Context-Sensitive Data-Dependence Analysis via Linear Conjunctive Language Reachability

Qirun Zhang Zhendong Su
University of California, Davis, United States
{qrzhang, su}@ucdavis.edu

Abstract

Many program analysis problems can be formulated as graph reachability problems. In the literature, context-free language (CFL) reachability has been the most popular formulation and can be computed in subcubic time. The context-sensitive data-dependence analysis is a fundamental abstraction that can express a broad range of program analysis problems. It essentially describes an interleaved matched-parenthesis language reachability problem. The language is not context-free, and the problem is well-known to be undecidable. In practice, many program analyses adopt CFL-reachability to exactly model the matched parentheses for either context-sensitivity or structure-transmitted data-dependence, *but not both*. Thus, the CFL-reachability formulation for context-sensitive data-dependence analysis is inherently an approximation.

To support more precise and scalable analyses, this paper introduces linear conjunctive language (LCL) reachability, a new, expressive class of graph reachability. LCL not only contains the interleaved matched-parenthesis language, but is also closed under all set-theoretic operations. Given a graph with n nodes and m edges, we propose an $O(mn)$ time approximation algorithm for solving *all-pairs* LCL-reachability, which is asymptotically better than known CFL-reachability algorithms. Our formulation and algorithm offer a new perspective on attacking the aforementioned undecidable problem — the LCL-reachability formulation is exact, while the LCL-reachability algorithm yields a sound approximation. We have applied the LCL-reachability framework to two existing client analyses. The experimental results show that the LCL-reachability framework is both more precise and scalable than the traditional CFL-reachability framework. This paper opens up the opportunity to exploit LCL-reachability in program analysis.

Categories and Subject Descriptors F.1.1 [*Models of Computation*]: Automata; F.2.2 [*Nonnumerical Algorithms and Problems*]: Computations on discrete structures; F.4.3 [*Formal Languages*]: Classes defined by grammars or automata

Keywords Context-free language reachability, linear conjunctive grammar, trellis automata, program analysis

POPL'17, January 15–21, 2017, Paris, France
© 2017 ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009848

1. Introduction

A variety of program analysis problems, such as interprocedural data flow analysis [33], program slicing [32], shape analysis [29], taint analysis [10], type-based flow analysis [26, 28], automatic specification inference [3] and pointer analysis [35, 40, 42, 43] can be formulated as formal language reachability (L-reachability) problems [30]. A problem instance of L-reachability contains (1) an edge-labeled graph G that abstracts the specific analysis problem and (2) an L-reachability formulation that captures desired analysis properties via computing reachability between nodes in G.

Context-free language (CFL) reachability [30] has been the most popular formulation developed over the past decades. In practice, the most widely instantiated CFL-reachability formulation is Dyck-reachability [16, 41]. Particularly, many program analyses use a Dyck language[1] to exactly model the *matched-parenthesis* property, which can be categorized as either *context-sensitivity* or *structure-transmitted data-dependence* [31]. Specifically, context-sensitivity describes the well-balanced procedure calls and returns as open and close parentheses, respectively. Similarly, the structure-transmitted data-dependence depicts yet another well-balanced property among language constructors, such as field accesses (*i.e.*, reads and writes [3, 35, 40]), list constructors (*i.e.*, car and cdr [31]), pointer indirections (*i.e.*, references and dereferences [42, 43]) or synchronizations (*i.e.*, lock and unlock [15, 27]). In the ideal case, a static analysis could dramatically improve its precision by leveraging both well-balanced properties. However, it is well-known that the precise analysis that simultaneously captures two or more well-balanced properties is undecidable [31]. It remains a challenging open problem to develop a general framework for context-sensitive and structure-transmitted data-dependence analysis.

A traditional (yet still popular) approximate solution to the undecidable problem is via a CFL-reachability-based approach. However, the analysis matching both well-balanced properties describes an interleaved matched-parenthesis language, which is not even context-free. On the other hand, CFLs are not closed under intersection [7, 9]. Therefore, a practical analysis must sacrifice the precision of either context-sensitivity or data-dependence by approximating it using a regular language [10, 35, 40]. Nevertheless, the size of the approximating regular language can be exponential in the size of the original CFL [17, 23, 37].

Moreover, CFL-reachability information is quite expensive to compute. Traditional CFL-reachability algorithms exhibit a subcubic time complexity [4, 30], and thus do not scale well in practice. Fast algorithms and implementations are known only for special cases [39, 41, 42]. Recent work by Tang *et al.* [36] intro-

[1] The Dyck language is an important subclass of CFL that essentially generates the well-matched parentheses. Formally, a Dyck language D_k of size k is generated by the following rules: $S \to S\,S \mid (_0\, S\,)_0 \mid \ldots \mid (_k\, S\,)_k \mid \epsilon$.

Type	L-reachability		Complexity	
	Formulation	Algorithm	Time	Space
CFL	sound	exact	$O(n^3/\log n)$	$O(n^2)$
TAL	sound	exact	$O(n^6)$	$O(n^2)$
LCL	exact	sound	$O(mn)$	$O(n^2)$

Table 1. Comparisons among L-reachability frameworks for context-sensitive data-dependence analysis, where "sound" stands for sound, over-approximation and "exact" a fully precise formulation or algorithm.

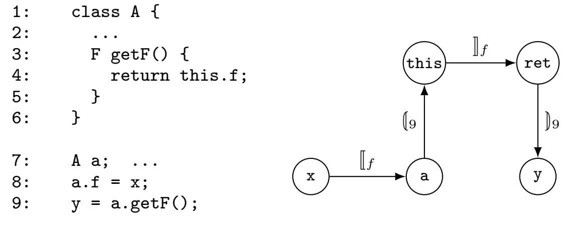
duces a new tree-adjoining language (TAL) reachability formulation for context-sensitive analysis without structure-transmitted data-dependence. The TAL-reachability algorithm is more scalable than CFL-reachability on the specific client analysis of constructing method summaries. However, TAL does not contain the interleaved matched-parenthesis language and the worst case time complexity of the TAL-reachability algorithm in general is $O(n^6)$.

In this paper, we study a new class of formal language reachability formulation called *linear conjunctive language (LCL) reachability*. This class of languages lies strictly between linear context-free and deterministic context-sensitive languages, and is incomparable with CFLs [24]. The most appealing properties of LCLs are that LCLs contain the interleaved matched-parenthesis language for context-sensitive structure-transmitted data-dependence analysis and are also closed under all set-theoretic operations [11, 13, 24]. The exact LCL-reachability problem is undecidable. We propose a sound approximation algorithm[2] for solving *all-pairs* LCL-reachability in $O(mn)$ time for a graph with n nodes and m edges.

In general, our LCL-reachability formulation yields a new perspective on solving the context-sensitive structure-transmitted data-dependence analysis problems. From a language-theoretic perspective, the formulation is exact compared with traditional CFL-reachability formulations. As a result, practical analyses only need to focus on the algorithmic aspects for developing a sound approximation for the reachability problem. From an algorithmic perspective, our algorithm is asymptotically faster than the subcubic CFL-reachability algorithm. Table 1 provides a summary comparison among the traditional CFL-reachability and our new LCL-reachability formulations.

We make the following main contributions in this paper:

- We present a new L-reachability formulation called linear conjunctive language (LCL) reachability. We instantiate the LCL-reachability formulation to context-sensitive structure-transmitted data-dependence analysis. To the best of our knowledge, this is the first exact L-reachability formulation of this important non-context-free language reachability problem.
- We propose a general algorithm for solving the *all-pairs* LCL-reachability problem. Moreover, we take the context-sensitive structure-transmitted date-dependence analysis as an example, and propose another refined LCL-reachability algorithm for this specific analysis. Given an input graph with n nodes and m edges, both our sound approximation algorithms solve the reachability problems in $O(mn)$ time, which is asymptotically better than the CFL-reachability algorithms.
- We apply the proposed LCL-reachability algorithm on two practical context-sensitive data-dependence analyses, *i.e.*, an alias analysis for Java [40] and a taint analysis for Android apps [10]. Our results illustrate significant precision and scalability advantages of the proposed LCL-reachability framework.



```
1:    class A {
2:      ...
3:      F getF() {
4:        return this.f;
5:      }
6:    }

7:    A a;  ...
8:    a.f = x;
9:    y = a.getF();
```

(a) A code snippet.

(b) Its graph representation.

Figure 1. A taint analysis example.

The rest of the paper is structured as follows. Section 2 motivates context-sensitive data-dependence analysis, and Section 3 reviews necessary background. Next, we present the LCL-reachability formulation (Section 4) and our LCL-reachability algorithm (Section 5). Then, Section 6 describes our evaluation setup and results, and Section 7 presents further discussions. Finally, Section 8 surveys related work, and Section 9 concludes.

2. Motivating Example

We give a concrete example to motivate context-sensitive structure-transmitted data-dependence analysis. Note that the context-sensitive data-dependence analysis considered by Tang *et al.* in their work on TAL-reachability [36] is without structure transmission. Unless otherwise stated, the data-dependence analysis discussed in latter sections is always structure-transmitted.

We consider a simple taint analysis for a Java-like language in Figure 1. In the code snippet shown in Figure 1(a), we would like to check whether a potential tainted value x flows to the variable y. Figure 1(b) gives a graphical view of the taint analysis. Each node represents a variable, and each edge an assignment. Specifically, on line 8, value x is put into field f, therefore, the directed edge is labeled with an open bracket $\llbracket_f$. Similarly, the return value ret gets the field value f from this, so the corresponding edge is labeled with a close bracket $\rrbracket_f$. Moreover, the parentheses $\llparenthesis_9$ and $\rrparenthesis_9$ depict the call and return of the procedure invocation occurred on line 9, respectively. In the L-reachability frameworks, variable y is tainted only if node y is reachable from node x in Figure 1(b) and the corresponding path string belongs to language L.

It is straightforward that both the calls/returns (*i.e.*, context-sensitivity) and putfield/getfield (*i.e.*, field-sensitivity) need to satisfy the balanced properties. Let the Dyck language D_m describe context sensitivity and D_n field sensitivity. It is clear that the path string "$\llbracket_f \llparenthesis_9 \rrbracket_f \rrparenthesis_9$" between nodes x and y belongs to the intersection of two Dyck languages $D_m \cap D_n$, which is not context-free.[3] The class of LCL contains $D_m \cap D_n$ since it contains Dyck languages and is closed under intersection [24, 25]. Therefore, an LCL-reachability-based analysis is able to report a potential tainted variable y. No other tainted variables will be reported for this example since all other path strings are not in $D_m \cap D_n$. Using a regular language R to approximate one Dyck language D, a CFL-reachability-based analysis may also report this tainted variable based on the CFL $D_m \cap R_n$ or $R_m \cap D_n$. However, the approximating regular language R may introduce additional spurious tainted sites, and thus decrease the analysis precision. Moreover, the regular approximation causes an exponential blowup in grammar size [17, 23, 37].

[2] The term approximation here reflects the general concept in program analysis, which should not be confused with its usage in "approximation algorithms".

[3] To perform the language intersection, the alphabets for both Dyck languages are extended to $\Sigma = \Sigma_{D_m} \cup \Sigma_{D_n}$.

3. Background and Definitions

Linear conjunctive languages can be described in terms of both the grammatical model (*i.e.*, Linear conjunctive grammars) and the operational model (*i.e.*, trellis automata). The two models are equivalent [25]. Unlike context-free grammars, most linear conjunctive grammars are specified using a direct automaton-to-grammar construction, which are quite complex and not intuitively meaningful [25]. For instance, the grammar size for Dyck language D_k could be $O(k^4)$ and the resulting grammar is not human-readable. Therefore, our LCL-reachability algorithm is based on trellis automata.

This section gives the background on our formulation for context-sensitive data-dependence analysis. Section 3.1 introduces the trellis automata. Section 3.2 formally defines the problem. We assume the reader is familiar with fundamental concepts in languages and automata and refer the reader to classical treatments [7, 9] of the subjects.

3.1 Trellis Automata

Trellis automata (TA) can be defined through the trellis representing their computation [5, 6]. A trellis automaton processes an input string of length $n \geqslant 1$ using a uniform triangular array with $\frac{n(n+1)}{2}$ processor nodes. Every processor node computes a value in a fixed finite set Q_l according to node labels l. In particular, each processor node in the bottom row reads a corresponding symbol directly from the input tape, and computes a value using a function $I_l : \Sigma \rightarrow Q_l$. The rest of the nodes compute the value using the function $\delta_l : Q_{l_1} \times Q_{l_2} \rightarrow Q_l$ from the output of both their left and right predecessors. The distance between any nodes and the topmost node is denoted as Level(k). The string is accepted if and only if the value computed by the topmost node belongs to the set of accepting states $F \subseteq Q_l$. Formally, we have:

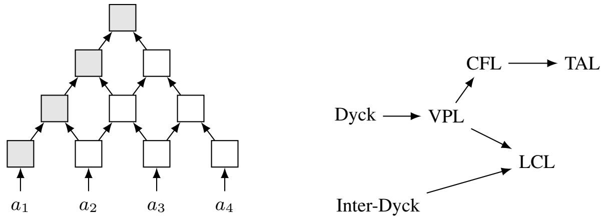
DEFINITION 1. *A trellis automaton is defined as a 6-tuple* $K = (\Sigma, L, Q_l, I_l, \delta_l, F)$, *where*

- Σ *is the input alphabet,*
- L *is a set of node labels,*
- Q_l *is a finite non-empty set of states for* l*-labeled nodes,*
- $I_l : \Sigma \rightarrow Q_l$ *is a function that generates the initial states,*
- $\delta_l : Q_{l_1} \times Q_{l_2} \rightarrow Q_l$ *is the set of transition functions, and*
- $F \subseteq Q_l$ *is the set of accepting states.*

The result of the computation on a string $w \in \Sigma^+$ *is denoted by* $\Delta : \Sigma^+ \rightarrow Q_l$, *which is defined inductively as* $\Delta(a) = I_l(a)$ *and* $\Delta(awb) = \delta_l(\Delta(aw), \Delta(wb))$, *for any* $a, b \in \Sigma$ *and* $w \in \Sigma^*$. *Finally, the language recognized by the automaton is* $L(K) = \{w \mid \Delta(w) \in F\}$.

DEFINITION 2. *A trellis automaton* K *is said to be* semihomogeneous *(STA) if the label of any node on Level*(k) *can be uniquely determined by the label of* any *ancestor on Level*$(k-1)$. K *is a* homogeneous *trellis automaton (HTA) if all nodes are labeled by the same symbol.*

Figure 2(a) shows an STA processing a string "$a_1a_2a_3a_4$" of length 4. Specifically, the leftmost gray nodes on each level are labeled by g and the remaining white nodes are labeled by w. It is straightforward to verify that the color of any node is uniquely determined by one of its ancestor(s). We define such an STA to be a *gray-white trellis automaton* (GWTA). The GWTA recognizes the inter-Dyck language and thus plays a pivotal role for our context-sensitive data-dependence algorithm. Moreover, the STA becomes an HTA if all nodes are white nodes. Let the languages recognized by STA and HTA be $L(STA)$ and $L(HTA)$, respectively. The two automata have the same expressive power:



(a) An STA.

(b) Relationship among languages.

Figure 2. Semihomogeneous trellis automata and LCL. In the right figure, the arrows denote inclusions.

THEOREM 1 (Culik II *et al.* [5, Thm. 2]). $L(STA) = L(HTA)$.

The class of $L(HTA)$ is also known as linear conjunctive languages (LCLs) [25]. We briefly compare LCLs to several important formal languages used in program analysis. Context-free language (CFL) is the most fundamental language class which has enabled many practical analyses [30]. Tree-adjoining language (TAL) [14] is a proper superset of CFL with increased expressiveness. Theoretically, TAL is more expensive to parse and formulate as a graph reachability problem. In practice, recent work by Tang *et al.* [36] demonstrates the effectiveness of applying TAL-reachability to compute method summaries. However, neither CFL nor TAL contains the interleaved matched-parenthesis languages (Inter-Dyck) for context-sensitive data-dependence analysis. On the other hand, LCL is strictly a superset of linear CFL, but is incomparable with CFL. Specifically, LCL contains some CFLs, such as Dyck, and some non-CFLs, such as Inter-Dyck. Moreover, the visibly pushdown language (VPL) class is introduced by Alur and Madhusudan for verifying "context-free"-style program properties [2]. LCL contains VPL. Figure 2(b) summarizes the relationship among these language classes.

3.2 Problem Statement

We give the formal definitions of the problems considered in this paper. Let $K = (\Sigma, L, Q_l, I_l, \delta_l, F)$ be a trellis automaton. Given a directed graph $G = (V, E)$ with each edge $(u, v) \in E$ labeled by a terminal $\mathcal{L}(u, v) \in \Sigma$, a path $p = v_0, v_1, \ldots, v_m$ realizes a string $R(p)$ by concatenating the edge labels along the path, *i.e.*, $R(p) = \mathcal{L}(v_0, v_1)\mathcal{L}(v_1, v_2)\ldots\mathcal{L}(v_{m-1}v_m)$. The realized string $R(p)$ is also defined as a path string. Consider a path $p = u, \ldots, v$, if the path string $R(p)$ can be derived from a trellis automaton state $q \in Q_l$, we summarize the path p as a *summary edge* (u, q, v). We also call state q a *summary*.

DEFINITION 3 (LCL-reachability). *Given an edge labeled digraph* $G = (V, E)$ *and a semihomogeneous trellis automaton* $K = (\Sigma, L, Q_l, I_l, \delta_l, F)$, *compute the summary edges* (u, q, v) *for all* $u, v \in V$, *where* $q \in F$.

Context-sensitive data-dependence analysis can be formulated as an LCL-reachability problem by restricting the STA to an GWTA which recognizes the intersection of two Dyck languages D_m and D_n. Let $\mathbb{D}_{mn}$ and $K_{D_{mn}}$ denote the language $D_m \cap D_n$ and its equivalent GWTA, respectively. We have,

DEFINITION 4. *Context-sensitive data-dependence analysis is an instance of LCL-reachability by restricting the trellis automation* K *to* $K_{D_{mn}}$.

THEOREM 2 (Reps [31]). *Context-sensitive data-dependence analysis is undecidable.*

COROLLARY 1. *The LCL-reachability problem is undecidable.*

Let ϕ and ϕ_{LCL} denote the sets of summary edges in an exact solution and a result obtained by an LCL-reachability algorithm, respectively. The soundness of an LCL-reachability algorithm is defined similarly as the soundness of a static analysis, *i.e.*, for each summary $e \in \phi_{LCL}$, it is undecidable to determine whether e is in ϕ and $e \notin \phi_{LCL}$ implies $e \notin \phi$. Formally, we have the following definition:

DEFINITION 5 (Sound Approximation). *An LCL-reachability algorithm is a sound approximation if and only if it computes a solution ϕ_{LCL} such that $\phi \subseteq \phi_{LCL}$.*

4. Formulation for Context-Sensitive Data-Dependence Analysis

This section describes the idea of constructing a GWTA $K_{D_{mn}}$ for context-sensitive data-dependence analysis.

4.1 Recognizing $\mathbb{D}_{mn}$

Constructing a TA as a language recognizer is challenging since TA is essentially a systolic system for which one has to deal with a large number of synchronized processes (*i.e.*, TA nodes) [12]. Many TA constructions involve nontrivial programming techniques [6] and the correctness proofs are difficult to obtain as well [11–13].

To make the TA construction more accessible, a restricted type of Turing machines named *deterministic simple Turing machine* (DSTM) was introduced by Ibarra *et al.* [11, 13]. The DSTM model not only provides a sequential machine characterization of TA but also enables more principled proofs of correctness. The DSTM shown in Figure 3(a) is equipped with a read-write work tape and a read-only input tape. The work tape, with a start marker \$, is infinite to the right, containing all blanks marked by λ initially . The definition of DSTM is formally given as follows:

DEFINITION 6 ([11, 13]). *A deterministic simple Turing machine (DSTM) is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where*

- *Q is a finite set of states,*
- *Σ is the input alphabet,*
- *Γ is the work tape alphabet, containing two special symbols \$ (start marker) and λ (blank marker),*
- *$q_0 \in Q$ is the start state,*
- *$F \subseteq Q$ is a finite set of accepting states, and*
- *$\delta : Q \times \Gamma \times (\Sigma \cup \{\epsilon\}) \rightarrow Q \times (\Gamma \setminus \{\lambda\}) \times \{-1, +1\}$ is a set of state transitions, where -1 and $+1$ denote left and right moves on work tape, respectively. And thus, M does not write λ (blank symbol) on work tape.*

The restriction on δ is as follows. For all $q, q' \in Q$, $Z, Z' \in \Gamma$, $a \in \Sigma \cup \{\epsilon\}$ and $d \in \{-1, +1\}$, if $\delta(q, Z, a) = (q', Z', d)$, then:

(1) $Z' = \$$ iff $Z = \$$;

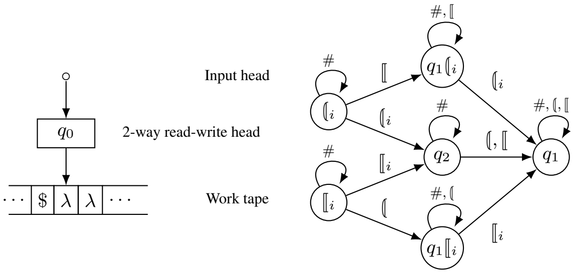
(2) If $q = q_0$ and $Z \neq \lambda$, then $a = \epsilon, q' = q_0, Z' = Z$ and $d = +1$: left-to-right sweep without altering the state and work tape;

(3) If $q = q_0$ and $Z = \lambda$, then $a \neq \epsilon, q' \neq q_0$ and $d = -1$: end of a left-to-right sweep with reading an input;

(4) If $q \neq q_0$ and $Z \neq \$$, then $a = \epsilon, q' \neq q_0$ and $d = -1$: right-to-left sweep with rewriting the work tape;

(5) If $q \neq q_0$ and $Z = \$$, then $a = \epsilon, q' = q_0$ and $d = +1$: end of a right-to-left sweep.

The restriction imposed by (1)−(5) on DSTM is that the read-write head can only make alternate sweeps (left-to-right and right-to-left between \$ and the leftmost λ) on the work tape. Moreover, on any left-to-right sweep, DSTM is required to remain in state q_0 without altering the contents of the work tape until it reaches the leftmost λ symbol. DSTM then reads an input from the input tape, enters a state other than q_0, rewrites a symbol from $\Gamma \setminus \{\lambda\}$, and proceeds to the left. On the right-to-left sweep, DSTM can arbitrarily rewrite symbols from $\Gamma \setminus \{\lambda\}$ on the work tape. Once DSTM reaches the start marker, it proceeds to the next left-to-right sweep. DSTM accepts a string if the input is exhausted and the machine enters an accepting state $q \in F$ with read-write head on the start marker \$.



(a) A DSTM. (b) Right-to-left sweep for $\mathbb{D}_{mn}$.

Figure 3. Illustrations on a DSTM.

EXAMPLE 1 (Dyck recognition [11, Example 4.1]). *Let D_m be the Dyck language of size m. It is straightforward to see that any non-empty string $x \in D_m$ reduces to ϵ by repeatedly applying the cancellation rule $\llparenthesis_i \rrparenthesis_i = \epsilon$. Given an input string $x \in \Sigma^+$, the basic idea of constructing a DSTM M is by applying the cancellation rules. Formally, $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2\} \cup \{\llparenthesis_1, \ldots, \llparenthesis_m\}$, $\Sigma = \{\llparenthesis_1, \rrparenthesis_1, \ldots, \llparenthesis_m, \rrparenthesis_m\}$, $\Gamma = \{\$, \lambda, \#\} \cup \{\llparenthesis_1, \ldots, \llparenthesis_m\}$, $F = \{q_2\}$, and δ is defined as follows, where $1 \leq i \leq m$, $Z \in \{\Gamma \setminus \{\lambda\}\}$:*

(R1) $\delta(q_0, Z, \epsilon) = (q_0, Z, +1)$,
(R2) $\delta(q_0, \lambda, \llparenthesis_i) = (q_1, \llparenthesis_i, -1)$,
(R3) $\delta(q_0, \lambda, \rrparenthesis_i) = (\llparenthesis_i, \#, -1)$,
(R4) $\delta(q_1, Z, \epsilon) = (q_1, Z, -1)$,
(R5) $\delta(\llparenthesis_i, \#, \epsilon) = (\llparenthesis_i, \#, -1)$,
(R6) $\delta(\llparenthesis_i, \llparenthesis_i, \epsilon) = (q_2, \#, -1)$,
(R7) $\delta(q_2, \#, \epsilon) = (q_2, \#, -1)$,
(R8) $\delta(q_2, \llparenthesis_i, \epsilon) = (q_1, \llparenthesis_i, -1)$,
(R9) $\delta(q_1, \$, \epsilon) = (q_0, \$, +1)$,
(R10) $\delta(q_2, \$, \epsilon) = (q_0, \$, +1)$.

In general, R1 indicates a left-to-right sweep. R2-R3 read an input and start the right-to-left sweep. R4-R8 represent the right-to-left sweep, and R9-R10 the end of a right-to-left sweep. We also give a detailed description of each rule. In the initial configuration, the machine M is in q_0 and the read-write head points to the start marker \$. The head then moves to the leftmost blank marker λ without altering either the state or work tape (R1). When reading an open parenthesis $\llparenthesis_i$ from the input head, M writes it down on the work tape immediately and enters a state q_1 indicating there is additional open parentheses on the work tape (R2). Then the right-to-left sweep begins. During the sweep, M remains in q_1 without altering any other work tape symbols (R4). If it hits the start marker \$, M begins a new left-to-right sweep (R9). When reading a close parenthesis $\rrparenthesis_i$ from the input head, M writes a symbol $\#$ down on the work tape and enters a state $\llparenthesis_i$ (R3). During the right-to-left sweep, it applies the cancellation rule by rewriting the matched parenthesis $\llparenthesis_i$ with $\#$ on the work tape, and enters q_2 indicating the matching (R6). The state q_2 transits to q_1 if M sees any open parentheses on the work tape (R8). When encountering *any # symbol on the work tape, the DSTM state remains unchanged (R5 and R7). Finally, M starts a left-to-right sweep to read the input* w.r.t. *R9 and R10.*

The language $\mathbb{D}_{mn}$ is the intersection of two Dyck languages D_m and D_n. For brevity, let D_m generate the matched pairs of $\{\llparenthesis_1, \rrparenthesis_1, \ldots, \llparenthesis_m, \rrparenthesis_m\}$ and D_n the pairs of $\{\llbracket_1, \rrbracket_1, \ldots, \llbracket_n, \rrbracket_n\}$. We extend the DSTM in Example 1 to handle the intersection of D_n. We assume the availability of the DSTM $M_m = (Q_m, \Sigma_m, \Gamma_m, \delta_m, q_{0_m}, F_m)$ for D_m in Example 1.

The resulting DSTM M for $\mathbb{D}_{mn}$. We construct a DSTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, such that $\Sigma = \Sigma_m \cup \{\llbracket_1, \rrbracket_1, \ldots, \llbracket_n, \rrbracket_n\}$, $\Gamma = \Gamma_m \cup \{\llbracket_1, \ldots, \llbracket_n\}$, $q_0 = q_{0_m}$ and $F = F_m$. The set of states Q is depicted as the set of nodes in Figure 3(b).

Next, we discuss the extension to state transitions δ. The transitions of the right-to-left sweep need nontrivial extensions. Figure 3(b) gives a more comprehensive representation of these transitions. Each node denotes a state and each directed edge ($q_1 \xrightarrow{Z_1} q_2$) a transition corresponding to a DSTM move $\delta(q_1, Z_1, \epsilon) = (q_2, Z_2, -1)$, where $Z_2 \in \{\Gamma \setminus \{\lambda\}\}$. Specifically, nodes q_1, q_2, $\llparenthesis_i$ and $\llbracket_i$ correspond to the states of handling the Dyck languages in Example 1, and nodes $q_1\llparenthesis_i$ and $q_1\llbracket_i$ correspond to handle the intersection. Let us consider processing an open parenthesis $\llparenthesis_i$ of D_m. As discussed above, the DSTM enters state q_1. During the sweep, it may encounter three kinds of work tape symbols $\llparenthesis$, $\llbracket$ and #, respectively. Due to R4 in Example 1, the state q_1 remains unchanged. For processing a close parenthesis $\rrparenthesis_i$, the DSTM first enters state $\llparenthesis_i$. During the sweep, it handles work tape symbols $\llparenthesis_i$ and # as R5 and R6 in Example 1. When encountering the brackets $\llbracket$ in D_n, it transits to a new state $q_1\llparenthesis_i$. The role of $q_1\llparenthesis_i$ is twofold. First, it acts just the same as state $\llparenthesis_i$ respecting the fact that the DSTM is on a move to match a $\llparenthesis_i$ symbol. Second, it is also similar to state q_1 such that it has already encountered additional brackets, and thus shall never transit to the matched state q_2. Furthermore, when state q_2 encounters additional parentheses or brackets or state $q_1\llparenthesis_i$ encounters a matched $\llparenthesis_i$, the DSTM enters q_1. The processing of brackets $\llbracket_i$ and $\rrbracket_i$ of D_n is similar to D_m. In particular, we introduce the addition state $q_1\llbracket_i$ to handle the language intersection with D_m. At the end of a right-to-left sweep, the DSTM starts a left-to-right sweep to read a new input symbol, which is the same as rules R1, R9 and R10 in Example 1.

LEMMA 1. *The DSTM M correctly recognizes $\mathbb{D}_{mn}$.*

Proof. $\mathbb{D}_{mn}$ is essentially the intersection of two Dyck languages D_m and D_n. Let Q denote "the DSTM M recognizes the input string s and reaches the accepting state q_2". We construct the proof by showing $Q \implies s \in \mathbb{D}_{mn}$ and $s \in \mathbb{D}_{mn} \implies Q$.

- $s \in \mathbb{D}_{mn} \implies Q$.
  We first consider the pairs of $\llparenthesis_i$ and $\rrparenthesis_i$ in D_m. M writes every $\llparenthesis_i$ on the work tape. When reading a $\rrparenthesis_i$, it writes # and begins a right-to-left sweep. During the sweep, since D_m is a Dyck language, every $\rrparenthesis_i$ should match a $\llparenthesis_i$. Before the matching, $\rrparenthesis_i$ cannot encounter any $\llparenthesis_j$, where $i \neq j$. Reading symbol # from the work tape does not change states in M, therefore, we can safely ignore them while discussing state transitions. Consider an actual right-to-left sweep. The $\rrparenthesis_i$ only encounters $\llparenthesis_i$ or $\llbracket$ (if any) on the work tape. There are two cases. Case 1: $\rrparenthesis_i$ encounters $\llparenthesis_i$ first. According to Figure 3(b), the state is q_2 with rewriting # on the work tape. If there are outstanding $\llbracket$s, DSTM transits to q_1 and starts a left-to-right sweep to read additional $\rrbracket$s. Case 2: $\rrparenthesis_i$ encounters $\llbracket$ first. The state transits to $q_1\llparenthesis_i$. It then matches the $\llparenthesis_i$, rewrites # and begins a new sweep to read $\rrbracket$s.
  Therefore, every $\llparenthesis_i$ will be rewritten as #. Due to the symmetry in $\mathbb{D}_{mn}$, every $\llbracket$ will be rewritten as # as well. Finally, for the last symbol, *i.e.*, either $\rrparenthesis$ or $\rrbracket$, in the input string, the work tape contains only # and the corresponding $\llparenthesis$ or $\llbracket$ to match. The final state is q_2 according to Figure 3(b).
- $Q \implies s \in \mathbb{D}_{mn}$.
  We use proof by contradiction. Due to the property discussed in Example 1, a string in $\mathbb{D}_{mn}$ finally reduces to ϵ by repeatedly applying the cancellation rule. Moreover, due to the DSTM construction, each time it applies the cancellation rule, it leaves a pair of #s on the work tape.
  Let us now suppose the DSTM is in q_2 but the input string $s \notin \mathbb{D}_{mn}$. There are four possibilities: s reduces to a string contains at least one $\llparenthesis_i$, $\rrparenthesis_i$, $\llbracket_i$ or $\rrbracket_i$. We only need to consider the parenthesis case since the bracket case is symmetric. We first consider the $\rrparenthesis_i$ case. The initial state is $\llparenthesis_i$. Since the symbol $\rrparenthesis_i$ is outstanding, during the right-to-left sweep, the DSTM M can never encounter a $\llparenthesis_i$ symbol on the work tape. Therefore, when reaching the start marker, the state can only be $q_1\llparenthesis_i$ and q_1 according to Figure 3(b). As a result, DSTM will not start a new left-to-right sweep. Then we consider the $\llparenthesis_i$ case. Again, since it is outstanding, during the right-to-left sweep, the DSTM can never apply rule (10) and (12) to rewrite the symbol. As a result, after encountering this outstanding $\llparenthesis_i$ on the work tape, the only possible states to reach are $q_1\llbracket_i$ and q_1 according to Figure 3(b). Consequently, neither of the four cases can reach state q_2, which is a contradiction with our hypothesis. □

4.2 Constructing $K_{D_{mn}}$

We proceed to discuss the conversion from a DSTM to $K_{D_{mn}}$. The intuition of the construction is based on the isomorphism between DSTMs and HTA [11]. It is worth noting that the construction from a DSTM to an HTA in the work of Ibarra and Kim [11] is flawed since the construction does not guarantee a valid start of left-to-right (*i.e.*, end of right-to-left) sweep. We discuss the issue in their work and provide a correction in Appendix A.[4] The GWTA employs the gray nodes to determine such sweeps.

Given a DSTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, we construct an equivalent GWTA $K = (\Sigma, L, Q_l, I_l, \delta_l, F')$, where $l \in \{g, w\}$. For white nodes, each state in $Q_w \in K$ is a pair of a state and a tape symbol in M, *i.e.*, $Q_w = \{\widetilde{q \diamond Z} \mid q \in Q, Z \in \Gamma\}$. Let Q_{eor} be the set of states corresponding to the end of any right-to-left sweep in M, *i.e.*, $Q_{eor} = \{q \mid \delta(q, \$, \epsilon) = (q_0, \$, +1)\}$. For gray nodes, their state set $Q_g \in K$ is a subset of Q_w, *i.e.*, $Q_g = \{\widetilde{q \diamond Z} \mid q \in Q_{eor}, Z \in \Gamma\}$. The accepting state $F' \subseteq Q_g$ is constructed as $F' = \{\widetilde{q \diamond Z} \mid q \in F, Z \in \Gamma\}$. We define the first element in pair $\widetilde{q \diamond Z}$ as a *q-position*, and the second as a *Z-position*.

Among all transitions in M, the transitions corresponding to a right-to-left sweep are used for constructing an equivalent K [11]. The transitions $I_l, \delta_l \in K$ are in the following forms, where $a \in \Sigma$, $q, q_1, q_2 \in Q$ and $Z, Z_1, Z_2 \in \Gamma$ of the DSTM M:

$$\begin{aligned} I_l(a) &= \widetilde{q \diamond Z} && \text{if } \delta(q_0, \lambda, a) = (q, Z, -1), \\ \delta_l(\widetilde{q_1 \diamond Z_1}, \widetilde{q_2 \diamond Z_2}) &= \widetilde{q \diamond Z} && \text{if } \delta(q_2, Z_1, \epsilon) = (q, Z, -1). \end{aligned}$$

Following the GWTA representation, we have $\delta_w : Q_w \times Q_w \to Q_w$ and $\delta_g : Q_g \times Q_w \to Q_g$. We define these transitions as the *LCL rules*. The LCL rules consist of the *I_l-rules* in the form $\widetilde{q \diamond Z} \overset{I}{:=} a$ and *δ_l-rules* in the form $\widetilde{q \diamond Z} \overset{\delta}{:=} \widetilde{q_1 \diamond Z_1}\widetilde{q_2 \diamond Z_2}$, respectively, according to the initial states I_l and state transitions δ_l in K.

All δ_l-rules are in the form $\widetilde{q \diamond Z} \overset{\delta}{:=} \delta_l(\widetilde{q_1 \diamond Z_1}, \widetilde{q_2 \diamond Z_2})$, which contain three q- and Z-positions, respectively. In contrast, each of these rules is generated by a single DSTM transition $\delta(q_2, Z_1, \epsilon) =$

[4] Based on our correction, it is theoretically possible to construct an HTA for the graph reachability algorithm. However, the resulting HTA may have many unnecessary states.

	LCL rules				Corresponding DSTM transitions
(1)	$\widetilde{q_1\diamond\llparenthesis_i}$	$\overset{I}{:=}$	$\llparenthesis_i$		$\delta(q_0, \lambda, \llparenthesis_i) = (q_1, \llparenthesis_i, -1)$
(2)	$\widetilde{\llparenthesis_i\diamond\#}$	$\overset{I}{:=}$	$\rrparenthesis_i$		$\delta(q_0, \lambda, \rrparenthesis_i) = (\llparenthesis_i, \#, -1)$
(3)	$\widetilde{q_1\diamond\llbracket_i}$	$\overset{I}{:=}$	$\llbracket_i$		$\delta(q_0, \lambda, \llbracket_i) = (q_1, \llbracket_i, -1)$
(4)	$\widetilde{\llbracket_i\diamond\#}$	$\overset{I}{:=}$	$\rrbracket_i$		$\delta(q_0, \lambda, \rrbracket_i) = (\llbracket_i, \#, -1)$
(5)	$\widetilde{q_1\diamond Z_{pb}}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond Z_{pb}}$	$\widetilde{q_1\diamond\text{␣}}$	$\delta(q_1, Z_{pb}, \epsilon) = (q_1, Z_{pb}, -1)$
(6)	$\widetilde{q_2\diamond\#}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\#}$	$\widetilde{q_2\diamond\text{␣}}$	$\delta(q_2, \#, \epsilon) = (q_2, \#, -1)$
(7)	$\widetilde{q_1\diamond Z_o}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond Z_o}$	$\widetilde{q_2\diamond\text{␣}}$	$\delta(q_2, Z_o, \epsilon) = (q_1, Z_o, -1)$
(8)	$\widetilde{\llparenthesis_i\diamond\#}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\#}$	$\widetilde{\llparenthesis_i\diamond\text{␣}}$	$\delta(\llparenthesis_i, \#, \epsilon) = (\llparenthesis_i, \#, -1)$
(9)	$\widetilde{q_1\llparenthesis_i\diamond\llbracket}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\llbracket}$	$\widetilde{\llparenthesis_i\diamond\text{␣}}$	$\delta(\llparenthesis_i, \llbracket, \epsilon) = (q_1\llparenthesis_i, \llbracket, -1)$
(10)	$\widetilde{q_2\diamond\#}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\llparenthesis_i}$	$\widetilde{\llparenthesis_i\diamond\text{␣}}$	$\delta(\llparenthesis_i, \llparenthesis_i, \epsilon) = (q_2, \#, -1)$
(11)	$\widetilde{q_1\llparenthesis_i\diamond Z_b}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond Z_b}$	$\widetilde{q_1\llparenthesis_i\diamond\text{␣}}$	$\delta(q_1\llparenthesis_i, Z_b, \epsilon) = (q_1\llparenthesis_i, Z_b, -1)$
(12)	$\widetilde{q_1\diamond\#}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\llparenthesis_i}$	$\widetilde{q_1\llparenthesis_i\diamond\text{␣}}$	$\delta(q_1\llparenthesis_i, \llparenthesis_i, \epsilon) = (q_1, \#, -1)$
(13)	$\widetilde{\llbracket_i\diamond\#}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\#}$	$\widetilde{\llbracket_i\diamond\text{␣}}$	$\delta(\llbracket_i, \#, \epsilon) = (\llbracket_i, \#, -1)$
(14)	$\widetilde{q_1\llbracket_i\diamond\llparenthesis}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\llparenthesis}$	$\widetilde{\llbracket_i\diamond\text{␣}}$	$\delta(\llbracket_i, \llparenthesis, \epsilon) = (q_1\llbracket_i, \llparenthesis, -1)$
(15)	$\widetilde{q_2\diamond\#}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\llbracket_i}$	$\widetilde{\llbracket_i\diamond\text{␣}}$	$\delta(\llbracket_i, \llbracket_i, \epsilon) = (q_2, \#, -1)$
(16)	$\widetilde{q_1\llbracket_i\diamond Z_p}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond Z_p}$	$\widetilde{q_1\llbracket_i\diamond\text{␣}}$	$\delta(q_1\llbracket_i, Z_p, \epsilon) = (q_1\llbracket_i, Z_p, -1)$
(17)	$\widetilde{q_1\diamond\#}$	$\overset{\delta}{:=}$	$\widetilde{\text{␣}\diamond\llbracket_i}$	$\widetilde{q_1\llbracket_i\diamond\text{␣}}$	$\delta(q_1\llbracket_i, \llbracket_i, \epsilon) = (q_1, \#, -1)$

Table 2. LCL rules for context-sensitive data-dependence analysis, where $\llparenthesis$ denotes the set of open parentheses and $\llbracket$ the set of open brackets. For brevity, we let $Z_o = \{\llparenthesis, \llbracket\}$, $Z_p = \{\#, \llparenthesis\}$, $Z_b = \{\#, \llbracket\}$ and $Z_{pb} = Z_p \cup Z_b$. Moreover, the accepting state of the equivalent trellis automaton is $\widetilde{q_2\diamond\#}$.

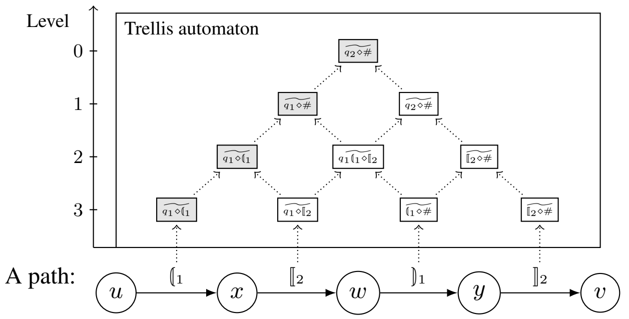

Level
Trellis automaton
0
1
2
3
$\widetilde{q_2\diamond\#}$
$\widetilde{q_1\diamond\#}$
$\widetilde{q_2\diamond\#}$
$\widetilde{q_1\diamond\llparenthesis_1}$
$\widetilde{q_1\llparenthesis_1\diamond\llbracket_2}$
$\widetilde{\llbracket_2\diamond\#}$
$\widetilde{q_1\diamond\llparenthesis_1}$
$\widetilde{q_1\diamond\llbracket_2}$
$\widetilde{\llparenthesis_1\diamond\#}$
$\widetilde{\llbracket_2\diamond\#}$
A path:
u
$\llparenthesis_1$
x
$\llbracket_2$
w
$\rrparenthesis_1$
y
$\rrbracket_2$
v

Figure 4. Computing LCL-reachability for $\mathbb{D}_{mn}$.

$(q, Z, -1)$, which contains only two q and Z symbols, respectively. Therefore, in each δ_l-rule, one q-position and one Z-position are unnecessary. They can be safely replaced by a ␣ symbol indicating "don't care". As a result, the δ_l-rules can be rewritten as $\widetilde{q\diamond Z} \overset{\delta}{:=} \delta_l(\widetilde{\text{␣}\diamond Z_1}, \widetilde{q_2\diamond\text{␣}})$.

The resulting GWTA $K_{D_{mn}}$. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be the DSTM for $\mathbb{D}_{mn}$ in Section 4.1. In the equivalent GWTA $K_{D_{mn}} = (\Sigma, L, Q_l, I_l, \delta_l, F')$, we have $L = \{g, w\}$ and $Q_{eor} = \{q_1, q_2\}$. Q_w, Q_g and F' can be constructed based on Q_{eor} as described above. The transitions $I_g : \Sigma \to Q_g$ and $\delta_g : Q_g \times Q_w \to Q_g$ can be built by restricting Q_w to $Q_g \subseteq Q_w$ in I_w and δ_w, respectively. We give the LCL rules for I_w and δ_w in Table 2. The right column in Table 2 shows the DSTM transitions for recognizing $\mathbb{D}_{mn}$. We generate the equivalent LCL rules in the left column according to the transformations discussed above. Specifically, rules (1)-(4) are I_l-rules and the remaining rules δ_l-rules. The correctness on the construction is stated as follows. We defer the proof to Appendix A.3 due to the space constraints.

THEOREM 3. *Given a DSTM, we can effectively construct an equivalent GWTA.*

EXAMPLE 2 (Recognizing a $\mathbb{D}_{mn}$ string). *Consider a path between nodes u and v in Figure 4. We discuss the steps for recognizing the path string using LCL rules in Table 2. The corresponding path string is "$\llparenthesis_1\llbracket_2\rrparenthesis_1\rrbracket_2$". The recognition trellis automaton is given above. We give the rule number in Table 2 to derive the corresponding summary edges. The edge labels from left to right are processed by the I_l-rules (1), (3), (2) and (4), respectively. In this example, for each gray node summary, we have its q position $q \in Q_{eor} = \{q_1, q_2\}$. The summaries generated on Level(2) are derived using δ_l-rules (5), (9), (13), respectively. The two summaries on Level(1) are generated using δ_l-rules (12) and (15), respectively. Finally, the topmost summary is generated using δ_l-rule (6). Since the topmost summary represents an accepting state, the path string is in $\mathbb{D}_{mn}$.*

5. Algorithm for Context-Sensitive Data-Dependence Analysis

This section discusses the LCL-reachability algorithm for context-sensitive data-dependence analysis. We first sketch a baseline LCL-reachability algorithm that is based on HTAs. The algorithm is general, and works for arbitrary LCLs. Then we restrict the automaton to a GWTA and present our main context-sensitive data-dependence analysis algorithm.

We give the baseline LCL-reachability algorithm to explain the key procedures in our main algorithm. The baseline algorithm is also a fundamental start-point to establish the complexity and correctness results of the main algorithm. Section 5.1 gives the baseline algorithm. Section 5.2 describes the extension for handling a GWTA. Section 5.3 provides the main algorithm, and Section 5.4 conducts the analysis.

5.1 Baseline LCL-Reachability Algorithm

The baseline LCL-reachability algorithm is based on HTAs, which is suitable for arbitrary LCLs. Due to Theorem 1, it also works for the context-sensitive data-dependence analysis problem. For simplicity, in this section, we assume an HTA that handles only white nodes in the GWTA described in Section 4.2.

The baseline algorithm is a dynamic-programming style algorithm. The algorithm maintains a worklist of *summary edges* in the form $(u, \widetilde{q\diamond Z}, v)$, where u and v represent a pair of nodes in the graph and $\widetilde{q\diamond Z}$ a state in the trellis automaton. For each of the worklist items, the processing involves two major steps: (1) *Traversing nodes* — The algorithm needs to determine the proper nodes to propagate the reachability information, and (2) *Generating summaries* — The algorithm also needs to generate the correct reachability information as the new summary edges *w.r.t.* the LCL rules. Those new summaries are then inserted into the worklist. Finally, the algorithm terminates until there are no new summary edges to be added. Next, we discuss the two steps in detail.

5.1.1 Traversing Nodes

Let us consider a path $p = u, x, w, y, v$ shown in Figure 4. The path string "$\llparenthesis_1\llbracket_2\rrparenthesis_1\rrbracket_2$" can be parsed by the GWTA above. In this section, we only consider the white nodes. The trellis representation can also be considered as a parsing tree. Each parsing tree node corresponds to a summary edge of two nodes in the given path. For example, the rightmost summary edges on Level(1) and Level(2) are $(x, \widetilde{q_2\diamond\#}, v)$ and $(w, \widetilde{\llbracket_2\diamond\#}, v)$, respectively. The summary edge on Level(k) is generated by two summary edges on Level($k+1$), for any non-bottom levels. For example, the rightmost summary $(x, \widetilde{q_2\diamond\#}, v)$ on Level(1) is generated by two summaries $(x, \widetilde{q_1\llparenthesis_1\diamond\llbracket_2}, y)$ and

$(w, \widetilde{\llbracket_2 \diamond \#}, v)$ on Level(2). Moreover, each summary edge on Level(k) contains exactly one more terminal than its two predecessors on Level$(k+1)$. For instance, the rightmost summary edge $(x, \widetilde{q_2 \diamond \#}, v)$ has one more "$\rrbracket_2$" and "$\llbracket_2$" than the predecessors $(x, \widetilde{q_1 \llparenthesis_1 \diamond \llbracket_2}, y)$ and $(w, \widetilde{\llbracket_2 \diamond \#}, v)$, respectively. Since the terminals correspond to the labels of the original graph edges, therefore, for generating new summary edges, only those *original graph edges* need to be traversed.

5.1.2 Generating Summaries

In the beginning, all edge labels $\mathcal{L}$ are converted to a summary edge according to I_l-rules $\widetilde{q \diamond Z} \overset{I}{:=} \mathcal{L}$. In a δ_l-rule $\widetilde{q \diamond Z} \overset{\delta}{:=} \delta_l(\widetilde{\textvisiblespace \diamond Z_1}, \widetilde{q_2 \diamond \textvisiblespace})$, we say a summary is a *left term* (L-term) if its q-position is filled with a $\textvisiblespace$ symbol. Similarly, a summary is a *right term* (R-term) if its Z-position is $\textvisiblespace$. Consider the rightmost summary edge $(w, \widetilde{\llbracket_2 \diamond \#}, v)$ on Level(2) in Figure 4, we say this summary is generated by an L-term $(w, \widetilde{\textvisiblespace \diamond \#}, y)$ and a R-term $(y, \widetilde{\llbracket_2 \diamond \textvisiblespace}, v)$. It is immediate that only the Z-position of an L-term and q-position of a R-term are useful for generating new summary edges. In our main algorithm, we use $L(u, v)$ to represent a set of Z-position symbols in an L-term between nodes u and v, and $R(u, v)$ to represent a set of q-position symbols in a R-term. A new summary edge $(u, \widetilde{q \diamond Z}, v)$ is generated if and only if $Z_1 \in L(u, v)$, $q_2 \in R(u, v)$, and there exists a valid δ_l-rule $\widetilde{q \diamond Z} \overset{\delta}{:=} \delta_l(\widetilde{\textvisiblespace \diamond Z_1}, \widetilde{q_2 \diamond \textvisiblespace})$.

LEMMA 2. *Given a directed graph $G = (V, E)$, every summary edge $(u, \widetilde{q \diamond Z}, v)$ is generated by $(u, \widetilde{\textvisiblespace \diamond Z_1}, y)$ and $(x, \widetilde{q_2 \diamond \textvisiblespace}, v)$, where nodes $u, v, x, y \in V$, edges $(y, v), (u, x) \in E$ and $\widetilde{q \diamond Z} \overset{\delta}{:=} \delta_l(\widetilde{\textvisiblespace \diamond Z_1}, \widetilde{q_2 \diamond \textvisiblespace})$.*

EXAMPLE 3 (Generating summaries). *Consider the path in Figure 4, where the path string is discussed in Example 2. We discuss generating the summary edge $(w, \widetilde{\llbracket_2 \diamond \#}, v)$* w.r.t. *the LCL rules on white nodes discussed in Section 4.2. In the beginning, for all nodes i and j in the path, the edge labels $\mathcal{L}(i, j)$ are converted to summary edges $(i, \widetilde{q \diamond Z}, j)$, where $\widetilde{q \diamond Z} \in I_w$. Therefore, we have $(w, \widetilde{\llparenthesis_1 \diamond \#}, y)$ and $(y, \widetilde{\llbracket_2 \diamond \#}, v)$. When processing $(w, \widetilde{\llparenthesis_1 \diamond \#}, y)$, we search all outgoing neighbors of y in the original graph. In our example, it is node v. As a result, we store the L-term $\widetilde{\textvisiblespace \diamond \#}$ in $L(w, v)$. The algorithm then searches for all R-terms $\widetilde{q \diamond \textvisiblespace} \in R(w, v)$ to generate a new summary edge. At this point, $R(w, v) = \emptyset$, so nothing is generated. Similarly, when processing $(y, \widetilde{\llbracket_2 \diamond \#}, v)$, we store the R-term $\widetilde{\llbracket_2 \diamond \textvisiblespace}$ to $R(w, v)$, and search for each L-term in $L(w, v)$. Consequently, we use $\widetilde{\textvisiblespace \diamond \#} \in L(w, v)$ and $\widetilde{\llbracket_2 \diamond \textvisiblespace} \in R(w, v)$ to match an LCL rule. According to Example 2, we have a matched LCL rule $\delta_w(\widetilde{\textvisiblespace \diamond \#}, \widetilde{\llbracket_2 \diamond \textvisiblespace}) = \widetilde{\llbracket_2 \diamond \#}$. Finally, we generate the new summary $(w, \widetilde{\llbracket_2 \diamond \#}, v)$.*

5.1.3 Baseline Algorithm

The baseline LCL-reachability algorithm is given in Algorithm 1. The inputs of the algorithm are an edge-labeled directed graph G and some LCL rules corresponding to an HTA. It computes a set $S(i, j)$ of summary edges $(i, \widetilde{q \diamond Z}, j)$, where $\widetilde{q \diamond Z}$ belongs to the accepting states of the corresponding trellis automaton.

The algorithm shares the same style of the popular worklist-based CFL-reachability algorithms [4, 30]. The sets of outgoing and incoming neighbors of node i are denoted as OUT(i) and IN(i). On line 1, it initializes the worklist with the items of I_l-rules for original graph edges $(i, \mathcal{L}, j) \in G$, *i.e.*, $I(\mathcal{L})$ is an initial state of the trellis automaton. The main algorithm proceeds as follows. For each item popped from the worklist on line 3, it obtains the L-term $\widetilde{\textvisiblespace \diamond Z}$ and R-term $\widetilde{q \diamond \textvisiblespace}$ from the summary edge on line 4. For brevity, we use Z and q to represent them while discussing the algorithms. For each summary edge (i, j), the algorithm generates new summary edges via searching the outgoing (lines 5-12) and incoming (lines 13-20) neighbors of nodes j and i, respectively. Specifically, for each outgoing edge (j, k), it stores the L-term Z in $L(i, k)$ on line 7 for generating a potential new summary edge between nodes i and k. It then searches for each R-term x in $R(i, k)$, and invokes the constant-time procedure FIND-RULE to find an LCL rule with both L-term Z and R-term x on lines 8-9. If the LCL rule is valid and the summary is new (line 10), the summary is inserted to $S(i, k)$ on line 11 and the corresponding edge is pushed to the worklist on line 12. Each incoming edge of node i is handled similarly on lines 13-20. Finally, the algorithm terminates when there are no new summary edges generated in the worklist. The output of the main algorithm is a set of summary edges $S(i, j)$. Any LCL-reachability query between nodes i and j can be answered in constant time by testing whether $\widetilde{q \diamond Z}$ is in $S(i, j)$, where $\widetilde{q \diamond Z}$ corresponds to a final trellis automata state.

Algorithm 1: The baseline LCL-reachability algorithm.

```
Input  : Edge-labeled directed graph $G = (V, E)$, a set of LCL rules;
Output : the set $\{S(i, j) \mid i, j \in V\}$
1  Initialize $W$ with summary edges of $I_l$-rules
2  while $W \neq \emptyset$ do
3      $(i, \widetilde{q \diamond Z}, j) \leftarrow$ SELECT-FROM$(W)$
4      Obtain $q$ and $Z$ from $\widetilde{q \diamond Z}$
5      foreach $k \in$ OUT$(j)$ do
6          if $Z \notin L(i, k)$ then
7              $L(i, k) \leftarrow L(i, k) \cup \{Z\}$
8              foreach $x \in R(i, k)$ do
9                  $\widetilde{q' \diamond Z'} \leftarrow$ FIND-RULE$(Z, x)$
10                 if $\widetilde{q' \diamond Z'} \neq \emptyset$ and $\widetilde{q' \diamond Z'} \notin S(i, k)$ then
11                     $S(i, k) \leftarrow S(i, k) \cup \{\widetilde{q' \diamond Z'}\}$
12                     $W \leftarrow (i, \widetilde{q' \diamond Z'}, k)$
13     foreach $k \in$ IN$(i)$ do
14         if $q \notin R(k, j)$ then
15             $R(k, j) \leftarrow R(k, j) \cup \{q\}$
16             foreach $X \in L(k, j)$ do
17                 $\widetilde{q' \diamond Z'} \leftarrow$ FIND-RULE$(X, q)$
18                 if $\widetilde{q' \diamond Z'} \neq \emptyset$ and $\widetilde{q' \diamond Z'} \notin S(k, j)$ then
19                     $S(k, j) \leftarrow S(k, j) \cup \{\widetilde{q' \diamond Z'}\}$
20                     $W \leftarrow (k, \widetilde{q' \diamond Z'}, j)$
```

5.2 Handling Gray Nodes

When extending Algorithm 1 for handling a GWTA, the key challenge is to handle the gray nodes since a summary edge can be both a gray node summary and a white node summary. For instance, $(x, \widetilde{q_1 \diamond \llbracket_2}, w)$ is a white node summary in Figure 4, and it becomes a gray node summary if we consider the path string of path $p = x, \ldots, v$. According to Section 4.2, Q_g is a subset of Q_w and the valid q-positions in Q_g are depicted as the set $Q_{eor} = \{q_1, q_2\}$. To distinguish gray node summaries from white node summaries, we introduce the concept of *spurious summaries*. Intuitively, any spurious summary edge cannot be a gray node summary. Formally,

DEFINITION 7. *A summary $\widetilde{q \diamond Z}$ is defined as a spurious summary if and only if*

(i) its q-position $q \notin Q_{eor}$ or

(ii) it is generated by a spurious L-term.

For example, the summary $\widetilde{q_1 \llparenthesis_1 \diamond \llbracket_2}$ on Level(2) of string "$\llbracket_2 \rrparenthesis_1$ in Figure 4 is a spurious summary since $q_1 \llparenthesis_1 \notin Q_{eor}$. Moreover, the above summary $\widetilde{q_2 \diamond \#}$ on Level(1) is also a spurious summary, since the left summary $\widetilde{q_1 \llparenthesis_1 \diamond \llbracket_2}$ is spurious.

To cope with the spurious summaries, we introduce new data structures to the LCL-reachability algorithm *w.r.t.* Definition 7. Handling (i) is straightforward since we can simply check if $q \in \{q_1, q_2\}$. For handling (ii), Algorithm 1 needs two extensions. First, we extend the worklist item from $(i, \widetilde{q\diamond Z}, j)$ to $(i, \widetilde{q\diamond Z}, j, good)$, where the flag $good$ is set to 1 if the current summary $\widetilde{q\diamond Z}$ is generated by a non-spurious (*i.e.*, gray node summary) L-term. Moreover, we introduce a new table $L_b(i,j)$ to represent the set of Z-positions of all spurious L-terms.

The basic idea of handling (ii) is as follows. For any summary $(i, \widetilde{q\diamond Z}, j)$ popped from the worklist, it uses the $good$ and its q-position to determine if it is a spurious summary. If it is spurious, we insert Z to $L_b(i,j)$ and otherwise $L(i,j)$. When generating new summaries, if the R-term symbol matches an L-term symbol $Z \in L(i,j)$ it sets $good$ to 1 in the new worklist item. The flag $good$ is set to 0 if it matches a $Z \in L_b(i,j)$. Finally, a spurious summary $\widetilde{q\diamond Z}$ can be changed to a non-spurious summary. We also update the information and change any summaries that depend on $\widetilde{q\diamond Z}$. Specifically, for a changed summary $(i, \widetilde{q\diamond Z}, j)$, we remove Z from $L_b(i,k)$, insert it to $L(i,k)$, and use $Z \in L(i,k)$ to update any summary edge between i and k.

5.3 Context-Sensitive Data-Dependence Analysis Algorithm

Refinement for $\mathbb{D}_{mn}$. In Algorithm 1, given a worklist item $(i, \widetilde{q\diamond Z}, j)$, it only traverses the neighbors k of nodes i and j to generate a new summary, without taking the edge labels (*i.e.*, $\mathcal{L}(j,k)$ and $\mathcal{L}(k,i)$) into consideration. Consequently, it might generate some *infeasible summaries*. For instance, any valid string in $\mathbb{D}_{mn}$ never begins with a close parenthesis, nor ends with an open parenthesis. In our main algorithm, we use a procedure FEASIBILITY() to eliminate those infeasible summaries. The procedure requires three parameters: a summary $\widetilde{q\diamond Z}$, an original graph edge label $\mathcal{L}(i,j)$, and a flag ty indicating the type (*i.e.*, incoming or outgoing) of edge (i,j). It returns a true or false value for the summary feasibility using a set of heuristics. Our reachability algorithm only inserts feasible summaries to the worklist. The current heuristics used for the main algorithm as follows.

- If ty is outgoing and $\mathcal{L}(i,j) \in \{\llparenthesis, \llbracket\}$, q must be q_1;
- If ty is outgoing and $\mathcal{L}(i,j) \in \rrparenthesis$, $q \notin \{\llbracket, q_1\llbracket\}$;
- If ty is outgoing and $\mathcal{L}(i,j) \in \rrbracket$, $q \notin \{\llparenthesis, q_1\llparenthesis\}$;
- If ty is incoming and $\mathcal{L}(i,j) \in \{\rrparenthesis, \rrbracket\}$, Z must be $\#$;
- If ty is incoming and $\mathcal{L}(i,j) \in \llparenthesis$, $Z \notin \llbracket$;
- If ty is incoming and $\mathcal{L}(i,j) \in \llbracket$, $Z \notin \llparenthesis$;

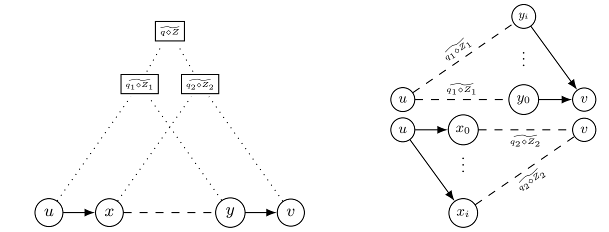
All rules can be interpreted according to Table 2 and state transitions in Figure 3(b). For instance, the first heuristic is obtained due to the fact that the q-position of both I_l-rules (1) and (3) is q_1, and q_1 never transits to other states.

Main algorithm. Our context-sensitive data-dependence analysis algorithm is given in Algorithm 2. To determine spurious summaries, the worklist is extended with an additional flag $good$ and the original $L(i,j)$ is separated into $L(i,j)$ and $L_b(i,j)$, as discussed in Section 5.2. The set $Good(i,j)$ represents the q-positions of any non-spurious summary. For each worklist item $(i, \widetilde{q\diamond Z}, j, good)$, the algorithm traverse its outgoing neighbors on lines 5-30 and incoming neighbors on lines 31-49, respectively.

- *Traversing outgoing edges* (j,k). The Z-position symbol Z can be either an entirely new symbol (line 6) or an old symbol in $L_b(i,k)$ which needs to be updated to $L(i,k)$ (line 20). For new symbols Z, we determine if it is a spurious summary on lines 7-9. Then we search for the q-position symbols $x \in R(i,k)$. If Z and x match a valid LCL rule, and the summary edge is also new (line 12), we insert the feasible summary edge to $S(i,k)$ on lines 13-14. If the current summary $\widetilde{q\diamond Z}$ is non-spurious, we insert an item $(i, \widetilde{q'\diamond Z'}, 1)$ to W and record it in $Good$ on lines 17-18. Otherwise, we insert an item $(i, \widetilde{q'\diamond Z'}, j, 0)$ to W on line 19. If the symbol Z exists in $L_b(i,k)$ but current summary edge $(i, \widetilde{q\diamond Z}, j)$ is not spurious (line 20), we adjust Z to $L(i,k)$ on lines 21-22. The remain steps to process the new summary on lines 24-30 is similar to the non-spurious case on lines 11-19.

Algorithm 2: The LCL-reachability algorithm for context-sensitive data-dependence analysis.

```
Input  : Graph $G = (V, E)$ and LCL rules for $\mathbb{D}_{mn}$
Output : the sets $\{S(i,j), Good(i,j) \mid i, j \in V\}$
1  Initialize $W$ with summary edges of $I_l$-rules
2  while $W \neq \emptyset$ do
3      $(i, \widetilde{q \diamond Z}, j, good) \leftarrow$ SELECT-FROM($W$)
4      Obtain $q$ and $Z$ from $\widetilde{q \diamond Z}$
5      foreach $k \in$ OUT($j$) do
6          if $Z \notin L(i,k)$ and $Z \notin L_b(i,k)$ then
7              if good and $q \in \{q_1, q_2\}$ then
8                  $L(i,k) \leftarrow L(i,k) \cup \{Z\}$
9              else $L_b(i,k) \leftarrow L_b(i,k) \cup \{Z\}$
10             foreach $x \in R(i,k)$ do
11                 $\widetilde{q'\diamond Z'} \leftarrow$ FIND-RULE($Z, x$)
12                 if $\widetilde{q'\diamond Z'} \neq \emptyset$ and $\widetilde{q'\diamond Z'} \notin S(i,k)$ then
13                     if FEASIBLE($\widetilde{q'\diamond Z'}, \mathcal{L}(j,k), out$) then
14                         $S(i,k) \leftarrow S(i,k) \cup \{\widetilde{q'\diamond Z'}\}$
15                     if $Z \in L(i,k)$ then
16                         if $q' \in \{q_1, q_2\}$ then
17                             $Good(i,k) \leftarrow Good(i,k) \cup \{q'\}$
18                         $W \leftarrow (i, \widetilde{q'\diamond Z'}, k, 1)$
19                     else $W \leftarrow (i, \widetilde{q'\diamond Z'}, k, 0)$
20         if $Z \in L_b(i,k)$ and good and $q \in \{q_1, q_2\}$ then
21             remove $Z$ from $L_b(i,k)$
22             $L(i,k) \leftarrow L(i,k) \cup \{Z\}$
23             foreach $x \in R(i,k)$ do
24                 $\widetilde{q'\diamond Z'} \leftarrow$ FIND-RULE($Z, x$)
25                 if $\widetilde{q'\diamond Z'} \neq \emptyset$ then
26                     if FEASIBLE($\widetilde{q'\diamond Z'}, \mathcal{L}(j,k), out$) then
27                         $S(i,k) \leftarrow S(i,k) \cup \{\widetilde{q'\diamond Z'}\}$
28                     if $q' \in \{q_1, q_2\}$ then
29                         $Good(i,k) \leftarrow Good(i,k) \cup \{q'\}$
30                     $W \leftarrow (i, \widetilde{q'\diamond Z'}, k, 1)$
31     foreach $k \in$ IN($i$) do
32         if $q \notin R(k,j)$ then
33             $R(k,j) \leftarrow R(k,j) \cup \{q\}$
34             foreach $X \in L_b(k,j)$ do
35                 $\widetilde{q'\diamond Z'} \leftarrow$ FIND-RULE($X, q$)
36                 if $\widetilde{q'\diamond Z'} \neq \emptyset$ and $\widetilde{q'\diamond Z'} \notin S(k,j)$ then
37                     if FEASIBLE($\widetilde{q'\diamond Z'}, \mathcal{L}(k,i), in$) then
38                         $S(k,j) \leftarrow S(k,j) \cup \{\widetilde{q'\diamond Z'}\}$
39                     $W \leftarrow (k, \widetilde{q'\diamond Z'}, j, 0)$
40             foreach $X \in L(k,j)$ do
41                 $\widetilde{q'\diamond Z'} \leftarrow$ FIND-RULE($X, q$)
42                 if $\widetilde{q'\diamond Z'} \neq \emptyset$ then
43                     if $q' \in \{q_1, q_2\}$ and $q' \notin Good(k,j)$ then
44                         $Good(k,j) \leftarrow Good(k,j) \cup \{q'\}$
45                         $W \leftarrow (k, \widetilde{q'\diamond Z'}, j, 1)$
46                     else if $\widetilde{q'\diamond Z'} \notin S(k,j)$ then
47                         $W \leftarrow (k, \widetilde{q'\diamond Z'}, j, 1)$
48                     if FEASIBLE($\widetilde{q'\diamond Z'}, \mathcal{L}(k,i), in$) then
49                         $S(k,j) \leftarrow S(k,j) \cup \{\widetilde{q'\diamond Z'}\}$
```


$\widetilde{q\diamond Z}$
$\widetilde{q_1\diamond Z_1}$
$\widetilde{q_2\diamond Z_2}$
u
x
y
v
y_i
y_0
x_0
x_i

(a) A trellis automaton representation. (b) A path representation.

Figure 5. Computing new summaries.

- *Traversing incoming edges* (k, i). For an incoming edge (k, i), the algorithm needs to search for the Z-positions X in both $L(k, j)$ and $L_b(k, j)$ to generate a new summary edge. The major steps of processing $Z \in L_b(k, j)$ on lines 35-39 is finding a matched LCL rule (line 35), inserting the feasible summary edge to $S(k, j)$ (line 38) and the spurious summary edge to W (line 39). In the processing of $Z \in L(k, j)$ on line 41-49, if the summary edge is non-spurious, we update the set *Good* and insert the corresponding item to W (lines 43-45). Otherwise, we insert the worklist item only if it is a new summary (lines 46-47). Finally, we insert the feasible summary to $S(k, j)$ on lines 48-49.

Upon termination, the table $S(i, j)$ contains all summary edges and $Good(i, j)$ the q-positions of all non-spurious summary edges. Given two nodes u and v, any reachability queries can be answered in $O(1)$ time by checking if $\widetilde{q_2\diamond\#} \in S(u, v)$ and $q_2 \in Good(u, v)$.

5.4 Algorithm Soundness and Complexity Analysis

In Section 5.4.1, we discuss the baseline LCL-reachability algorithm in Algorithm 1 that only generates summaries according to Section 5.1.2, *i.e.*, without taking infeasible or spurious summaries into consideration. Based on the discussion, we then give the analysis for Algorithm 2 in Section 5.4.2.

5.4.1 Analysis for Algorithm 1

Let ϕ denote the exact solution and ϕ_{LCL} the solution obtained by our algorithm. We establish the soundness by showing that $(u, \widetilde{q_f\diamond Z_f}, v) \in \phi \implies (u, \widetilde{q_f\diamond Z_f}, v) \in \phi_{LCL}$ for all u and v, where $\widetilde{q_f\diamond Z_f}$ represents an accepting state. Initially, our algorithm correctly processes all edge labels according to the I_l-rules. We consider an arbitrary summary edge $(u, \widetilde{q\diamond Z}, v)$ in Figure 5. Figure 5(a) gives the trellis automaton representation of $(u, \widetilde{q\diamond Z}, v)$. Without loss of generality, we suppose the summary is generated via a δ_l-rule $\widetilde{q\diamond Z} \overset{\delta}{:=} \widetilde{\sqcup\diamond Z_1}\,\widetilde{q_2\diamond\sqcup}$. Figure 5(b) shows the corresponding path considered in our algorithm. The Z-position $Z_1 \in L(u, v)$ is generated by any summary edge $(u, \widetilde{q_1\diamond Z_1}, y')$, where (y', v) is a directed edge in the input graph. Our algorithm enumerates all such summary edges $(u, \widetilde{q_1\diamond Z_1}, y')$ for Z_1. Let $X = \{x_0, \ldots, x_i\}$ and $Y = \{y_0, \ldots, y_i\}$. It is straightforward that $y \in Y$. Similarly, $q_2 \in R(u, v)$ is generated by enumerating all $x' \in X$. Therefore, $x \in X$ and the summary $\widetilde{q\diamond Z}$ is inserted to $S(u, v)$. According to Lemma 2, every summary edge can be generated in a similar manner. As a result, the algorithm eventually inserts the summary $\widetilde{q_f\diamond Z_f}$ to $S(u, v)$.

Next we discuss the time complexity of Algorithm 1. Let M be the equivalent DSTM corresponding to the input LCL rules, where q denotes the number of states and Z the size of work tape alphabet in M. Given a graph with n nodes and m edges, there are $O(qZ)$ summary edges for each node pair. On line 5, the algorithm traverses the outgoing neighbors k of each summary edge $(i, \widetilde{q\diamond Z}, j)$. Moreover, for each k, the algorithm also searches for the q-positions $x \in R(i, k)$. Due to the construction, $|x|$ is equal to q. Let Δ_j denote the out-degree of node j. The total number of steps involved for each summary edge is $q \cdot \Sigma_{(i,j)}\Delta_j = q \cdot \Sigma_i(\Sigma_j \Delta_j) = qnm$. Similarly, the total number of steps of the loop on lines 13-20 is Znm. As a result, the time complexity of processing all summary edges in Algorithm 1 is $O((qZ)(q + Z)mn)$. Combining the analysis, we state the following theorem:

THEOREM 4. *Given a directed graph with n nodes and m edges, an HTA with the equivalent DSTM M, Algorithm 1 computes a sound approximation of LCL-reachability in $O(|\tilde{M}|mn)$ time with $O(n^2)$ space, where $|\tilde{M}| = O(q^2Z + qZ^2)$, q and Z are the numbers of states and work tape symbols in M. Any LCL-reachability query can be answered in $O(1)$ time.*

5.4.2 Analysis for Algorithm 2

This section discusses the soundness and complexity of our context-sensitive data-dependence analysis algorithm. Due to Lemma 1, Algorithm 2 is based on a correct set of LCL rules.

Let ϕ_b and ϕ_s denote the sets of infeasible and spurious summaries, and ϕ_{LCL}, ϕ_{csdd} the sets of summaries obtained by Algorithms 1 and 2. Again, the set ϕ denotes the exact solution. Based on Theorem 4, our basic idea to establish the soundness is to show $\phi_{csdd} = \phi_{LCL} \setminus \{\phi_b \cup \phi_s\}$ and $e \notin \phi$ for all $e \in \phi_b \cup \phi_s$. The $e \notin \phi$ part is immediate due to the discussions in Sections 5.2 and 5.3. We also note that, unlike infeasible summaries, Algorithm 2 indeed inserts spurious summaries to the output set S.

According to Lemma 2, both Algorithms 1 and 2 generate new summary edges $S(i, j)$ using L-terms $L(i, j)$ and R-terms $R(i, j)$. One of the major structural differences between Algorithms 1 and 2 is that Algorithm 1 only traverses $L(i, j)$ and $R(i, j)$ to generate the summary edge $e \in S$. We denote the $L(i, j)$ in Algorithm 1 as $L_1(i, j)$. However, Algorithm 2 uses four loops to traverse $L_b(i, j)$, $L(i, j)$ and $R(i, j)$. Due to lines 7-9 and lines 21-22, we have $L_1(i, j) = L_b(i, j) \cup L(i, j)$. In other words, the graph traversals in the two algorithms are identical. Let S_1 and S_2 be the output sets S of Algorithms 1 and 2. For a worklist summary edge in Algorithm 2, the traversal inserts every element in S_1 except the infeasible summaries. Consequently, we have $S_1 = S_2 \cup \phi_b$, *i.e.*, $S_2 = \phi_{LCL} \setminus \phi_b$.

On the other hand, Algorithm 2 uses the set *Good* to remember all non-spurious summaries. On lines 16, 28 and 43, Algorithm 2 determines the set of non-spurious summaries if and only if $q' \notin \{q_1, q_2\}$ and the Z-position of L-term is in $L(i, j)$, which is precisely $\overline{\phi_s}$. Finally, according to the query answering discussed in Section 5.3, we have $\phi_{csdd} = S_2 \cap \overline{\phi_s}$. Putting everything together, we have,

$$\phi_{csdd} = S_2 \cap \overline{\phi_s} = \{\phi_{LCL} \setminus \phi_b\} \setminus \phi_s = \phi_{LCL} \setminus \{\phi_b \cup \phi_s\}.$$

The analysis on time complexity is also based on Theorem 4. As mentioned earlier, the graph traversal parts of both Algorithms 1 and 2 are identical. We only need to bound the number of summary edges inserted to W. We first note that, there are six sites inserting a summary edge to W in Algorithm 2. Among them, on line 30, an item is inserted only if $Z \in L_b(i, k)$. Z is then removed from L_b and is never added back again. Similarly, on line 45, an item is inserted to W only if $q' \notin Good(k, j)$, and q' is added to $Good(k, j)$ immediately. For the other four sites, an item is inserted only if $\widetilde{q'\diamond Z'} \notin S$, and it is immediately added to S. As a result, a summary edge $(i, \widetilde{q'\diamond Z'}, j)$ can be added into W for at most three times.[5] Let x and X denote the number of R-terms and L-terms to generate a summary. According to the discussion of Theorem 4, the total number of steps for processing each summary

[5] A tighter analysis can bound the number to 2.

is $3(|x| + |X|)mn = O((|x| + |X|)mn)$. On the other hand, the number of summaries is bounded by the number of LCL rules for $\mathbb{D}_{m'n'}$. In Table 2, the number of all rules is dominated by rules (9), (11), (14) and (16) with $O(m'n')$ different trellis automaton states. Therefore, the total number of processing all summary edges is $O((m'n')(|x| + |X|)mn)$. Moreover, according to the GWTA $K_{D_{mn}}$ in Section 4.2, we have $|x| = 2 + 2m' + 2n'$ and $|X| = 1 + m' + n'$. Finally, we have the following theorem.

THEOREM 5. *Given a directed graph with n nodes and m edges, an interleaved matched-parenthesis language $\mathbb{D}_{m'n'}$, Algorithm 2 computes a sound approximation in $O(|\tilde{\mathbb{D}}|mn)$ time with $O(n^2)$ space for context-sensitive data-dependence analysis, where $|\tilde{\mathbb{D}}| = O((m' + n')m'n')$, m' and n' denote the sizes of the two Dyck languages in $\mathbb{D}_{m'n'}$. Any query can be answered in $O(1)$ time.*

6. Evaluation

To demonstrate the utility of the proposed LCL-reachability framework, we apply it to two practical context-sensitive data-dependence analyses. Specifically, we compare the proposed LCL-reachability algorithm against the traditional CFL-reachability algorithm. The experimental results demonstrate that the LCL-reachability framework considerably improves existing approximations based on CFL-readability in terms of both precision and scalability.

6.1 Experimental Setup

Client analyses Our evaluation is based on two practical client analyses: A context-sensitive field-sensitive alias analysis for Java [40] and a context-sensitive field-sensitive taint analysis for Android apps [10]. In both applications, context-sensitivity is matched using a Dyck language D_m of size m, where each open parenthesis "$\llparenthesis_i$" represents a method call and each close parenthesis "$\rrparenthesis_i$" the corresponding method return. Filed-sensitivity is matched using another Dyck language D_n of size n:

- In the alias analysis, each field-sensitivity edge $(u, \llbracket_f, v)$ denotes that the field f of object u may point to v. For each field points-to edge $(u, \llbracket_f, v)$, the analysis builds an inverse edge $(v, \rrbracket_f, u)$. Similarly, the context-sensitivity edges representing calls and returns are also augmented with the corresponding inverse edges. As a result, the graphs of the alias analysis are bidirected. The bidirectedness is a prerequisite for CFL-reachability-based formulations of pointer analysis [30].
- In the taint analysis, each field-sensitivity edge $(u, \llbracket_f, v)$ denotes that the value of u flows to the filed f of variable v. Similarly, the $(u, \rrbracket_f, v)$ represents that variable v reads the field f of u.

In both analyses, the parentheses in D_m and brackets in D_n should be matched simultaneously. As a result, they each perform a context-sensitive data-dependence analysis using the non-context-free language $D_m \cap D_n$.

Evaluated algorithms We apply both the LCL-reachability algorithm and the traditional CFL-reachability algorithm to compute the *all-pairs* L-reachability in the two client applications, where L is $\mathbb{D}_{mn} = D_m \cap D_n$. The LCL-reachability formulation can precisely handle $\mathbb{D}_{mn}$. However, the CFL-reachability formulation only precisely describes either D_m or D_n. Therefore, we consider four CFL-reachability variants for approximation. We first consider the precise handling of field-sensitivity D_n. To cope with context-sensitivity D_m using CFL-reachability, two traditional treatments are: (1) making it context-insensitive and (2) approximating D_m using a regular language R_m. Specifically, the context-insensitive field-sensitive (CIFS) variant is essentially a Dyck language D_n that deems any node connected by an edge labeled by $\llparenthesis_i, \rrparenthesis_i \in D_m$ as reachable. The regular-approximating context-sensitive field-sensitive (CRFS) variant is $D_n \cap R_m$, where R_m is the approximation regular language shown in Figure 6. We follow the standard method to construct R_m [7, 9]. Since the size of R_m grows exponentially in terms of k, we set $k = 1$ in our CFL-reachability algorithm to make it scale. Finally, the context-sensitive field-insensitive (CSFI) and context-sensitive and regular-approximating field-sensitive (CSFR) variants can be similarly defined. Our evaluation compares the LCL-reachability algorithm (Lin) against the CFL-reachability algorithm based on the CIFS, CRFS, CSFI and CSFR approximations.

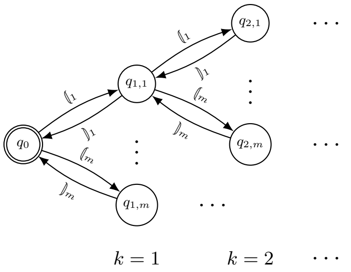


Figure 6. The finite state automaton approximating a Dyck language D_m. There is an additional accepting state q_x for accepting arbitrary parentheses beyond k. We omit q_x here for brevity.

Graph collection We select the benchmark program according to the original client analyses. Specifically, for the Java alias analysis [40], we use the standard DaCapo-2006-10-MR2 suite [1]. For the Android taint analysis [10], we use the 15 Contagio malware apps from the artifacts of the original reference. We adopt the implementations of both analyses to derive the graphs.[6] All benchmark programs are processed using Oracle JDK 1.7.0_51.

Implementation We implement all algorithms in C++, compiled with g++-4.9.3 at the -O3 optimization level. The algorithms share similar data structures for storing summary edges. The core data structure used in our algorithms is the sparse bitmap, which is taken from the GCC compiler. To obtain information on running time and memory consumption, we repeat each experiment three times and report the averages. All experiments were conducted on a 64bit machine with an Xeon X7542 2.6GHz processor and 128 GB memory, running Ubuntu 14.04.

6.2 Graph Characteristics

Table 3 gives the graph characteristics of the alias analysis and taint analysis, respectively. In general, the graphs of the alias analysis are significantly larger than those of the taint analysis. On the other hand, the graphs of both applications are sparse. On average, we have $|E|/|V| \approx 2.5$. However, the graph densities of the taint analysis, ranging from 2.0 to 3.8, appear to vary more significantly than those of the alias analysis. Finally, we also note that the numbers of calls are much larger than those of fields in both applications.

6.3 Precision Comparison

Tables 4 and 5 give the experimental results for both client analyses. We compare precision in terms of S-summary edge counts. In Table 5, we can see that the context-sensitive field-sensitive CFL-reachability variants (*i.e.*, CRFS and CSFR) are more precise than the CIFS and CSFI variants. However, they are also more expensive to compute, and are not scalable for the alias analysis. In both client analyses, it is consistent that the CFL-reachability framework gains more precision benefits from context-sensitivity (*i.e.*, the CSFI

[6] The alias analysis for Java is at `http://www.ics.uci.edu/~guoqingx/tools/alias.htm`, and the taint analysis for Android is at `https://github.com/proganalysis/type-inference`.

Program	**Graph**			
	V	E	#calls	#fields
antlr	29,831	69,768	7,103	1,246
bloat	36,181	82,241	10,400	1,360
chart	67,535	161,186	21,637	3,132
eclipse	30,981	72,354	7,392	1,346
fop	61,016	142,748	18,597	2,857
hsqldb	27,494	65,277	6,308	1,160
jython	36,162	104,260	11,137	1,398
luindex	28,595	67,462	6,519	1,228
lusearch	29,530	71,286	6,685	1,275
pmd	31,333	73,232	7,186	1,322
xalan	27,358	64,735	6,217	1,152

(a) Alias analysis graphs.

Program	**Graph**			
	V	E	#calls	#fields
Backflash	544	2,048	455	6
BattaryDoctor	1,674	4,790	1,132	87
DroidKungFu	734	1,983	367	39
Fakebanker	434	1,103	209	25
Fakedaum	1,144	2,603	715	42
Faketaobao	222	450	114	10
Jollyserv	488	998	236	28
Loozfon	152	323	78	3
Phospy	4,402	15,660	2,795	118
Roidsec	553	2,026	362	13
Scipiex	1,809	5,820	844	173
simhosy	4,253	13,768	2,894	330
Skullkey	18,862	69,599	12,316	1,340
Uranai	568	1,246	357	8
Zertsecurity	281	710	166	9

(b) Taint analysis graphs.

Table 3. Graph characteristics of two client analyses. The last two columns give the numbers of different call and field edges in the graphs.

Program	**Precision** (#S-pairs)					**Time** (s)			**Space** (MB)		
	CIFS	CSFI	Lin	$\cap_1$	$\cap_2$	CIFS	CSFI	Lin	CIFS	CSFI	Lin
antlr	2,153,091	1,827,705	1,519,527	1,511,902	1,511,512	165.46	768.11	7.76	674.42	913.20	3822.73
bloat	2,305,351	1,904,837	1,588,287	1,580,459	1,580,065	195.15	1161.30	11.38	800.62	1142.45	5632.24
chart	6,181,087	5,746,258	4,868,772	4,648,923	4,648,289	1170.75	7190.24	70.96	1588.62	2405.58	18024.00
eclipse	2,300,305	1,826,423	1,515,782	1,507,400	1,506,946	190.66	794.00	9.20	709.95	948.41	4218.52
fop	4,733,519	4,069,404	3,575,835	3,367,863	3,367,356	824.71	4393.99	45.07	1443.70	2166.72	14220.40
hsqldb	2,094,016	1,814,974	1,513,500	1,503,197	1,502,866	149.07	675.71	7.46	630.21	834.00	3534.08
jython	2,786,450	1,878,085	1,557,348	1,548,146	1,547,706	243.20	1225.47	13.33	840.79	1271.31	5420.17
luindex	2,151,119	1,820,530	1,513,628	1,505,988	1,505,573	161.44	698.43	7.33	657.75	866.41	3484.07
lusearch	2,199,729	1,824,220	1,517,576	1,507,395	1,507,027	171.55	718.47	9.75	644.55	895.12	3909.12
pmd	2,353,255	1,879,584	1,570,281	1,561,769	1,561,279	191.28	796.97	10.22	709.25	950.11	4181.22
xalan	2,078,462	1,814,712	1,511,005	1,502,976	1,502,586	147.38	665.81	7.47	622.53	828.00	3390.27

Table 4. Evaluation results of the alias analysis. The S-pairs column shows the numbers of S-summary edges, where S denotes the start symbol in CFL-reachability and the accepting states in LCL-reachability, respectively. The $\cap_1$ column shows the numbers of common S-pairs of CIFS and CSFI, and the $\cap_2$ column shows those for CIFS, CSFI, and Lin, respectively. Since all algorithms obtain over-approximate results, the fewer S-pairs, the better the precision.

Program	**Precision** (#S-pairs)					**Time** (s)					**Space** (MB)				
	CIFS	CRFS	CSFI	CSFR	Lin	CIFS	CRFS	CSFI	CSFR	Lin	CIFS	CRFS	CSFI	CSFR	Lin
Backflash	32,081	11,679	7,115	6,819	597	0.09	996.11	0.44	7.10	0.02	11.4	2,054.8	14.0	56.7	16.1
BattaryDoctor	109,662	-	15,978	-	3,071	1.86	-	2.25	-	0.12	43.6	-	47.6	-	97.0
DroidKungFu	41,072	-	11,813	9,523	1,510	0.44	-	0.89	178.34	0.03	18.6	-	19.9	686.3	23.8
Fakebanker	12,098	4,439	2,463	2,363	542	0.08	180.40	0.08	8.90	0.01	9.4	1,038.5	10.4	138.1	9.6
Fakedaum	59,104	-	6,480	6,237	1,560	0.50	-	0.59	206.43	0.04	24.3	-	27.4	1,246.2	42.9
Faketaobao	3,196	1,179	732	676	274	0.02	5.28	0.02	0.34	0.00*	4.7	163.9	5.5	21.5	3.4
Jollyserv	22,960	12,449	1,463	1,182	801	0.19	937.41	0.06	5.91	0.02	10.9	1,519.1	11.2	200.3	11.9
Loozfon	3,044	1,865	646	618	218	0.01	2.15	0.02	0.06	0.00*	3.4	37.5	4.2	5.0	2.6
Phospy	-	-	-	-	1,158K	-	-	-	-	35.57	-	-	-	-	3,383.3
Roidsec	81,485	-	18,598	16,592	611	0.38	-	1.88	55.49	0.02	-	16.7	18.1	167.8	16.8
Scipiex	976,255	-	71,759	-	146,913	333.50	-	21.69	-	2.74	476.2	-	93.3	-	478.6
simhosy	1,711,493	-	171,052	-	30,736	833.30	-	177.33	-	1.46	1,242.8	-	262.7	-	670.9
Skullkey	-	-	-	-	2,703K	-	-	-	-	190.40	-	-	-	-	23,290.8
Uranai	24,802	11,379	1,062	874	587	0.07	554.68	0.07	0.90	0.02	9.9	1,210.1	13.0	47.6	14.0
Zertsecurity	24,534	9,361	2,512	1,705	479	0.08	131.64	0.06	1.06	0.01	7.5	453.5	7.0	26.6	6.0

Table 5. Evaluation results of the taint analysis. The time budget for all algorithms is 1,000 seconds. Each "-" mark indicates a timeout, and each "$*$" a running time less than 0.01 seconds.

variants are more precise than the CIFS variants) since there are more call edges.

Tables 4 and 5 also demonstrate that the LCL-reachability algorithm is the most precise. Specifically, for the alias analysis, our algorithm is 1.2 times more precise than the best CSFI variant of CFL-reachability. For the taint analysis, the precision improvement is more dramatic, *i.e.*, it achieves 6.5 times precision improvements versus the best CSFR variant. We have also tried to increase the k of R_n in the CSFR variant. But the precision gain is marginal since the numbers of the field edges are significantly smaller than those of the call edges. We note that the precision improvement of LCL-reachability in the alias analysis is not as significant as the taint analysis. This may be due to the fact that the graphs in the alias analysis are bidirected — bidirectedness introduces more spurious summary edges since the LCL-reachability algorithm computes an over-approximation of the reachability information. Section 7.2 gives a detailed discussion on this approximation.

In general, the precision comparison indicates that the LCL-reachability framework benefits from the precision of both context- and field-sensitivities. It is more precise than the traditional CFL-reachability framework. In addition, Table 4 also gives the numbers of alias pairs by counting the common S-edges of two CFL-reachability algorithms (column $\cap_1$) and all three evaluated algorithms (column $\cap_2$). We notice that the CFL-reachability algorithm that combines both CIFS and CSFI is slightly more precise than the LCL-reachability algorithm. It is interesting to further investigate the connection between LCL-reachability and CFL-reachability that combines CIFS and CSFI.

6.4 Performance Comparison

Tables 4 and 5 list the running time and memory consumption for each algorithm. In CFL-reachability, a more precise variant computes more intermediate summary edges. Therefore, it requires more running time and memory. For instance, in Table 4, the running time of the more precise CSFI variant is 4.8 times longer than that of CIFS. In Table 5, the most precise CSFS variant requires 124.3 times more time than the cheapest CIFS variant on average.

The LCL-reachability algorithm is faster than the traditional CFL-reachability algorithm. In terms of the asymptotic time complexity, the LCL-reachability algorithm runs in $O(mn)$ time since each new summary edge is obtained using only original edges in the input graph. And the graphs in the two client analyses are all sparse graphs. On the contrary, the CFL-reachability algorithm exhibits a subcubic time complexity. From Tables 4 and 5, we can see that our proposed LCL-reachability algorithm is the fastest one while achieving the best precision at the same time. In both client analyses, it computes the results in minutes. For most benchmark programs, it finishes within 10 seconds. Specifically, in Table 4, our algorithm is 19.1 times faster than the CIFS CFL-reachability variant. In Table 5, it is 62.0 times faster than the best CIFS variant.

The space complexities of both LCL-reachability and CFL-reachability algorithm are quadratic. In practice, the comparison of memory consumption varies in different client analysis. For the alias analysis, the LCL-reachability algorithm consumes 4.8 times more memory than the CSFI CFL-reachability variant. However, the memory requirement of LCL-reachability is still modest. For the taint analysis, LCL-reachability's memory consumption is comparable to those of the CIFS and CSFI CFL-reachability variants.

7. Further Discussions

7.1 Extending the Framework

In practice, the class of LCL is capable of modeling more properties encountered in the client analysis. For example, the taint analysis considered in Section 6 obtains the value flows that start and end in the same method (*i.e.*, the D_m in $\mathbb{D}_{mn}$ properly matches all method calls and returns). In practice, the client taint analysis may also be interested in obtaining the value flows between different methods [10, 22]. The DSTM discussed in Section 4.1 can be easily extended, by introducing additional transitions to bypass the "$\llbracket$" and "$\#$" symbols during a right-to-left sweep, to accept the Dyck language augmented with outstanding open and close parentheses. We also note that many CFLs in the existing CFL-reachability formulations belong to the class of LCL, such as the alias analysis for C [43], the interprocedural dataflow analysis [33] and the polymorphic flow analysis [28].

The LCL-reachability framework can also be applied in demand-driven analyses to compute the reachability between any two given nodes. During graph traversal, a path string in LCL can be recognized in quadratic time, while it takes subcubic time to recognize a CFL string. Therefore, theoretically, the LCL-reachability framework is more efficient than the CFL-reachability framework. However, many CFLs (*e.g.*, the Dyck language) encountered in practice are deterministic CFLs which can be recognized in linear time. In this case, adopting CFL-reachability has a clear advantage. It is interesting future research to investigate better demand-driven LCL-reachability algorithms.

7.2 Understanding the Approximation

Algorithms 1 and 2 are sound approximations of the exact LCL-reachability problem. We briefly discuss two main sources of approximation in our algorithms.

The path length problem. Consider Figure 4, each summary edge on Level(k) is generated by two summary edges on Level$(k+1)$. For chain graphs, this always holds. Therefore, our algorithm computes the exact solution. When computing summary edges for arbitrary graphs shown in Figure 5(a), the algorithm cannot guarantee that two summaries $\widetilde{q_1 \diamond Z_1}$ and $\widetilde{q_2 \diamond Z_2}$ are on the same level. It is infeasible to simultaneously track the level and utilize the memorization on summary edges — the resulting algorithm may not terminate. Therefore, our algorithm approximates the exact solution by treating $\widetilde{q_1 \diamond Z_1}$ and $\widetilde{q_2 \diamond Z_2}$ as always on the same level.

The fixed path problem. According to Lemma 2, the LCL-reachability algorithm generates new summary edges $S(u, v)$ using both L- and R-terms. This appears to be the main source of the imprecision of LCL-reachability. Consider the path between nodes u and v in Figure 5(b). In the chain graph case, there is only one path connecting nodes u and v. Again, our algorithm computes the exact solution. However, for arbitrary graphs, the $Z_1 \in L(u, v)$ can be generated via any $y \in \{y_0, \ldots, y_i\}$. Similarly, $q_1 \in R(u, v)$ can be generated via any $x \in \{x_0, \ldots, x_i\}$. They belong to two different paths. Our algorithm approximates the situation by generating a new summary between u and v, without guaranteeing that u, x, y and v are on the same path. A formal treatment of the approximation is extremely helpful to understand the nature of LCL-reachability, which we leave as an open problem for future research.

7.3 Design Choices for Practical Analyses

Based on an exact L-reachability formulation, the proposed LCL-reachability framework enables more design choices for practical analyses than the traditional CFL-reachability framework. Analysis designers can implement more suitable STAs for specific analysis problems. For instance, Section 5 describes LCL-reachability algorithms based on two kinds of STAs: HTAs for Algorithm 1 and a GWTA for Algorithm 2. According to Section 5.4.2, Algorithm 2 achieves better precision for the context-sensitive data-dependence analysis problem since $\phi_{csdd} \subseteq \phi_{LCL}$. Algorithm 2 based on a GWTA also benefits from the set of heuristics designed for ruling out infeasible and spurious summaries. Moreover, we also notice that when the input graph is a tree, our LCL-reachability algorithm is able to obtain the exact solution.[7] In practice, it would be interesting to study important characteristics (*e.g.*, treewidth) of the graphs generated from real-world programs to develop better analysis algorithms. Performance wise, LCL-reachability operates on summary edges that are paired with one DSTM state and one DSTM tape symbol. It would thus also be interesting to develop better data structures and algorithms for indexing and querying the set of summary edges, or design efficient persistence schemes [38] to keep the intermediate results for staged analyses.

[7] Any two nodes in the input tree are connected via only one path. Therefore, the LCL-reachability problem can be reduced to LCL parsing.

8. Related Work

Context-sensitive data-dependence analysis is a rich abstraction for many program analyses [31]. The problem is known to be undecidable [31]. Therefore, every solution must resort to approximations. In practice, many analyses approximate context sensitivity; examples include a variety of context-insensitive and context-sensitive interprocedural analyses [35, 40, 41]. In a seminal work, Sharir and Pnueli describe the functional and call string approaches to interprocedural analysis [34]. The latter one also corresponds to the k-CFA approach for functional programming languages [21]. In set constraint formulations, Kodumal and Aiken propose a general annotated framework to simultaneously match regular and context-free properties [17]. These frameworks are very similar to the CS-k variant considered in our evaluation, with the typical exponential time complexity in k. In program verification, the reachability problem of multistack pushdown systems (MPDS) has been studies to model concurrent programs [18–20]. Many existing MPDS algorithms work with bounded constraints. Moreover, the formal language recognized by MPDS is too expressive to be adopted in our reachability problem.

In general, nearly all current approaches to context-sensitive data-dependence analyses employ approximate formulations since the actual analysis requires matching the intersection of two context-free properties [10, 35, 40]. The predominant framework for context-sensitive data-dependence analysis is CFL-reachability [30]. CFL-reachability depicts a program analysis problem using graph reachability. In the literature, many practical analyses have adopted the CFL-reachability formulation [3, 10, 26, 35, 40, 41]. Traditional CFL-reachability algorithms exhibit an $O(n^3)$ time complexity [30, 33]. The complexity is improved to $O(n^3/\text{log}n)$ by Chaudhuri using the Four Russians' trick [4]. Extensive effort has been devoted to improve the scalability of CFL-reachability algorithms [8, 35, 39, 42]. On the other hand, the time complexity for CFL-reachability is extremely hard to improve as it contains the CFL-recognition problem [30]. Improved results are only known for special cases [16, 41]. Recently, Tang *et al.* propose a new TAL-reachability formulation, which improves the scalability of CFL-reachability on constructing compact method summaries [36]. However, TAL is a super class of CFL and the reachability algorithm exhibits an $O(n^6)$ complexity. LCL-reachability is the first precise formulation for context-sensitive data-dependence analysis. The $O(mn)$ time complexity of LCL-reachability algorithm is asymptotically faster than all previous approaches.

9. Conclusion

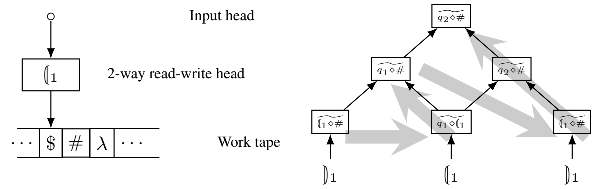
This paper has presented a new LCL-reachability framework. We have instantiated the framework on context-sensitive structure-transmitted data-dependence analysis, a rich abstraction for many practical program analyses. Our work provides a new perspective for this undecidable problem. The LCL-reachability formulation precisely models the problem since LCL contains the interleaved matched-parenthesis language, while our algorithm computes a sound approximation. We have also applied the LCL-reachability framework on two practical client analyses. The evaluation results show that our proposed LCL-reachability algorithm is both more precise and efficient than algorithms based on the traditional CFL-reachability formulation.

Acknowledgments

We would like to thank Shouyuan Chen, Martin Velez, and the anonymous reviewers for helpful comments on earlier drafts of this paper. We particularly appreciate the POPL reviewers' suggestions on investigating the GWTA and discussing its precision improvements. This research was supported in part by the United States National Science Foundation (NSF) Grants 1319187, 1528133, and 1618158. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

LCL rules					Corresponding DSTM transitions
(1)	$\widetilde{q_1 \diamond \llbracket_1}$	$\overset{I}{:=}$	$\llbracket_1$		$\delta(q_0, \lambda, \llbracket_1) = (q_1, \llbracket_1, -1)$
(2)	$\widetilde{\llbracket_1 \diamond \#}$	$\overset{I}{:=}$	$\rrbracket_1$		$\delta(q_0, \lambda, \rrbracket_1) = (\llbracket_1, \#, -1)$
(3)	$\widetilde{q_1 \diamond Z}$	$\overset{\delta}{:=}$	$\widetilde{\textvisiblespace \diamond Z}$	$\widetilde{q_1 \diamond \textvisiblespace}$	$\delta(q_1, Z, \epsilon) = (q_1, Z, -1)$
(4)	$\widetilde{q_2 \diamond \#}$	$\overset{\delta}{:=}$	$\widetilde{\textvisiblespace \diamond \#}$	$\widetilde{q_2 \diamond \textvisiblespace}$	$\delta(q_2, \#, \epsilon) = (q_2, \#, -1)$
(5)	$\widetilde{q_1 \diamond \llbracket_1}$	$\overset{\delta}{:=}$	$\widetilde{\textvisiblespace \diamond \llbracket_1}$	$\widetilde{q_2 \diamond \textvisiblespace}$	$\delta(q_2, \llbracket_1, \epsilon) = (q_1, \llbracket_1, -1)$
(6)	$\widetilde{\llbracket_1 \diamond \#}$	$\overset{\delta}{:=}$	$\widetilde{\textvisiblespace \diamond \#}$	$\widetilde{\llbracket_1 \diamond \textvisiblespace}$	$\delta(\llbracket_1, \#, \epsilon) = (\llbracket_1, \#, -1)$
(7)	$\widetilde{q_2 \diamond \#}$	$\overset{\delta}{:=}$	$\widetilde{\textvisiblespace \diamond \llbracket_1}$	$\widetilde{\llbracket_1 \diamond \textvisiblespace}$	$\delta(\llbracket_1, \llbracket_1, \epsilon) = (q_2, \#, -1)$

Table 6. LCL rules for D_1, where $Z = \{\llbracket_1, \#\}$. The accepting state of the trellis automaton is $\widetilde{q_2 \diamond \#}$.


Input head
$\llbracket_1$
2-way read-write head
$\cdots$ \$ # λ $\cdots$
Work tape
$\widetilde{q_2 \diamond \#}$
$\widetilde{q_1 \diamond \#}$
$\widetilde{q_2 \diamond \#}$
$\widetilde{\llbracket_1 \diamond \#}$
$\widetilde{q_1 \diamond \llbracket_1}$
$\widetilde{\llbracket_1 \diamond \#}$
$\rrbracket_1$
$\llbracket_1$
$\rrbracket_1$

(a) A DSTM.

(b) An HTA.

Figure 7. Illustrations on DSTM and HTA. The HTA is obtained through the original part (ii) construction.

A. On the Equivalence of HTA and DSTMs

The work of Ibarra and Kim [11] has shown that the deterministic homogeneous trellis automata (HTA) and the deterministic simple Turing machine (DSTM) are equivalent. The proof [11, Thm. 5.1] contains two parts: (i) Given an HTA, one can construct a DSTM that recognizes the same language; and (ii) given a DSTM, one can construct an equivalent HTA as well. However, the proof of part (ii) is not constructive.

This section gives a counterexample based on part (ii) construction of the original proof, discusses the root cause, and gives a corrected proof establishing the equivalence result.

A.1 A Counterexample

Let us use the DSTM $D_m = (Q_m, \Sigma, \Gamma_m, \delta_m, q_{0_m}, F_m)$ in Example 1 to recognize D_1. The same DSTM also appears in the original reference [11, Example 4.1]. Following the part (ii) construction, we obtain an HTA $K = (\Sigma, Q, I, \delta, F)$, where $Q = \{\widetilde{q \diamond Z} \mid q \in Q_m, Z \in \Gamma_m\}$, $F = \{\widetilde{q_2 \diamond \#}\}$. The transition functions δ and I are shown in Table 6.

Now, let us use the two devices to process an input "$\rrbracket_1 \llbracket_1 \rrbracket_1$". We give the DSTM and HTA in Figure 7. The DSTM configuration after processing the first character $\rrbracket_1$ is given in Figure 7(a). The DSTM head now points to the start marker \$ which corresponds to an end-of-right-to-left sweep. The DSTM intends to start a left-to-right sweep according to rules R9 and R10 in Example 1. However, the DSTM stops moving since $\delta(\llbracket_1, \$, \epsilon)$ is not a valid transition. According to the definition, the DSTM correctly rejects the input string. On the other hand, the HTA in Figure 7(b) processes the string and outputs an accepting state $\widetilde{q_2 \diamond \#}$.

A.2 Discussion

Let us revisit the restrictions on DSTMs in Definition 6. All state transitions can be divided into four categories: left-to-right (L2R) transitions, end-of-left-to-right (EOL) transitions, right-to-left (R2L) transitions and end-of-right-to-left (EOR) transitions. Among them, all EOR transitions are in the form $\delta(q, \$, \epsilon) = (q_0, \$, +1)$, where $q \in Q \setminus \{q_0\}$ due to the DSTM restriction (5).

The major problem of the part (ii) construction is that it implicitly assumes the EOR transition to be total, *i.e.*, the EOR transitions always handle q for all $q \in Q \setminus \{q_0\}$. However, the totality of EOR transitions is not a requirement for DSTMs. Therefore, for some state q' that is not depicted by any EOR transition, the DSTM stops moving and rejects the string. On the contrary, the constructed HTA processes the string and may accept it as described in Section A.1.

The key idea for our correction is to depict the non-totality of EOR transitions. Due to the isomorphism between HTA and DSTMs [11], we observe that the leftmost nodes on each HTA level correspond to EOR transitions.[8] Therefore, we use the white nodes to denote the original HTA nodes, and introduce gray nodes in our GWTA. Specifically, the set of gray node states $Q_g = \{\widetilde{q \diamond Z} \mid q \in Q_{eor}, Z \in \Gamma\}$ is a subset of Q_w, and explicitly describes all valid EOR transitions.

Finally, we discuss the GWTA processing on string "$\rrbracket_1 \llbracket_1 \rrbracket_1$". According to the GWTA construction, we have $Q_{eor} = \{q_1, q_2\}$. On the bottom level of the trellis automaton in Figure 7(b), we have $\widetilde{\llbracket_1 \diamond \#} \notin Q_g$ because $\llbracket_1 \notin Q_{eor}$. Since the GWTA is built in a bottom-up manner, the ancestor of node $\widetilde{\llbracket_1 \diamond \#}$ will not be generated due to the gray node transition $\delta_g : Q_g \times Q_w \rightarrow Q_g$. Consequently, the GWTA rejects the string.

A.3 Proof of Theorem 3

Proof. To establish the correctness, it suffices to show that every DSTM transition can be simulated by the GWTA. According to Section A.2, there are four cases to consider for all DSTM transitions:

- *L2R transitions:* They enable a DSTM to read the next input character. Any trellis automaton can naturally handle them.
- *R2L transitions:* Each state transition of white nodes $\delta_w : Q_w \times Q_w \rightarrow Q_w$ correspond to each R2L transition due to the isomorphism between the two models.
- *EOL transitions:* Every EOL transition reads a new input character which is simulated by the initial function I_l.
- *EOR transitions:* The EOR transitions specify the valid states upon reaching the start marker \$. According to Section A.2, those states are depicted as Q_g. Moreover, GWTA propagates the Q_g information for every gray node on Level(k) to its ancestor on Level($k-1$) using $\delta_g : Q_g \times Q_w \rightarrow Q_g$.

Finally, the GWTA handles all four kinds of DSTM transitions. □

A.4 The equivalence of HTA and DSTMs

The part (i) of the original proof can be reused to give the construction from an HTA to a DSTM. To establish the equivalence result, it suffices to show the construction from a DSTM to an HTA. Due to Theorem 3, we can construct an equivalent GWTA from a DSTM. A GWTA is essentially an STA. According to Theorem 1, we can also effectively construct an equivalent HTA from an STA. This completes the part (ii) construction. Based on the discussion, we obtain the equivalence result:

THEOREM 6. $L(DSTM) = L(HTA)$.

[8] A more formal discussion on the isomorphism can be found in the work of Okhotin [25]. Informally, a trellis automaton simulates DSTM moves along the gray arrows shown in Figure 7(b).

References

[1] DaCapo benchmark suite. `http://dacapobench.org/`.

[2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.

[3] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *POPL*, pages 553–566, 2015.

[4] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL*, pages 159–169, 2008.

[5] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automata I. *International Journal of Computer Mathematics*, 15:195–212, 1984.

[6] K. Culik II, J. Gruska, and A. Salomaa. Systolic trellis automatat II. *International Journal of Computer Mathematics*, 16:3–22, 1984.

[7] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[8] N. Hollingum and B. Scholz. Towards a scalable framework for context-free language reachability. In *CC*, pages 193–211, 2015.

[9] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[10] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for Android. In *ISSTA*, pages 106–117, 2015.

[11] O. H. Ibarra and S. M. Kim. Characterizations and computational complexity of systolic trellis automata. *Theor. Comput. Sci.*, 29:123–153, 1984.

[12] O. H. Ibarra, M. A. Palis, and S. M. Kim. Designing systolic algorithms using sequential machines. In *FOCS*, pages 46–55, 1984.

[13] O. H. Ibarra, S. M. Kim, and S. Moran. Sequential machine characterizations of trellis and cellular automata and applications. *SIAM J. Comput.*, 14(2):426–447, 1985.

[14] A. K. Joshi, L. S. Levy, and M. Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163, 1975.

[15] V. Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *LICS*, pages 27–36, 2009.

[16] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, pages 207–218, 2004.

[17] J. Kodumal and A. Aiken. Regularly annotated set constraints. In *PLDI*, pages 331–341, 2007.

[18] S. La Torre and M. Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *CONCUR*, pages 203–218, 2011.

[19] S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170, 2007.

[20] S. La Torre, M. Napoli, and G. Parlato. Scope-bounded pushdown languages. *Int. J. Found. Comput. Sci.*, 27(2):215–234, 2016.

[21] M. Might, Y. Smaragdakis, and D. V. Horn. Resolving and exploiting the *k*-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI*, pages 305–315, 2010.

[22] A. Milanova, W. Huang, and Y. Dong. CFL-reachability and context-sensitive integrity types. In *PPPJ*, pages 99–109, 2014.

[23] M. Nederhof. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44, 2000.

[24] A. Okhotin. On the closure properties of linear conjunctive languages. *Theor. Comput. Sci.*, 1-3(299):663–685, 2003.

[25] A. Okhotin. On the equivalence of linear conjunctive grammars and trellis automata. *Informatique Théorique et Applications*, 38(1):69–88, 2004.

[26] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via CFL reachability. In *SAS*, pages 88–106, 2006.

[27] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[28] J. Rehof and M. Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[29] T. W. Reps. Shape analysis as a generalized path problem. In *PEPM*, pages 1–11, 1995.

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

[31] T. W. Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

[32] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay. Speeding up slicing. In *SIGSOFT FSE*, pages 11–20, 1994.

[33] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[34] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, 1981.

[35] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI*, pages 387–400, 2006.

[36] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, and H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, pages 83–95, 2015.

[37] L. G. Valiant. Regularity and related problems for deterministic pushdown automata. *J. ACM*, 22(1):1–10, 1975.

[38] X. Xiao, Q. Zhang, J. Zhou, and C. Zhang. Persistent pointer information. In *PLDI*, pages 463–474, 2014.

[39] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.

[40] D. Yan, G. H. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.

[41] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*, pages 435–446, 2013.

[42] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su. Efficient subcubic alias analysis for C. In *OOPSLA*, pages 829–845, 2014.

[43] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208, 2008.