;
            if $pts(q)$ is changed then
                add $q$ to $W$;
```

time processing the edge $p \xrightarrow{\texttt{Field}_1} q$, a new field object is generated from existing one and propagated from $pts(p)$ to $pts(q)$, then propagated back to $pts(p)$ via $q \xrightarrow{\texttt{Copy}} p$ for a new round of field derivation, resulting in infinitely deriving fields $o_{[1]}, o_{[2]}, \cdots$ from the base object $o$. As the point-to sets of $p$ and $q$ is continuously changing, such loops will not terminate.



```
typedef struct A {int idx; /* f0 */ A* next; /* f1 */ } A;

A* p, q;
for(...){
  p=malloc(...);//o
  q=&p->next;
  p=q;
}
```

$\{o\} \subseteq p$

$p+1 \subseteq q$

$q \subseteq p$

(a) C code (b) Constraints (c) Solving a PWC on the constraint graph

**Figure 5.3.** Positive weight cycle and infinite derivations.

To avoid infinite derivations, existing works [15, 96] manually set an upper bound of the number of derivations for each abstract object. This trades off precision with practicality. For a stack and global object, its number of fields can be statically determined based on its declared types. However, the number of fields of a dynamically allocated heap object may be unknown without actually running the program. Thus, on one hand, if the upper bound is small, solving positive weight cycles will be highly imprecise although it can possibly be quicker. On the other hand, if the upper bound is set to be large enough, solving positive weight cycles will be more precise but very inefficient.

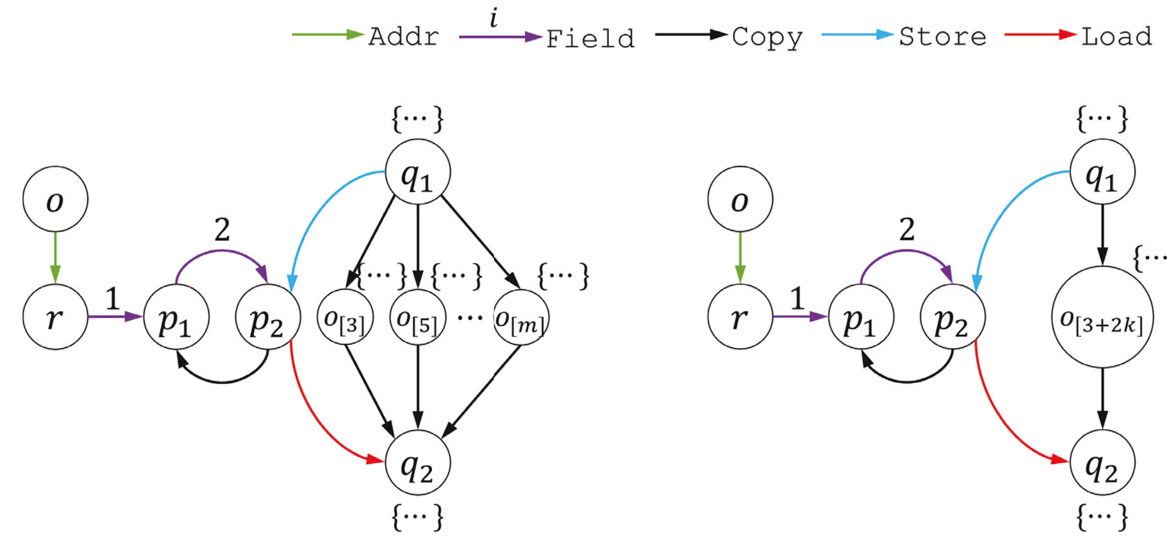
As a special instance of CFL-reachability, Andersen's pointer analysis has its properties which makes a large variety of offline/online graph simplification techniques [38, 51, 52, 95, 97, 111] available. An important property is that nodes in a cycle comprised of Copy edges are equivalent. Such equivalence manifests in that nodes in the cycle take part in identical point-to set propagation, and such nodes will have identical point-to sets after the overall constraint solving. Thus, collapsing a cycle consisting of Copy edges into a node will not change the pointer analysis result, hence is safe. In the literature, there are many existing techniques to detect and collapse cycles in the preprocessing stage [51] or during the constraint solving [38, 51, 95, 97].

However, in field-sensitive analysis, nodes in a PWC are not equivalent as they do not have identical point-to sets. This is obvious Figure 5.3(c). Moreover, contemporary programs contain a large amount of field accesses and nodes in PWCs usually occupy a large property of all nodes. Only merging cycles comprised of `Copy` edges while leaving PWCs without any optimization cannot effectively improve the efficiency. To achieve higher efficiency, some existing works [97, 121] treat `Field` edges in cycle the same as `Copy` edges and collapse PWCs. This obviously discounts the precision of field-sensitive analysis. Similarly, other graph simplification techniques like edge contraction [52, 111] are not suitable for handling `Field` edges in field-sensitive analysis.

### 5.1.3 Derivation Equivalence Based Constraint Solving

This chapter present a *derivation equivalence algorithm* DEA, a fast and precise approach to handle positive weight cycles. Rather than cycle elimination and edge contractions which merges equivalent original nodes in the graph, our technique collapses items dynamically derived during constraint solving by capturing derivation equivalence. Our insight is that two items derived from the same base object are *derivation equivalent* if they are always contained by the sets of the same nodes. When iteratively processing the cycles, such derivation equivalent items are produced using a particular initial value with some constant strides, which depends on the weights of the weighted edges. Based on this property, we propose a new *stride-based field representation* to capture the initial value and the constant strides of the items derived from the same positive weight cycle and avoid infinite derivation while preserving the precision, upon which, our DEA is constructed.

For the convenience of demonstration, our DEA is presented in the form of set constraint analysis under the client of field-sensitive Andersen's pointer analysis. According to the interconvertability between set constraint analysis and CFL-reachability [86], our technique is also applicable to CFL-reachability. The following part of this section uses an exmple to illustrate our solution.



$pts(r) = \{o\}$

$pts(p_1) = \{o_{[1]}, o_{[3]}, o_{[5]}, \cdots, o_{[m]}\}$

$pts(p_2) = \{o_{[3]}, o_{[5]}, \cdots, o_{[m]}\}$

**(a)** Traditional PWC solving.

$pts(r) = \{o\}$

$pts(p_1) = \{o_1\} \cup \{o_{[3+2k]}\}$

$pts(p_2) = \{o_{[3+2k]}\}$

**(b)** Our approach.

**Figure 5.4.** An example.

Figure 5.4 is an example showing the redundant derivations when solving a PWC on the constraint graph by the traditional approach [96] based on the inference rules in Figure 5.2. The example consists of five types of constant edges corresponding to the five types of instructions in Table 5.1. There is a PWC comprised of two edges $p_1 \xrightarrow{\texttt{Field}_2} p_2$ and $p_2 \xrightarrow{\texttt{Copy}} p_1$. Pointer $r$ initially points to $o$ because of $o \xrightarrow{\texttt{Addr}} r$ (Rule [Addr]). The point-to set of $p_1$ has the field $o_{[1]}$ derived from the object $o$ after solving $r \xrightarrow{\texttt{Field}_1} p_1$ (Rule [Field-1]). In the traditional PWC solving (Figure 5.4(a)), since the PWC formed by $p_1 \xrightarrow{\texttt{Field}_2} p_2$ and $p_2 \xrightarrow{\texttt{Copy}} p_1$ has a positive weight 2, a sequence of field objects starting from $o_{[3]}$ with a stride 2 are iteratively derived and added into the point-to set of $p_2$ (Rule [Field-2]) and then propagated back to the point-to set of $p_1$ (Rule [Copy]), until the upper bound of derivation number, e.g., $m$, is reached. These field objects are derivation equivalent because all of them are always pointed to by both $p_1$ and $p_2$ in this PWC, incurring redundant derivations. Even worse, handling the edge $q_1 \xrightarrow{\texttt{Store}} p_2$

that flowing into this PWC and the edge $p_2 \xrightarrow{\texttt{Load}} q_2$ that flowing out of this PWC adds redundant Copy edges (e.g., $q_1 \xrightarrow{\texttt{Copy}} o_{[3]}$ and $o_{[3]} \xrightarrow{\texttt{Copy}} q_2$) via Rules [Store] and [Load], which means that the point-to set of $q_1$ will be repeatly propagated to such field nodes and then to $q_2$ multiple times, as also illustrated in Figure 5.4(a).

In fact, such redundant field derivations and unnecessary point-to set propagations is avoidable. We achieve this by merging derivation equivalent fields into a stride-based polynomial representation $o_{[i+k\times s]}$, where $i$ is the starting field, $s$ is the stride corresponding to the weight of the PWC, and $k$ denotes a non-negative integer. Figure 5.4(b) illustrates the new representation $o_{[3+2k]}$ for collapsing derivation equivalent fields $\{o_{[3]}, \{o_{[5]}, \cdots\}$ in Figure 5.4(a). After the first time traversing the PWC, the stride of the fields is determined. Once the field representation $o_{[i+k\times s]}$ is derived and propagated into the point-to sets of $p_1$ and $p_2$, there is not any more iteration for processing the PWC is needed as the point-to information involving the PWC is determined. As a result, this field representation successfully reduces the number of points-to targets during points-to propagation and the number of Copy edges added into the constraint graph when solving Store/Load edges, while preserving the precision.

## 5.2 Our Solution

Our technique includes a stride-based field representation for collapsing derivation equivalent point-to targets, a series of rules for solving set constraints involving the field representation, and an efficient derivation equivalent algorithm DEA.

### 5.2.1 Stride-based Field Representation

**Definition 5.2** (*Stride-based Field Representation (SFR)*)**.** We use $\sigma = \langle o, i, S \rangle$ to denote a single object or a sequence of fields in classical modeling starting from $i$-th field following the strides in the set $S$. $\langle o,i,S\rangle$ represents objects or fields as follows:

$$\texttt{FieldExpansion}(\langle o,i,S\rangle) = \begin{cases} \{o\} & \text{if } S = \emptyset \wedge i = 0 \\ \{o_{[j]} \,\big|\, j = i + \sum_{n=1}^{|S|} k_n s_n,\ j \leq max, k_n \in \mathbb{N}, s_n \in S\} & \text{otherwise} \end{cases}$$

where

$max$ denotes the upper bound of field number [95] of object $o$;

$s_n \in S$ is the $n$-th element of the stride set $S$ which models precisely field derivations when a `Field` edge resides in one or multiple PWC; and

$k \in \mathbb{N}$ is a non-negative integer.

We use $\langle o,0,\emptyset\rangle$ to represent the entire object $o$ and its single field $o_{[i]}$ is denoted by $\langle o,i,\{0\}\rangle$. Stride-based field representation unifies the notations of an object and its fields. The expansion of an stride-based field representation fully represents the objects and fields in the classical modeling, while it reduces the number of points-to targets during constraint solving. With the upper bound of field number $max$ set, stride-based field representation is at least as precise as the classical modeling.

From the perspective of field expansion (Definition 5.3), two stride-based field representations can be disjointed or overlapping (Definition 5.4).

**Definition 5.3** (*Field Expansion*)**.** The field expansion of a stride-based field representation, denoted by $\texttt{FieldExpand}(\sigma)$, is to expand the field representation $\sigma$ back into a set holding the objects or fields that $\sigma$ represents.

**Definition 5.4** (*Overlapping and Disjointed SFRs*)**.** Two stride-based field representations $\sigma$ and $\sigma'$ are overlapping, denoted as $\sigma \sqcap \sigma' \neq \emptyset$, if

$$\texttt{FieldExpand}(\sigma) \cap \texttt{FieldExpand}(\sigma') \neq \emptyset.$$

We say that two stride-based field representations are *disjointed* if $\sigma \sqcap \sigma' = \emptyset$. A special case is the *subset relation*, denoted as $\sigma \sqsubseteq \sigma'$, which means that

$$\texttt{FieldExpand}(\sigma) \subseteq \texttt{FieldExpand}(\sigma').$$

*Example* 5.1. Given two stride-based field representations $\sigma = \langle o, 1, \{2\}\rangle$ and $\sigma' = \langle o, 1, \{5,6\}\rangle$, $\texttt{FieldExpand}(\sigma) = \{o_{[j]} \big| j = 1 + 2k,\ k \in \mathbb{N}\} = \{o_{[1]}, o_{[3]}, o_{[5]}, \cdots\}$ and $\texttt{FieldExpand}(\sigma') = \{o_{[j]} \big| j = 1 + 5k_1 + 6k_2,\ k_1, k_2 \in \mathbb{N}\} = \{o_{[1]}, o_{[6]}, o_{[7]}, o_{[11]}, o_{[12]}, \cdots\}$.

Since $\texttt{FieldExpand}(\sigma) \cup \texttt{FieldExpand}(\sigma') \neq \emptyset$, $\sigma \sqcap \sigma' \neq \emptyset$, i.e., $\sigma$ and $\sigma'$ are overlapping.

### 5.2.2 Inference Rules

Figure 5.5 gives the inference rules for solving set constraints of field-sensitive points-to analysis based on the stride-based field representation. Object and field nodes on the constraint graph are now represented by the unified SFRs.

**Addr and Copy.** Rule `[E-Addr]` initializes the points-to set of $p$ with object $o$ represented by $\langle o, 0, \emptyset\rangle$ for each $o \xrightarrow{\texttt{Addr}} p$. Similar to `[Copy]` in Figure 5.2, `[E-Copy]` simply propagates the points-to set of $p$ to that of $q$ when analyzing $p \xrightarrow{\texttt{Copy}} q$.

**Field.** After the first time traversing a positive weight cycle, the weight of the cycle can be determined. Our approach needs to collect the weights of positive weight cycles as the strides of the derivation equivalent fields.

**Definition 5.5** (Path and cycle)**.** A *path* $p \overset{*}{\Rightarrow} q$ on the constraint graph $G = \langle V, E\rangle$ is a sequence of edges leading from $p$ to $q$. A path $p \overset{*}{\Rightarrow} p$ is a *cycle* if all its edges are distinct and the only node occurring twice in this path is $p$.

**Definition 5.6** (Weight of a PWC)**.** A PWC, denoted as $\mathscr{C}$, is a cycle containing only `Copy` and `Field` edges and at least one edge is a `Field` with a positive weight. The weight of $\mathscr{C}$ is $\mathscr{W}_{\mathscr{C}} = \sum_{e \in \mathscr{C}} wt_e$, where $e$ is a `Field` or `Copy` edge and $wt_e$ is the weight of $e$ ($wt_e$ is 0 if $e$ is a `Copy` edge). The set of weights of all the PWCs containing $e$ is $\{\mathscr{W}_{\mathscr{C}} \mid \forall \mathscr{C} \subseteq E : e \in \mathscr{C}\}$.

Unlike rule `[Field-1]` and `[Field-2]` in Figure 5.2 which generate a single field object when processing $p \xrightarrow{\texttt{Field}_i} q$, `[E-Field]` generates an SFR $\sigma = \langle o, i + j, S \cup S'\rangle$ representing

$$\texttt{[E-AddrOf]}\ \frac{o \xrightarrow{\texttt{Addr}} p \in E \quad \sigma = \langle o, 0, \emptyset \rangle}{\sigma \in pts(p)} \qquad \texttt{[E-Copy]}\ \frac{p \xrightarrow{\texttt{Copy}} q \in E}{pts(p) \subseteq pts(q)}$$

$$\texttt{[E-Field]}\ \frac{\begin{array}{c} p \xrightarrow{\texttt{Field}_i} q \in E \quad \langle o, j, S \rangle \in pts(p) \quad S' = Strides(p \xrightarrow{\texttt{Field}_i} q) \\ \sigma = \langle o, i+j, S \cup S' \rangle \qquad \nexists\, \sigma' \in pts(q) : \sigma \sqsubseteq \sigma' \end{array}}{\sigma \in pts(q)}$$

$$\texttt{[E-Store]}\ \frac{p \xrightarrow{\texttt{Store}} q \in E \quad \sigma \in pts(q)}{p \xrightarrow{\texttt{Copy}} \sigma \in E} \qquad \texttt{[E-Load]}\ \frac{p \xrightarrow{\texttt{Load}} q \in E \quad \sigma \in pts(p)}{\forall \sigma' : \sigma \sqcap \sigma' \neq \emptyset \Rightarrow \sigma' \xrightarrow{\texttt{Copy}} q \in E}$$

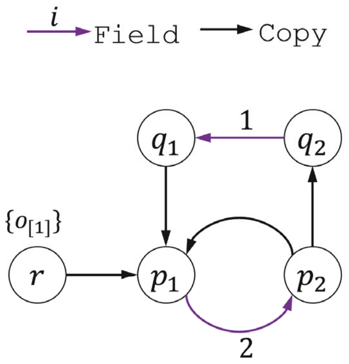
$$Strides(e) = \begin{cases} \{0\} & \text{if edge } e \text{ is not in any PWC} \\ \{\mathscr{W}_{\mathscr{C}} \mid \forall \mathscr{C} \subseteq E : e \in \mathscr{C}\} & \text{otherwise} \quad \text{(Definition 5.6)} \end{cases}$$

**Figure 5.5.** SFR-based rules for field-sensitive pointer analysis.

a sequence of fields starting from the $(i+j)$-th field following strides $S \cup S'$, where (1) $S' = \{0\}$ if $p \xrightarrow{\texttt{Field}_i} q$ does not reside in any PWC and (2) $S' = \{\mathscr{W}_{\mathscr{C}} \mid \forall \mathscr{C} \subseteq E : p \xrightarrow{\texttt{Field}_i} q \in \mathscr{C}\}$ otherwise. Here $S'$ denotes a set holding the weights of all the positive weight cycles with each $\mathscr{C}$ containing $p \xrightarrow{\texttt{Field}_i} q$ on the constraint graph (Definitions 5.5 and 5.6). If $p \xrightarrow{\texttt{Field}_i} q$ is involved in multiple PWCs, $\sigma$ is derived to collapse as many equivalent fields as possible by combining $S$ and $S'$.

The premise of [E-Field] ensures that $\sigma$ represents the derivation equivalent fields such that the targets added to the points-to sets of all these fields are always identical when solving each cycle $\mathscr{C}$. The conclusion of [E-Field] ensures early termination and avoids redundant derivations, since an SFR $\sigma$ can only be generated and added to $pts(p)$ if there no SFR $\sigma'$ already exists in $pts(p)$ such that the expressiveness of $\sigma'$ can cover $\sigma$, i.e., $\sigma \sqsubseteq \sigma'$ (Definition 5.4). Examples 5.2 and 5.3 give two scenarios in which a Field edge resides in single and multiple PWCs.

*Example* 5.2 ([e-field] for a single PWC). Let us revisit the example in Figure 5.4 to



**(a)** Constraint graph.

$$\begin{aligned} & & pts(r) &= \{\sigma_1\} \\ \sigma_1 &= \langle o, 1, \{0\}\rangle & pts(p_1) &= \{\sigma_1, \sigma_2, \sigma_3\} \\ \sigma_2 &= \langle o, 3, \{0,2,3\}\rangle & pts(p_2) &= \{\sigma_2\} \\ \sigma_3 &= \langle o, 4, \{0,2,3\}\rangle & pts(q_1) &= \{\sigma_3\} \\ \sigma_2 \sqcap \sigma_3 &\neq \emptyset & pts(q_2) &= \{\sigma_2\} \end{aligned}$$

**(b)** Solution in SFR.

$$\begin{aligned} pts(r) &= \{o_{[1]}\} \\ pts(p_1) &= \{o_{[1]}, o_{[3]}, o_{[4]}, o_{[5]}, o_{[6]}, o_{[7]}, o_{[8]}, \cdots\} \\ pts(p_2) &= \{o_{[3]}, o_{[5]}, o_{[6]}, o_{[7]}, o_{[8]}, \cdots\} \\ pts(q_1) &= \{o_{[4]}, o_{[6]}, o_{[7]}, o_{[8]}, o_{[9]}, \cdots\} \\ pts(q_2) &= \{o_{[3]}, o_{[5]}, o_{[6]}, o_{[7]}, o_{[8]}, \cdots\} \end{aligned}$$

**(c)** Solution in classical field modeling.

**Figure 5.6.** Solving $p_1 \xrightarrow{\texttt{Field}_2} p_2$, which resides in multiple cycles, with SFRs.

explain [E-Field]. The Field edge $r \xrightarrow{\texttt{Field}_1} p_1$ is not involved in any PWC, therefore, [E-Field] generates an SFR $\sigma = \langle o, 1, \{0\}\rangle$ with $S' = \{0\}$, representing only field $o_{[1]}$ and then adds $\sigma$ into $pts(p_1)$. Together with $p_2 \xrightarrow{\texttt{Copy}} p_1$, the second Field edge $p_1 \xrightarrow{\texttt{Field}_2} p_2 \in \mathscr{C}$ forms a positive weight cycle $\mathscr{C}$ with its weight $\mathscr{W}_{\mathscr{C}} = 2$. Given $pts(p_1) = \{\langle o, 1, \{0\}\rangle\}$, a new SFR $\sigma = \langle o, 1+2, \{0\} \cup \{2\}\rangle = \langle o, 3, \{0,2\}\rangle$ is derived and added into $pts(p_2)$ after processing $p_1 \xrightarrow{\texttt{Field}} p_2$. The SFR $\langle o, 3, \{0,2\}\rangle$ is then propagated back to $p_1$ via $p_2 \xrightarrow{\texttt{Copy}} p_1$. In the second iteration for solving $p_1 \xrightarrow{\texttt{Field}_2} p_2$, the newly derived SFR $\langle o, 5, \{0,2\}\rangle$ is discarded and not added into $pts(p_1)$ since the expressiveness of $\langle o, 5, \{0,2\}\rangle$ is covered by $\langle o, 3, \{0,2\}\rangle$, i.e., $\langle o, 5, \{0,2\}\rangle \sqsubseteq \langle o, 3, \{0,2\}\rangle$ (Definition 5.4).

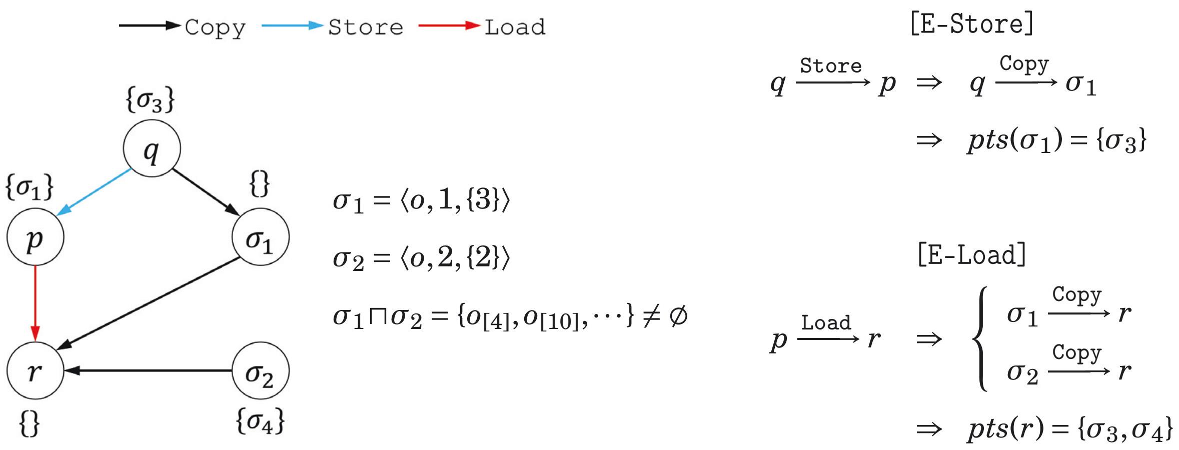
*Example* 5.3 ([E-Field] for multiple PWCs). Figure 5.6 compares our stride-based field representation (Figure 5.6(b)) with the classical modeling (Figure 5.6(c)) to show that [E-Field] requires significantly fewer field derivations to resolve $p_1 \xrightarrow{\texttt{Field}_2} p_2$ when it is involved in two PWCs, i.e., $\mathscr{C}_1$ formed by $p_1 \xrightarrow{\texttt{Field}_2} p_2$ and $p_2 \xrightarrow{\texttt{Copy}} p_1$, and $\mathscr{C}_2$ formed by $p_1 \xrightarrow{\texttt{Field}_2} p_2$, $p_2 \xrightarrow{\texttt{Copy}} q_2$, $q_2 \xrightarrow{\texttt{Field}_1} q_1$ and $q_1 \xrightarrow{\texttt{Copy}} p_1$. The weights of $\mathscr{C}_1$ and $\mathscr{C}_2$ are 2 and 3, respectively. Thus, $S' = \{2,3\}$. Initially, $r$ points to $\sigma_1 = \langle o, 1, \{0\}\rangle$, which is propagated to $p_1$ along $r \xrightarrow{\texttt{Copy}} p_1$.

We first consider resolving $\mathscr{C}_1$. A new SFR $\sigma_2 = \langle o, 1+2, \{0\} \cup \{2,3\}\rangle = \langle o, 3, \{0,2,3\}\rangle$ is derived and added to $pts(p_2)$ after processing $p_1 \xrightarrow{\texttt{Field}_2} p_2$. $\sigma_2$ is then propagated back and added to $pts(p_1)$ along $p_2 \xrightarrow{\texttt{Copy}} p_1$. The second iteration for processing $p_1 \xrightarrow{\texttt{Field}_2} p_2$ avoids adding $\langle o, 5, \{0,2,3\}\rangle$ into $pts(p_2)$ because $\langle o, 5, \{0,2,3\}\rangle$ is a subset of $\sigma_2$, resulting in early termination.

Similarly, when resolving $\mathscr{C}_2$ which contains two `Field` edges, our approach generates $\sigma_3 = \langle o, 3+1, \{0,2,3\}\rangle$ after processing $q_2 \xrightarrow{\texttt{Field}_1} q_1$ and then propagates $\sigma_3$ to $p_1$ via $q_1 \xrightarrow{\texttt{Copy}} p_1$. Given this new $\sigma_3$ in $pts(p_1)$, $\langle o, 4+2, \{0,2,3\}\rangle$ is derived when processing $p_1 \xrightarrow{\texttt{Field}_2} p_2$ again. However, $\langle o, 4+2, \{0,2,3\}\rangle$ is a subset of $\sigma_2$, hence is not added to $pts(p_2)$. Note that though $\sigma_2$ and $\sigma_3$ are overlapping due to the intersecting PWCs, $\sigma_2$ successfully captures the equivalent fields that are always pointed by $p_1, p_2, q_2$ and $\sigma_3$ captures the equivalent fields that are always pointed by $p_1, q_1$, avoiding redundant derivations. For each PWC, our approach generates only one SFR, requiring at most two iterations to converge the analysis. In contrast, the classical field modeling performs redundant derivations until it reaches the maximum number of fields of this object, as shown in Figure 5.6(c).

**Store and Load.** Unlike `[Store]` and `[Load]` in Figure 5.2, our handling of `Store` and `Load` is asymmetric for both efficiency and precision purposes. For each $\sigma \in pts(q)$, while processing $p \xrightarrow{\texttt{Store}} q$, `[E-Store]` is similar to `[Store]` by adding a `Copy` edge from $p$ to $\sigma$ to the graph, which propagates $pts(p)$ to $pts(\sigma)$ afterward. For each SFR $\sigma \in pts(p)$, while processing $p \xrightarrow{\texttt{Load}} q$, `[E-Load]` adds copy edges from $\sigma'$ to $q$ for all $\sigma'$ overlapping with $\sigma$ (Definition 5.4), which propagates $pts(\sigma')$ to $pts(q)$ afterward.

In `[E-Load]`, we consider $\sigma'$ on top of $\sigma$ because a field $o_{[i]}$ in the classical field modeling may belong to one or multiple SFRs. For example, in Figure 5.6, $o_{[6]}$ belongs to $\sigma_2$ and $\sigma_3$ when resolving a `Field` edge which is involved in multiple PWCs or in one


Copy
Store
Load
$\{\sigma_3\}$
$q$
$\{\sigma_1\}$
$\{\}$
$p$
$\sigma_1$
$r$
$\sigma_2$
$\{\}$
$\{\sigma_4\}$
$\sigma_1 = \langle o, 1, \{3\}\rangle$
$\sigma_2 = \langle o, 2, \{2\}\rangle$
$\sigma_1 \sqcap \sigma_2 = \{o_{[4]}, o_{[10]}, \cdots\} \neq \emptyset$
[E-Store]
$q \xrightarrow{\texttt{Store}} p \;\Rightarrow\; q \xrightarrow{\texttt{Copy}} \sigma_1$
$\Rightarrow\; pts(\sigma_1) = \{\sigma_3\}$
[E-Load]
$p \xrightarrow{\texttt{Load}} r \;\Rightarrow\; \begin{cases} \sigma_1 \xrightarrow{\texttt{Copy}} r \\ \sigma_2 \xrightarrow{\texttt{Copy}} r \end{cases}$
$\Rightarrow\; pts(r) = \{\sigma_3, \sigma_4\}$

**(a)** Constraint graph.

**(b)** Constraint solving.

**Figure 5.7.** Solving $q \xrightarrow{\texttt{Store}} p$ and $p \xrightarrow{\texttt{Load}} r$ for overlapping SFRs.

PWC containing multiple `Field` edges. We use $\mathcal{M}_{o_{[i]}}$ to denote a set of all SFRs containing $o_{[i]}$, i.e., any two SFRs in $\mathcal{M}_{o_{[i]}}$ share common fields including at least $o_{[i]}$. According to Definition 5.2, any change to the point-to sets of $\sigma \in \mathcal{M}_{o_{[i]}}$ also applies to those of $o_{[i]}$ during our constraint resolution. If $*q$ at a `Load` refers to an SFR $\sigma \in \mathcal{M}_{o_{[i]}}$, it also refers to $\sigma' \in \mathcal{M}_{o_{[i]}}$ that overlaps with $\sigma$ for each field $o_{[i]} \in \texttt{FieldExpand}(\sigma)$ (Definition 5.4). Therefore, `[E-Load]` maintains the correctness that $pts(o_{[i]})$ obtains the union of the points-to sets of all SFRs in $\mathcal{M}_{o_{[i]}}$. Since a points-to target in $pts(\sigma)$ must be in the points-to set of every field in $\texttt{FieldExpand}(\sigma)$ (i.e, for any $\sigma \in \mathcal{M}_{o_{[i]}}$, $pts(\sigma)$ is always a subset of $pts(o_{[i]})$), ensuring that no spurious points-to targets other than $pts(o_{[i]})$ will be propagated to $p$ at the `Load`. Thus, our handling of PWCs is precision preserving, i.e., the points-to set of a variable after field expansion resolved under SFRs is the same as that the classical field representation.

*Example* 5.4 (`[E-Load]` and `[E-Store]`). Figure 5.7 illustrates the resolving of $q \xrightarrow{\texttt{Store}} p$ and $p \xrightarrow{\texttt{Load}} r$ with the initial points-to sets $pts(p) = \{\sigma_1\}$, $pts(q) = \{\sigma_3\}$, $pts(\sigma_2) = \{\sigma_4\}$ and $pts(r) = pts(\sigma_1) = \emptyset$. $\sigma_1$ and $\sigma_2$ are both derived from object $o$ with overlapping fields, e.g., $o_{[4]}$ and $o_{[10]}$. When resolving $q \xrightarrow{\texttt{Store}} p$, `[E-Store]` adds a new `Copy` edge from $q$ to $\sigma_1$, propagating $\sigma_3 \in pts(q)$ to $pts(\sigma_1)$, but not $pts(\sigma_2)$ though $\sigma_1 \sqcap \sigma_2 \neq \emptyset$, as illustrated in Figure 5.7(b). This avoids, for example, introducing the spurious point-to target $\sigma_3$ to the points-to set of $o_{[2]}$ which only resides in $\sigma_2$ but not in $\sigma_1$. In contrast, [E-Load] resolves $p \xrightarrow{\texttt{Load}} r$ by adding two Copy edges $\sigma_1 \xrightarrow{\texttt{Copy}} r$ and $\sigma_2 \xrightarrow{\texttt{Copy}} r$, as also depicted in Figure 5.7(b). Since $\sigma_1 \sqcap \sigma_2 = \{o_{[4]}, o_{[10]}, \cdots\}$ and $\sigma_1 \in pts(p)$, if $^*p$ at Load $r = *p$ refers to an overlapping field e.g., $o_{[4]}$ shared by $\sigma_1$ and $\sigma_2$, the points-to set of $r$ is the union of $pts(\sigma_1)$ and $pts(\sigma_2)$, i.e., $pts(r) = \{\sigma_3, \sigma_4\}$, achieving the precise field-sensitive results.

### 5.2.3 DEA: a Derivation Equivalence Algorithm

Our precision-preserving handling of PWCs (i.e., the inference rules in Figure 5.5) can be integrated into existing constraint solving algorithms for field-insensitive Andersen's analysis, e.g., the state-of-the-art cycle elimination approaches [38, 51, 56, 95, 97]. This section presents an overall derivation equivalence algorithm DEA, which is constructed by instantiating our inference rules on top of *wave propagation* [97]. We choose wave propagation as the dynamic-programming constraint solving strategy as it is the most competitive one over the existing techniques [38, 51, 56, 95] for analyzing large size programs. In Algorithm 9, all the Addr edges are processed only once to initialize the worklist $W$ (line 3), followed by a *while* loop for the main phase of constraint solving, which has three phases.

**Cycle Elimination and Weights of PWCs.** In Line 5, we use Nuutilia et al.'s algorithm [92] to detect and merge strongly connected components (SCCs), which is an improvement over the original algorithm developed by Tarjan et al. [127]. Then, the weight $W_{\mathscr{C}}$ of each PWC $\mathscr{C}$ can be calculated from the detected SCCs (Line 6).

**Points-to Set Propagation via Direct Edges.** We propagate points-to information along each Copy edge based on [E-Copy] (Lines 9–11). New SFRs are derived and added to the points-to sets of the destination node of each Field edge based on [E-Field] (Lines 12–16). A variable $p$ is pushed into a new worklist $W_{ind}$ if there exists an incoming Store

**Algorithm 9:** An Algorithm

```
Function DEA($G = \langle V, E \rangle$)
    $W \leftarrow \emptyset$; $W_{ind} \leftarrow \emptyset$;
    for each $o \xrightarrow{\texttt{Addr}} p \in E$ do $pts(p) \leftarrow pts(p) \cup \{\langle o, 0, \emptyset \rangle\}$; $W$.push($p$) ;   /* [E-Addr] */
    while $W \neq \emptyset$ do
        Detect and collapse cycles comprised of Copy edges using the algorithm in [92];
        Calculate the weight of each PWC in $G$;
        while $W \neq \emptyset$ do
            $p \leftarrow W$.pop_front();
            for each $p \xrightarrow{\texttt{Copy}} q \in E$ do   /* [E-Copy] */
                $pts(q) \leftarrow pts(q) \cup pts(p)$;
                if $pts(q)$ is changed then $W$.push($q$) ;
            for each $p \xrightarrow{\texttt{Field}_i} q \in E$ do   /* [E-Field] */
                $S' \leftarrow Strides(p \xrightarrow{\texttt{Field}_i} q)$
                for each $\langle o, j, S \rangle \in pts(p)$ do
                    $\sigma \leftarrow \langle o, i + j, S \cup S' \rangle$
                    if $\nexists\, \sigma' \in pts(q) : \sigma \sqsubseteq \sigma'$ then $pts(q) \leftarrow pts(q) \cup \{\sigma\}$; $W$.push($q$) ;
            if $\exists\, q \xrightarrow{\texttt{Store}} p \in E$ or $\exists\, p \xrightarrow{\texttt{Load}} q \in E$ then push $p$ into $W_{ind}$ ;
            while $W_{ind} \neq \emptyset$ do
                $p \leftarrow W_{ind}$.pop_front();
                for each $q \xrightarrow{\texttt{Store}} p \in E$ do   /* [E-Store] */
                    for each $\sigma \in pts(p)$ do
                        if $q \xrightarrow{\texttt{Copy}} \sigma \notin E$ then $E \leftarrow E \cup \{q \xrightarrow{\texttt{Copy}} \sigma\}$; $W$.push($q$) ;
                for each $p \xrightarrow{\texttt{Load}} q \in E$ do   /* [E-Load] */
                    for each $\sigma' \in \{\sigma' \sqcap \sigma \neq \emptyset \mid \sigma \in pts(p)\}$ do
                        if $\sigma' \xrightarrow{\texttt{Copy}} q \notin E$ then $E \leftarrow E \cup \{\sigma' \xrightarrow{\texttt{Copy}} q\}$; $W$.push($\sigma'$) ;
            for each $p \xrightarrow{\texttt{Copy}} q$ added by UpdateCallgraph do
                $W$.push($p$)
```

edge to $p$ or an outgoing Load edge from $p$ for later handling of Load and Store edges (lines 17).

**Processing Indirect Edges.** Lines 20–25 handle Store and Load edges via [E-Store] and [E-Load]. New Copy edges are added to $G$, and the source node of each newly added Copy edge is added to worklist $W$ for points-to propagation in the next iteration. Lines 26–27 update the callgraph according to the literature [96, 121], which creates new Copy edges (e.g., $p \xrightarrow{\texttt{Copy}} q$) for parameter/retval passings when a new callee function is discovered at a callsite using the points-to results of function pointers obtained from this points-to resolution round. The source node $v$ of the Copy edge is added to $W$ to be processed in the next iteration until a fixed point is reached, i.e., no changes are made to the points-to set of any node. Other field-sensitive analyses (e.g., [96]) can also be implemented under the same constraint solving algorithm by simply replacing the lines for handling the five types of constraints with the inference rules in Figure 5.2.

## 5.3 Implementation of Field-Sensitive Pointer Analysis for C/C++

We perform our pointer analysis on top of the LLVM-IR of a program, as in [15, 77, 79, 123, 142]. The set of all variables $\mathscr{V}$ is separated into two subsets, $\mathscr{A}$ which contains all possible abstract objects and their fields, i.e., *address-taken variables*, and $\mathscr{P}$ which contains all *top-level variables*, including stack virtual registers (symbols starting with "%") and global variables (symbols starting with "@") which are explicit, i.e., directly accessed. Address-taken variables in $\mathscr{A}$ are implicit, i.e., accessed indirectly at LLVM's Load or Store instructions via top-level variables.

After the SSA conversion, a program is represented by five types of instructions: Addr, Copy, Field, Store and Load (Table 5.1). Top-level variables are put directly in

```
            p = &a;
p = &a;     t1 = &b;
a = &b;     *p = t1;

q = &c;     q = &c;
*p = *q;    t2 = *q;
            *p = t2;
-----------------------
 C code     LLVM IR
```

**Figure 5.8.** C code fragment and its LLVM IR.

SSA form, while address-taken variables are only accessed indirectly via Load or Store. For an Addr $p = \&o$, known as an *allocation site*, $o$ is a stack or global variable with its address taken or a dynamically created abstract heap object (e.g., via malloc()). Parameter passings and returns are treated as Copy constraints. A field object denoted by $o_{[i]}$ is derived from $o$ when analyzing Field q = &p → $f_i$ (LLVM's getelementptr instruction), where $\mathrm{f}_i$ denotes the $i$-th field of $o$ and $i$ is a constant integer.

Figure 5.8 shows a code fragment and its corresponding partial SSA form, where $\mathtt{p},\mathtt{q},\mathtt{t1},\mathtt{t2} \in \mathscr{P}$ and $\mathtt{a},\mathtt{b},\mathtt{c} \in \mathscr{A}$. Note that a is indirectly accessed at a store $*\mathtt{p} = \mathtt{t1}$ by introducing a top-level pointer t1 in the partial SSA form. Complex statements such as $*\mathtt{p} = *\mathtt{q}$ are decomposed into basic instructions by introducing a top-level pointer t2.

Our approach is implemented on top of LLVM-12.0.0 and its sub-project SVF [123, 124, 126]. A state-of-the-art constraint resolution strategy, *wave propagation* [97] is incorporated for cycle detection and computing dynamic transitive closures on top of the same constraint graph for both DEA and the baseline (i.e., constraint solving using rules in Figure 5.2). Indirect calls via function pointers are resolved on-the-fly during points-to resolution. A C++ virtual call p → foo() is translated into four low-level LLVM instructions for our pointer analysis. (1) a Load vtptr = *p, obtaining virtual table pointer vtptr by dereferencing pointer p to the object; (2) a Field vfn = &vtptr→idx, obtaining the entry in the vtable at a designated offset idx for the target function; (3)

a Load `fp = *vfn`, obtaining the address of the function, and (4) a function call `fp(p)`. Following [53, 94, 124], an allowlist is maintained to summarize all the side-effects of external calls (e.g., `memcpy`, `xmalloc` and `_Znwm` for C++ `new`) [120].

## 5.4 Experimental Evaluation

We compare DEA with the state-of-the-art field-sensitive pointer analysis for C/C++ using 11 open-source programs. The experimental result shows that DEA significantly accelerates field-sensitive pointer analysis while preserving precision.

### 5.4.1 Experimental Setup

To evaluate the effectiveness of our implementation, we chose 11 large-scale open-source C/C++ projects downloaded from Github, including `git-checkout` (a sub project of Git for version control), `json-conversions` and `json-ubjson` (two main Json libraries for modern C++ environment), `llvm-as-new` and `llvm-dwp` (tools in LLVM-12.0.0 compiler), `opencv_perf_core` and `opencv_test_dnn` (two main libraries in OpenCV), `python` and `redis-server` (a distributed database server). The source code of each program is compiled into bit code files Clang-12.0.0 [83] and then linked together using `WLLVM` [129] to produce whole program bc files.

Table 5.2 collects the basic characteristics about the 11 programs before the main pointer analysis phase. The statistics include the LLVM IR's lines of code (LOC) of a program, the number of pointers (#Pointers), the number of fields of the largest struct in the program, also known as the maximum number of fields using the upper bound for deriving fields of a heap object, and the number of each of the five types of constraint edges in the initial constraint graph. The reason that #Field is not much smaller than #Copy is twofold (1) `Field` refers to LLVM's `getelementptr` instruction, which is used to get the addresses of subelements of aggregates, including not only structs but also arrays

**Table 5.2.** Basic characteristics of the benchmarks (IR's lines of code, number of pointers, number of five types of instructions on the initial constraint graph, and maximum number of fields of the largest struct in each program).

| Benchmark | LOC | #Pointers | MaxFields | #Field | #Copy | #Store | #Load | #AddrOf |
|---|---|---|---|---|---|---|---|---|
| git-checkout | 1253K | 624K | 302 | 93201 | 88406 | 41620 | 60723 | 33380 |
| json-conversions | 355K | 264K | 64 | 27685 | 36557 | 37960 | 36872 | 43448 |
| json-ubjson | 330K | 233K | 64 | 24064 | 35813 | 34577 | 26288 | 34165 |
| llvm-as-new | 729K | 597K | 121 | 307167 | 77944 | 287634 | 41960 | 17435 |
| llvm-dwp | 1796K | 897K | 632 | 100877 | 101849 | 116205 | 142943 | 121541 |
| llvm-objdump | 728K | 353K | 121 | 61117 | 57743 | 56493 | 40314 | 16767 |
| opencv_perf_core | 1014K | 715K | 64 | 122744 | 192419 | 59599 | 79466 | 24450 |
| opencv_test_dnn | 889K | 635K | 64 | 105550 | 174080 | 52304 | 70332 | 22786 |
| python | 539K | 420K | 171 | 84779 | 74524 | 49215 | 56434 | 18340 |
| redis-server | 706K | 374K | 332 | 52178 | 60111 | 24542 | 39205 | 13175 |
| Xalan | 2192K | 807K | 133 | 110184 | 181804 | 35940 | 68812 | 53926 |

and nested aggregates. (2) In low-level LLVM IR, a `Copy` only refers to an assignment between two virtual registers, such as casting or parameter passing. An assignment "p = q" in high-level C/C++ is not translated into a `Copy`, but a `Store`/`Load` manipulated indirectly through registers on LLVM's partial SSA form.

All our experiments were conducted on a platform consisting of a 3.50GHz Intel Xeon Quad Core CPU with 128 GB memory, running Ubuntu Linux 18.04. The baseline is the constraint solving using rules in Figure 5.2 and running within the same dynamic programming strategy, i.e., wave propagation [97], as DEA.

### 5.4.2 Results and Analysis

Table 5.3 compares DEA with baseline for each of the 11 programs evaluated in terms of the following three analysis results after constraint resolution, the total number of address-taken variables (*#AddrTakenVar*), the total number of fields derived when

**Table 5.3.** Comparing the results produced by DEA with those by baseline, including the total number of address-taken variables, number of fields and the number of fields derived when resolving PWCs, and the number of `Copy` edges connected to/from the field object nodes derived when resolving PWCs

| **Benchmark** | **#AddrTakenVar** | | **#Field** | | **#FieldByPWC** | |
|---|---|---|---|---|---|---|
| | **Baseline** | **DEA** | **Baseline** | **DEA** | **Baseline** | **DEA** |
| git-checkout | 135576 | 73967 | 121574 | 59965 | 68045 | 6436 |
| json-conversions | 62397 | 40993 | 40943 | 19539 | 22330 | 926 |
| json-ubjson | 60721 | 34987 | 49211 | 23477 | 27000 | 1266 |
| llvm-as-new | 24427 | 16124 | 19304 | 11001 | 9770 | 1467 |
| llvm-dwp | 145247 | 91945 | 109650 | 56348 | 62383 | 9081 |
| llvm-objdump | 16130 | 12007 | 11235 | 7112 | 5119 | 996 |
| opencv_perf_core | 60625 | 44061 | 40196 | 23632 | 18894 | 2330 |
| opencv_test_dnn | 53064 | 37957 | 35177 | 20070 | 17366 | 2259 |
| python | 30848 | 23713 | 21530 | 14395 | 9531 | 2396 |
| redis-server | 13109 | 9581 | 8165 | 4637 | 4234 | 706 |
| Xalan | 90314 | 62859 | 61466 | 34011 | 32226 | 4771 |
| *Max reduction* | 45.4% | | 52.3% | | 95.9% | |
| *Average reduction* | 32.4% | | 44.4% | | 86.6% | |

resolving all `Field` edges (*#Field*), and the number of fields derived only when resolving `Field` edges involving PWCs (*#FieldByPWC*). Both DEA and baseline use `LLVM Sparse Bitvectors` as the points-to set implementation. The peak memory usage by DEA is 7.33G observed in git-checkout. DEA produces identical points-to results to those by baseline, confirming that DEA's precision is preserved.

From the results produced by baseline, we can see that the number of fields (Column 4 in Table 5.3) occupies a large proportion of the total address-taken variables (Column 2) in modern large-scale C/C++ programs. On average, 72.5% of the address-taken variables are field objects. In programs `git-checkout` (written in C) and `json-ubjson` (written in C++) with heavy use of structs and classes, the percentages for both are higher than 80%.

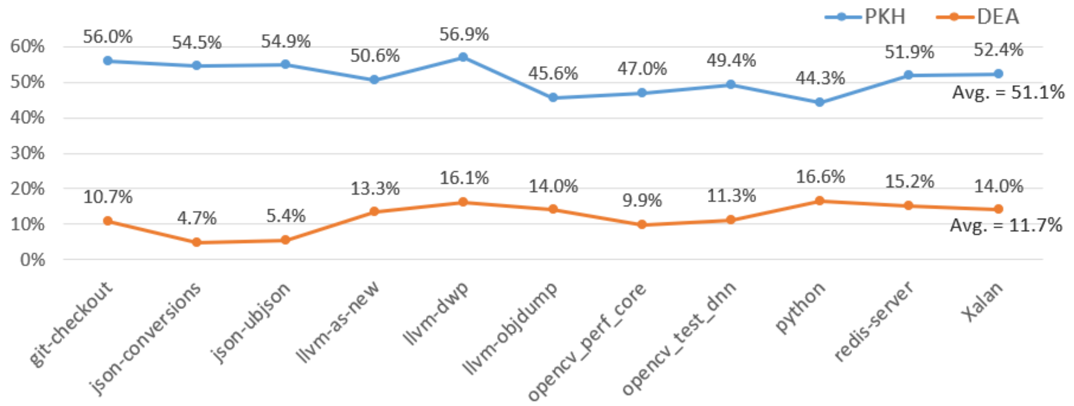


**Figure 5.9.** Percentages of fields derived when solving PWCs out of the total number of fields, i.e., $\frac{\#FieldByPWC}{\#Field} * 100$

In 8 of the 11 programs, over 50% of the fields are derived from PWCs.

Columns 4-5 of Table 5.3 compare the total number of field objects produced by baseline and DEA respectively. Columns 6-7 give more information about the number of fields derived only when resolving PWCs by baseline and DEA, we can see that these fields are significantly reduced by DEA with an average reduction rate of 86.6%, demonstrating that DEA successfully captured the derivation equivalence to collapse a majority of fields into SFRs when resolving PWCs.

Figure 5.9 further compares DEA with baseline in terms of percentages of fields derived from resolving PWCs out of the total number of fields for the 11 programs. The average percentage of 51.1% in baseline (blue line) is reduced to only 11.7% (orange line) in DEA with a reduction of 39.4%.

In `git-checkout`, `json-conversions` and `json-ubjson`, DEA achieves over 90% reduction in solving PWCs because these programs have relatively large numbers of address-taken variables (Table 5.3) and relatively more nodes involving PWCs (Table 5.4). On average, over 85% of redundant field derivations involving PWCs are avoided with the maximum reduction rate of 95.9% in `json-conversions`, confirming the effectiveness of our field collapsing in handling PWCs.

Table 5.4 gives the constraint graph information after points-to resolution. Column 2

**Table 5.4.** Constraint graph information (*#NodeInPWC* denotes the number of nodes involving PWCs by baseline; *#SFR* denotes the number of stride-based field representatives, generated by DEA; *#CopyByPWC*, denotes the number of Copy edges flowing into and going out of fields derived when solving PWCs; *#CopyProcessed* denotes the number of processing Copy edges.)

| **Benchmark** | **#NodeInPWC** | **#SFR** | **#CopyByPWC** | | **#CopyProcessed** | |
|---|---|---|---|---|---|---|
| | **Baseline** | **DEA** | **Baseline** | **DEA** | **Baseline** | **DEA** |
| git-checkout | 2840 | 2172 | 12372 | 2046 | 3868834 | 1128617 |
| json-conversions | 3631 | 1641 | 13490 | 2622 | 2253266 | 319960 |
| json-ubjson | 4271 | 1753 | 4311 | 1037 | 5621768 | 575884 |
| llvm-as-new | 1752 | 2085 | 9739 | 2789 | 2513940 | 688238 |
| llvm-dwp | 7263 | 1463 | 15062 | 2128 | 2802988 | 779424 |
| llvm-objdump | 1581 | 1761 | 7105 | 2013 | 2177990 | 647582 |
| opencv_perf_core | 1373 | 2030 | 4948 | 1973 | 4800563 | 655095 |
| opencv_test_dnn | 1007 | 777 | 4008 | 1577 | 5095795 | 460127 |
| python | 3817 | 1942 | 8530 | 3854 | 3495769 | 971376 |
| redis-server | 2783 | 1405 | 3380 | 1408 | 1288753 | 390783 |
| Xalan | 4874 | 2909 | 21935 | 7671 | 5143418 | 1554627 |
| *Max reduction* | | | 85.9% | | 91.0% | |
| *Avg. reduction* | | | 70.3% | | 77.3% | |

lists the number of nodes involving PWCs by baseline. For each SFR *Rep*() generated by DEA, Column 3 gives the numbers of SFRs generated by DEA. The average numbers of overlapping SFRs for the 11 programs evaluated are all below 1, which means that the majority of the SFRs either represent a single object/field or represent a sequence of fields that do not overlap with one another.

Columns 4-5 give the numbers of Copy edges flowing into and going out of field nodes derived when resolving PWCs by baseline and DEA respectively. DEA on average reduces the Copy edges in Column 4 by 70.3% with a maximum reduction rate of 85.9% Columns 6-7 give the number of processing times of Copy edges during points-to propagation by the two approaches. Since the number of Copy edges is significantly reduced by DEA,

**Table 5.5.** Total analysis times and the times of the three analysis stages, including *CycleDec* cycle detection (Lines 5–6 of Algorithm 9), *PtsProp*, propagating point-to information via `Copy` and `Field` edges (Lines 9–17), *ProcessLdSt*, adding new `Copy` edges when processing `Load`/`Store` and update callgraph (Lines 20–27).

| **Benchmark** | **CycleDec** | | **PtsProp** | | **ProcessLdSt** | | **TotalTime** | | **speedup** |
|---|---|---|---|---|---|---|---|---|---|
| | **Baseline** | **DEA** | **Baseline** | **DEA** | **Baseline** | **DEA** | **Baseline** | **DEA** | |
| git-checkout | 3117.8 | 4600.0 | 138233.5 | 26668.1 | 3870.2 | 1472.5 | 145221.6 | 32740.6 | 4.4 |
| json-conversions | 4436.2 | 561.6 | 12248.2 | 939.2 | 17.6 | 11.5 | 16702.0 | 1512.3 | 11.0 |
| json-ubjson | 25.1 | 6.0 | 18635.2 | 1817.3 | 52.4 | 23.2 | 18712.7 | 1846.6 | 10.1 |
| llvm-as-new | 22.6 | 11.9 | 10920.4 | 1728.9 | 541.9 | 221.2 | 11484.9 | 1962.0 | 5.9 |
| llvm-dwp | 3134.1 | 1457.7 | 120654.4 | 22177.2 | 1671.2 | 747.5 | 125459.8 | 24382.4 | 5.1 |
| llvm-objdump | 22.2 | 22.2 | 10617.3 | 2158.4 | 254.8 | 109.7 | 10894.4 | 2290.2 | 4.8 |
| opencv_perf_core | 338.5 | 299.3 | 30049.9 | 3018.5 | 2125.5 | 991.7 | 32513.9 | 4309.5 | 7.5 |
| opencv_test_dnn | 67.0 | 64.2 | 3145.5 | 248.8 | 366.1 | 122.2 | 3578.6 | 435.2 | 8.2 |
| python | 51.6 | 18.8 | 167556.9 | 22674.4 | 939.9 | 474.8 | 168548.3 | 23168.0 | 7.3 |
| redis-server | 525.1 | 428.6 | 11088.3 | 1315.2 | 99.8 | 49.8 | 11713.2 | 1793.5 | 6.5 |
| Xalan | 412.3 | 118.1 | 146617.8 | 21729.4 | 352.5 | 218.1 | 147382.7 | 22065.6 | 6.7 |
| *Average speedup* | | | | | | | | | 7.1 |

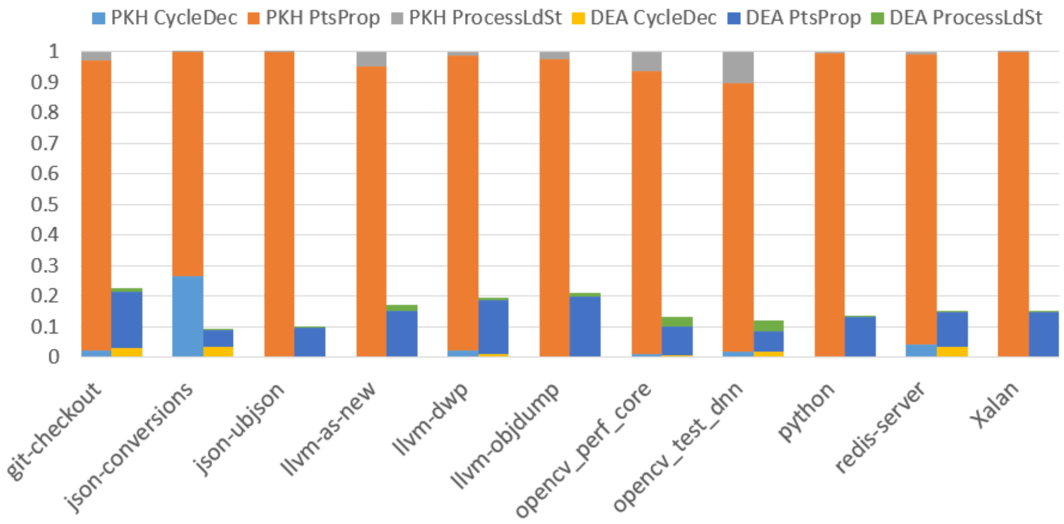


**Figure 5.10.** Comparing the time distribution of the three analysis phases of DEA with that of baseline (normalized with baseline as the base).

the processing times of Copy edges are reduced accordingly with an average/maximum reduction rate of 77.3%/91.0%.

Table 5.5 compares DEA with baseline in terms of the overall analysis times and the times collected for each of the three analysis phases. The total pointer analysis time consists of three major parts, as also discussed in Algorithm 9, and comprises (1) cycle detection, (2) propagating point-to sets via Copy and Field edges, and (3) processing Store and Load by adding new Copy edges into the constraint graph. Overall, DEA has a best speed up of 11.0× (observed in json-conversions) with an average speed up of 7.1× among the 11 programs.

Figure 5.10 gives the analysis time distributions of the three analysis phases in Table 5.5 for both baseline and DEA, where the phases are highlighted in different colors. The time cost of *PtsProp* (Columns 4-5) occupies a large percentage in resolution time by baseline. This is because *PtsProp* in field-sensitive pointer analysis needs to perform heavy set union operations for handling both Copy and Field edges. Worse, PWCs which need to be fully resolved by baseline incur a large number of redundant field derivations and unnecessary Copy edges until a pre-defined maximum number is reached, resulting in high analysis overhead in the *PtsProp* phase. In contrast, as depicted in Figure 5.10, the analysis overhead introduced by *PtsProp* is greatly reduced by DEA, though it occupies a noticeable portion of the total analysis time, showing that DEA effectively cuts down the overhead introduced by PWCs (i.e., redundant points-to propagation, and unnecessary `Copy` edges connecting to/from derivation equivalent fields) to help constraint resolution converge more quickly.

### 5.4.3 Summary

Our derivation algorithm DEA significantly boost the performance of solving positive weight cycles. By capturing derivation equivalence, DEA avoids infinite derivations and greatly reduces overhead during constraint solving, making the analysis converge more quickly. By applying it to field-sensitive Andersen's pointer analysis for C/C++, DEA on average achieves a speed up of 7.1× over state-of-the-art field-sensitive analysis [96] equipped with a popular cycle elimination technique wave propagation [97] for analyzing 11 open-source large-scale C/C++ programs.

CHAPTER 6

# CONCLUSION AND FUTURE WORKS

This dissertation focuses on scaling dynamic transitive closure based program analysis. Our works improve three fundamental static analysis frameworks, i.e., CFL-reachability, recursive state machine and set constraint analysis. We develop three techniques to handle three factors causing inefficiency: (1) a partially ordered CFL-reachability algorithm POCR for eliminating transitive redundancy in the dynamic solving procedure; (2) an RSM-guided graph folding GF for reducing redundant nodes and edges in the input graph; and (3) a derivation equivalence algorithm DEA for precisely and efficiently solving positive weight cycles whilst avoiding infinite derivations. Experimental results on benchmarks including SPEC 2017 and real-world open-source programs show the effectiveness of our techniques. In particular, the three techniques scale dynamic transitive closure based program analysis by reducing on-the-fly redundant derivations (POCR and DEA) or redundant input information (GF). Moreover, the three static analysis frameworks are interchangeable, which provides the feasibility of combining the three techniques to deal with particular real-world problems in the future.

Upon this dissertation, our future works will still center around precise and efficient program analysis. First, in view that CFL-reachability and set constraint are interchangeable and the implementations our optimization techniques are only for CFL-reachability

(Chapters 3 and 4) or set constraint (Chapter 5), it is worth design a transformer that can automatically transform our optimization techniques and make them suitable to both frameworks. Second, elegant analysis techniques/tools are desirable. This includes the optimization and extension of existing frameworks, algorithms and data structures and the development of new frameworks. Third, adapting the proposed techniques to complex real-world end-to-end applications, such as compiler optimization, code summarization, bug detection and model checking, is also pursuable.

APPENDIX A

# APPENDIX

## A.1 Proof of the Soundness of Algorithm 2

We only prove the case for $\mathtt{CheckStree}$, as the other case is similar. Specifically, let $succ(A, v_j)$ denote the final set of successors of $v_j$ in the gound trouth (i.e., $v_k \in succ(A, v_j)$) if and only if there exists a path between $v_j$ and $v_k$ such that the path string is derivable from $A$. We show that $\mathtt{CheckStree}(X, A, v_i, v_j, v_j)$ ensures that for all $v_k \in succ(A, v_j)$, $v_i \xrightarrow{X} v_k$ will have been added to $G$ at the end of the algorithm execution.

Throughout this proof, we assume that the predecessor trees and successor trees are automatically maintained to satisfy the properties discussed in Section 3.2.1. Moreover, since the updates of the trees involve adding new (transitive) edges, we assume that those edges will be properly processed by the original algorithm (Algorithm 1).

Since $v_i \xrightarrow{X} v_j$ is popped out from the worklist, this edge must already be in the graph, since for each edge, we always push it into the worklist and add it into the graph simultaneously. Let's consider two versions of the successor tree: $stree(A, v_j)_0$ denoting the successor tree at the time of adding $v_i \xrightarrow{X} v_j$ into the graph, and $stree(A, v_j)$ denoting the successor tree at the end of the algorithm execution.

If $v_i \in stree(A, v_j) \setminus stree(A, v_j)_0$, then $v_k$ is added to the successor tree after the insertion of the edge $v_i \xrightarrow{X} v_j$ to the graph. Since we assume that when the successor trees are updated, the corresponding edges will be processed by Algorithm 1, this ensures that $v_i \xrightarrow{X} v_k$ will be added to the graph since the edge $v_i \xrightarrow{X} v_j$ is already in the graph.

If $v_k \in \mathit{stree}(A,v_j)_0$, then consider the successor tree $\mathit{stree}(A,v_j)_1$ at the time of processing the edge $v_i \xrightarrow{X} v_j$. We have $\mathit{stree}(A,v_j)_0 \subset \mathit{stree}(A,v_j)_1$ since the successor trees are non-shrinking and processing the edge happens after adding the edge to the graph. We prove that $\mathtt{CheckStree}(X,A,v_i,v_j,v_j)$ ensures for all $v_k \in \mathit{stree}(A,v_j)_1$, $v_i \xrightarrow{X} v_k$ will eventually be added to the graph, and this naturally ensures that for all $v_k \in \mathit{stree}(A,v_j)_0$, $v_i \xrightarrow{X} v_k$ will eventually be added to the graph.

For proving this, we consider two cases. If $\mathtt{CheckStree}(X,A,v_i,v_j,v_j)$ traverses all vertices in $\mathit{stree}(A,v_j)_1$, then the proof is completed. Otherwise, $\mathtt{CheckStree}(X,A,v_i,v_j,v_j)$ stops the traversal at vertices $u_1,\cdots,u_t \in \mathit{stree}(A,v_j)_1$. From this we know that the edges $v_i \xrightarrow{X} u_1,\cdots,v_i \xrightarrow{X} u_t$ has already been added to the graph, and they may or may not have been popped out from the worklist and processed. Then we can do similar reasoning to the processing of those edges. This recursive reasoning must eventually stop because each edge is processed only once. So the proof is completed.

## A.2 Proof of Property 4.2

The dependency of global transition on local ones (Property 3.1) indicates that $\forall w_B = \langle b_i,\cdots,b_j\rangle \in B^*$, $\forall s_1 \in B \times N$ and $\forall L \subseteq \Sigma$,

$$\begin{array}{lrcl}
\text{(i)} & s_1 \xrightarrow{\ell} s_2 \in \Delta & \Leftrightarrow & \langle w_B,s_1\rangle \xrightarrow{\ell} \langle w_B,s_2\rangle \in \Delta, \\
\text{(ii)} & s_1 \overset{L}{\preceq} s_2 & \Leftrightarrow & \langle w_B,s_1\rangle \overset{L}{\preceq} \langle w_B,s_2\rangle, \\
\text{(iii)} & s_1 \overset{L}{\simeq} s_2 & \Leftrightarrow & \langle w_B,s_1\rangle \overset{L}{\simeq} \langle w_B,s_2\rangle.
\end{array}$$

Analogously, $\forall s_1 \in B^2 \times N$,

$$s_1 \xrightarrow{\ell_1} s_2 \xrightarrow{\ell_2} s_3 \in R \quad \Leftrightarrow \quad \forall \langle w_B,s_1\rangle \in S,\ \langle w_B,s_1\rangle \xrightarrow{\ell_1} \langle w_B,s_2\rangle \xrightarrow{\ell_2} \langle w_B,s_3\rangle \in R.$$

Thus, given $\ell_1, \ell_2 \in \Sigma$, $L \subseteq \Sigma$, and $N_1 \subseteq N$,

$$\forall s_1 \in B^2 \times N, \quad \exists s_1 \xrightarrow{\ell_1} s_2 \in R \ s.t. \ s_1 \overset{L}{\simeq} s_2$$

indicates

(iv) $$\forall s_1 \in B^{\alpha'} \times N, \ s.t. \ \alpha' > 2, \quad \exists s_1 \xrightarrow{\ell_1} s_2 \in R \ s.t. \ s_1 \overset{L}{\simeq} s_2,$$

and

$$\forall s_1 \xrightarrow{\ell_1} s_2 \xrightarrow{\ell_2} s_3 \in R \ s.t. \ s_1 \in B^2 \times N, \quad s_3 \overset{L}{\preceq} s_1$$

indicates

(v) $$\forall s_1 \xrightarrow{\ell_1} s_2 \xrightarrow{\ell_2} s_3 \in R \ s.t. \ s_1 \in B^{\alpha'} \times N \wedge \alpha' > 2, \quad s_3 \overset{L}{\preceq} s_1.$$

Comparing (iv) and (v) with the four rules in Figure 5, we can conclude that if the four rules in Figure 5 hold when $\alpha = 2$, they also hold whenever $\alpha > 2$. Thus, Principle ② has Property 4.1.

## A.3 Proof of Property 4.3

Property 4.1 indicates that the four rules in Figure A.15 hold if the rules in Figure 5 hold for all $\alpha \in \{0,1,2\}$. Then the problem is reduced to proving the following lemma:

**Lemma A.1.** *With $\boldsymbol{P}_{xy}$ denoting an xy-FEQ class that each path passes through x and y via at most one xy-subpath and $p_{G'}$ denoting the xy-folded path of $\boldsymbol{P}_{xy}$, the four rules in Figure A.15 ensures that (i) $p_{G'}$ is a reachable path iff $\boldsymbol{P}_{xy}$ contains a reachable path, and (ii) when the paths of $\boldsymbol{P}_{xy}$ do not end with x or y, $p_{G'}$ has a corresponding sub-transition chain from $s_0$ to $s_k$ iff $\boldsymbol{P}_{xy}$ also contains a path corresponding to a sub-transition chain from $s_0$ to $s_k$.*

$$\textit{Rule [x-x]'}: \quad \forall s_x \xrightarrow{\ell_{xy}} s_y \xrightarrow{\ell_{yx}} s'_x \in R \;\; s.t.\; s_x \in B^* \times Nr(L\_x) \wedge \ell_{xy} \in L_{xy} \wedge \ell_{yx} \in L_{yx}\;, \quad s'_x \overset{L_{x\_}}{\preceq} s_x\;.$$

$$\textit{Rule [y-y]'}: \quad \forall s_y \xrightarrow{\ell_{yx}} s_x \xrightarrow{\ell_{xy}} s'_y \in R \;\; s.t.\; s_y \in B^* \times Nr(L\_y) \wedge \ell_{xy} \in L_{xy} \wedge \ell_{yx} \in L_{yx}\;, \quad s'_y \overset{L_{y\_}}{\preceq} s_y\;.$$

$$\textit{Rule [x-y]'}: \quad \forall s_x \in B^* \times Nr(L\_x) \wedge \forall \ell_{xy} \in L_{xy}\;, \quad \exists s_x \xrightarrow{\ell_{xy}} s_y \in R \quad s.t.\;\; s_x \overset{L_{y\not{x}}}{\simeq} s_y\;.$$

$$\textit{Rule [y-x]'}: \quad \forall s_y \in B^* \times Nr(L\_y) \wedge \forall \ell_{yx} \in L_{yx}\;, \quad \exists s_y \xrightarrow{\ell_{yx}} s_x \in R \quad s.t.\;\; s_y \overset{L_{x\not{y}}}{\simeq} s_x\;.$$

**Figure A.15.** The above four rules holds if the rules in Figure 5 hold for all $\alpha \in \{0,1,2\}$.

Notably, according to Equation 2 (in Section 4.1), $Q_x \subseteq B^* \times Nr(L\_x)$ and $Q_y \subseteq B^* \times Nr(L\_y)$.

Figure 7 displays the four basic types of $xy$-FEQ classes where each path contains at most one $xy$-subpath. It is obvious that the four basic types are mutually exclusive and cover all paths containing at most one $xy$-subpath. The following proves Lemma A.1 by cases.

**Type 1 (Figure 7(a))** Each $xy$-FEQ classes of this type has an *invariant element*, which does not contain any edge joining $x$ and $y$. Notably, the invariant element and its corresponding sub-transition chains are never changed by folding $(x, y)$.

**Type 1.1** First, we consider the subtype where the $\boldsymbol{P}_{xy}$ is comprised of paths starting and ending both with $x$, which is folded into a single node $p_{G'} = x$. Each corresponding sub-transition chain of $p_{G'}$ can be denoted by $p'_R = s_0$ where $s_0 \in B^* \times Nr(L\_x)$. (1) In this subtype, the invariant element is the path consisting of a single node $x$. (2) Then we consider the path $x \xrightarrow{\ell_{xy_1}} y \xrightarrow{\ell_{yx_1}} x \xrightarrow{\ell_{xy_2}} \cdots \xrightarrow{\ell_{yx_k}} x$, which is comprised of at least two edges joining $x$ and $y$. For the case $x \xrightarrow{\ell_{xy_1}} y \xrightarrow{\ell_{yx_1}} x$, which corresponds to $s_0 \xrightarrow{\ell_{xy_1}} s_{y_1} \xrightarrow{\ell_{yx_1}} s_{x_1}$ (if exists) where $s_0 \in B^* \times Nr(L\_x)$, rule [x-x]' ensures that $s_{x_1} \overset{L_{x\_}}{\preceq} s_0$. Similarly, for the case $x \xrightarrow{\ell_{xy_1}} y \xrightarrow{\ell_{yx_1}} x \xrightarrow{\ell_{xy_2}} y \xrightarrow{\ell_{yx_2}} x$, which corresponds to $s_0 \xrightarrow{\ell_{xy_1}} s_{y_1} \xrightarrow{\ell_{yx_1}} s_{x_1} \xrightarrow{\ell_{xy_2}} s_{y_2} \xrightarrow{\ell_{yx_2}} s_{x_2}$ (if exists), rule [x-x]' ensures that $s_{x_1} \overset{L_{x\_}}{\preceq} s_0$ and $s_{x_2} \overset{L_{x\_}}{\preceq} s_{x_1}$. According to Definition 4.5, $\overset{L}{\preceq}$ is a transitive relation, i.e., $s_{x_2} \overset{L_{x\_}}{\preceq} s_{x_1}$ and $s_{x_1} \overset{L_{x\_}}{\preceq} s_0$ indicates that $s_{x_2} \overset{L_{x\_}}{\preceq} s_0$. Analogously, for the case $x \xrightarrow{\ell_{xy_1}} y \xrightarrow{\ell_{yx_1}} x \xrightarrow{\ell_{xy_2}} \cdots \xrightarrow{\ell_{yx_k}} x$ corresponding to $s_0 \xrightarrow{\ell_{xy_1}} s_{y_1} \xrightarrow{\ell_{yx_1}} s_{x_1} \xrightarrow{\ell_{xy_2}} \cdots \xrightarrow{\ell_{yx_k}}$

$s_{x_k}$ (if exists), Rule [x-x]' ensures that $s_{x_k} \overset{L_{x\_}}{\preceq} s_0$, which means that $s_0 \in F$ whenever $s_{x_k} \in F$. Taking the invariant element into consideration, for each sub-transition chain $p'_R = s_0$ of $p_{G'}$, (i) $\boldsymbol{P}_{xy}$ always contains path corresponding to a sub-transition chain from $s_0$ to $s_0$, and (ii) for any path of $\boldsymbol{P}_{xy}$ corresponding to sub-transition chain from $s_0$ to $s_{x_k}$, $s_{x_k} \overset{L_{x\_}}{\preceq} s_0$.

**Type 1.2** Second, we consider the subtype where each $\boldsymbol{P}_{xy}$ is comprised of paths ending but not starting with $x$, and use $p_{G'}$ to denote the $xy$-folded path of $\boldsymbol{P}_{xy}$. (1) In this subtype, the invariant element of $\boldsymbol{P}_{xy}$ is the path $v_i \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_2} x$ not containing any edge joining $x$ and $y$. (2) Then we consider the path $p_G = v_i \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_2} x \xrightarrow{\ell_{xy_1}} y \xrightarrow{\ell_{yx_1}} x \xrightarrow{\ell_{xy_2}} \cdots \xrightarrow{\ell_{yx_k}} x$, where the $xy$-subpath is comprised of at least two edges joining $x$ and $y$ (back and forth). $p_G$ can always be seen as the concatenation of two subpaths $p_{G1}p_{G2}$, where $p_{G1} = v_i \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_2} x$ is exactly the invariant element, and $p_{G2} = x \xrightarrow{\ell_{xy_1}} y \xrightarrow{\ell_{yx_1}} x \xrightarrow{\ell_{xy_2}} \cdots \xrightarrow{\ell_{yx_k}} x$ is a path belonging to Type 1.1. Then, for $p_G$, each corresponding $p_R = s_0 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_2} s_x \xrightarrow{\ell_{xy_1}} s_y \xrightarrow{\ell_{yx_1}} s_{x_1} \xrightarrow{\ell_{xy_2}} \cdots \xrightarrow{\ell_{yx_k}} s_{x_k}$ can be seen as the concatenation of $p_{R1} = s_0 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_2} s_x$ and $p_{R2} = s_x \xrightarrow{\ell_{xy_1}} s_y \xrightarrow{\ell_{yx_1}} s_{x_1} \xrightarrow{\ell_{xy_2}} \cdots \xrightarrow{\ell_{yx_k}} s_{x_k}$, where $p_{R1}$ corresponds to $p_{G1}$, $p_{R2}$ corresponds to $p_{G2}$, $s_0 \in B^* \times Nr(L_{\_v_i})$ and $s_x \in B^* \times Nr(L_{\_x})$.

Obviously, for the case that $p_{R1}$ does not exist, $p_{R2}$ does not exist, either. For the case that $p_{R1}$ exists, as discussed in Type 1.1, there is always $s_{x_k} \overset{L_{x\_}}{\preceq} s_x$ in $p_{R2}$. Moreover, $p_{R1}$ is invariant as $p_{G1}$ is invariant, which means that the $xy$-folded path of $p_{G2}$ corresponds to $p'_{R2} = s_x$. Therefore, $p_{G'}$ has a corresponding sub-transition chain from $s_0$ to $s_x$ iff (i) $\boldsymbol{P}_{xy}$ has a path corresponding to a sub-transition chain from $s_0$ to $s_x$ and (ii) for any path of $\boldsymbol{P}_{xy}$ having a corresponding sub-transition chain from $s_0$ to $s_{x_k}$, $s_{x_k} \overset{L_{x\_}}{\preceq} s_x$.

Notably, it is valid that $s_0 = s_{init}$ and $s_x \in F$. Thus, for each $\boldsymbol{P}_{xy}$ of Type 1.1 and Type 1.2, whose $xy$-folded path is $p_{G'}$, Rule [x-x]' ensures that $p_{G'}$ is a reachable path iff $\boldsymbol{P}_{xy}$ contains a reachable path.

**Type 1.3** Finally, we consider the subtype where each $\boldsymbol{P}_{xy}$ is comprised of paths not ending with $x$, and use $p_{G'}$ to denote the $xy$-folded path of $\boldsymbol{P}_{xy}$. Each path $p_G$ belonging to $\boldsymbol{P}_{xy}$ can be seen as the concatenation of two subpaths $p_{G1}p_{G2}$ where $p_{G1} \in \boldsymbol{P}_{xy_1}$ belongs to Type 1.1 or Type 1.2 and $p_{G2} = x \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_4} v_k$ is a path not containing any edge joining $x$ and $y$, which is invariant before and after folding $(x, y)$. Correspondingly, $p_{G'}$ can be seen as the concatenation of two subpaths $p_{G1'}p_{G2'}$ where $p_{G1'}$ is the invariant element of $\boldsymbol{P}_{xy_1}$ and $p_{G2'} = p_{G2}$.

As discussed in Type 1.1 and 1.2, $p_{G1'}$ has a corresponding sub-transition chain from $s_0$ to $s_x$ iff (i) $\boldsymbol{P}_{xy_1}$ has a path corresponding to a sub-transition chain from $s_0$ to $s_x$ and (ii) for any path of $\boldsymbol{P}_{xy}$ having a corresponding sub-transition chain from $s_0$ to $s_{x_k}$, $s_{x_k} \overset{L_{x\_}}{\preceq} s_x$. This means that the corresponding $p_{R2}$ of $p_{G2}$ starts with $s_x$ or $s_{x_k}$, while the corresponding $p'_{R2}$ of $p_{G2}$ only starts with $s_x$. (1) For the case that $p_{R2}$ and $p'_{R2}$ both start with $s_x$, $p_{R2} = p'_{R2}$ as $p_{G2'} = p_{G2}$. (2) For the case that $p_{R2}$ starts with $s_{x_k}$ while $p'_{R2}$ starts with $s_x$, Definition 4.5 ensures that if $p_{R2}$ ends with $s_k$, $p'_{R2}$ also ends with $s_k$. Thus, $p_{G'}$ has a corresponding sub-transition chain from $s_0$ to $s_x$ then to $s_k$ iff $\boldsymbol{P}_{xy_1}$ has a path corresponding to a sub-transition chain from $s_0$ to $s_x$ then to $s_k$. Therefore, for each $\boldsymbol{P}_{xy}$ of Type 1.3, whose $xy$-folded path is $p_{G'}$, Rule [x-x]' ensures that $p_{G'}$ has a corresponding sub-transition chain from $s_0$ to $s_k$ iff $\boldsymbol{P}_{xy}$ also contains a path corresponding to a sub-transition chain from $s_0$ to $s_k$.

**Type 2 (Figure 7(b)).** For each $xy$-FEQ class $\boldsymbol{P}_{xy}$ belonging to Type 2, we use $p_{G'}$ to denote the $xy$-folded path of $\boldsymbol{P}_{xy}$. A path $p_G$ of $\boldsymbol{P}_{xy}$ can always be seen as the concatenation of two subpaths $p_{G1}$ and $p_{G2}$, where $p_{G1} \in \boldsymbol{P}_{xy_1}$ belongs to Type 1.1 or Type 1.2, and $p_{G2} = x \xrightarrow{\ell_{xy}} y \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_4} v_k$ contains only one edge joining $x$ and $y$. Correspondingly, $p_{G'}$ can be seen as the concatenation of two subpaths $p_{G1'}p_{G2'}$ where $p_{G1'}$ is the invariant element of $\boldsymbol{P}_{xy_1}$ and $p_{G2'} = x \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_4} v_k$.

As discussed in Type 1.1 and 1.2, $p_{G1'}$ has a corresponding sub-transition chain from

$s_0$ to $s_x$ iff (ii) $\boldsymbol{P}_{xy_1}$ has a path corresponding to a sub-transition chain from $s_0$ to $s_x$ and (ii) for any path of $\boldsymbol{P}_{xy}$ having a corresponding sub-transition chain from $s_0$ to $s_{x_k}$, $s_{x_k} \overset{L_{x\_}}{\preceq} s_x$. This means that the corresponding $p_{R2}$ of $p_{G2}$ starts with $s_x$ or $s_{x_k}$, while the corresponding $p'_{R2}$ of $p_{G2'}$ only starts with $s_x$. According to Definition 4.5, for all $\ell_{xy} \in L_{xy}$, $s_{x_k} \xrightarrow{\ell_{xy}} s_y \in \Delta$ indicates that $s_x \xrightarrow{\ell_{xy}} s_y \in \Delta$. Moreover, Rule [x-y]' ensures that $s_y \overset{L_{y\_}}{\simeq} s_x$ and $s_y \overset{L_{y\_}}{\simeq} s_{x_k}$.

Thus, (1) for the case that $p_{G2}$ is comprised of only one edge, i.e., $p_{G2} = x \xrightarrow{\ell_{xy}} y$ and $p_{G2'} = x$ corresponds to $p'_{R2} = s_x$, no matter $p_{R2}$ start with $s_x$ or $s_{x_k}$, $p_{R2}$ ends with $s_y$ such that $s_y \overset{L_{y\_}}{\simeq} s_x$, which means that $s_x \in F$ iff $s_y \in F$; (2) for the case that $p_{G2}$ is comprised of multiple edges, i.e., $p_{G2} = x \xrightarrow{\ell_{xy}} y \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_4} v_k$ and $p_{G2'} = x \xrightarrow{\ell_3} \cdots \xrightarrow{\ell_4} v_k$, $p_{G2'}$ corresponds to a sub-transition chain from $s_x$ to $s_k$ iff $p_{G2}$ corresponds to a sub-transition chain from $s_x$ or $s_{x_k}$ to $s_k$. Therefore, for each $\boldsymbol{P}_{xy}$ of Type 2, whose $xy$-folded path is $p_{G'}$, Rule [x-x]' and Rule [x-y]' ensures that (i) $p_{G'}$ is a reachable path iff $\boldsymbol{P}_{xy}$ contains a reachable path, and (ii) when the paths of $\boldsymbol{P}_{xy}$ do not end with $x$ or $y$, $p_{G'}$ has a corresponding sub-transition chain from $s_0$ to $s_k$ iff $\boldsymbol{P}_{xy}$ contains a path corresponding to a sub-transition chain from $s_0$ to $s_k$.

**Type 3 and Type 4 (Figure 7(c) and Figure 7(d))** These two types are symmetric to Type 1 and Type 2 with respect to $x$ and $y$. Hence, similar to Type 1 and Type 2, Rule [y-y]' and Rule [y-x]' ensures that for each $xy$-FEQ class $\boldsymbol{P}_{xy}$ of Type 3 and Type 4, whose $xy$-folded path is $p_{G'}$, (i) $p_{G'}$ is a reachable path iff $\boldsymbol{P}_{xy}$ contains a reachable path, and (ii) when the paths of $\boldsymbol{P}_{xy}$ do not end with $x$ or $y$, $p_{G'}$ has a corresponding sub-transition chain from $s_0$ to $s_k$ iff $\boldsymbol{P}_{xy}$ contains a path corresponding to a sub-transition chain from $s_0$ to $s_k$.

Putting the discussions of Types 1–4 together, Lemma A.1 is proved, which means that Principle ② has Property 4.2.

# BIBLIOGRAPHY

[1] R. AGRAWAL, A. BORGIDA, AND H. V. JAGADISH, *Efficient management of transitive relationships in large data and knowledge bases*, ACM SIGMOD Record, 18 (1989), pp. 253–262.

[2] A. V. AHO, M. S. LAM, R. SETHI, AND J. D. ULLMAN, *Compilers: principles, techniques, & tools*, Pearson Education India, 2007.

[3] A. AIKEN, *Introduction to set constraint-based program analysis*, Science of Computer Programming, 35 (1999), pp. 79–111.

[4] A. AIKEN AND E. L. WIMMERS, *Solving systems of set constraints*, in LICS, vol. 92, Citeseer, 1992, pp. 329–340.

[5] J. ALBERT, D. GIAMMARRESI, AND D. WOOD, *Normal form algorithms for extended context-free grammars*, Theoretical Computer Science, 267 (2001), pp. 35–47.

[6] R. ALUR, M. BENEDIKT, K. ETESSAMI, P. GODEFROID, T. REPS, AND M. YANNAKAKIS, *Analysis of recursive state machines*, ACM Transactions on Programming Languages and Systems (TOPLAS), 27 (2005), pp. 786–818.

[7] R. ALUR AND P. MADHUSUDAN, *Visibly pushdown languages*, in Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, 2004, pp. 202–211.

[8] ——, *Adding nesting structure to words*, Journal of the ACM (JACM), 56 (2009), pp. 1–43.

[9] A. AMBAINIS, Y. FILMUS, AND F. LE GALL, *Fast matrix multiplication: limitations of the coppersmith-winograd method*, in Proceedings of the forty-seventh annual ACM symposium on Theory of Computing, 2015, pp. 585–593.

[10] L. O. ANDERSEN, *Program analysis and specialization for the C programming language*, PhD thesis, University of Cophenhagen, 1994.

[11] J.-M. AUTEBERT, J. BERSTEL, AND L. BOASSON, *Context-free languages and pushdown automata*, in Handbook of formal languages, Springer, 1997, pp. 111–174.

[12] D. AVOTS, M. DALTON, V. B. LIVSHITS, AND M. S. LAM, *Improving software security with a C pointer analysis*, in ICSE '05, ACM, 2005, pp. 332–341.

[13] N. AYEWAH, W. PUGH, D. HOVEMEYER, J. D. MORGENTHALER, AND J. PENIX, *Using static analysis to find bugs*, IEEE software, 25 (2008), pp. 22–29.

[14] L. BACHMAIR, H. GANZINGER, AND U. WALDMANN, *Set constraints are the monadic class*, in [1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science, IEEE, 1993, pp. 75–83.

[15] G. BALATSOURAS AND Y. SMARAGDAKIS, *Structure-sensitive points-to analysis for C and C++*, in SAS '16, Springer, 2016, pp. 84–104.

[16] M. BENERECETTI, S. MINOPOLI, AND A. PERON, *Analysis of timed recursive state machines*, in 2010 17th International Symposium on Temporal Representation and Reasoning, IEEE, 2010, pp. 61–68.

[17] M. BERNDL, O. LHOTÁK, F. QIAN, L. HENDREN, AND N. UMANEE, *Points-to analysis using bdds*, in PLDI '03, vol. 38, ACM, 2003, pp. 103–114.

[18] J. BERSTEL, *Transductions and context-free languages*, Springer-Verlag, 2013.

[19] D. BEYER AND M. E. KEREMOGLU, *Cpachecker: A tool for configurable software verification*, in Proceedings of the 23rd International Conference on Computer Aided Verification, 2011.

[20] A. BOUAJJANI, M. MÜLLER-OLM, AND T. TOUILI, *Regular symbolic analysis of dynamic networks of pushdown systems*, in International Conference on Concurrency Theory, Springer, 2005, pp. 473–487.

[21] S. CERI, G. GOTTLOB, L. TANCA, ET AL., *What you always wanted to know about datalog(and never dared to ask)*, IEEE transactions on knowledge and data engineering, 1 (1989), pp. 146–166.

[22] T. M. CHAN AND R. WILLIAMS, *Deterministic apsp, orthogonal vectors, and more: Quickly derandomizing razborov-smolensky*, in Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms, SIAM, 2016, pp. 1246–1255.

[23] W. CHARATONIK AND L. PACHOLSKI, *Negative set constraints with equality*, in Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science, IEEE, 1994, pp. 128–136.

[24] ——, *Set constraints with projections are in nexptime*, in Proceedings 35th Annual Symposium on Foundations of Computer Science, IEEE, 1994, pp. 642–653.

[25] K. CHATTERJEE, B. CHOUDHARY, AND A. PAVLOGIANNIS, *Optimal dyck reachability for data-dependence and alias analysis*, Proc. ACM Program. Lang., 2 (2018), pp. 30:1–30:30.

[26] K. Chatterjee, R. Ibsen-Jensen, A. Pavlogiannis, and P. Goyal, *Faster algorithms for algebraic path properties in recursive state machines with constant treewidth*, in Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2015, pp. 97–109.

[27] Chaudhuri and Swarat, *Subcubic algorithms for recursive state machines*, in Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2008, pp. 159–169.

[28] Checkstyle, *Checkstyle*, https://checkstyle.sourceforge.io, (2022).

[29] B. Chess and G. McGraw, *Static analysis for security*, IEEE security & privacy, 2 (2004), pp. 76–79.

[30] N. Chomsky and M. P. Schützenberger, *The algebraic theory of context-free languages*, in Studies in Logic and the Foundations of Mathematics, vol. 26, Elsevier, 1959, pp. 118–161.

[31] P. A. Chou, *Recognition of equations using a two-dimensional stochastic context-free grammar*, in Visual Communications and Image Processing IV, vol. 1199, SPIE, 1989, pp. 852–865.

[32] E. M. Clarke, *Model checking*, in Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17, Springer, 1997, pp. 54–56.

[33] M. Colón, S. Sankaranarayanan, and H. Sipma, *Linear invariant generation using non-linear constraint solving*, in CAV, vol. 3, Springer, 2003, pp. 420–432.

[34] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977, pp. 238–252.

[35] D. Dietsch, M. Heizmann, A. Nutz, C. Schatzle, and F. Schussele, *Ultimate taipan with symbolic interpretation and fluid abstractions*, 2020.

[36] J. Eremondi, *Set constraints, pattern match analysis, and smt*, in International Symposium on Trends in Functional Programming, Springer, 2020, pp. 121–141.

[37] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, *Efficient algorithms for model checking pushdown systems*, in International Conference on Computer Aided Verification, Springer, 2000, pp. 232–247.

[38] M. FÄHNDRICH, J. S. FOSTER, Z. SU, AND A. AIKEN, *Partial online cycle elimination in inclusion constraint graphs*, in Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, 1998, pp. 85–96.

[39] FINDBUGS, *Findbugs*, https://findbugs.sourceforge.net/findbugs2.html, (2022).

[40] A. FINKEL, B. WILLEMS, AND P. WOLPER, *A direct symbolic approach to model checking pushdown systems*, Electronic Notes in Theoretical Computer Science, 9 (1997), pp. 27–37.

[41] O. GAUWIN, A. MUSCHOLL, AND M. RASKIN, *Minimization of visibly pushdown automata is np-complete*, arXiv preprint arXiv:1907.09563, (2019).

[42] R. GILLERON, S. TISON, AND M. TOMMASI, *Solving systems of set constraints with negated subset relationships*, in Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science, IEEE, 1993, pp. 372–380.

[43] S. GINSBURG AND S. GREIBACH, *Deterministic context free languages*, in 6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965), IEEE, 1965, pp. 203–220.

[44] R. C. GONZALEZ AND M. G. THOMASON, *Syntactic pattern recognition: an introduction*, (1978).

[45] A. GOSAIN AND G. SHARMA, *Static analysis: A survey of techniques and tools*, in Intelligent Computing and Applications, Springer, 2015, pp. 581–591.

[46] T. J. GREEN, S. S. HUANG, B. T. LOO, W. ZHOU, ET AL., *Datalog and recursive query processing*, Foundations and Trends® in Databases, 5 (2013), pp. 105–195.

[47] D. GROVE AND C. CHAMBERS, *A framework for call graph construction algorithms*, ACM Transactions on Programming Languages and Systems (TOPLAS), 23 (2001), pp. 685–746.

[48] D. GROVE, G. DEFOUW, J. DEAN, AND C. CHAMBERS, *Call graph construction in object-oriented languages*, in Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 1997, pp. 108–124.

[49] S. GULWANI, S. SRIVASTAVA, AND R. VENKATESAN, *Program analysis as constraint solving*, in Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008, pp. 281–292.

[50] T. HAO, X. WANG, L. ZHANG, X. BING, Z. LU, AND M. HONG, *Summary-based context-sensitive data-dependence analysis in presence of callbacks*, in Acm Sigplan-sigact Symposium on Principles of Programming Languages, 2015.

[51] B. HARDEKOPF AND C. LIN, *The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code*, in Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 290–299.

[52] ——, *Exploiting pointer and location equivalence to optimize pointer analysis*, in International Static Analysis Symposium, Springer, 2007, pp. 265–280.

[53] ——, *Semi-sparse flow-sensitive pointer analysis*, in POPL '09, vol. 44, ACM, 2009, pp. 226–238.

[54] D. L. HEINE AND M. S. LAM, *A practical flow-sensitive and context-sensitive c and c++ memory leak detector*, (2003), p. 168.

[55] N. HEINTZE AND J. JAFFAR, *A decision procedure for a class of set constraints*, in [1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science, IEEE, 1990, pp. 42–51.

[56] N. HEINTZE AND O. TARDIEU, *Ultra-fast aliasing analysis using cla: A million lines of c code in a second*, in PLDI '01, vol. 36, ACM, 2001, pp. 254–263.

[57] M. HEIZMANN, C. SCHILLING, AND D. TISCHNER, *Minimization of visibly pushdown automata using partial max-sat*, in International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2017, pp. 461–478.

[58] M. HIND, *Pointer analysis: Haven't we solved this problem yet?*, in Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2001, pp. 54–61.

[59] J. E. HOPCROFT, R. MOTWANI, AND J. D. ULLMAN, *Automata theory, languages, and computation*, International Edition, 24 (2006), pp. 171–183.

[60] J. E. HOPCROFT AND J. D. ULLMAN, *Formal languages and their relation to automata*, Addison-Wesley Longman Publishing Co., Inc., 1969.

[61] S. HORWITZ, T. REPS, AND M. SAGIV, *Demand interprocedural dataflow analysis*, ACM SIGSOFT Software Engineering Notes, 20 (1995), pp. 104–115.

[62] G. HOTZ, *Normal-form transformations of context-free grammars*, Acta Cybernetica, 4 (1978), pp. 65–84.

[63] Y. E. IOANNIDIS AND R. RAMAKRISHNAN, *Efficient transitive closure algorithms*, tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 1988.

[64] ISO90, *ISO/IEC. international standard ISO/IEC 9899, programming languages - C*, (1990).

[65] G. F. ITALIANO, *Amortized efficiency of a path retrieval data structure*, Theoretical Computer Science, 48 (1986), pp. 273–281.

[66] R. JOHNSON AND K. PINGALI, *Dependence-based program analysis*, in Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, 1993, pp. 78–89.

[67] H. JORDAN, B. SCHOLZ, AND P. SUBOTIĆ, *Soufflé*, (2016).

[68] ——, *Soufflé: On synthesis of program analyzers*, in International Conference on Computer Aided Verification, Springer, 2016, pp. 422–430.

[69] A. K. JOSHI AND Y. SCHABES, *Tree-adjoining grammars*, in Handbook of formal languages, Springer, 1997, pp. 69–123.

[70] G. A. KILDALL, *A unified approach to global program optimization*, in Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1973, pp. 194–206.

[71] D. E. KNUTH, *Semantics of context-free languages*, Mathematical systems theory, 2 (1968), pp. 127–145.

[72] V. KUMAR, P. MADHUSUDAN, AND M. VISWANATHAN, *Visibly pushdown automata for streaming xml*, in Proceedings of the 16th international conference on World Wide Web, 2007, pp. 1053–1062.

[73] C. LATTNER AND V. ADVE, *Llvm: A compilation framework for lifelong program analysis & transformation*, in International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE, 2004, pp. 75–86.

[74] C. LATTNER, A. LENHARTH, AND V. ADVE, *Making context-sensitive points-to analysis with heap cloning practical for the real world*, in PLDI'07, vol. 42, ACM, 2007, pp. 278–289.

[75] F. LE GALL, *Powers of tensors and fast matrix multiplication*, in Proceedings of the 39th international symposium on symbolic and algebraic computation, 2014, pp. 296–303.

[76] Y. LEI AND Y. SUI, *Fast and precise handling of positive weight cycles for field-sensitive pointer analysis*, in International Static Analysis Symposium, Springer, 2019, pp. 27–47.

[77] O. LHOTÁK AND K.-C. A. CHUNG, *Points-to analysis with efficient strong updates*, in POPL '11, 2011, pp. 3–16.

[78] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, *Static analysis of android apps: A systematic literature review*, Information and Software Technology, 88 (2017), pp. 67–95.

[79] L. Li, C. Cifuentes, and N. Keynes, *Boosting the performance of flow-sensitive points-to analysis using value flow*, in FSE '11, 2011, pp. 343–353.

[80] Y. Li, Q. Zhang, and T. Reps, *Fast graph simplification for interleaved dyck-reachability*, in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 780–793.

[81] P. Linz and S. H. Rodger, *An introduction to formal languages and automata*, Jones & Bartlett Learning, 2022.

[82] P. Liu, Y. Li, B. Swain, and J. Huang, *Pus: A fast and highly efficient solver for inclusion-based pointer analysis*, in International Conference on Software Engineering (ICSE'22), 2022.

[83] LLVM, *Clang static analyzer*, https://clang-analyzer.llvm.org, (2022).

[84] Y. Lu, L. Shang, X. Xie, and J. Xue, *An incremental points-to analysis with cfl-reachability*, in International Conference on Compiler Construction, Springer, 2013, pp. 61–81.

[85] J. Mantas, *Methodologies in pattern recognition and image analysis—a brief survey*, Pattern Recognition, 20 (1987), pp. 1–6.

[86] D. Melski and T. Reps, *Interconvertibility of a class of set constraints and context-free-language reachability*, Theoretical Computer Science, 248 (2000), pp. 29–98.

[87] A. Miné, *Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics*, in LCTES '06, vol. 41, ACM, 2006, pp. 54–63.

[88] D. E. Muller and P. E. Schupp, *The theory of ends, pushdown automata, and second-order logic*, Theoretical Computer Science, 37 (1985), pp. 51–75.

[89] N. A. Naeem and O. Lhoták, *Typestate-like analysis of multiple interacting objects*, ACM Sigplan Notices, 43 (2008), pp. 347–366.

[90] P. Nappa, D. Zhao, P. Subotić, and B. Scholz, *Fast parallel equivalence relations in a datalog compiler*, in 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2019, pp. 82–96.

[91] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*, springer, 2015.

[92] E. NUUTILA AND E. SOISALON-SOININEN, *On finding the strongly connected components in a directed graph*, Information Processing Letters, 49 (1994), pp. 9–14.

[93] E. M. NYSTROM, H.-S. KIM, AND W.-M. W. HWU, *Importance of heap specialization in pointer analysis*, in PASTE '04, ACM, 2004, pp. 43–48.

[94] OPEN64, *Implementing next generation points-to in open64*. `www.affinic.com/documents/open64workshop/2010/slides/8_Ravindran.ppt`.

[95] D. J. PEARCE, P. H. KELLY, AND C. HANKIN, *Online cycle detection and difference propagation for pointer analysis*, in Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation, IEEE, 2003, pp. 3–12.

[96] ——, *Efficient field-sensitive pointer analysis of C*, TOPLAS, 30 (2007), p. 4.

[97] F. M. Q. PEREIRA AND D. BERLIN, *Wave propagation and deep propagation for pointer analysis*, in 2009 International Symposium on Code Generation and Optimization, IEEE, 2009, pp. 126–135.

[98] PMD, *Pmd*, https://pmd.github.io/latest, (2022).

[99] A. POTAMIANOS AND H.-K. J. KUO, *Statistical recursive finite state machine parsing for speech understanding.*, in INTERSPEECH, Citeseer, 2000, pp. 510–513.

[100] P. PRATIKAKIS, J. S. FOSTER, AND M. HICKS, *Existential label flow inference via cfl reachability*, in International Static Analysis Symposium, Springer, 2006, pp. 88–106.

[101] G. K. PULLUM AND G. GAZDAR, *Natural languages and context-free languages*, Linguistics and Philosophy, 4 (1982), pp. 471–504.

[102] P. PURDOM, *A sentence generator for testing parsers*, BIT Numerical Mathematics, 12 (1972), pp. 366–375.

[103] R. C. READ, *Graph theory and computing*, Academic Press, 2014.

[104] J. REHOF AND M. FÄHNDRICH, *Type-base flow analysis: from polymorphic subtyping to cfl-reachability*, ACM SIGPLAN Notices, 36 (2001), pp. 54–66.

[105] REPS, THOMAS, HORWITZ, SUSAN, SAGIV, AND MOOLY, *Precise interprocedural dataflow analysis via graph reachability*, in Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1995, pp. 49–61.

[106] T. REPS, *Shape analysis as a generalized path problem*, in Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, 1995, pp. 1–11.

[107] T. Reps, A. Lal, and N. Kidd, *Program analysis using weighted pushdown systems*, in International Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, 2007, pp. 23–51.

[108] T. Reps, S. Schwoon, S. Jha, and D. Melski, *Weighted pushdown systems and their application to interprocedural dataflow analysis*, Science of Computer Programming, 58 (2005), pp. 206–263.

[109] J. C. Reynolds, *Automatic computation of data set definitions*, (1969).

[110] L. Roditty and U. Zwick, *A fully dynamic reachability algorithm for directed graphs with an almost linear update time*, in Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, 2004, pp. 184–191.

[111] A. Rountev and S. Chandra, *Off-line variable substitution for scaling points-to analysis*, Acm Sigplan Notices, 35 (2000), pp. 47–56.

[112] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, *On fast large-scale program analysis in datalog*, in Proceedings of the 25th International Conference on Compiler Construction, 2016, pp. 196–206.

[113] M. P. Schützenberger, *On context-free languages and push-down automata*, Information and control, 6 (1963), pp. 246–264.

[114] L. Shang, X. Xie, and J. Xue, *On-demand dynamic summary-based points-to analysis*, in Proceedings of the Tenth International Symposium on Code Generation and Optimization, 2012, pp. 264–274.

[115] J. Späth, K. Ali, and E. Bodden, *Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems*, Proceedings of the ACM on Programming Languages, 3 (2019), pp. 1–29.

[116] M. Sridharan and R. Bodík, *Refinement-based context-sensitive points-to analysis for java*, ACM SIGPLAN Notices, 41 (2006), pp. 387–400.

[117] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, *Demand-driven points-to analysis for java*, ACM SIGPLAN Notices, 40 (2005), pp. 59–76.

[118] V. Strassen et al., *Gaussian elimination is not optimal*, Numerische mathematik, 13 (1969), pp. 354–356.

[119] Z. Su, M. Fähndrich, and A. Aiken, *Projection merging: Reducing redundancies in inclusion constraint graphs*, in Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2000, pp. 81–95.

[120] Y. SUI, *Side-effects of external apis*. `https://github.com/SVF-tools/SVF/blob/master/lib/Util/ExtAPI.cpp`.

[121] Y. SUI, X. FAN, H. ZHOU, AND J. XUE, *Loop-oriented array-and field-sensitive pointer analysis for automatic SIMD vectorization*, in LCTES '16, ACM, 2016, pp. 41–51.

[122] Y. SUI, Y. LI, AND J. XUE, *Query-directed adaptive heap cloning for optimizing compilers*, in Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2013, pp. 1–11.

[123] Y. SUI AND J. XUE, *On-demand strong update analysis via value-flow refinement*, in FSE '16, ACM, 2016, pp. 460–473.

[124] Y. SUI AND J. XUE, *Svf: Interprocedural static value-flow analysis in LLVM*, in CC '16, 2016, pp. 265–266.

[125] Y. SUI AND J. XUE, *Svf: interprocedural static value-flow analysis in llvm*, in Proceedings of the 25th international conference on compiler construction, 2016, pp. 265–266.

[126] Y. SUI, D. YE, AND J. XUE, *Detecting memory leaks statically with full-sparse value-flow analysis*, IEEE Transactions on Software Engineering, 40 (2014), pp. 107–122.

[127] R. TARJAN, *Depth-first search and linear graph algorithms*, SIAM journal on computing, 1 (1972), pp. 146–160.

[128] W. THOMAS, *Languages, automata, and logic*, in Handbook of formal languages, Springer, 1997, pp. 389–455.

[129] TRAVITCH, *Whole-program llvm*, https://github.com/travitch/whole-program-llvm, (2022).

[130] W. T. TUTTE AND W. T. TUTTE, *Graph theory*, vol. 21, Cambridge university press, 2001.

[131] I. WALUKIEWICZ, *Model checking ctl properties of pushdown systems*, in International Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, 2000, pp. 127–138.

[132] H. WANG, X. XIE, Y. LI, C. WEN, Y. LI, Y. LIU, S. QIN, H. CHEN, AND Y. SUI, *Typestate-guided fuzzer for discovering use-after-free vulnerabilities*, in 42nd International Conference on Software Engineering, ACM, 2020.

[133] K. WANG, A. HUSSAIN, Z. ZUO, G. XU, AND A. AMIRI SANI, *Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code*, ACM SIGARCH Computer Architecture News, 45 (2017), pp. 389–404.

[134] ——, *Graspan-cpp*, https://github.com/Graspan/graspan-cpp, (2020).

[135] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, *Using datalog with binary decision diagrams for program analysis*, in Asian Symposium on Programming Languages and Systems, Springer, 2005, pp. 97–118.

[136] R. Williams, *Faster all-pairs shortest paths via circuit complexity*, in Proceedings of the forty-sixth annual ACM symposium on Theory of computing, 2014, pp. 664–673.

[137] V. V. Williams, *Multiplying matrices faster than coppersmith-winograd*, in Proceedings of the forty-fourth annual ACM symposium on Theory of computing, 2012, pp. 887–898.

[138] V. V. Williams and R. R. Williams, *Subcubic equivalences between path, matrix, and triangle problems*, J. ACM, 65 (2018), pp. 27:1–27:38.

[139] G. Xu, A. Rountev, and M. Sridharan, *Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis*, in European Conference on Object-Oriented Programming, Springer, 2009, pp. 98–122.

[140] Z. Xu, L. Zheng, and H. Chen, *A toolkit for generating sentences from context-free grammars*, in 2010 8th IEEE International Conference on Software Engineering and Formal Methods, IEEE, 2010, pp. 118–122.

[141] M. Yannakakis, *Graph-theoretic methods in database theory*, in Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, 1990, pp. 230–242.

[142] S. Ye, Y. Sui, and J. Xue, *Region-based selective flow-sensitive pointer analysis*, in SAS '14, 2014, pp. 319–336.

[143] D. M. Yellin, *Speeding up dynamic transitive closure for bounded degree graphs*, Acta Informatica, 30 (1993), pp. 369–384.

[144] H. Yuan and P. Eugster, *An efficient algorithm for solving the dyck-cfl reachability problem on trees*, in European Symposium on Programming, Springer, 2009, pp. 175–189.

[145] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su, *Fast algorithms for dyck-cfl-reachability with applications to alias analysis*, in Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, 2013, pp. 435–446.

[146] Q. Zhang and Z. Su, *Context-sensitive data-dependence analysis via linear conjunctive language reachability*, in Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, 2017, pp. 344–358.

[147] Q. ZHANG, X. XIAO, C. ZHANG, H. YUAN, AND Z. SU, *Efficient subcubic alias analysis for c*, in Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, 2014, pp. 829–845.

[148] D. ZHAO, P. SUBOTIC, M. RAGHOTHAMAN, AND B. SCHOLZ, *Towards elastic incrementalization for datalog*, in 23rd International Symposium on Principles and Practice of Declarative Programming, 2021, pp. 1–16.

[149] X. ZHENG AND R. RUGINA, *Demand-driven alias analysis for c*, in Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2008, pp. 197–208.

[150] Z. ZUO, K. WANG, A. HUSSAIN, A. A. SANI, Y. ZHANG, S. LU, W. DOU, L. WANG, X. LI, C. WANG, ET AL., *Systemizing interprocedural static analysis of large-scale systems code with graspan*, ACM Transactions on Computer Systems (TOCS), 38 (2021), pp. 1–39.