Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks

Hao Tang[1] Xiaoyin Wang[2] Lingming Zhang[3] Bing Xie[1] * Lu Zhang[1] Hong Mei[1]

[1]Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing, 100871, China
[2]Department of Computer Science, University of Texas at San Antonio, TX 78249, USA
[3]Department of Computer Science, University of Texas at Dallas, TX 75080, USA
{tanghao13, xiebing, zhanglu, meih}@sei.pku.edu.cn, xiaoyin.wang@utsa.edu, lingming.zhang@utdallas.edu

Abstract

Building a summary for library code is a common approach to speeding up the analysis of client code. In presence of callbacks, some reachability relationships between library nodes cannot be obtained during library-code summarization. Thus, the library code may have to be analyzed again during the analysis of the client code with the library summary. In this paper, we propose to summarize library code with tree-adjoining-language (TAL) reachability. Compared with the summary built with context-free-language (CFL) reachability, the summary built with TAL reachability further contains *conditional reachability* relationships. The *conditional reachability* relationships can lead to much lighter analysis of the library code during the client code analysis with the TAL-reachability-based library summary. We also performed an experimental comparison of context-sensitive data-dependence analysis with the TAL-reachability-based library summary and context-sensitive data-dependence analysis with the CFL-reachability-based library summary using 15 benchmark subjects. Our experimental results demonstrate that the former has an 8X speed-up over the latter on average.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Program analysis; F.4.3 [*Mathematical Logic and Formal Languages*]: Formal Languages - Classes defined by grammars or automata

General Terms Algorithms, Languages

Keywords TAL reachability, CFL reachability, context-sensitive analysis, summary-based analysis, tree adjoining languages

1. Introduction

Leveraging libraries is ubiquitous in modern software development. The existence of various software libraries makes it possible for developers to reuse products of numerous previous software developers. As a result, a modern software application is typically composed of a large volume of library code and a small volume of client code. For example, *compress*, a typical Java based application from SPECjvm2008 benchmark suite [2], involves 17640 Control-Flow Graph nodes while 91.6% of them are from libraries. The huge size of used library code in an application becomes a burden for various program analysis tasks. One common approach to this problem is to build a summary for each used library. Based on the library summary, the analysis time of the client code becomes much shorter. The summary for the library can also be built once and reused many times for analyzing all its client applications.

Data-dependence analysis [15] aims to identify the def-use chains in a program, and has many applications, such as slicing [31] and impact analysis [3]. For context-sensitive data-dependence analysis, a typical way to build the summary of a library is to use context-free-language (CFL) reachability [29], where a CFL (typically the parenthesis-matching language) is used to filter out invalid paths in the dependence graph. However, the possible existence of callbacks[1] poses an obstacle for summarizing the library code with CFL reachability. When there are no callbacks, the analysis of a client application can be confined to the client code with only queries of the summary of the library code. That is to say, it suffices that we know all the reachability relationships between library nodes after summarizing the library code. In presence of callbacks, even with the library summary, the library code should still be analyzed as well as the client code, because callbacks may incur new reachability (i.e., dependence) between library nodes.

To further motivate our research, let us consider the following example for context-sensitive data-dependence analysis.

```
1  package library;
2  public abstract class AbstractClass {
3        public final int method1(int x1) {
4              int y1 = x1 + 1;
5              int z1 = method2(y1) + 1;
6              return z1;
7        }
8        private final int method2(int x2) {
9              int y2 = x2 + 2;
10             int z2 = method3(y2) + 2;
11             return z2;
12       }
13       abstract public int method3(int x3);
14 }
```

* Corresponding author

POPL'15 January 15-17 2015, Mumbai, India

http://dx.doi.org/10.1145/2676726.2676997

[1] Callbacks are invocations of the methods/functions in the client code from the library code. Due to the convenience of customizing/extending libraries, many major programming languages provide some mechanism (e.g., dynamic binding) to support callbacks.

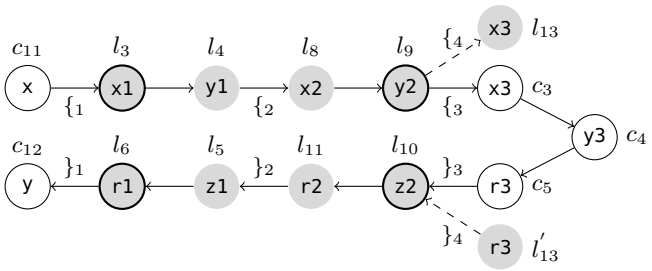


Figure 1: Data-Dependence Graph for the Example

As shown in the above code, there is an abstract class defined in the library. The abstract class provides a public method named `method1` for its clients, and requires its clients to implement a public method named `method3`, which may lead to a callback. The client code is listed below, where `ConcreteClass` extends the abstract class in the library with an implementation of `method3`, resulting in a callback.

```
1 package client;
2 class ConcreteClass extends library.AbstractClass {
3       public int method3(int x3) {
4             int y3 = x3 * 3;
5             return y3;
6       }
7 }
8 public class Main {
9       public static void main(String[] args) {
10            ConcreteClass p = new ConcreteClass();
11            int x = 4;
12            int y = p.method1(x);
13            System.out.println(y);
14      }
15 }
```

Supposing that it is required to determine whether the value of `y` in the `main` method after calling `method1` depends on the value of `x`. This problem can be turned into the problem of determining whether it is reachable from node c_{11} to node c_{12} in Figure 1, where the gray/white nodes are from the library/client code, and the ith line in client/library code is labeled as c_i/l_i. In this figure, the edges labeled with $\{_i/\}_i$ ($i \in [1, 4]$) correspond to direct data dependence incurred by the four method invocations/returns[2]; and nodes l_6, l_{11}, and c_5 represent the result values immediately before the method returns (denoted as r_1, r_2, and r_3). Obviously, it is reachable from node c_{11} to node c_{12}, if we use CFL reachability to analyze both the library code and the client code, i.e., the value of `y` depends on the value of `x`.

However, when summarizing only the library code with CFL reachability, as nodes c_3, c_4, and c_5 are not available yet, it is unknown whether it is reachable from node l_3 to node l_6. Only when the client code is available can it be determined that node c_3 is reachable to node c_5. Then the library graph should be analyzed again to determine that it is actually reachable from node l_3 to node l_6. With this information, node c_{11} can be determined to be reachable to node c_{12}. Note that nodes l_4, l_5, l_8, and l_{11} are essential here to determine this reachability relationship even if we already have the CFL-reachability-based summary of the library code. Without these four nodes, we cannot know that it is reachable from node l_3 to node l_6.

[2] Note that $\{_4/\}_4$ is marked as dashed line because it is the invocation/return to an abstract method.

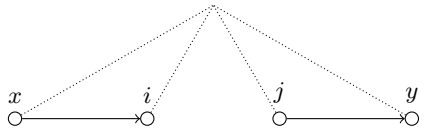


Figure 2: Graphical Representation of C-Reachable$_{i,j}(x,y)$

1.1 Our Solution

The basic idea of our approach is to further calculate *conditional reachability* relationships as well as unconditional reachability relationships[3] when summarizing the library code. Below, we define *conditional reachability* in Definition 1.1.

DEFINITION 1.1. ***(Conditional Reachability)*** *Let G be a directed graph and x, y, i, and j be four nodes in G. It is said that there is a conditional reachability relationship among x, y, i, and j (denoted as C-Reachable$_{i,j}(x,y)$), if it is reachable from x to y when a new unconditional reachability relationship from i to j is added.*

For intuitiveness of presentation, if C-Reachable$_{i,j}(x, y)$, we also state that there is a *conditional reachability* relationship from x to y depending on the reachability from i to j. Figure 2 depicts the graphical representation of C-Reachable$_{i,j}(x,y)$. Similar to the reachability relationships in CFL reachability, *conditional reachability* relationships can also be labeled with different symbols to distinguish *conditional reachability* relationships of different types.

For the motivating example, there is actually a *conditional reachability* relationship from node l_3 to node l_6 depending on the reachability from node l_9 to node l_{10} (i.e., C-Reachable$_{l_9,l_{10}}(l_3,l_6)$). If we obtain this *conditional reachability* relationship when summarizing the library code, we can immediately determine that it is reachable from node l_3 to node l_6 after we analyze the client code to obtain the reachability from node c_3 to node c_5. That is, we turn the *conditional reachability* relationship into an unconditional one. In fact, calculating *conditional reachability* during library-code summarization has the following two benefits. First, we can save some computation when analyzing the client code, as such computation has already been performed when analyzing the library code. In our example, the computation for determining C-Reachable$_{l_9,l_{10}}(l_3,l_6)$ actually involves some computation for determining the reachability from node l_3 to node l_6 under CFL reachability. Second, since we have the *conditional reachability* relationship during library-code summarization, some nodes (i.e., nodes l_4, l_5, l_8, and l_{11} in our example) in the library graph become unnecessary for analyzing client code. Thus, we can further avoid some unnecessary computation involving these nodes when analyzing the client code. Of course, calculating *conditional reachability* would incur extra computation during library-code summarization.

As a *conditional reachability* relationship involves four nodes in the graph, it is beyond the power of CFL reachability to calculate *conditional reachability*. Therefore, we turn the problem of summarizing *conditional reachability* relationships for the library code as a problem of calculating tree-adjoining-language (TAL) reachability. Tree-adjoining languages, each of which can be defined with a tree-adjoining grammar (TAG) [11], are a family of mildly context-sensitive languages that can represent much context sensitivity in natural languages[4]. As there are constructs (denoted as second-order non-terminals in this paper) in a TAG to represent two strings with some yet-unknown symbols between the two strings, such a construct can be used to represent

[3] To avoid ambiguity, we also refer to the reachability relationships calculated via CFL reachability as unconditional reachability relationships.

[4] Note that the meaning of *context sensitivity* in the formal language theory is completely different from that in program analysis.

conditional reachability, where the reachability between two library nodes depends on the reachability in client code.

Based on the definition of *conditional reachability*, we can summarize a library that contains at most one callback on any path. If multiple callbacks exist, our approach retains a small number of selected library nodes in the summary, and uses these nodes to chain multiple conditional reachability relationships. In such a case, the library is not fully-summarized because some analysis is still required on the selected library nodes during the client analysis phase. However, the client analysis phase can still be significantly accelerated (as validated in our evaluation) because only a small set of library nodes need to be analyzed.

To evaluate the performance enhancement that our approach achieves for analyzing client code, we empirically compared our approach with the approach that adopts CFL reachability to summarize the library code using 15 Java applications as subjects. Our evaluation results demonstrate that the analysis of client code using the TAL-reachability-based summary achieves an average 8X speed-up over the analysis using the CFL-reachability-based summary.

1.2 Contributions and Paper Organization

In summary, this paper makes the following main contributions:

- A concise formalism for defining tree-adjoining-language (TAL), serving as the Chomsky Normal Form for TAGs.
- The proposal of TAL reachability, and the first (polynomial) algorithm for calculating TAL reachability.
- A formalization of the problem of summarizing library code with callbacks for context-sensitive data-dependence analysis as a TAL-reachability problem.
- A sound technique for context-sensitive data-dependence analysis of client code based on the summary calculated with TAL reachability.
- An experimental evaluation of our approach on a number of benchmark subjects, comparing with the approach using CFL reachability to summarize the library code.

2. Tree-Adjoining Languages and Grammars

For natural language processing, researchers have investigated some formalisms to define mildly context-sensitive grammars to parse sentences in a natural language, which is typically a context-sensitive language. In particular, there are four intensively investigated formalisms for defining mildly context-sensitive grammars: tree-adjoining grammars [11], head grammars [23], linear indexed grammars [8], and combinatory categorial grammars [39]. According to [41], these four formalisms actually define the same family of languages, which are parsable in $O(n^6)$ time (where n is the length of the sentence for parsing). Thus, in this paper, we refer to the language family described by all the four grammars as tree-adjoining languages (TALs). As all the four existing grammars are not suitable for defining TAL reachability in Section 3, we present a new and concise formalism that defines the same language family as TALs (proof given in the Appendix). Our formalism is actually the Chomsky Normal Form for TAGs, which is suitable for defining TAL reachability and analyzing its complexity.

We define our tree-adjoining grammars (TAGs) by extending the formalism for defining CFGs. A context-free grammar G is typically defined as a four-tuple, denoted as G=(N, T, P, S), where N is a set of non-terminals, T is a set of terminals, P is a set of productions, and S is a non-terminal in N. Each terminal in T is a character and each non-terminal in N represents a set of strings, each being the concatenation of some terminals.

In our formalism for TAGs, there is a new type of non-terminals, each of which represents a set of ordered pairs of strings. To distinguish the two types of non-terminals, we refer to non-terminals representing strings as first-order non-terminals and non-terminals representing ordered pairs of strings as second-order non-terminals in this paper. For the ease of presentation, we use upper-case letters (e.g., A) to denote first-order non-terminals, upper-case letters in the blackboard typeface (e.g., $\mathbb{A}$) to denote second-order non-terminals, and lower-case letters to denote terminals. Furthermore, for two strings (denoted as α and β), if the pair of α and β is in the set of pairs of strings that $\mathbb{A}$ represents, we denote that $\mathbb{A}$ can represent $\alpha \circ \beta$. Note that, as each pair is ordered, $\alpha \circ \beta$ (indicating that α appears before β) and $\beta \circ \alpha$ (indicating that β appears before α) denote two different pairs. Similarly, if γ is in the set of strings that A represents, we denote that A can represent γ.

In the formalism for defining CFGs, there is only one operator (i.e., the concatenation operator) to connect non-terminals (i.e., first-order non-terminals) and/or terminals. In the formalism of TAGs, there are a number of extra operators to connect second-order non-terminals and first-order non-terminals. Specifically, there is one operator to connect two second-order non-terminals, and the result is a set of pairs of strings. This operator is called the *adjoining operator*, whose semantics is defined in Definition 2.1.

DEFINITION 2.1. *(**Adjoining Operator**) Let $\mathbb{A}$ and $\mathbb{B}$ be two second-order non-terminals. The semantics of the adjoining operator is as follows:*

- $\Cap(\mathbb{A}, \mathbb{B}) \doteq \{\alpha\gamma \circ \delta\beta \mid \alpha \circ \beta \in \mathbb{A},\ \gamma \circ \delta \in \mathbb{B}\}$.

In Definition 2.1 and throughout the rest of this paper, we adopt the convention in the formalism of CFGs to denote the concatenation operator. That is to say, when we put two symbols (each being a first-order non-terminal, a terminal, or a string) immediately side by side, we mean the concatenation between them. For example, we use $\alpha\beta$ to denote the concatenation of α and β.

The formalism of TAGs also includes four operators to connect a second-order non-terminal and a first-order non-terminal. The result of each such operator is also a set of pairs of strings. In fact, the four operators are extended from the concatenation operator between first-order non-terminals, and the differences lie in the position for concatenation. Thus, they are referred to as the extended concatenation operators. Definition 2.2 provides the semantics of the four operators.

DEFINITION 2.2. *(**Extended Concatenation Operators**) Let $\mathbb{A}$ be a second-order non-terminal and B be a first-order non-terminal. The semantics of the four extended concatenation operators are as follows:*

- $\LeftCap(\mathbb{A}, B) \doteq \{\gamma\alpha \circ \beta \mid \alpha \circ \beta \in \mathbb{A},\ \gamma \in B\}$.
- $\RightCap(\mathbb{A}, B) \doteq \{\alpha\gamma \circ \beta \mid \alpha \circ \beta \in \mathbb{A},\ \gamma \in B\}$.
- $\LeftCap(\mathbb{A}, B) \doteq \{\alpha \circ \gamma\beta \mid \alpha \circ \beta \in \mathbb{A},\ \gamma \in B\}$.
- $\RightCap(\mathbb{A}, B) \doteq \{\alpha \circ \beta\gamma \mid \alpha \circ \beta \in \mathbb{A},\ \gamma \in B\}$.

These four extended concatenation operators can also be used for connecting a second-order non-terminal and a terminal. For example, $\LeftCap(\mathbb{A}, a)$ represents all pairs of strings in the form of $a\alpha \circ \beta$.

The formalism of TAGs has an operator to construct a set of pairs of strings from two first-order non-terminals, referred to as the pairing operator, whose semantics is provided in Definition 2.3.

DEFINITION 2.3. *(**Pairing Operator**) Let A and B be two first-order non-terminals. The semantics of pairing operator is:*

- $\oplus(A, B) \doteq \{\alpha \circ \beta \mid \alpha \in A,\ \beta \in B\}$.

The pairing operator can also be used to the situation where either A or B is a terminal or ϵ (i.e., the empty string). For example, $\oplus(a, \epsilon)$ represents the pair $a \circ \epsilon$.

In contrast to the pairing operator, the formalism of TAGs has a de-pairing operator to obtain a set of strings from one second-order non-terminal. Definition 2.4 provides the semantics of the de-pairing operator.

DEFINITION 2.4. *(**De-Pairing Operator**) Let $\mathbb{A}$ be a second-order non-terminal. The semantics of de-pairing operator is:*

- $\ominus(\mathbb{A}) \doteq \{\alpha\beta \mid \alpha \circ \beta \in \mathbb{A}\}$.

Based on the preceding operators and the concatenation operator used for defining CFGs, the tree-adjoining grammars (TAGs) are defined in Definition 2.5.

DEFINITION 2.5. *(**Tree-Adjoining Grammar**) A tree-adjoining grammar G is a five-tuple, denoted as $G = (N_1, N_2, T, P, S)$, where N_1 is a set of first-order non-terminals, N_2 is a set of second-order non-terminals, T is a set of terminals, P is a set of productions, and S is a first-order non-terminal in N_1. Productions in P conform to the following rules.*

- *The left-hand side of a production must be a first-order non-terminal or a second-order non-terminal.*
- *If the left-hand side of a production is a first-order non-terminal, the right-hand side of the production must be in one of the following forms: 1) the de-pairing operator on one second-order non-terminal, 2) the concatenation of two symbols (each of which is either a first-order non-terminal or a terminal), 3) one first-order non-terminal, 4) one terminal, or 5) ϵ.*
- *If the left-hand side of a production is a second-order non-terminal, the right-hand side of the production must be in one of the following forms: 1) one adjoining operator on two second-order non-terminals, 2) one extended concatenation operator on one second-order non-terminal and one first-order non-terminal (or terminal), 3) the pairing operator on two symbols (each of which is a first-order non-terminal, a terminal, or ϵ), or 4) one second-order non-terminal.*

EXAMPLE 2.1. *The following TAG defines $\{a^n b^n c^n d^n \mid n \geq 0\}$:*

$S \rightarrow \ominus(\mathbb{S}_0)$
$\mathbb{S}_0 \rightarrow \cap_{\triangleright}(\mathbb{S}_3, d) \mid \oplus(\epsilon, \epsilon)$
$\mathbb{S}_1 \rightarrow \measuredangle\cap(\mathbb{S}_0, a)$
$\mathbb{S}_2 \rightarrow \between(\mathbb{S}_1, b)$
$\mathbb{S}_3 \rightarrow \curlywedge(\mathbb{S}_2, c)$

THEOREM 2.1. *The language family defined by CFGs is a true subset of the language family defined by TAGs.*

Proof. The second rule in Definition 2.5 defines a normal form for each production whose left-hand side is a first-order non-terminal. Except for the de-pairing operator, this normal form is actually a variant of Chomsky Normal Form. Therefore, any CFG, when transformed to the normal form required by this rule, is actually a TAG with no second-order non-terminals. As we know that the language defined in Example 2.1 is not a CFL, the CFL family is a true subset of the TAL family. □

3. TAL Reachability

As demonstrated by Reps [30], the notion of CFL reachability can be generalized to $\mathcal{L}$-reachability, where $\mathcal{L}$ can be any family of languages. By confining $\mathcal{L}$ to the family of TALs, we have the problem of TAL reachability. In the following, we present, to our knowledge, the first algorithm for TAL reachability, which is actually a polynomial algorithm .

ALGORITHM 1: Calculate TAL Reachability

Input: Directed graph G and productions of TAL language L.
Output: TAL-reachability relationships for G.

1 /***Initialize the work list***/
2 **for** *each edge $e(i, j)$ (labeled with a and $1 \leq i, j \leq n$) in G* **do**
3 | Add (a, i, j) to R_1; Add (a, i, j) to W;
4 **end**
5 **for** *each i ($1 \leq i \leq n$)* **do**
6 | Add (ϵ, i, i) to R_1; Add (ϵ, i, i) to W;
7 **end**
8 /***Add reachability for productions***/
9 **while** *W is not empty* **do**
10 | Select and remove the first item (denoted as ϖ) from W;
11 | **if** *ϖ is in the form of (X, i, j)* **then**
12 | | Try $Z \rightarrow X$ for (X, i, j)
13 | | Try $Z \rightarrow XY$ for (X, i, j)
14 | | Try $Z \rightarrow YX$ for (X, i, j)
15 | | Try $\mathbb{Z} \rightarrow \measuredangle\cap(\mathbb{Y}, X)$ for (X, i, j)
16 | | Try $\mathbb{Z} \rightarrow \between(\mathbb{Y}, X)$ for (X, i, j)
17 | | Try $\mathbb{Z} \rightarrow \curlywedge(\mathbb{Y}, X)$ for (X, i, j)
18 | | Try $\mathbb{Z} \rightarrow \cap_{\triangleright}(\mathbb{Y}, X)$ for (X, i, j)
19 | | Try $\mathbb{Z} \rightarrow \oplus(X, Y)$ for (X, i, j)
20 | | Try $\mathbb{Z} \rightarrow \oplus(Y, X)$ for (X, i, j)
21 | **end**
22 | **else if** *ϖ is in the form of $(\mathbb{X}, i, j, k, l)$* **then**
23 | | Try $Z \rightarrow \ominus(\mathbb{X})$ for $(\mathbb{X}, i, j, k, l)$
24 | | Try $\mathbb{Z} \rightarrow \mathbb{X}$ for $(\mathbb{X}, i, j, k, l)$
25 | | Try $\mathbb{Z} \rightarrow \measuredangle\cap(\mathbb{X}, Y)$ for $(\mathbb{X}, i, j, k, l)$
26 | | Try $\mathbb{Z} \rightarrow \between(\mathbb{X}, Y)$ for $(\mathbb{X}, i, j, k, l)$
27 | | Try $\mathbb{Z} \rightarrow \curlywedge(\mathbb{X}, Y)$ for $(\mathbb{X}, i, j, k, l)$
28 | | Try $\mathbb{Z} \rightarrow \cap_{\triangleright}(\mathbb{X}, Y)$ for $(\mathbb{X}, i, j, k, l)$
29 | | Try $\mathbb{Z} \rightarrow \Cap(\mathbb{X}, \mathbb{Y})$ for $(\mathbb{X}, i, j, k, l)$
30 | | Try $\mathbb{Z} \rightarrow \Cap(\mathbb{Y}, \mathbb{X})$ for $(\mathbb{X}, i, j, k, l)$
31 | **end**
32 **end**

3.1 Algorithm

Our algorithm for TAL reachability (depicted in Algorithm 1) is a dynamic programming algorithm, which can be viewed as an extension of the work-list algorithm for CFL reachability proposed by Melski and Reps [20].

Our algorithm maintains a work list (i.e., W) containing two categories of items. Each item in the first category represents the adding of a terminal or a first-order non-terminal. Such an item is in the form of (X, i, j), which represents the adding of X between the i-th node and the j-th node in G. Each item in the second category represents the adding of a second-order non-terminal. Such an item is in the form of $(\mathbb{X}, i, j, k, l)$, which indicates the addition of a second-order non-terminal between i, j, k, and l. Furthermore, we use R_1 to store the calculated reachability information of terminals and first-order non-terminals, and R_2 to store the calculated reachability information of second-order non-terminals.

In Algorithm 1, the two initialization steps (Lines 2 to 7) add reachability information into R_1 and items into W for 1) terminals labeled on edges of G, and 2) ϵ between a node and itself. The main body of the algorithm is a large loop (Lines 9 to 32), in which we iteratively deal with the items in W. We further divide each iteration of the loop into two parts. The first part (Lines 11 to 21) deals with items in the first category, and each line between Line 12 and Line 20 deals with one type of productions involving X at the right-hand side. The second part (Lines 22 to 31) deals with items in the second category, and each line between Line 22 and Line 29 deals with one type of productions involving $\mathbb{X}$ at the right-hand side. Note that, Lines 12 to 14 deal with basic CFL productions,

which has also been described in Melski and Reps's algorithm for CFL reachability [20].

ALGORITHM 2: Try $\mathbb{Z} \rightarrow \curvearrowleft(\mathbb{Y}, X)$ for (X, i, j)

```
1 for each production in the form of Z → ◿(Y, X) do
2     for each (Y, j, k, l, p) in R2 (1 ≤ k, l, p ≤ n) do
3         if (Z, i, k, l, p) ∉ R2 then
4             Add (Z, i, k, l, p) to R2;
5             Add (Z, i, k, l, p) to W;
6         end
7     end
8 end
```

Algorithm 2 depicts the details of Line 15 in Algorithm 1. As the right-hand side of such a production indicates that X appears immediately before the first string of $\mathbb{Y}$, Algorithm 2 looks for reachability information of $\mathbb{Y}$ in R_2 satisfying this requirement. If such information exists, we add reachability information of $\mathbb{Z}$ with the corresponding nodes. The algorithms for Lines 16, 17, and 18 are similar to Algorithm 2. The differences lie in the different ways that X concatenates with $\mathbb{Y}$.

The details of Line 19 in Algorithm 1 are depicted in Algorithm 3. As such a production does not require Y to be concatenated with X, Algorithm 3 looks for reachability information of Y in R_1 with no further requirement. If such information exists, we add reachability information of $\mathbb{Z}$ with corresponding nodes. The details of Line 20 in Algorithm 1 are similar to Algorithm 3.

ALGORITHM 3: Try $\mathbb{Z} \rightarrow \oplus(X, Y)$ for (X, i, j)

```
1 for each production in the form of Z → ⊕(X, Y) do
2     for each (Y, k, l) in R1 (1 ≤ k, l ≤ n) do
3         if (Z, i, j, k, l) ∉ R2 then
4             Add (Z, i, j, k, l) to R2;
5             Add (Z, i, j, k, l) to W;
6         end
7     end
8 end
```

The details of Line 23 in Algorithm 1 are depicted in Algorithm 4. To concatenate the two strings represented by $\mathbb{X}$, we require that the first string should appear immediately before the second string. If this requirement is satisfied, we add reachability information of Z between nodes i and l.

Line 24 in Algorithm 1 is similar to Line 12 in Algorithm 1. The difference is that Line 24 deals with productions each having two second-order non-terminals, while Line 12 deals with productions each having two first-order non-terminals.

Line 25 to Line 28 in Algorithm 1 are similar to Line 15 to Line 18 in Algorithm 1. The difference is that, for Line 25 to Line 28, we look in R_1 for first-order non-terminal Y that can concatenate with second-order non-terminal $\mathbb{X}$.

ALGORITHM 4: Try $Z \rightarrow \ominus(\mathbb{X})$ for $(\mathbb{X}, i, j, k, l)$

```
1 for each production in the form of Z → ⊖(X) do
2     if j=k ∧ (Z, i, l) ∉ R1 then
3         Add (Z, i, l) to R1;
4         Add (Z, i, l) to W;
5     end
6 end
```

The details of Line 29 in Algorithm 1 are depicted in Algorithm 5. In particular, we look for reachability information of $\mathbb{Y}$ in R_2 such that the first string of $\mathbb{Y}$ appears immediately after the first string of $\mathbb{X}$ and the second string of $\mathbb{Y}$ appears immediately before the second string of $\mathbb{X}$. The details of Line 30 in Algorithm 1 are similar to Algorithm 5.

ALGORITHM 5: Try $\mathbb{Z} \rightarrow \circledcirc(\mathbb{X}, \mathbb{Y})$ for $(\mathbb{X}, i, j, k, l)$

```
1 for each production in the form of Z → ⊚(X, Y) do
2     for each (Y, j, p, q, k) in R2 (1 ≤ p, q ≤ n) do
3         if (Z, i, p, q, l) ∉ R2 then
4             Add (Z, i, p, q, l) to R2;
5             Add (Z, i, p, q, l) to W;
6         end
7     end
8 end
```

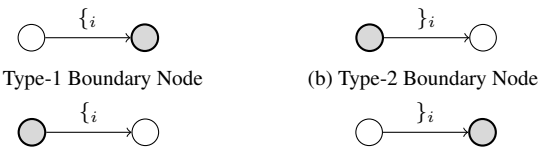
3.2 Time Cost

As our TAL-reachability algorithm does not allow one piece of reachability information to be added more than once, we have at most $O((|N_1|+|T|)*n^2)$ items in the form of (X, i, j) and at most $O(|N_2|*n^4)$ items in the form of $(\mathbb{X}, i, j, k, l)$. Then, we focus on analyzing the asymptotic time cost of dealing with one item in the form of (X, i, j) and one item in the form of $(\mathbb{X}, i, j, k, l)$, respectively. We summarize the result of our analysis as Theorem 3.1.

Lines 12 to 20 deal with each item in the form of (X, i, j). Among them, Line 13 requires the same asymptotic time as Line 14; Line 15 requires the same asymptotic time as Line 16, 17, or 18; and Line 19 requires the same asymptotic time as Line 20. Executing Line 12 once requires $O(|N_1|)$ time asymptotically, because given X, there are at most $O(|N_1|)$ productions in the form of $Z \rightarrow X$. Given X, there are at most $O(|N_1|*(|N_1|+|T|))$ productions in the form of $Z \rightarrow XY$, and for each such production there are at most $O(n)$ distinct pieces of reachability information of Y such that X appears immediately before Y. Thus, executing Line 13 once requires $O(|N_1|*(|N_1|+|T|)*n)$ time asymptotically. Given X, there are at most $O(|N_2|^2)$ productions in the form of $\mathbb{Z} \rightarrow \curvearrowleft(\mathbb{Y}, X)$, and for each such production there are at most $O(n^3)$ distinct pieces of reachability information of $\mathbb{Y}$ such that X appears immediately before the first string of $\mathbb{Y}$. Thus, executing Line 15 once requires $O(|N_2|^2*n^3)$ time asymptotically. Given X, there are at most $O(|N_2|*(|N_1|+|T|))$ productions in the form of $\mathbb{Z} \rightarrow \oplus(X, Y)$, and for each such production there are at most $O(n^2)$ distinct pieces of reachability information of Y. Thus, executing Line 19 once requires $O(|N_2|*(|N_1|+|T|)*n^2)$ time asymptotically. In summary, Lines 12 to 20 deal with one item in the form of (X, i, j) in $O((|N_1|+|N_2|+|T|)^2*n^3)$ time asymptotically.

Lines 23 to 30 deal with each item in the form of $(\mathbb{X}, i, j, k, l)$. Among them, Line 23 requires the same asymptotic time as Line 24; Line 25 requires the same asymptotic time as Line 26, 27, or 28; and Line 29 requires the same asymptotic time as Line 30. Executing Line 23 once requires $O(|N_2|)$ time asymptotically. Given $\mathbb{X}$, there are at most $O(|N_2|*(|N_1|+|T|))$ productions in the form of $\mathbb{Z} \rightarrow \curvearrowleft(\mathbb{X}, Y)$, and for each such production there are at most $O(n)$ distinct pieces of reachability information of Y such that Y appears immediately before the first string of $\mathbb{X}$. Thus, executing Line 25 once requires $O(|N_2|*(|N_1|+|T|)*n)$ time asymptotically. Given $\mathbb{X}$, there are at most $O(|N_2|^2)$ productions in the form of $\mathbb{Z} \rightarrow \circledcirc(\mathbb{X}, \mathbb{Y})$, and for each such production there are at most $O(n^2)$ distinct pieces of reachability information of $\mathbb{Y}$ such that the first string of $\mathbb{Y}$ appears immediately after the first string of $\mathbb{X}$ and the second string of $\mathbb{Y}$ appears immediately before the second string of $\mathbb{X}$. Thus, executing Line 29 once requires $O(|N_2|^2*n^2)$ time asymptotically. In summary, Lines 22 to 30 deal with one item in the form of $(\mathbb{X}, i, j, k, l)$ in $O((|N_1|+|N_2|+|T|)^2*n^2)$ time asymptotically.

THEOREM 3.1. *The asymptotic time cost of our TAL-reachability algorithm is $O(l^3*n^6)$, where l is $|N_1|+|N_2|+|T|$ and n is the number of nodes in G.*



(a) Type-1 Boundary Node (b) Type-2 Boundary Node

(c) Type-3 Boundary Node (d) Type-4 Boundary Node

Figure 3: Four Types of Boundary Nodes in Summary

4. Context-Sensitive Data-Dependence Analysis with TAL Reachability

On top of TAL reachability, we propose a novel algorithm for context-sensitive data-dependence analysis to calculate the reachability relationships between each pair of client nodes. Our algorithm consists of three steps. First, we build a summary containing *conditional reachability* information via analyzing the library code (Section 4.1). Second, we further process the summary to make it suitable for the analysis of client code (Section 4.2). Third, we use CFL reachability to analyze the client code based on the post-processed summary (Section 4.3). In Section 4.4, we demonstrate the soundness of our algorithm.

4.1 Summarizing Library Code with TAL Reachability

Definition 4.1 defines the TAL that we use to summarize the library code. In this grammar, S represents unconditional reachability with exactly matched parentheses, and $\mathbb{S}$ represents *conditional reachability* with exactly matched parentheses. Similarly, S_1 and $\mathbb{S}_1$ are for unconditional and conditional reachability with unmatched opening parentheses, and S_2 and $\mathbb{S}_2$ are for unconditional and conditional reachability with unmatched closing parentheses.

DEFINITION 4.1. ***(TAG for Library-Code Summarization)***

$S \rightarrow \ominus(\mathbb{S}) \mid SS \mid e$

$\mathbb{S} \rightarrow \circledast(\mathbb{S}, \mathbb{S}) \mid \measuredangleleft\!\!\cap(\mathbb{S}, S) \mid \cap\!\!\measuredangleright(\mathbb{S}, S) \mid \cap\!\!\measuredangleleft(\mathbb{S}, S) \mid \cap_{\triangleright}(\mathbb{S}, S) \mid \oplus(\{_i, \}_i)$

$S_1 \rightarrow \ominus(\mathbb{S}_1) \mid S_1S_1 \mid S_1S \mid SS_1 \mid \{_i$

$\mathbb{S}_1 \rightarrow \measuredangleleft\!\!\cap(\mathbb{S}, S_1) \mid \cap_{\triangleright}(\mathbb{S}, S_1) \mid \measuredangleleft\!\!\cap(\mathbb{S}_1, S_1) \mid \cap_{\triangleright}(\mathbb{S}_1, S_1)$

$S_2 \rightarrow \ominus(\mathbb{S}_2) \mid S_2S_2 \mid S_2S \mid SS_2 \mid \}_i$

$\mathbb{S}_2 \rightarrow \measuredangleleft\!\!\cap(\mathbb{S}, S_2) \mid \cap_{\triangleright}(\mathbb{S}, S_2) \mid \measuredangleleft\!\!\cap(\mathbb{S}_2, S_2) \mid \cap_{\triangleright}(\mathbb{S}_2, S_2)$

If we use the TAL defined in Definition 4.1 to calculate TAL reachability on a dependence graph G, we actually obtain exactly the same unconditional reachability relationships as we use the CFL[5] defined in Definition 4.2 to calculate CFL reachability on G. However, when calculating TAL reachability, we also obtain *conditional reachability* relationships as well as unconditional reachability relationships. In fact, our TAL-reachability algorithm always first determines a *conditional reachability* relationship, and if the *conditional reachability* relationship is in the form of C-Reachable$_{i,i}(x, y)$ for some i, our TAL-reachability algorithm (specifically Algorithm 4) then determines an unconditional reachability relationship from x to y. Therefore, if we are summarizing a library, using the TAL in Definition 4.1 allows us to summarize more information than using the CFL in Definition 4.2.

DEFINITION 4.2. ***(CFG for Context-Sensitive Data-Dependence Analysis)***

$S \rightarrow SS \mid T_i\}_i \mid e$

$T_i \rightarrow \{_iS$

$S_1 \rightarrow S_1S_1 \mid S_1S \mid SS_1 \mid \{_i$

$S_2 \rightarrow S_2S_2 \mid S_2S \mid SS_2 \mid \}_i$

[5] If we use this CFL to analyze both the library code and the client code, we can calculate all the context-sensitive data-dependence relationships.

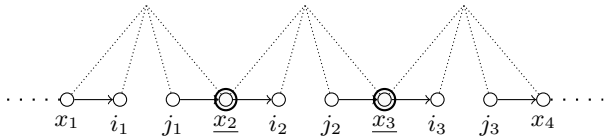


Figure 4: Chaining Nodes

4.2 Post-Processing

To make the summary obtained in Section 4.1 suitable for client-code analysis, we need to remove some unnecessary reachability relationships from the summary. Let us consider our example discussed in Section 1. Besides C-Reachable$_{l_9,l_{10}}(l_3,l_6)$, our TAL-reachability algorithm also calculates some other *conditional reachability* relationships using the TAL in Definition 4.1, e.g., C-Reachable$_{l_9,l_{10}}(l_4,l_5)$. In fact, C-Reachable$_{l_9,l_{10}}(l_4,l_5)$ does not provide further help for client-code analysis, but may incur unnecessary computation. To identify which information in the summary is unnecessary, we first identify in the summary a set of nodes such that only the reachability information among them is sufficient for client-code analysis. In this paper, we denoted these nodes as the *fundamental summary nodes*. It should be noted that, the post-processing is an important way to reduce the cost of client analysis when multiple callbacks exist. As the size of graph representing library summary shrinks, the graph will require much less unnecessary operations (adding non-terminals to edges) during client analysis.

4.2.1 Fundamental Summary Nodes

First, all the *boundary nodes* are *fundamental summary nodes*, since the reachability transfers between the summary and the client through them. Actually, there are four types of such *boundary nodes* in the summary. Figure 3, in which we use gray nodes to represent *boundary* nodes in the summary and white nodes to represent client nodes, depicts four boundary nodes each of a different type. The boundary node (referred to as the type-1 boundary node) in Figure 3(a) receives a value from the client via a call of a library method. The boundary node (referred to as the type-2 boundary node) in Figure 3(b) returns a value to the client via a method return. The boundary node (referred to as the type-3 boundary node) in Figure 3(c) passes a value to the client via a callback. The boundary node (referred to as the type-4 boundary node) in Figure 3(d) receives a value from the client via a return of a callback.

Second, it is still insufficient only keeping all the *boundary nodes*. Given one type-1 boundary node (denoted as x) and one type-2 boundary node (denoted as y) in the summary, there may be a chain of two or more *conditional reachability* relationships between them. If all the *conditional reachability* relationships in the chain are turned into unconditional, we have an unconditional reachability relationship between x and y. Note that, as such a chain may depend on more than one unconditional reachability relationship provided by the client, it cannot be represented by just one *conditional reachability* relationship. Therefore, we need to keep all the nodes that may connect two or more *conditional reachability* relationships. We refer to these nodes as the *chaining nodes*. Figure 4 depicts a sub-chain. In this figure, both nodes x_2 and x_3 are *chaining nodes*.

Third, there are two ways for one *conditional reachability* relationship (denoted as C-Reachable$_{i,j}(x,y)$) to become unconditional. The first way is that the client provides an unconditional reachability relationship to make the *conditional reachability* relationship become unconditional. In such a case, i must be a type-3 boundary node and j a type-4 boundary node. The other way is that a chain of *conditional reachability* relationships eventually provides an unconditional reachability relationship to make the

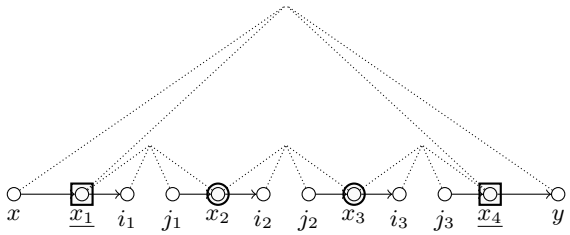


Figure 5: Hidden Chaining Nodes

conditional reachability relationship become unconditional. In such a case, either i or j may not be in the set of boundary nodes or the set of chaining nodes. We refer to such nodes as the *hidden chaining nodes*, which should also be kept in the summary. Figure 5, in which node x_1 and node x_4 are *hidden chaining nodes*, depicts such a situation. We further distinguish *hidden chaining nodes* into two types: type-1 and type-2. In Figure 5, x_1 is a type-1 and x_4 is a type-2 *hidden chaining node*.

Note that a *fundamental summary node* can be of multiple types, e.g., a type-3 *boundary node* may also be *chaining node*.

4.2.2 Identifying Fundamental Summary Nodes

Chaining nodes can be identified before calculating TAL reachability. The algorithm for identifying *chaining nodes* is depicted in Algorithm 6. Obviously, the asymptotical time cost of Algorithm 6 is $O(n + E)$, where n is the number of nodes and E is the number of edges. Essentially, a node with at least an incoming edge labeled with a closing parenthesis and at least an outgoing edge labeled with an opening parenthesis is a *chaining node*. However, due to the existence of edges labeled with e, we need to perform a propagation of edge labels (which is also based on a work-list algorithm) before checking the preceding condition. Algorithm 6 can ensure that, for any node (denoted as x) with at least an incoming edge labeled with a closing parenthesis and any node (denoted as y) with at least an outgoing edge labeled with an opening parenthesis, if there is a path from x to y with all edges labeled with e, there must be a *chaining node* along the path (including x and y). Note that the change of edge labels in Algorithm 6 should be confined to the algorithm itself so as not to impact other parts of our analysis.

After identifying the *chaining nodes* and calculating TAL reachability, we identify *hidden chaining nodes* in the following way. Supposing there is a *conditional reachability* relationship among x, y, i, and j (i.e., C-Reachable$_{i,j}(x,y)$) and the label is $\mathbb{S}$, if we know that y is a *chaining node*, we mark x as a type-1 *hidden chaining node*; and if we know that x is a *chaining node*, we mark y as a type-2 *hidden chaining node*.

It is straightforward to identify *boundary nodes*. The only issue for identifying *boundary nodes* is that we can only identify a set of potential boundary nodes when the client code is still unavailable. When the client code becomes available, we can refine the set to obtain real *boundary nodes*.

4.2.3 Discarding Unnecessary Reachability Information

As we keep only *fundamental nodes* in the summary, reachability information related to at least one node not being a *fundamental summary node* is naturally discarded. We further remove some reachability information as follows.

We keep all unconditional reachability relationships between two *fundamental summary nodes*. However, for a *conditional reachability* relationship in the form of C-Reachable$_{i,j}(x,y)$, we keep it in the summary only if both x and y are *fundamental summary nodes*; i is a type-3 boundary node or a type-1 hidden chaining node; and j is a type-4 boundary node or a type-2 hidden chaining node. All the other reachability information should be discarded.

ALGORITHM 6: Identify Chaining Nodes

```
1  for each node x with at least an incoming edge labeled with a closing
   parenthesis in G do
2      Add x to C; Add x to W;
3  end
4  while W is not empty do
5      Select and remove the first item (denoted as ϖ) from W;
6      for each outgoing edge (denoted as oe) labeled with e of ϖ do
7          Change the label of oe to an closing parenthesis;
8          Suppose that oe connects ϖ to y;
9          if y ∉ C then
10             Add y to C; Add y to W;
11         end
12     end
13 end
14 for each node x in G do
15     if there is at least one incoming edge of x labeled with a closing
       parenthesis and there is at least one outgoing edge of x labeled with
       an opening parenthesis then
16         Mark x as a chaining node
17     end
18 end
```

Furthermore, if there is already an unconditional reachability relationship of a certain type (i.e., S, S_1, or S_2) between two nodes (denoted as x and y), it is thus not useful to keep any *conditional reachability* relationships of the corresponding type (i.e., $\mathbb{S}$ for S, $\mathbb{S}_1$ for S_1, and $\mathbb{S}_2$ for S_2) in the form of C-Reachable$_{i,j}(x,y)$ for any i and j. The reason is that the only aim to keep a *conditional reachability* relationship is that the *conditional reachability* relationship may become an unconditional one during client-code analysis. In fact, we use CFL reachability to calculate all the unconditional reachability relationships before calculating TAL reachability; and when there is already an unconditional reachability relationship from x to y, we prevent our TAL-reachability algorithm from adding any *conditional reachability* relationship in the form of C-Reachable$_{i,j}(x,y)$ to R_2 in the first place.

4.3 Analyzing Client Code

With the post-processed summary, we use CFL reachability to analyze the client code. Specifically, we use the CFL defined in Definition 4.2 to analyze the graph composed of the client graph, the *fundamental summary nodes* with the reachability relationships among them, and edges each between a client node and a *boundary node* in the summary. Our CFL-reachability algorithm extends Melski and Reps's work-list algorithm [20] as follows.

First, for each unconditional reachability relationship in the summary, we put it into the work list during initialization, because it may trigger some productions in Definition 4.2 (e.g., $T_k \rightarrow \{_k S$, $S \rightarrow T_k\}_k$, and $S \rightarrow SS$).

Second, for each *conditional reachability* relationship in the summary, we do not put it into the work list, because we do not want to have new *conditional reachability* relationships. However, when we have a new unconditional reachability relationship between two summary nodes (denoted as i and j), we check whether this unconditional reachability relationship can turn a *conditional reachability* relationship into unconditional. That is to say, we check whether there exist two summary nodes (denoted as x and y) such that C-Reachable$_{i,j}(x, y)$.

4.4 Soundness

We demonstrate that our algorithm for context-sensitive data-dependency analysis using TAL-reachability-based summary is sound in Theorem 4.1.

THEOREM 4.1. *(**Soundness**) Our algorithm using TAL-reachability-based summary produces sound results for context-sensitive data-dependency analysis.*

In stead of directly proving Theorem 4.1, we prove that our algorithm produces exactly the same results as the CFL-reachability-based algorithm using the CFL defined in Definition 4.2 (i.e., Theorem 4.2). Since the CFL-reachability-based algorithm is sound, our algorithm is also sound.

THEOREM 4.2. *(**Equivalence**) Our algorithm using TAL-reachability-based summary produces exactly the same results for context-sensitive data-dependency analysis as the CFL-reachability-based algorithm using the CFL in Definition 4.2.*

Proof. To prove Theorem 4.2, we need to demonstrate that the following two properties hold for any two nodes (denoted as x and y) in the client graph. First, if our algorithm determines a reachability relationship from x to y, the CFL-reachability-based algorithm also determines such a reachability relationship. Second, if the CFL-reachability-based algorithm determines a reachability relationship from x to y, our algorithm also determines such a reachability relationship. For conciseness, we consider only the reachability of exactly matched parentheses, as the proof for the other two types of reachability can be naturally extended from the current proof.

The first property obviously holds. In fact, when our algorithm determines a reachability from x to y, our algorithm can ensure that there is at least a path from x to y labeled with exactly matched parentheses.

Let G be the union of the client graph and the library graph, N be the union of the set of client nodes and the set of $fundamental\ summary\ nodes$, $x{\in}N$ and $y{\in}N$. We actually prove that if there is a parenthesis-matched path from x to y (denoted as $path_{x,y}$) in G, our algorithm determines a reachability relationship from x to y. Note that this target property is stronger than the second property.

We prove the target property via mathematical induction on the basis of the number (denoted as p) of pairs of matched parentheses in $path_{x,y}$. When p=0, our target property obviously holds. Since there is no parenthesis in $path_{x,y}$, $path_{x,y}$ can be divided into a series of pieces, each of which is composed of nodes all in the library graph or nodes all in the client graph. As our algorithm keeps all the unconditional reachability between $boundary\ nodes$ in the library graph during library-code summarization and post-processing, our CFL-reachability algorithm for client-code analysis definitely determines $path_{x,y}$.

Let us assume that, for any $0{\leq}r{\leq}q$, our target property holds when p=r. In the following, we prove that our target property holds when p=q+1. Let us denote the first opening parenthesis in $path_{x,y}$ as OP and the closing parenthesis matched with OP in $path_{x,y}$ as CP. There are four situations for $path_{x,y}$.

1. OP does not connect two library nodes and CP is not the last closing parenthesis in $path_{x,y}$.
2. OP does not connect two library nodes and CP is the last closing parenthesis in $path_{x,y}$.
3. OP connects two library nodes and CP is not the last closing parenthesis in $path_{x,y}$.
4. OP connects two library nodes and CP is the last closing parenthesis in $path_{x,y}$.

For Situation 1, let CP connect node i to node j. As CP is not the last closing parenthesis, there is a parenthesis-matched sub-path from x to j in $path_{x,y}$, where the number of pairs of matched parentheses is at most q; and there is a parenthesis-matched sub-path from j to y in $path_{x,y}$, where the number of pairs of matched parentheses is at most q. Since OP does not connect two library nodes, CP must not connect two library nodes. Therefore, j is a client node or a $boundary\ node$ in the library. According to the inductive assumption, our algorithm determines a reachability relationship from x to j and a reachability relationship from j to y. As a result, our algorithm determines a reachability relationship from x to y.

For Situation 2, let OP connect node i to node j and CP connect node k to node l. Thus, there is a parenthesis-matched sub-path from j to k in $path_{x,y}$, where the number of pairs of matched parentheses is q. Since OP does not connect two library nodes, CP must not connect two library nodes. Therefore, j is a client node or a $boundary\ node$ in the library, and so is k. According to the inductive assumption, our algorithm determines a reachability relationship from j to k. As a result, our algorithm determines a reachability relationship from x to y.

Situation 3 is similar to Situation 1. However, supposing that CP connects node i to node j, j may not be a $fundamental\ summary\ node$. Let us consider the first opening parenthesis (denoted as OP') between j and y along $path_{x,y}$. Let OP' connect node k to node l. As j is a library node and a library node can connect to a client node only through an edge labeled with a parenthesis, k must be a library node and all the nodes between j and k along $path_{x,y}$ are library nodes. Thus, our Algorithm 6 can identify at least one $chaining\ node$ in the sub-path from j to k. Note that, if l is a client node, there must be another library node (denoted as l') and an edge labeled with an opening parenthesis from k to l'. Let node u be such a chaining node. Our algorithm determines a reachability relationship from x to u and a reachability relationship from u to y based on the inductive assumption.

Situation 4 is similar to Situation 2 but more complicated. Supposing that OP connects node i to node j and CP connects node k to node l, any of the four nodes may not be a $fundamental\ summary\ node$. As there is no parenthesis between x and i along $path_{x,y}$, the entire sub-path from x to i is in the library graph. Similarly, the entire sub-path from l to y is in the library graph. Now, we consider all pairs of matched library parentheses nested with the pair of OP and CP along $path_{x,y}$. Let OP' (connecting node i' to node j') and CP' (connecting node k' and node l') denote the opening parenthesis and the closing parenthesis in the innermost pair. It can be proven that there exists a node (denoted as u) in the sub-path from j' to k' along $path_{x,y}$ such that u is a type-3 boundary node or a type-1 hidden chaining node and there is no parenthesis between j' and u along $path_{x,y}$. Similarly, there exists a node (denoted as v) in the sub-path from j' to k' along $path_{x,y}$ such that v is a type-4 boundary node or a type-2 hidden chaining node and there is no parenthesis between v and k' along $path_{x,y}$. According to the inductive assumption, our algorithm determines a reachability relationship from u to v. Furthermore, our algorithm keeps a $conditional\ reachability$ relationship in the form of C-Reachable$_{u,v}(x,y)$.

To show the existence of u, let us consider the first opening parenthesis (denoted as OP'') in the sub-path from j' to k' along $path_{x,y}$. Supposing that OP'' connects node i'' to node j'' and the matched closing parenthesis of OP'' (denoted as CP'') connects node k'' to l''. If j'' is a client code, i'' is thus a type-3 boundary node and can serve as node u. Suppose that j'' is a library node. Then, CP'' must not be the last parenthesis in the sub-path from j' to k' along $path_{x,y}$; otherwise, the pair of OP'' and CP'' would nest inwardly with the pair of OP' and CP'. Thus, we can find a $chaining\ node$ (denoted as u') between l'' and k' along $path_{x,y}$ such that there is no parenthesis between l'' and u'. Due to the existence of u', i'' is thus a type-1 hidden chaining node and can serve as node u. □

5. Experimental Evaluations

In this section, we consider the cost our TAL-based approach in comparison with the CFL-based approach. As our main consideration is whether the TAL-reachability-based summary can boost client code analysis, our first experiment (**E1**) focuses on the cost comparison between our TAL-based approach and the CFL-based approach for analyzing *client code*. Furthermore, to understand how much more computation the calculation of TAL reachability would incur, our second experiment (**E2**) focuses on the cost comparison between our TAL-based approach and the CFL-based approach for summarizing *library code*. More details can be found in our project website at `http://www.utdallas.edu/~lxz144130/tal.html`.

5.1 Implementation

We implemented our TAL-Reachability algorithm in Java. More specifically, we use hash tables to store R_1 and R_2 in Algorithm 1 to 5 because reachability information in R_1 and R_2 is typically sparse. We use several indexing facilities to access reachability in R_1 and R_2 when trying different productions in Algorithm 1. For R_1, we use two hash tables to index all the reachability information in the form of (X, i, j) by $<X, i>$ and $<X, j>$, respectively. For R_2, we use six hash tables to index $(\mathbb{X}, i, j, k, l)$ by $<\mathbb{X}, i>$, $<\mathbb{X}, j>$, $<\mathbb{X}, k>$, $<\mathbb{X}, l>$, $<\mathbb{X}, i, l>$ and $<\mathbb{X}, j, k>$, respectively. Here, we use HashSet and HashMap from the Java Standard Library to maintain the reachability information (i.e., R_1 and R_2) and the indexing information for R_1 and R_2, respectively.

Furthermore, we also implemented a TAL-based context-sensitive data-dependence analysis tool on top of Wala (version 1.3.5) [6] for Java applications. We used Wala to build the data dependence graph of the program. When building the data dependency graph, we do not directly perform alias analysis but retain all the information necessary for alias analysis (assignment information between object variables). Therefore, when TAL-reachability algorithm is applied, the alias information will be computed (in a flow-insensitive and field-insensitive but context-sensitive way) together with data dependence among various code elements. We chose to target Java applications, because Java is one of the most popular programming languages and callbacks are ubiquitous (even inevitable) in Java libraries due to the support of method overriding in Java. Our tool consists of (1) a summarization component that analyzes and generates TAL-based summaries for *library code*, and (2) an analysis component that performs context-sensitive data-dependence analysis for the *client code* using the generated summaries.

5.2 Experimental Setup

All our experiments were performed on a PC with 4-core 8-thread Intel Core i7-3770 CPU (3.4GHz) and 4 Gigabyte RAM running JVM 1.7.0-55 on Ubuntu Linux 13.10.

Benchmarks. We used 11 Java programs from SPECjvm2008 [2][7] as our benchmark. The SPECjvm benchmark suite has been widely used in the program analysis research [37, 38, 44, 45]. In addition, we also randomly selected 4 programs from GitHub [1] to evaluate our approach. The first 5 columns in Table 1 depict the statistics of the subjects used in our evaluation. Column 1 and 2 lists the subjects' name and size in lines of code; Columns 3 to 5 present the total number of nodes, the number of $fundamental\ summary\ nodes$, and the number of nodes in client code for each subject's data-dependence graph. Note that, each of our subjects contains a Java library and a client program that invokes the library in different ways. We analyze all the methods (in either client or library code) that are reachable from the main method of the client program, and we compute summaries for only the main Java library in the benchmark (while ignoring other invoked libraries such as java.lang). Our statistics are based on the part of client and library code that are involved in our analysis.

Compared Techniques. Since the summary information for library code is reusable, we may simply compare our approach in analyzing the client code with TAL-reachability-based summaries with the traditional CFL-based approach in analyzing the whole code base. However, that is not fair because the CFL-based approach can also pre-compute summary information for the part of library code that does not involve any callbacks. Note that, when using CFL reachability to summarize library code, if one node is labeled as not reachable to another node, the reachability is still possible during client-code analysis due to callbacks.

To enable a fair comparison between our TAL-based approach and the traditional CFL-based approach, we implemented a two-step CFL-based approach containing two components: (1) a summarization component that uses CFL reachability to pre-compute summary information for the *library code*, and (2) an analysis component that performs data-dependence analysis for the *client code* on the basis of the CFL-reachability-based library summaries. Similar to the implementation of our approach, we implemented the CFL-based approach using Melski and Reps's generic CFL-reachability algorithm [20]. Thus, we ensure that the CFL-based approach is the same with our approach except that it computes and uses CFL-reachability-based summaries instead of TAL-based summaries for library code.

To investigate the efficiency of our approach for analyzing client code (i.e., **E1**), we compared the second component of our approach with that of the CFL-based approach. Furthermore, to measure the overhead incurred by our TAL-based summary computation (i.e., **E2**), we compared our first component with that of the CFL-based approach.

Measurements. Similar to existing studies of CFL-reachability-based analysis [37, 44, 45], we measured both the time cost and the peak memory usage of our TAL-based analysis .

5.3 Results

The main experimental results are shown in Table 1. Columns 6 to 9 show the results for client code analysis (**E1**), and Columns 10 to 13 show the results for library summarization (**E2**).

E1: Client-Code Analysis. From Columns 6 to 9 of Table 1, we make the following observations. First, the time cost of our TAL-based approach is much lower than that of the CFL-based approach. For example, to analyze the client code for all the subjects, the CFL-based approach needs 7877 milliseconds, while our TAL-based approach needs only 956 milliseconds, indicating a speed-up of 8.24X. In an extreme case, our TAL-based approach achieves a speed-up of 25.83X for the xml subject. In addition, our TAL-based approach outperforms the CFL-based approach for all the studied subjects in terms of time cost (with at least a speed-up of 2X). The reason is that, although our TAL-based approach leverages the same CFL-reachability algorithm when analyzing client code, it is able to utilize more intensive summaries for the library code, so that more parts of the library code need not to be analyzed again. Second, the peak memory consumption of the CFL-based approach is also larger than that of our TAL-based approach. This is also expected because our approach maintains reachability information between the $fundamental\ summary\ nodes$, whose number is much smaller than the number of all library nodes. As depicted in Table 1, in total, the number of $fundamental\ summary\ nodes$ is only 6.35% of the number of library nodes. Furthermore, for each subject, the number of $fundamental\ summary\ nodes$ is always less than 10% of the number of library nodes. In sum-

[6] http://wala.sourceforge.net

[7] Note that the $serial$ program is not included because our approach runs out of memory during library-code summarization.

Subjects	Size (LOC)	#Nodes			Client-Code Analysis				Library Summarization			
		Total	Fund.	Client	CFL		TAL		CFL		TAL	
					T(ms)	M(MB)	T(ms)	M(MB)	T(ms)	M(MB)	T(ms)	M(MB)
check	10,672	19951	1539	3347	348	152	67	58	267	107	645	132
compiler	9,607	16726	1220	536	330	153	53	73	309	130	661	141
compress	10,058	17640	1309	1483	330	159	64	75	321	130	670	142
crypto	21,503	23310	1825	3216	459	174	73	66	339	126	773	207
derby	21,568	24513	1687	1106	684	250	78	93	435	194	998	610
helloworld	9,458	16299	1207	296	309	130	36	48	283	105	640	138
mpegaudio	40,246	56493	2175	27576	1047	378	215	196	575	239	4421	387
scimark	10,326	18297	1287	2027	331	161	67	78	322	131	665	149
startup	11,104	19933	1549	621	450	168	74	65	351	132	941	268
sunflow	9,102	15700	1121	85	290	123	33	44	293	100	663	127
xml	47,556	73786	4501	2312	2945	756	114	184	1433	587	5811	741
btree	5,168	7813	406	1103	105	80	48	56	179	68	264	155
mushroom	2,866	3865	272	7	80	64	8	48	169	76	183	93
parser	5,545	7968	385	112	108	86	13	53	199	74	269	158
sample	2,894	3969	247	28	61	64	13	48	162	73	184	99
Total	217,673	326263	20730	43855	7877		956		5637		17788	

Table 1: Experimental results

mary, without sacrificing any precision, our approach is able to accelerate the context-sensitive data-dependence analysis on client code by almost a magnitude.

E2: Library Summarization. From Columns 10 to 13 of Table 1, we are able to make the following observations. First, the time cost of our approach is higher than that of the CFL-based approach when summarizing library code. Note that our TAL-based approach uses CFL reachability as a preprocessing step for summarizing the library code. To summarize the library code for all subjects, our TAL-based approach needs 17788 milliseconds, while the CFL-based approach needs only 5637 milliseconds, indicating an average slow-down of 3.16X. Second, the average peak memory consumption of our approach is 236.5MB, which is also higher than that of the CFL-based approach (i.e., 151.5MB). The reason for the slow-down and extra memory consumption is that our approach further computes the TAL-reachability information for library code as well as the CFL-reachability information. As this phase is typically performed much less frequently than the client-code analysis, the average slow-down of 3.16X can be affordable.

6. Discussion

6.1 Factors Affecting Client-Analysis Speed-up

Our experimental results in Section 5 demonstrate that the speed-up of our approach for client-code analysis varies much for different subjects (i.e., from 2X to 25X). Here, we further qualitatively investigate the factors that may affect the speed-up.

The first factor is the number of callbacks the client code provides. If the client does not provide any callbacks, our TAL-based summaries are essentially the same as the CFL-based summaries. Intuitively, the more callbacks the client provides, the more possibly our approach achieves a large speed-up. Also, since the impacts of different callbacks on the analysis are different, the overall impact does not depend only on the number of callbacks.

The second factor is the depth of nested method calls in the library code. Since our approach folds up nested method calls in library-code summarization, our client-code analysis can avoid analyzing those deeply nested method calls again. TAL reachability works best when some deeply nested method calls are callbacks.

The third factor is the number of long chains of $conditional\ reachability$ relationships that are eventually turned into unconditional. Since our library-code summarization can speed up only the determination of each $conditional\ reachability$ relationship to be unconditional in such a chain, the existence of many such long chains may affect our approach negatively.

6.2 Other Applications of TAL Reachability

Although we only investigate one application of TAL reachability, there may be many other applications in the following categories:

First, since many analyses can be modeled as CFL-reachability problems [3, 31, 40], TAL reachability should be useful for improving summary-based versions of these analyses. That is to say, we can use TAL reachability to calculate a more intensive summary of library code to boost client-code analysis.

Second, TAL reachability may provide further help for analyzing applications in which some code in the code base is not suitable for static analysis [43]. Two common cases are dynamically loaded code and dynamically generated code. Since dynamically loaded or generated code is typically impossible or very hard to analyze, to ensure the quality (e.g., reliability and security) of the whole program, dynamic analyzers and checkers are often enforced at run time. Similar to summarizing library code, it may be beneficial to acquire more intensive analysis results of the static code with TAL reachability so as to reduce the overhead of runtime enforcement.

Third, TAL reachability may be a useful tool for analyzing partial code. In other words, we can use TAL reachability to acquire useful properties of part of a code base regardless of the missing code (e.g., client code, dynamically loaded code, and dynamically generated code). For example, the $conditional\ reachability$ information calculated by our algorithm can be easily extended to acquire the dependence between any two nodes in the library graph regardless of the client code: definite dependence, possible dependence, and impossible dependence. This dependence information can then be used for compiling and verifying the library.

Fourth, TAL reachability may also be useful for analyzing evolving software systems. Modern software systems usually undergo various revisions [5]. When the developers are changing some code files, they can run TAL-reachability-based analysis for the unchanged files (including libraries) meanwhile at the backend. Then, when the developers finish the revision, they can directly use the pre-computed information for unchanged files to speed-up program analysis for verifying the changed files.

6.3 Considering Field Sensitivity

In program analysis, field sensitivity is another important factor that impacts the precision of analysis [18, 37, 45]. Our approach performs flow-insensitive and field-insensitive data dependency analysis, and uses a simple heap abstraction that does not consider type structures. The rich heap abstractions that enable field-sensitivity can also be represented with CFL reachability. However, unfortunately, data-dependence analysis considering both kinds of sensi-

tivity is undecidable [30]. One typical solution is to use a regular language to approximate one CFL and keep the other CFL intact. For example, we can regularize the CFL for field sensitivity (denoted as RL_f) and keep the CFL for context sensitivity (denoted as CFL_c) [12, 13]. Then, data-dependence analysis considering both kinds of sensitivity can be approximated as a CFL-reachability problem using $RL_f \cap CFL_c$. This paradigm can be extended to other constraints that can be represented as CFL reachability (e.g., synchronization sensitivity [26]). Of course, the generic algorithm by Melski and Reps [20] would typically not scale to large programs when using $RL_f \cap CFL_c$.

To extend our approach for field sensitivity, we can use a similar strategy: regularizing the CFL for field sensitivity to intersect it with our TAL in Definition 4.1 (denoted as TAL_c). Note that our TAL-reachability algorithm essentially provides a way to calculate the intersection of a regular language and a tree-adjoining language, as a regular language can be represented as paths in a graph. Similar to Melski and Reps's generic algorithm for CFL reachability, our generic algorithm for TAL reachability may also have scalability issues for large programs when using $RL_f \cap TAL_c$. One possible solution is to extend Kodumal and Aiken's regularly annotated set constraints [13] to TAL reachability.

7. Related Work

CFL Reachability. Based on the framework of CFL reachability proposed by Yannakakis [46], researchers used CFL reachability for various program-analysis problems, such as inter-procedural slicing [31], inter-procedural dataflow analysis [32], shape analysis [10, 28, 33], constant propagation [35] and pointer analysis [37, 45]. Researchers also investigated demand-driven algorithms for solving CFL-reachability problems [9, 38]. Reps [29] provided an early survey of program analysis based on CFL reachability. Rehof and Fähndrich [27] used CFL reachability to achieve an $O(n^3)$ algorithm for type-based flow analysis with polymorphic subtyping, which improves a previous $O(n^8)$ algorithm. Pratikakis et al. used CFL reachability for label-flow analysis [24] and race detection [25]. Zheng and Rugina [48] used CFL reachability for context-insensitive demand-driven alias analysis, which directly computes aliasing information without pre-computing points-to information. Chaudhuri [4] provided the first sub-cubic algorithm for CFL reachability. Some analyses require the consideration of satisfying multiple CFL reachability properties. However, it is undecidable considering two or more CFL reachability properties simultaneously [26, 30]. Therefore, it is needed to keep one CFL reachability property intact and approximate all the other CFL reachability properties as regular reachability properties [13, 37, 45]. In this paper, we propose TAL reachability via extending CFL reachability. TAL reachability allows us to calculate more intensive and compact summaries than CFL reachability for library code.

Mildly Context-Sensitive Grammars. As mentioned previously, researchers intensively investigated four formalisms for defining mildly context-sensitive grammars: tree-adjoining grammars [11], head grammars [23], linear indexed grammars [8], and combinatory categorial grammars [39]. According to [41], these four formalisms are actually four sets of different notations for defining the same family of mildly context-sensitive languages (i.e., the TAL family), which are parsable in $O(n^6)$ time (where n is the length of the sentence for parsing) [36]. In this paper, we provide a concise formalism, which is the Chomsky Normal Form for defining the TAL family. Compared with the four existing formalisms, our formalism is more suitable for defining TAL reachability, which has not been investigated previously. There are also another two less intensively investigated formalisms (i.e., linear context-free rewriting systems [42] and minimalist grammars [17]) for defining a larger language family than the TAL family.

Staged Program Analysis. Summary-based analysis belongs to a wider notion of staged program analysis, where multiple passes of program analysis cooperate with each other to achieve the purpose.

We are aware of two research efforts on summary-based program analysis that try to build summaries for library code with callbacks. Specifically, Lattner et al. [16] leverages heap cloning to achieve context-sensitive stages analysis. However, their approach sacrifices context-sensitivity when recursion exists, while our approach achieves full context-sensitivity with recursions. Madhavan et al. [19] provide a general framework to deal with callbacks. Due to its generality, it identifies abstract domain parts affected by callbacks, and left these parts for analyses when client code is available, while our approach is able to generate partial summaries for these parts with conditional reachability, and the partial summaries are validated to be effective in our experiments. Our conditional reachability is relevant to conditional dependencies, which were defined and studied by Komondoor and Ramalingam [14] in another scenario. But the conditions in conditional dependencies refer to existing branch predicates, while the conditions in conditional reachability in our paper refer to the unknown client code.

Furthermore, CFL reachability [32] naturally provides a basis of summarizing library code. Dillig et al. [7] proposed a summary-based flow-sensitive analysis (for verifying program memory safety properties), in which strong updates are considered in summary building. However, their analysis does not consider callbacks. Researchers have also considered summarizing library code to speed up dataflow analysis [34]. However, their approach is based on the "functional approach" proposed by Sharir and Pnueli [22], and is not solvable in polynomial-time of the program size [32]. In addition, their approach simply folds intra-procedural nodes or invocation nodes which transitively invoke only library functions (no callbacks). Our TAL-based analysis can also be applied the folded graph produced by their approach to make data-flow analysis even faster, since our approach can reduce the analysis cost globally, e.g., our algorithm can also fold invocation nodes that can transitively invoke client code (dealing with callbacks).

For demand-driven interprocedural analysis, the main disadvantage of top-down approaches lies in repeated analysis and the main disadvantage of bottom-up approaches lies in analyzing information unnecessary for the demand [47]. Our approach is essentially a bottom-up approach. However, since we target analyzing reachability relationships between all pairs of nodes (not specific pairs of nodes), we do not need to avoid analysis unnecessary for any specific demand. Thus, it is natural to use bottom-up approach here.

Unlike summary-based program analysis, where library code is distinguished from client code, other staged analyses use different passes on the same code base. Naik and Aiken [21] proposed a two-stage analysis, where conditional must-not-aliasing information is calculated to ensure sound race detection. Xu et al. [45] proposed a two-stage points-to analysis, which uses conditional must-not-aliasing information to accelerate CFL-reachability calculation. Chugh et al. [6] proposed to use static information (i.e., residual policies) calculated in the first pass to reduce the runtime overhead for analyzing dynamically generated code in JavaScript.

8. Conclusion and Future Work

In this paper, we propose a novel framework for program analysis called TAL reachability. Based on this framework, we further propose a novel technique to summarize library code with callbacks for context-sensitive data-dependency analysis. Since our technique calculates $conditional\ reachability$, we have more intensive summary information without further approximation.

We further experimentally evaluated our approach on a set of Java benchmark programs, and our experimental results demonstrate that, compared with a similar CFL-based approach, our

TAL-based approach is able to achieve a speed-up of over 8X for client-code analysis on average. The largest speed-up even reaches over 25X. Our TAL-based approach naturally concedes some slow-down for library-code summarization; however our experimental results demonstrate that the slow-down is typically acceptable.

Since TAL reachability provides a new framework for static program analysis, we believe that there will be a lot of future work on top of TAL reachability. In the near future, we plan to investigate the following issues. First, although our current research focuses on data-dependence analysis, TAL reachability is applicable to various other program-analysis problems. We plan to investigate these problems (e.g., points-to analysis [37, 44, 45]) using TAL reachability. Second, our current algorithm for TAL reachability is a generic algorithm; it does not consider the special characteristics of the graph. We plan to investigate faster TAL-reachability algorithms, especially for specific program-analysis problems. Third, our current research considers only context sensitivity in our analysis. We plan to further consider field sensitivity in our analysis, where a major issue is to scale the field-sensitive, context-sensitive library-code summarization using TAL reachability. Fourth, when analyzing libraries, it is possible that library l_1 invokes another library l_2 whose summaries has been computed. We plan to handle this such cases by feeding the summaries of l_2 for TAL-reachability analysis of the client library l_1.

Acknowledgment

The authors from Peking University are sponsored by the National 973 Program of China No. 2015CB352201, the National 863 Program of China No. 2013AA01A605, the Science Fund for Creative Research Groups of China No. 61121063, and the National Science Foundation of China No. 911118004, 61228203, and 61225007.

References

[1] GitHub Home. `https://github.com/`.

[2] SPECjvm2008 Benchmark Suite. `http://www.spec.org/jvm2008/`.

[3] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

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

[5] L. Chen, J. Qian, Y. Zhou, P. Wang, and B. Xu. Identifying extract class refactoring opportunities for internetware. *Science China Information Sciences*, 57(7):1–18, 2014.

[6] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proc. PLDI*, pages 50–62, 2009.

[7] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Proc. PLDI*, pages 567–577, 2011.

[8] G. Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94, 1988.

[9] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. FSE*, pages 104–115, 1995.

[10] S. Itzhaky, N. Bjørner, T. W. Reps, M. Sagiv, and A. V. Thakur. Property-directed shape analysis. In *Proc. CAV*, pages 35–51, 2014.

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

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

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

[14] R. Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *Proc. WCRE*, pages 110–119, 2007.

[15] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. POPL*, pages 207–218, 1981.

[16] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. PLDI*, pages 278–289, 2007.

[17] A. Lecomte and C. Retoré. Extending lambek grammars: a logical account of minimalist grammars. In *Proc. ACL*, pages 362–369, 2001.

[18] S. Litvak, N. Dor, R. Bodík, N. Rinetzky, and M. Sagiv. Field-sensitive program dependence analysis. In *Proc. FSE*, pages 287–296, 2010.

[19] R. Madhavan, G. Ramalingam, and K. Vaswani. Modular heap analysis for higher-order programs. In *Proc. SAS*, pages 370–387, 2012.

[20] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *TCS*, 248(1-2):29–98, 2000.

[21] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proc. POPL*, pages 327–338, 2007.

[22] M. Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.

[23] C. Pollard. *Gneralized Phrase Structure Grammars, Head Grammars and Natural Language*. PhD thesis, Stanford University, 1984.

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

[25] P. Pratikakis, J. S. Foster, and M. W. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proc. PLDI*, pages 320–331, 2006.

[26] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS*, 22(2):416–430, 2000.

[27] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *Proc. POPL*, pages 54–66, 2001.

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

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

[30] T. Reps. Undecidability of context-sensitive data-dependence analysis. *TOPLAS*, 22(1):162–186, 2000.

[31] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proc. FSE*, pages 11–20, 1994.

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

[33] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *Proc. ESOP*, pages 220–236, 2007.

[34] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *Proc. CC*, pages 2–16. Springer, 2006.

[35] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167(1-2): 131–170, 1996.

[36] Y. Schabes and K. Vijay-Shanker. Deterministic left to right parsing of tree adjoining languages. In *Proc. ACL*, pages 276–283, 1990.

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

[38] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proc. OOPSLA*, pages 57–76, 2005.

[39] M. Steedman. Combinators and grammars. In R. Oehrle, E. Bach, and D. Wheeler, editors, *Categorial Grammars and Natural Language Structures*, pages 417–442, 1988.

[40] C. Sun, N. Xi, S. Gao, Z. Chen, and J. Ma. Automated enforcement for relaxed information release with reference points. *Science China Information Sciences*, 57(11):1–19, 2014.

[41] K. Vijay-Shanker and D. J. Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27(6):511–546, 1994.

[42] D. Weir. *Characterizing mildly context-sensitive grammar formalisms*. PhD thesis, University of Pennsylvania, 1988.

[43] T. Xie, L. Zhang, X. Xiao, Y.-F. Xiong, and D. Hao. Cooperative software testing and analysis: Advances and challenges. *JCST*, 29(4): 713–723, 2014.

[44] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proc. ISSTA*, pages 225–235, 2008.

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

[46] M. Yannakakis. Graph-theoretic methods in database theory. In *Proc. PODS*, pages 230–242, 1990.

[47] X. Zhang, R. Mangal, M. Naik, and H. Yang. Hybrid top-down and bottom-up interprocedural analysis. In *Proc. PLDI*, pages 249–258, 2014.

[48] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *Proc. POPL*, pages 351–363, 2008.

Appendix

THEOREM 1. *Our formalism presented in Section 2 defines the same language family that TAGs define.*

In the following, we first present the formalism of Head Grammars [41], which is equivalent to the formalism of TAGs. Then, we prove that our formalism is equivalent to the formalism of Head Grammars, thus proving Theorem 1.

DEFINITION 1. *(**Head Grammars** [41]) A head grammar HG is a four tuple $G = (N, T, S, P)$, where N is a finite set of non-terminals, T is a finite set of terminals, $S \in N$ is the start symbol, and P is a finite set of productions of the form $A \rightarrow f(\sigma_1, \sigma_2, ..., \sigma_n)$, where $A \in N, n \geq 1, f \in \{W, C_{1,n}, C_{2,n}, ..., C_{n,n}\}$, $\sigma_1, \sigma_2, ..., \sigma_n \in N \cup (T^* \times T^*)$, and $n = 2$ when $f = W$. $C_{i,n}$ and W are defined as follows:*

- $C_{i,n} : (T^* \times T^*)^n \rightarrow (T^* \times T^*)$ *is a concatenation operation where* $C_{i,n}(u_1 \uparrow v_1, ..., u_i \uparrow v_i, ..., u_n \uparrow v_n) = u_1 v_1 ... u_i \uparrow v_i ... u_n v_n$
- $W : (T^* \times T^*)^2 \rightarrow (T^* \times T^*)$ *is the wrapping operation where* $W(u_1 \uparrow v_1, u_2 \uparrow v_2) = u_1 u_2 \uparrow v_2 v_1$

The language defined by HG is $\{\mu\nu \in T^* | S \Rightarrow \mu \uparrow \nu\}$. Non-terminals in a Head Grammar are actually equivalent to second-order non-terminals in our formalism; and there is an implicit de-pairing operation on the start symbol to form a string in the language defined by the Head Grammar.

We transform Theorem 1 into the following two lemmas:

LEMMA 1. *(**Backward Subsumption**) The language family defined by our formalism can be defined by the Head Grammars.*

LEMMA 2. *(**Forward Subsumption**) The language family defined by the Head Grammars can be defined by our formalism.*

Proof (*Lemma 1*). We show that all the production rules in our formalism can be expressed by the rules in Head Grammars:

Adjoining operator. For any two string pairs $\mathbb{A}$ and $\mathbb{B}$ in the form $\alpha \circ \beta$ and $\gamma \circ \delta$, $\Cap(\mathbb{A}, \mathbb{B}) = \alpha\gamma \circ \delta\beta$. This can be exactly expressed by the W operator in Head Grammars, since $W(\alpha \uparrow \beta, \gamma \uparrow \delta) = \alpha\gamma \uparrow \delta\beta$[8].

Extended concatenation operators. For a string pair in the form of $\alpha \circ \beta$ and a string in the form of γ, all the four extended concatenation operators can be expressed using the W or $C_{i,n}$ operator in Head Grammars:

- $\curvearrowleft\!\!\cap(\alpha \circ \beta, \gamma) = \gamma\alpha \circ \beta$. This can be expressed by $C_{2,2}(\epsilon \uparrow \gamma, \alpha \uparrow \beta)$, which will produce $\epsilon\gamma\alpha \uparrow \beta$, and finally $\gamma\alpha \uparrow \beta$.
- $\curvearrowright\!\!\cap(\alpha \circ \beta, \gamma) = \alpha\gamma \circ \beta$. This can be expressed by $W(\alpha \uparrow \beta, \gamma \uparrow \epsilon)$, which will produce $\alpha\gamma \uparrow \epsilon\beta$, and finally $\alpha\gamma \uparrow \beta$.
- $\curvearrowright\!\!\cap(\alpha \circ \beta, \gamma) = \alpha \circ \gamma\beta$. This can be expressed by $W(\alpha \uparrow \beta, \epsilon \uparrow \gamma)$, which will produce $\alpha\epsilon \uparrow \gamma\beta$, and finally $\alpha \uparrow \gamma\beta$.
- $\cap\!\!\curvearrowright(\alpha \circ \beta, \gamma) = \alpha \circ \beta\gamma$. This can be expressed by $C_{1,2}(\alpha \uparrow \beta, \epsilon \uparrow \gamma)$, which will produce $\alpha \uparrow \beta\epsilon\gamma$, and finally $\alpha \uparrow \beta\gamma$.

Concatenation operator. Although the formalism of Head Grammars contains only our second-order non-terminals, it can simulate the traditional concatenation operator as follows. We can use a non-terminal in Head Grammars representing $\alpha \uparrow \epsilon$ to simulate a non-terminal in CFGs (i.e., a first-order non-terminal in our formalism). Then the W operator serves as the traditional concatenation operator for two such non-terminals.

Pairing operator. Let A and B be simulated by $\alpha \uparrow \epsilon$ and $\beta \uparrow \epsilon$, respectively. We can simulate $\oplus(A, B)$ as $C_{1,2}(\alpha \uparrow \epsilon, \beta \uparrow \epsilon)$.

De-pairing operator. In Head Grammars, there is actually an implicit de-pairing operator on the start symbol. Our formalism allows a first-order non-terminal de-paired from a second-order non-terminal to concatenate with other second-order non-terminals. The $C_{i,n}$ operator can simulate this situation. For example, de-pairing $\alpha \circ \beta$ can be expressed by $C_{2,2}(\alpha \uparrow \beta, \epsilon \uparrow \epsilon)$ since it will produce $\alpha\beta \uparrow \epsilon$. Note that we can use the $C_{i,n}$ operator together with $\epsilon\uparrow\epsilon$ and the W operator to manipulate the concatenation position. For example, $W(\alpha\uparrow\beta, C_{2,2}(\gamma \uparrow \delta, \epsilon \uparrow \epsilon))$ results in $\alpha\gamma\delta \uparrow \beta$.

Therefore, all the production rules in our formalism can be expressed by the formalism of Head Grammars. □

Proof (*Lemma 2*). We further show that all the production rules in Head Grammars can be expressed by our formalism.

W operator. As shown above, the W operator in Head Grammars is equivalent to our adjoining operator $\Cap$, thus can be expressed by our formalism.

$C_{i,n}$ operator. We apply mathematical induction on n ($n \geq 1$), and i can be arbitrary integer within the range $[1, n]$.

- Case-1: $n = 1$, i can only be 1:
  - $i = 1$: $C_{1,1}(u_1 \uparrow v_1) = u_1 \uparrow v_1$. This is the identity production rule and is trivially supported by our formalism.
- Case-2: $n = 2$, i can be 1 or 2:
  - $i = 1$: $C_{1,2}(u_1 \uparrow v_1, u_2 \uparrow v_2) = u_1 \uparrow v_1 u_2 v_2$. This can be expressed by our formalism as: $\cap\!\!\curvearrowright(u_1 \circ v_1, \ominus(u_2 \circ v_2))$.
  - $i = 2$: $C_{2,2}(u_1 \uparrow v_1, u_2 \uparrow v_2) = u_1 v_1 u_2 \uparrow v_2$. This can be expressed by our formalism as: $\curvearrowleft\!\!\cap(u_2 \circ v_2, \ominus(u_1 \circ v_1))$.
- Now we assume $C_{i,n}$ can be expressed by our formalism when $n = k$ ($k \geq 1$) for any $i \in [1, k]$, then for $n = k + 1$ and $i = i'$, $i' \in [1, k + 1]$:

$$\begin{aligned} & C_{i',k+1}(u_1 \uparrow v_1, ..., u_{i'} \uparrow v_{i'}, ..., u_{k+1} \uparrow v_{k+1}) \\ = \; & u_1 v_1 ... u_{i'} \uparrow v_{i'} ... u_{k+1} v_{k+1} \\ = \; & C_{2,2}(u_1 \uparrow v_1, C_{i'-1,k}(u_2 \uparrow v_2, ..., u_{i'} \uparrow v_{i'}, ..., u_{k+1} \uparrow v_{k+1})) \end{aligned}$$

  According to Case-2, $C_{2,2}$ can be expressed by our formalism. In addition,$i' - 1$ is less than or equal to k. Then, according to our inductive assumption, $C_{i'-1,k}$ can also be expressed by our formalism. Thus, $C_{i,k+1}$ can be expressed by our formalism.

Based on mathematical induction, $C_{i,n}(n \geq 1,\ 1 \leq i \leq n)$ can be expressed by our formalism. □

[8] Note that the $\uparrow$ operator in Head Grammars is equivalent to our $\circ$ operator.