Undecidability of Context-Sensitive Data-Dependence Analysis

THOMAS REPS
University of Wisconsin

A number of program-analysis problems can be tackled by transforming them into certain kinds of graph-reachability problems in labeled directed graphs. The edge labels can be used to filter out paths that are not of interest: A path P from vertex s to vertex t only counts as a "valid connection" between s and t if the word spelled out by P is in a certain language. Often the languages used for such filtering purposes are languages of matching parentheses:

- In some cases, the matched-parenthesis condition is used to filter out paths with mismatched calls and returns. This leads to so-called "context-sensitive" program analyses, such as context-sensitive interprocedural slicing and context-sensitive interprocedural dataflow analysis.
- In other cases, the matched-parenthesis condition is used to capture a graph-theoretic analog of McCarthy's rules: "`car(cons(x,y)) = x`" and "`cdr(cons(x,y)) = y`". That is, in the code fragment

```
c = cons(a,b);
d = car(c);
```

  the fact that there is a "structure-transmitted data dependence" from `a` to `d`, but not from `b` to `d`, is captured in a graph by using parentheses that match as the labels on edges that run from `a` to `c` and `c` to `d`, and using parentheses that do not match as the labels on edges that run from `b` to `c` and `c` to `d`.

However, structure-transmitted data-dependence analysis is context-*insensitive*, because there are no constraints that filter out paths with mismatched calls and returns. Thus, a natural question is whether these two kinds of uses of parentheses can be combined to create a context-*sensitive* analysis for structure-transmitted data dependences. This paper answers the question in the negative: In general, the problem of context-sensitive, structure-transmitted data-dependence analysis is undecidable.

The results of this paper imply that, in general, both context-sensitive set-based analysis and ∞-CFA (when data constructors and selectors are taken into account) are also undecidable.

CR Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*computability theory*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*decision problems*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*path and circuit problems*;

General Terms: Languages, Theory

Additional Key Words and Phrases: Context-sensitive program analysis, graph-reachability problem, structure-transmitted data dependence, set-based analysis, set constraints, control-flow analysis, ∞-CFA, linear matched-parenthesis language

This work was supported in part by the National Science Foundation under grants CCR-9625667 and CCR-9619219, by the United States-Israel Binational Science Foundation under grant 96-00337, by a grant from IBM, and by a Vilas Associate Award from the University of Wisconsin.

Author's address: Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.
E-mail: reps@cs.wisc.edu.

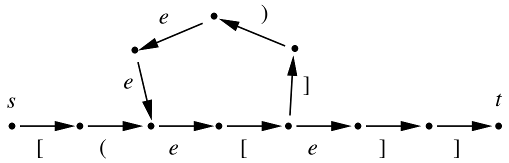
1. INTRODUCTION

A number of program-analysis problems can be tackled by transforming them into certain kinds of graph-reachability problems in labeled directed graphs [19,8,5,4,6,14,18,24,26,15,25,22,27]. It is useful to consider not just ordinary reachability (*e.g.*, transitive closure), but a generalization in which the edge labels are used, in effect, to filter out paths that are not of interest [4,14,24,26,15,25,22,27]: A path P from vertex s to vertex t only counts as a "valid connection" between s and t if the word spelled out by P (*i.e.*, the concatenation, in order, of the labels on the edges of P) is in a certain language.

Definition 1.1. Let L be a language over alphabet Σ_L, and let G be a graph whose edges are labeled with members of Σ_L. Each path in G defines a word over Σ_L, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an *L-path* if its word is a member of L. The (single-source/single-target) *L-path problem* is to determine whether there exists an L-path in G from a given source vertex s to a given target vertex t.

Let $\mathcal{L}$ be a family of languages, and $L \in \mathcal{L}$ be a language over alphabet Σ_L. An *instance of the $\mathcal{L}$-reachability problem* is an L-path problem $< L, \Sigma_L, G, s, t >$. □

Example. When the family of languages $\mathcal{L}$ in Def. 1.1 is the context-free languages, we have the *CFL-reachability* problem [32]. Consider the graph and the context-free grammar shown below. Note that $L(matched)$ is the context-free language that consists of strings of matched parentheses and square brackets, with zero or more e's interspersed.



$$
\begin{array}{lcl}
\textit{matched} & \rightarrow & \textit{matched} \;\; \textit{matched} \\
 & | & (\;\; \textit{matched} \;\;) \\
 & | & [\;\; \textit{matched} \;\;] \\
 & | & e \\
 & | & \varepsilon
\end{array}
$$

In this graph, there is exactly one $L(matched)$-path from s to t: The path goes exactly once around the cycle, and generates the word "$[(e[])eee[e]]$". □

A number of program-analysis problems can be viewed as instances of the CFL-reachability problem [27]. In program-analysis problems, the languages used for such filtering purposes are often languages of matching parentheses. In some cases, the matched-parenthesis condition is used to filter out paths with mismatched calls and returns in order to implement so-called "context-sensitive" program analyses.

Example 1.2. The use of CFL-reachability in context-sensitive program analysis, as opposed to ordinary graph reachability, is illustrated by the following example:[1]

---

[1]In this example, we use C syntax; in later examples, we use C augmented with the operator `cons`, which denotes a pairing constructor, and the operators `car` and `cdr`, which select the first and second components of a pair, respectively. This notation is used to simplify the way storage-allocation operations are expressed in our examples. The use of these operators does *not* imply that our results apply only to the analysis of LISP programs.

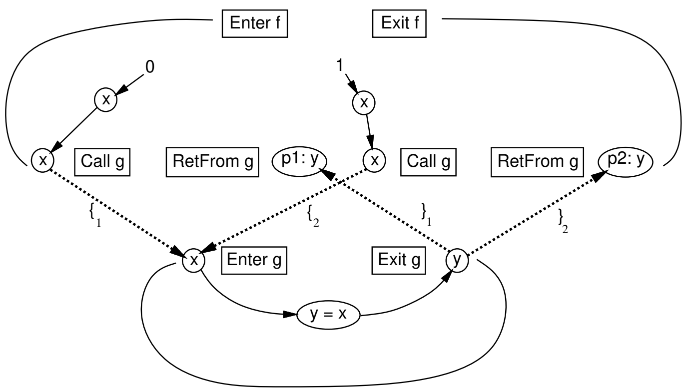
Source Code

```
int x, y;

void g() {
     y = x;
}

void f() {
     x = 0;
     g();    /* Call site 1 */
 p1: /* Could y be 0? 1? */
     x = 1;
     g();    /* Call site 2 */
 p2: /* Could y be 0? 1? */
}
```

Corresponding Data-Dependence Graph



The vertices in the graph labeled with variables or assignment statements represent the values of variables at various points in the program. (The rectangular boxes labeled `Call`, `RetFrom`, `Enter`, and `Exit` indicate various control points by which control is transferred from caller to callee and vice versa; they are included in the graph to indicate the sites in the program to which the various variable-vertices are associated.) The edges in the graph represent data dependences. For instance, the edge

$$\texttt{y = x} \longrightarrow \texttt{y}$$

in the dependence graph for procedure `g` indicates that the value of `y` on exit from `g` could be (and, in fact, must be) the value assigned to variable `y` by the statement `y = x`.

Note that data-dependence edges for dependences transmitted from the caller to the callee (*i.e.*, from `f` to `g`) are labeled by the symbols "$\{_1$" and "$\{_2$", whereas data-dependence edges for dependences transmitted from the callee back to the caller are labeled by the symbols "$\}_1$" and "$\}_2$". In particular, the data-dependence edge from the vertex that represents `x` before the first call on `g` to the vertex for `x` on entry to `g` is labeled by $\{_1$; the data-dependence edge from the vertex that represents `x` before the second call on `g` to the vertex for `x` on entry to `g` is labeled by $\{_2$; and the data-dependence edges from the vertex that represents `y` on exit from `g` to the vertices `p1:y` and `p2:y` are labeled with $\}_1$ and $\}_2$, respectively.

A context-insensitive analysis that tracks dependences between constants and variables in the program will report that `y` could have either the value 0 or 1 at `p1`, and either 0 or 1 at `p2`. The reason is that a context-insensitive analysis *ignores* the fact that the only paths that can possibly be feasible execution paths are those in which returns are matched with corresponding calls. For instance, the existence of the (mismatched) path

$$0 \longrightarrow \texttt{x} \longrightarrow \texttt{x} \xrightarrow{\{_1} \texttt{x} \longrightarrow \texttt{y = x} \longrightarrow \texttt{y} \xrightarrow{\}_2} \texttt{p2:y} \tag{1.3}$$

in the graph shown above serves as "evidence" that the value of `y` at `p2` could be 0.

In contrast, a context-sensitive analysis only reports possible transmissions of values along paths in which returns are matched with corresponding calls. For this example, it reports that `y` could have only the value 0 at `p1` and 1 at `p2`. The context-sensitive analysis can be expressed as a CFL-reachability problem with respect to a language of matched indexed curly braces. It would say that path (1.3) is not a valid connection between 0 and `p2:y`, because the label $\{_1$ on the edge that represents the data dependence between `x` before the first call on `g` and `x` at the beginning of `g` does not match the label $\}_2$ on the edge that represents the data dependence between `y` at the end of a call on `g` and `p2:y`. On the other hand, because $\{_2$ matches with $\}_2$, the (matched) path

$$1 \longrightarrow \texttt{x} \longrightarrow \texttt{x} \xrightarrow{\{_2} \texttt{x} \longrightarrow \texttt{y = x} \longrightarrow \texttt{y} \xrightarrow{\}_2} \texttt{p2:y} \qquad (1.3)$$

does count as a valid connection between 1 and `p2:y`, and this path serves as evidence that the value of `y` at `p2` could be 1. □

Problems in which CFL-reachability has been used to devise context-sensitive program analyses include interprocedural slicing [14,24] and interprocedural dataflow analysis [26,15].

CFL-reachability—and, in particular, a matched-parenthesis constraint—has also been used to capture a graph-theoretic analog of McCarthy's rules [21]: "`car(cons(x,y)) = x`" and "`cdr(cons(x,y)) = y`" [25,22].

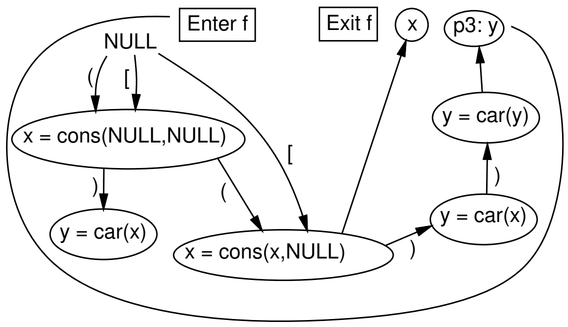
Example 1.5. Consider the following example program, which serves to illustrate the use of CFL-reachability in context-insensitive, structure-transmitted data-dependence analysis:

Source Code

```
List *x, *y;

void f() {
     x = cons(NULL,NULL);
     y = car(x);
 p1: /* Could y be NULL here? */
     x = cons(x,NULL);
     y = car(x);
 p2: /* Could y be NULL here? */
     y = car(y);
 p3: /* Could y be NULL here? */
}
```

Corresponding Data-Dependence Graph



In this graph, there are also labels on the data-dependence edges, but here they serve a different purpose than the labels used in Example 1.2: Here an edge labeled "(" corresponds to a data construction in which the value is placed in the first position of a `cons`; an edge labeled "[" corresponds to a data construction in which the value is placed in the second position of a `cons`; an edge labeled ")" corresponds to a selection via `car`; an edge labeled "]" corresponds to a selection via `cdr`.

The matched-parenthesis path

$$\texttt{NULL} \xrightarrow{(} \texttt{x=cons(NULL,NULL)} \xrightarrow{(} \texttt{x=cons(x,NULL)}$$
$$\xrightarrow{)} \texttt{y=car(x)} \xrightarrow{)} \texttt{y=car(y)} \longrightarrow \texttt{p3:y}$$

from `NULL` to program point `p3` serves as evidence that the value of `y` at `p3` could be `NULL`. The path

$$\texttt{NULL} \xrightarrow{(} \texttt{x=cons(NULL,NULL)} \xrightarrow{(} \texttt{x=cons(x,NULL)} \xrightarrow{)} \texttt{y=car(x)}$$

does not serve as evidence that the value of `y` at `p2` could be `NULL`, because the first occurrence of "(" has no matching ")". In fact, there is *no* matched-parenthesis path from `NULL` to the node `y=car(x)` in the lower right-hand corner of the above graph, and a (context-insensitive) structure-transmitted data-dependence analysis would conclude that the value of `y` at `p2` cannot be `NULL`. □

The structure-transmitted data-dependence analyses given in [25] and [22] are context-*insensitive*, because there are no constraints that filter out paths with mismatched calls and returns. Thus, a natural question is whether the two kinds of uses of matching delimiters illustrated in Examples 1.2 and 1.5 can be combined to create a context-*sensitive* analysis for structure-transmitted data dependences. The following interprocedural variation on Example 1.5 illustrates what we would hope to gain from a context-sensitive, structure-transmitted data-dependence analysis:

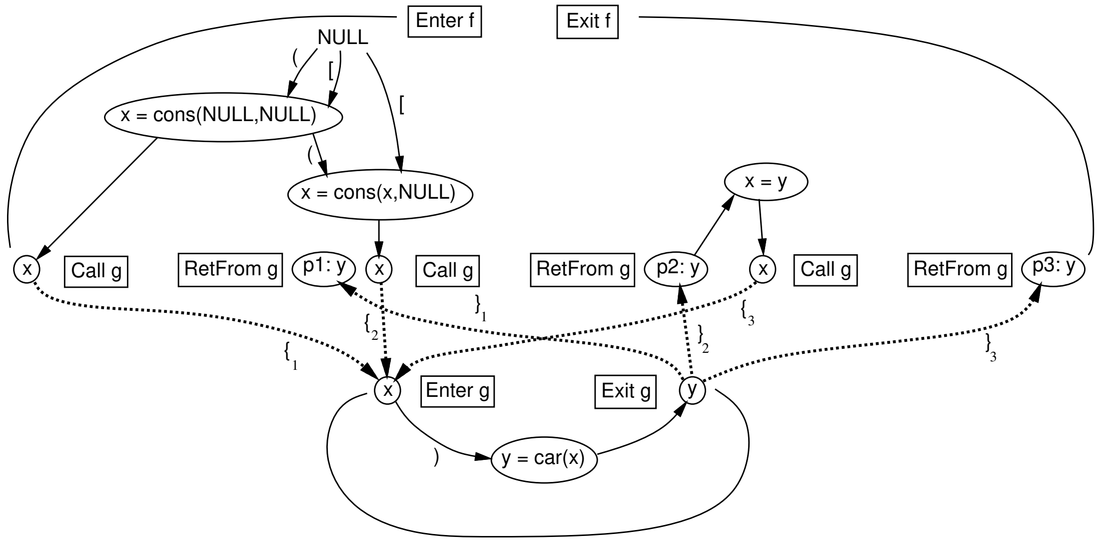
Example 1.6. Consider the following example program:

```
List *x, *y;

void g() {
     y = car(x);
}

void f() {
     x = cons(NULL,NULL);
     g();
 p1: /* Could y be NULL here? */
     x = cons(x,NULL);
     g();
 p2: /* Could y be NULL here? */
     x = y;
     g();
 p3: /* Could y be NULL here? */
}
```

This program has the following data-dependence graph:



For this example, a context-sensitive analysis should report that variable `y` can have the value `NULL` at `p1` and `p3`, but not at `p2`. The following path from `NULL` to program point `p3` serves as evidence that the value of `y` at `p3` could be `NULL`:

$$\texttt{NULL} \xrightarrow{(} \texttt{x=cons(NULL,NULL)} \xrightarrow{(} \texttt{x=cons(x,NULL)} \longrightarrow \texttt{x}$$

$$\xrightarrow{\{_2} \texttt{x} \xrightarrow{)} \texttt{y=car(x)} \longrightarrow \texttt{y} \xrightarrow{\}_2} \texttt{p2:y} \longrightarrow \texttt{x=y} \longrightarrow \texttt{x}$$

$$\xrightarrow{\{_3} \texttt{x} \xrightarrow{)} \texttt{y=car(x)} \longrightarrow \texttt{y} \xrightarrow{\}_3} \texttt{p3:y}$$

In contrast, a context-insensitive analysis would also use the following path from `NULL` to program point `p2` as evidence that the value of `y` at `p2` could be `NULL`:

$$\texttt{NULL} \xrightarrow{(} \texttt{x=cons(NULL,NULL)} \xrightarrow{(} \texttt{x=cons(x,NULL)} \longrightarrow \texttt{x}$$
$$\xrightarrow{\{_2} \texttt{x} \xrightarrow{)} \texttt{y=car(x)} \longrightarrow \texttt{y} \xrightarrow{\}_2} \texttt{p2:y} \longrightarrow \texttt{x=y} \longrightarrow \texttt{x}$$
$$\xrightarrow{\{_3} \texttt{x} \xrightarrow{)} \texttt{y=car(x)} \longrightarrow \texttt{y} \xrightarrow{\}_2} \texttt{p2:y}$$

The problem with this path is that there is a mismatch between the call edge $\texttt{x} \xrightarrow{\{_3} \texttt{x}$ and the return edge $\texttt{y} \xrightarrow{\}_2} \texttt{p2:y}$. Consequently, this path would be excluded from consideration by a context-sensitive analysis. □

The other fact to note about Example 1.6 is that the path used by the (hypothetical) context-sensitive analysis has two *interleaved* patterns of matched symbols: "(" symbols match with ")" symbols, and "$\{_i$" symbols match with "$\}_i$" symbols; however, the two patterns of matched symbols are interleaved.[2] This observation serves to motivate the study of interleaved matched-parenthesis languages carried out in Sections 2 and 3.[3]

This paper shows that it is impossible to create an algorithm that captures all, and only, such interleaved matched-parenthesis paths: In general, the problem of context-sensitive, structure-transmitted data-dependence analysis is undecidable. In particular, the paper shows that L-reachability in graphs, where L is a language of strings formed by interleaving two matched-parenthesis languages in a certain way, is undecidable. In other words, you can capture either (i) the matching of call edges and return edges (to reflect the fact that when a procedure finishes it returns to the site of the most recent call), or (ii) "`car(cons(x,y) = x` cancellation", but not both simultaneously, in any amount of time. (Of course, there may be useful algorithms that compute approximate, but safe, solutions to the L-reachability problem for this language L, *cf.* [12].)

In terms of the programming-language features needed for this result to apply, higher-order functions are not required: The main result of this paper implies that context-sensitive, structure-transmitted data-dependence analysis is undecidable for *first-order* languages (both functional and imperative).

The remainder of this paper is organized into four sections: Our undecidability result is shown by a reduction from a variant of Post's Correspondence Problem (PCP); Section 2 defines PCP and discusses a variant of it that is particularly suited to our needs. Example 1.6 motivates our interest in languages with interleaved patterns of matching delimiters; Section 3 shows that a certain L-reachability problem—where L is a language of strings formed by interleaving two languages of matching parentheses—is undecidable. Section 4 relates the result from Section 3 to the undecidability of context-sensitive, structure-transmitted

---

[2] In this paper, the term "interleaved" is used in a somewhat restricted sense, compared to the standard usage in formal-language theory (*cf.* [13, pp. 282]). The exact nature in which patterns of matched symbols are allowed to be woven together is defined precisely in Section 3.

[3] The above observation also provides a hint that our program-analysis problem may be undecidable:

- As we will see in Section 3, a natural way to express a language with interleaved patterns of matched delimiters is as the intersection of two (deterministic) context-free languages.
- An L-reachability problem in a graph can be viewed as a problem of determining whether the intersection of L with a regular language is empty (where the regular language in question is related to the language of paths in the graph). Consequently, context-sensitive, structure-transmitted data-dependence analysis is really a problem of determining whether the intersection of two (deterministic) context-free languages and a regular language is empty.
- The context-free languages are closed under intersection with regular languages.
- The general question of whether the intersection of two (deterministic) context-free languages is empty is undecidable [13, pp. 247].

However, because we are dealing with the intersection of two matched-parenthesis languages (*i.e.*, the intersection of two languages in a very restricted subset of the deterministic context-free languages) this line of reasoning only suggests—but does not prove—that it might not be possible to have an algorithm for context-sensitive, structure-transmitted data-dependence analysis.

data-dependence analysis. Section 5 discusses what our results imply about other program-analysis problems.

2. A VARIANT OF POST'S CORRESPONDENCE PROBLEM

Our undecidability result is shown by a reduction from a variant of Post's Correspondence Problem:

Definition 2.1. An instance of *Post's Correspondence Problem*, or PCP, consists of two lists of strings, X and Y, where X and Y each consist of k strings in $\{0, 1\}^+$:

$$X = x_1, x_2, \cdots, x_k$$
$$Y = y_1, y_2, \cdots, y_k$$

The instance of PCP has a solution if there exists a nonempty sequence of integers $i_1, i_2, \cdots, i_j, \cdots, i_m$ such that, for all $1 \le j \le m$,

$$1 \le i_j \le k \text{ and } x_{i_1}\, x_{i_2} \cdots x_{i_j} \cdots x_{i_m} = y_{i_1}\, y_{i_2} \cdots y_{i_j} \cdots y_{i_m}.$$

□

Example 2.2. Consider the following instance of PCP, where k is 3:

$$X = 0101,\ 101,\ 111$$
$$Y = 01,\ 011,\ 0111101$$

This instance of PCP has the solution 1, 2, 3, 1 because

$$x_1\, x_2\, x_3\, x_1 = 01011011110101 = y_1\, y_2\, y_3\, y_1.$$

□

PCP is known to be undecidable; for proofs, see Hopcroft and Ullman [13, pp. 193-198], Lewis and Papadimitriou [20, pp. 289-293], or Harrison [7, pp. 249-256].

For our purposes, it is more convenient to work with the following variant of PCP:

Definition 2.3. (Parenthesis-PCP) Given an instance of PCP,

$$X = x_1, x_2, \cdots, x_k$$
$$Y = y_1, y_2, \cdots, y_k$$

we define the corresponding instance of *parenthesis-PCP*, or P-PCP, as

$$\bar{X} = \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_k$$
$$\bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \cdots, \bar{y}_k^R$$

where, for $1 \le i \le k$,

- $\bar{x}_i$ is the string in $\{\, (, [\, \}^+$ equal to x_i with 0 replaced by "(" and 1 replaced by "[".
- $\bar{y}_i$ is the string in $\{\,),]\, \}^+$ equal to y_i with 0 replaced by ")" and 1 replaced by "]".
- The superscript "R" on a string denotes the reversed string.

A solution to an instance of P-PCP is defined with the aid of the following linear context-free grammar:[4]

$$
\begin{array}{lcl}
\mathit{balanced} & \longrightarrow & (\ \mathit{balanced}\ ) \\
 & | & [\ \mathit{balanced}\ ] \\
 & | & (\ \#\ ) \\
 & | & [\ \#\ ]
\end{array}
$$

An instance of P-PCP has a solution if there exists a nonempty sequence of integers $i_1, i_2, \cdots, i_j, \cdots, i_m$ such that, for all $1 \le j \le m$,

[4]A *linear* context-free grammar is one in which at most one nonterminal appears on the right-hand side of each production.

$$1 \le i_j \le k \text{ and } \bar{x}_{i_1}\, \bar{x}_{i_2} \cdots \bar{x}_{i_j} \cdots \bar{x}_{i_m} \,\#\, \bar{y}^R_{i_m} \cdots \bar{y}^R_{i_j} \cdots \bar{y}^R_{i_2}\, \bar{y}^R_{i_1} \in L(\mathit{balanced}).$$

□

Example 2.4. The instance of P-PCP that corresponds to Example 2.2 is

$$\begin{array}{lcl} \bar{X} & = & ([([,\quad [([,\quad [[[\\ \bar{Y}^R & = &)],\qquad)]],\quad)]]]])] \end{array}$$

This instance of P-PCP has the solution 1, 2, 3, 1 because

$$x_1\, x_2\, x_3\, x_1 \,\#\, y^R_1\, y^R_3\, y^R_2\, y^R_1 = ([([[([[[[([([\#])])]]]])]])]) \in L(\mathit{balanced}).$$

□

Clearly an instance of PCP has a solution if and only if the corresponding instance of P-PCP has a solution.

For our purposes, it is convenient to define two additional concepts:

Definition 2.5. If an instance of P-PCP

$$\begin{array}{lcl} \bar{X} & = & \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_k \\ \bar{Y}^R & = & \bar{y}^R_1, \bar{y}^R_2, \cdots, \bar{y}^R_k \end{array}$$

has a solution $i_1, i_2, \cdots, i_m$, we say that the following string *exhibits the solution in tagged form*:[5]

$$\{_{i_1}\, \bar{x}_{i_1}\, \{_{i_2}\, \bar{x}_{i_2} \cdots \{_{i_m}\, \bar{x}_{i_m} \,\#\, \bar{y}^R_{i_m}\, \}_{i_m} \cdots \bar{y}^R_{i_2}\, \}_{i_2}\, \bar{y}^R_{i_1}\, \}_{i_1}. \tag{2.6}$$

In general, suppose that $i_1, i_2, \cdots, i_j, \cdots, i_p$, is some nonempty sequence of integers such that, for all $1 \le j \le p$, $1 \le i_j \le k$. Regardless of whether $i_1, i_2, \cdots, i_j, \cdots, i_p$ is actually a solution to the instance of P-PCP, we say that a string of the form

$$\{_{i_1}\, \bar{x}_{i_1}\, \{_{i_2}\, \bar{x}_{i_2} \cdots \{_{i_j}\, \bar{x}_{i_j} \cdots \{_{i_p}\, \bar{x}_{i_p} \,\#\, \bar{y}^R_{i_p}\, \}_{i_p} \cdots \bar{y}^R_{i_j}\, \}_{i_j} \cdots \bar{y}^R_{i_2}\, \}_{i_2}\, \bar{y}^R_{i_1}\, \}_{i_1}$$

exhibits a candidate solution in tagged form. □

For instance, because the sequence 2, 1, 2, 3, 1 is not a solution to Example 2.4, the following string exhibits a candidate solution in tagged form (but *not* a solution in tagged form):

$$\{_2 [([\{_1 ([([\{_2 [([\{_3 [[[\{_1 ([([\#]) \}_1])]]]]) \}_3]]) \}_2]) \}_1]]) \}_2. \tag{2.7}$$

In contrast, the following string exhibits a candidate solution to Example 2.4 in tagged form that is also a solution in tagged form:

$$\{_1 ([([\{_2 [([\{_3 [[[\{_1 ([([\#]) \}_1])]]]]) \}_3]]) \}_2]) \}_1. \tag{2.8}$$

3. AN UNDECIDABLE L-REACHABILITY PROBLEM

In this section, we show that a certain L-reachability problem—where L is a language of strings formed by interleaving two linear languages of matching parentheses—is undecidable. Throughout the remainder of the paper, we assume that we have been given a fixed, but arbitrary, instance of P-PCP consisting of the k pairs of strings

$$\begin{array}{lcl} \bar{X} & = & \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_k \\ \bar{Y}^R & = & \bar{y}^R_1, \bar{y}^R_2, \cdots, \bar{y}^R_k \end{array}$$

Here our interest in interleaved matched-parenthesis languages stems from the fact that a string that exhibits a P-PCP solution in tagged form has two interleaved patterns of matched delimiters:

---

[5]For technical reasons having to do with the details of the construction given in Section 3, we will also work with a version of (2.6) in which each occurrence of "{ " is immediately preceded by two occurrences of the symbol e, and each occurrence of " }" is immediately followed by two occurrences of e.

(i) The string $\bar{x}_{i_1}\ \bar{x}_{i_2} \cdots \bar{x}_{i_j} \cdots \bar{x}_{i_m}\ \#\ \bar{y}^R_{i_m} \cdots \bar{y}^R_{i_j} \cdots \bar{y}^R_{i_2}\ \bar{y}^R_{i_1}$ is in $L(\mathit{balanced})$.

(ii) The string $\{_{i_1}\ \{_{i_2} \cdots \{_{i_j} \cdots \{_{i_m}\ \#\ \}_{i_m} \cdots \}_{i_j} \cdots \}_{i_2}\ \}_{i_1}$ is in $L(\mathit{balanced}')$, the language of balanced strings made up of $\{_i$ and $\}_i$, for $1 \le i \le k$, defined by the following linear context-free grammar:

$$
\begin{array}{rcll}
\mathit{balanced}' & \longrightarrow & \{_i\ \ \mathit{balanced}'\ \ \}_i & \text{for } 1 \le i \le k \\
 & | & \{_i\ \ \#\ \ \}_i & \text{for } 1 \le i \le k
\end{array}
$$

It is important to note that, in general, in a string that exhibits a P-PCP solution in tagged form, *the two balancing processes can be out of sync*. For instance, in the following prefix of string (2.8)

$$\{_1 ([([\ \{_2 [([\ \{_3 [[[\ \{_1 ([([\#])\ \}_1$$

the last symbol, namely "$\}_1$", does not match with the seventh-from-last symbol, namely "[". However, we can capture the structure of P-PCP solutions in tagged form via the intersection of two linear context-free languages:

(i) The language $L(S_1)$ consists of strings of $L(\mathit{balanced})$ with an arbitrary number of symbols of the form "$\{_i$" interspersed among the open-parenthesis and open-bracket symbols, and an arbitrary number of symbols of the form "$\}_i$" interspersed among the close-parenthesis and close-bracket symbols:

$$
\begin{array}{rcl}
S_1 & \longrightarrow & O\ (\ O\ S_1\ C\ )\ C \\
 & | & O\ [\ O\ S_1\ C\ ]\ C \\
 & | & O\ (\ O\ \#\ C\ )\ C \\
 & | & O\ [\ O\ \#\ C\ ]\ C
\end{array}
$$

$$
\begin{array}{rcll}
O & \longrightarrow & \{_i\ O & \text{for } 1 \le i \le k \\
 & | & \varepsilon &
\end{array}
$$

$$
\begin{array}{rcll}
C & \longrightarrow & \}_i\ C & \text{for } 1 \le i \le k \\
 & | & \varepsilon &
\end{array}
$$

(ii) The language $L(S_2)$ consists of all possible candidate solutions, in tagged form, to the given instance of P-PCP:

$$
\begin{array}{rcll}
S_2 & \longrightarrow & \{_i\ \ \bar{x}_i\ \ S_2\ \ \bar{y}_i^R\ \ \}_i & \text{for } 1 \le i \le k \\
 & | & \{_i\ \ \bar{x}_i\ \ \#\ \ \bar{y}_i^R\ \ \}_i & \text{for } 1 \le i \le k
\end{array}
$$

Languages $L(S_1)$ and $L(S_2)$ capture the two interleaved patterns of matched delimiters noted above: $L(S_2)$ consists of all possible candidate solutions, in tagged form, to the given instance of P-PCP; $L(S_1)$ consists of all strings such that when all "$\{_i$" and "$\}_i$" symbols are excluded, we are left with a string in $L(\mathit{balanced})$. Consequently, $L(S_1) \cap L(S_2)$ consists of exactly the solutions, in tagged form, to the given instance of P-PCP. For example, for Example 2.4, strings (2.7) and (2.8) are both in $L(S_2)$, but only string (2.8) is in $L(S_1)$; that is,

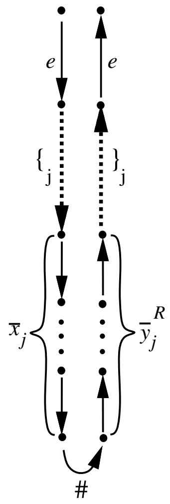
$$\{_1 ([([\ \{_2 [([\ \{_3 [[[\ \{_1 ([([\#])\ \}_1])]]]])\ \}_3]])\ \}_2])\ \}_1 \ \in\ L(S_1) \cap L(S_2).$$

Let $\mathcal{L}_{P\text{-}PCP}$ denote the family of languages $L(S_1) \cap L(S_2)$ that arise from instances of P-PCP.

THEOREM 3.1. *$\mathcal{L}_{P\text{-}PCP}$-reachability is undecidable.*

PROOF. For any instance of P-PCP, we show how to construct a graph (with two distinguished vertices, s and t) such that there is an $L(S_1) \cap L(S_2)$-path from s to t if and only if the instance of P-PCP has a solution. Thus, if $\mathcal{L}_{P\text{-}PCP}$-reachability were decidable, this construction, followed by the application of the algorithm for solving $L(S_1) \cap L(S_2)$-reachability, would provide a decision procedure for P-PCP (and hence for PCP).

Fig. 1 shows a schematic diagram that illustrates the construction. For an instance of P-PCP $\bar{X} = \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_j, \cdots, \bar{x}_k, \bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \cdots, \bar{y}_j^R, \cdots, \bar{y}_k^R$, the graph contains k regions of the form



Call the left part of such a region an *x-string segment*, and the right part a $\bar{y}^R$*-string segment*. Note that the j^{th} $\bar{x}$-string segment begins with the sequence "$e\{_j$", whereas the j^{th} $\bar{y}^R$-string segment ends with the sequence "$\}_j e$". The dotted edges labeled e around the outsides of Fig. 1 serve to connect each $\bar{x}$-string

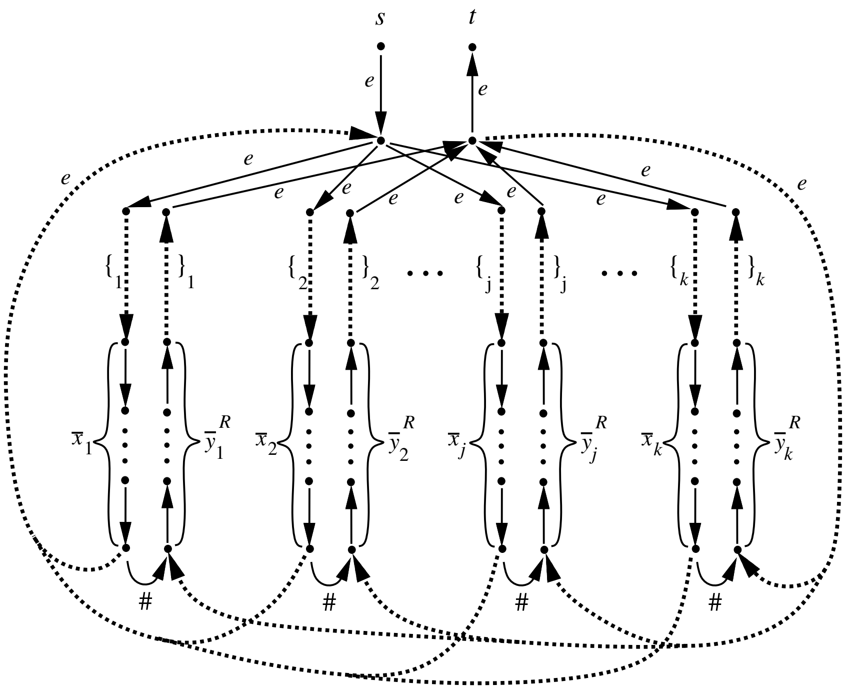


Figure 1. A schematic diagram of the graph that would be constructed for an instance of P-PCP $\bar{X} = \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_j, \cdots, \bar{x}_k; \bar{Y}^R = \bar{y}_1^R, \bar{y}_2^R, \cdots, \bar{y}_j^R, \cdots, \bar{y}_k^R$.

segment to all of the other $\bar{x}$-string segments, and each $\bar{y}^R$-string segment to all of the other $\bar{y}^R$-string segments. Any number of $\bar{x}$-string segments can be concatenated together to form a path in any order; however, each such segment is labeled in the word of the path by the appropriate "$\{$" symbol. Similarly, any number of $\bar{y}^R$-string segments can be concatenated together to form a path in any order; however, each such segment is labeled in the word of the path by the appropriate "$\}$" symbol.

By following one of the edges that is labeled with "#", a path can pass from an $\bar{x}$-string segment to a $\bar{y}^R$-string segment. However, once a #-edge is taken, the path can only be extended with $\bar{y}^R$-string segments. Consequently, all paths from s to t are of the form:

$$s \xrightarrow{e} \text{some number of } \bar{x}\text{-string segments} \xrightarrow{\#} \text{some number of } \bar{y}^R\text{-string segments} \xrightarrow{e} t$$

To complete the proof, we observe that

- If there is an $L(S_1) \cap L(S_2)$-path P from s to t, a solution to the instance of P-PCP can be read off from the word of P by reading it as a P-PCP solution in tagged form.[6]
- If the instance of P-PCP has the solution $i_1, i_2, \cdots, i_j, \cdots, i_m$, then we can find an $L(S_1) \cap L(S_2)$-path P from s to t by
  - (i) following the e edge from s,
  - (ii) choosing $\bar{x}$-string segments in the order $\bar{x}_{i_1}, \bar{x}_{i_2}, \cdots, \bar{x}_{i_j}, \cdots, \bar{x}_{i_m}$,
  - (iii) following the #-edge from the i_m^{th} $\bar{x}$-string segment to the i_m^{th} $\bar{y}^R$-string segment,
  - (iv) choosing $\bar{y}^R$-string segments in the order $\bar{y}^R_{i_m}, \cdots, \bar{y}^R_{i_j}, \cdots, \bar{y}^R_{i_2}, \bar{y}^R_{i_1}$,
  - (v) following the e edge to t.

  The word of this path exhibits the solution to the instance of P-PCP in tagged form, and hence is an $L(S_1) \cap L(S_2)$-path from s to t. □

For the particular case of the instance of P-PCP introduced in Example 2.4, the construction used in Theorem 3.1 is illustrated in Fig. 2.

4. UNDECIDABILITY OF CONTEXT-SENSITIVE, STRUCTURE-TRANSMITTED DATA-DEPENDENCE ANALYSIS

In this section, we discuss why Theorem 3.1 implies that it would be impossible to create a precise algorithm for context-sensitive, structure-transmitted data-dependence analysis. In particular, the graphs used in the construction given in Theorem 3.1 can be encoded in programs by encoding each edge in Fig. 1 as a data dependence in the program.

For example, Fig. 3 shows a C program fragment that corresponds to the instance of P-PCP given in Example 2.4. A context-sensitive, structure-transmitted data-dependence analysis should report that variable `x` may have the value `NULL` at program point `t` (which corresponds to the fact that this instance of P-PCP has the solution 1, 2, 3, 1). In Fig. 3, the symbols "$\{_j$" and "$\}_j$" correspond to data dependences associated with the call from procedure `f` to procedure `fj` and the return from `fj` to `f`, respectively; the symbol "(" corresponds to a data construction in which the value is placed in the first position of a `cons`; the symbol "[" corresponds to a data construction in which the value is placed in the second position of a `cons`; the symbol ")" corresponds to a selection via `car`; the symbol "]" corresponds to a selection via `cdr`; the symbol "#" corresponds to a data dependence that occurs when a (recursive) call on `f` is finally bypassed.

It should be noted that questions of the kind posed in Fig. 3 (*i.e.*, "Does a given variable have a given value at a particular point in a program?") often turn out to be undecidable in their most general form, and

---

[6]Here we see the reason for the remark made in the footnote in Def. 2.5: In the word of path P, each occurrence of "{ " is immediately preceded by two occurrences of the symbol e, and each occurrence of " }" is immediately followed by two occurrences of e. These occurrences of e should be ignored in recovering a P-PCP solution in tagged form from the word of P.

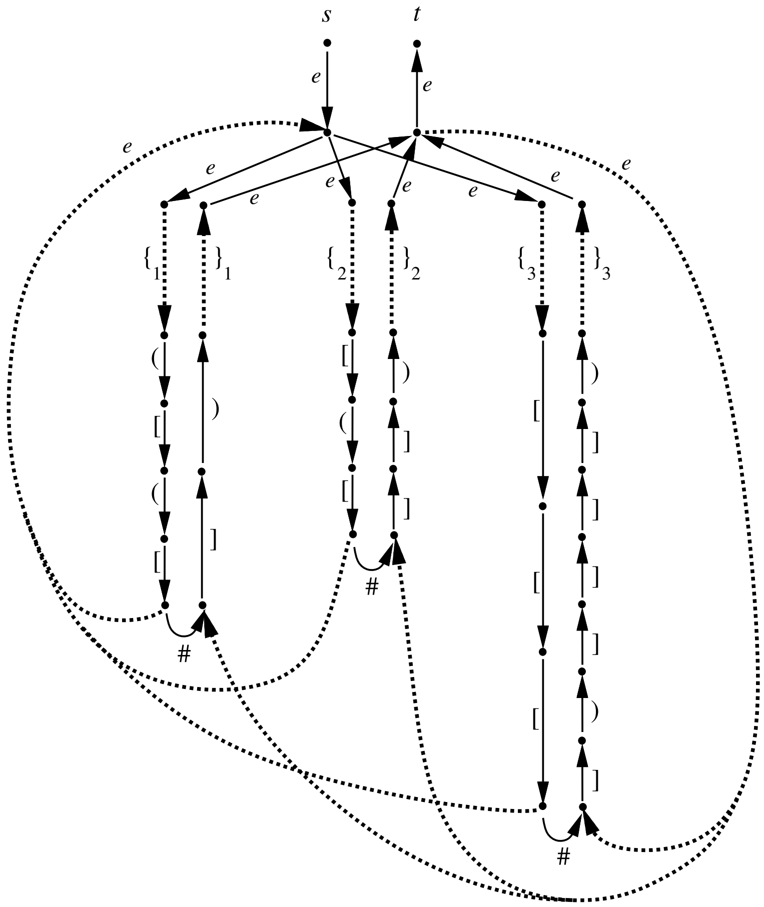


Figure 2. The $L(S_1) \cap L(S_2)$-reachability problem that would be constructed for the instance of P-PCP given in Example 2.4. This graph contains a path whose word is $\{_1 ([([\{_2 [([\{_3 [[[\{_1 ([([\#]) \}_1])]]]]) \}_3]]) \}_2]) \}_1$, which indicates that the instance of P-PCP given in Example 2.4 has the solution 1, 2, 3, 1.

often there are several independent reasons why the problem is undecidable (*e.g.*, it is undecidable whether a given statement is ever executed; it is undecidable whether a given path is ever executed; *etc.*). This paper shows that context-sensitive, structure-transmitted data-dependence analysis is undecidable even if a conservative approximation is made that usually—*i.e.*, for many other program-analysis problems—overcomes other sources of undecidability. In particular, we assume that all paths in which returns are matched with corresponding calls are executable.

The idea illustrated in Fig. 3 carries over to all of the graphs that arise in the construction used in Theorem 3.1 (see Fig. 1): In general, the pair of strings $\bar{x}_j$, $\bar{y}_j^R$ is encoded by a procedure of the form

```
void fj() {
    x = cons(...x...);               /* Construction expression encoding x̄j */
    if (. . .) {
        f();
    }
    x = ...car(...cdr(...x...)...)...;  /* Selection expression encoding ȳj^R */
}
```

Procedure `f` has the form

```
void f() {
    if (. . .) f1();
    else if (. . .) f2();
    .
    .
    .
    else if (. . .) fk−1();
    else fk();
}
```

Procedure `main` is the same as in Fig. 3.

A context-sensitive, structure-transmitted data-dependence analysis would determine that `x` could have the value `NULL` at program point `t` (in `main`) if and only if the given instance of P-PCP has a solution. Consequently, context-sensitive, structure-transmitted data-dependence analysis is undecidable (*i.e.*, an algorithm for context-sensitive, structure-transmitted data-dependence analysis cannot exist).

```
List *w, *x, *y;

void f1() {
    x = cons(NULL,cons(cons(NULL,cons(x,NULL)),NULL));   /* Encodes ([([    */
    if (. . .) {
        f();
    }
    x = car(cdr(x));                                     /* Encodes ])      */
}

void f2() {
    x = cons(NULL,cons(cons(NULL,x),NULL));              /* Encodes [([     */
    if (. . .) {
        f();
    }
    x = car(cdr(cdr(x)));                                /* Encodes ]])     */
}

void f3() {
    x = cons(NULL,cons(NULL,cons(NULL,x)));              /* Encodes [[[     */
    if (. . .) {
        f();
    }
    x = car(cdr(cdr(cdr(cdr(car(cdr(x)))))));            /* Encodes ])]]]]) */
}

void f() {
    if (. . .) f1();
    else if (. . .) f2();
    else f3();
}

void main() {
 s: x = NULL;
    f();
 t:                                               /* Could x be NULL here? */
}
```

Figure 3. The C program scheme that would be constructed for the instance of P-PCP given in Example 2.4. This program has the dependence graph given in Fig. 2.

5. IMPLICATIONS FOR OTHER PROGRAM-ANALYSIS FRAMEWORKS

Set Constraints and Set-Based Analysis

Following earlier work by Reynolds [28] and Jones and Muchnick [17], a number of people in recent years have explored the use of set constraints for analyzing programs. Set constraints are typically used to collect a superset of the set of values that the program's variables may hold during execution. Typically, a set variable is created for each program variable at each program point; set constraints are generated that approximate the program's behavior; program analysis then becomes a problem of finding the least solution of the set-constraint problem. Set constraints have been used both for program analysis [28,17,1,10,11], and for type inference [2,3].

Melski and Reps have obtained a number of results on the relationship between certain classes of set constraints and CFL-reachability [22]. Their results establish relationships in both directions: They showed that CFL-reachability problems and a subclass of what have been called *definite set constraints* [9] are equivalent. That is, given a CFL-reachability problem, it is possible to construct a set-constraint problem whose answer gives the solution to the CFL-reachability problem; likewise, given a set-constraint problem, it is possible to construct a CFL-reachability problem whose answer gives the solution to the set-constraint problem. It is also shown in [22] that CFL-reachability is equivalent to a class of set constraints that was designed to be useful for (context-insensitive) analysis of programs written in a higher-order language—so-called "set-based analysis" [10]. The results of Sections 3 and 4 imply that it would be impossible to create a precise context-sensitive version of set-based analysis.

Control-Flow Analysis

In [30], Sharir and Pnueli defined two methods for carrying out interprocedural dataflow analysis so as to ensure that the propagation of dataflow information respects the fact that when a procedure finishes it returns to the site of the most recent call. In one of their methods, the so-called "call-strings approach", each piece of dataflow information is tagged with a call string that records the history of uncompleted procedure calls along which that data has propagated. The call string on a piece of information is updated whenever a propagation step associated with a call statement or return statement is performed. The information that would be obtained, in principle, if call strings were allowed to grow arbitrarily long is called the call-strings-∞ solution.

Sharir and Pnueli show that, for distributive dataflow-analysis problems over a finite semilattice, it is possible to restrict the length of call strings to some fixed length (where the bound on the length required is quadratic in the size of the lattice and linear in the number of call sites in the program) and yet still obtain a result that is equivalent in precision to the call-strings-∞ solution. By suitable means, approximate (but safe) solutions can also be obtained using shorter call strings; limiting call strings to length k defines the call-strings-k solution.

In considering algorithms for interprocedural dataflow analysis, one should be careful not to confuse two separate issues:

(i) Whether an algorithm computes a solution equal in precision to the call-strings-∞ solution.

(ii) Whether an algorithm computes its solution by *actually tracking* entities labeled by call strings (*e.g.*, of some length k).

A type-(ii) algorithm typically has worst-case running time that is exponential in k. However, for suitably restricted classes of interprocedural dataflow-analysis problems, there are algorithms with property (i), yet their worst-case running times are *polynomial* in the size of the program;[7] these algorithms use dynamic

---

[7]In their Theorem 7-3.4, Sharir and Pnueli establish that the greatest fixed point of a certain set of equations equals the meet-over-all-valid-paths (MOVP) solution. In their Theorem 7-4.6, they establish that the MOVP solution equals the call-strings-∞ solution. As for algorithms, Sharir and Pnueli give a worklist algorithm for finding the greatest fixed point of the set of equations. However, Section 7-3 of [30] presents a fairly general framework: in particular, the only assumption about the semilattice is that it is finite. Because of the way their dynamic-programming algorithm tabulates information, when the size of the semilattice is exponential in the size of the program, the algorithm may use time exponential in the size of the program.

To achieve a polynomial time bound, the tricks are:

programming, rather than utilizing entities labeled with explicit call strings [26,29]. For the same class of problems, a type-(ii) algorithm will, in general, have exponential running time.

These results provide an interesting contrast with those that have been obtained on a program-analysis problem of interest to the functional-programming community: the problem of "k-CFA", or "control-flow analysis" (for higher-order programming languages) using call strings of length k [31,16,12,23]. The goal of control-flow analysis is to track data and control flow in the presence of first-class (anonymous) functions, data constructors, and selectors. Many of the algorithms that have been given for k-CFA are type-(ii) algorithms (in the sense mentioned above), in that they actually track entities labeled by call strings of length $\leq k$. In general, the running time of these algorithms is exponential in k.

Similar to the concept of the call-strings-∞ solution to an interprocedural dataflow-analysis problem, the ∞-CFA solution is what would be obtained, in principle, if call strings were allowed to grow arbitrarily long. The results of Sections 3 and 4 imply that, in general, when data constructors and selectors are to be taken into account, ∞-CFA is undecidable. That is, in the presence of data constructors and selectors, the ∞-CFA solution cannot be computed.

ACKNOWLEDGEMENTS

I am grateful for discussions that I had about the problem with F. Nielson, G. Ramalingam, S. Horwitz, and D. Melski.

REFERENCES

1. Aiken, A. and Murphy, B.R., "Implementing regular tree expressions," pp. 427-447 in *Func. Prog. and Comp. Arch., Fifth ACM Conf.,* (Cambridge, MA, Aug. 26-30, 1991), *Lec. Notes in Comp. Sci.,* Vol. 523, ed. J. Hughes, Springer-Verlag, New York, NY (1991).
2. Aiken, A. and Murphy, B.R., "Static type inference in a dynamically typed language," pp. 279-290 in *Conf. Rec. of the Eighteenth ACM Symp. on Princ. of Prog. Lang.,* (Orlando, FL, Jan. 1991), ACM, New York, NY (1991).
3. Aiken, A. and Wimmers, E.L., "Type inclusion constraints and type inference," pp. 31-41 in *Sixth Conf. on Func. Prog. and Comp. Arch.,* (Copenhagen, Denmark), (June 1993).
4. Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *SIGPLAN Not.* **23**(7) pp. 47-56 (July 1988).
5. Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *SIGPLAN Not.* **23**(7) pp. 57-66 (July 1988).
6. Cooper, K.D. and Kennedy, K., "Fast interprocedural alias analysis," pp. 49-59 in *Conf. Rec. of the Sixteenth ACM Symp. on Princ. of Prog. Lang.,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
7. Harrison, M.A., *Introduction to Formal Language Theory,* Addison-Wesley, Reading, MA (1978).
8. Hecht, M.S., *Flow Analysis of Computer Programs,* North-Holland, New York, NY (1977).
9. Heintze, N. and Jaffar, J., "A decision procedure for a class of set constraints," Tech. Rep. CMU-CS-91-110, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA (1991).
10. Heintze, N., "Set-based analysis of ML programs," Tech. Rep. CMU-CS-93-193, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA (July 1993).
11. Heintze, N. and Jaffar, J., "Set constraints and set-based analysis," in *2nd Workshop on Principles and Practice of Constraint Programming*, (May 1994).
12. Heintze, N. and McAllester, D., "Linear-time subtransitive control flow analysis," *Proc. of the ACM SIGPLAN 97 Conf. on Prog. Lang. Design and Implementation,* (Las Vegas, Nevada, June 15-18, 1997), *SIGPLAN Not.* **32**(5) pp. 261-272 (May 1997).
13. Hopcroft, J.E. and Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading, MA (1979).

- To restrict attention to a certain class of semilattices. However, this class can include semilattices whose size is exponential in the size of the program (*e.g.*, the powerset of the program points, the powerset of the program's variables, *etc*.).
- To tabulate the Sharir-Pnueli ϕ functions pointwise (*e.g.*, on singletons, rather than entire sets).

This is essentially what is done in [26] for the class of finite-distributive-subset problems, and in [29] for the larger class of distributive environment problems.

14. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (Jan. 1990).
15. Horwitz, S., Reps, T., and Sagiv, M., "Demand interprocedural dataflow analysis," *SIGSOFT 95: Proc. of the Third ACM SIGSOFT Symp. on the Found. of Softw. Eng.,* (Wash., DC, Oct. 10-13, 1995)*, ACM SIGSOFT Softw. Eng. Notes* **20**(4) pp. 104-115 (1995).
16. Jagannathan, S. and Weeks, S., "A unified treatment of flow analysis in higher-order languages," pp. 393-406 in *Conf. Rec. of the Twenty-Second ACM Symp. on Princ. of Prog. Lang.,* (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
17. Jones, N.D. and Muchnick, S.S., "Flow analysis and optimization of Lisp-like structures," pp. 102-131 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones, Prentice-Hall, Englewood Cliffs, NJ (1981).
18. Khedker, U.P. and Dhamdhere, D.M., "A generalized theory of bit vector data flow analysis," *ACM Trans. Program. Lang. Syst.* **16**(5) pp. 1472-1511 (Sept. 1994).
19. Kou, L.T., "On live-dead analysis for global data flow problems," *J. ACM* **24**(3) pp. 473-483 (July 1977).
20. Lewis, H.R. and Papadimitriou, C.H., *Elements of the Theory of Computation,* Prentice-Hall, Englewood Cliffs, NJ (1981).
21. McCarthy, J., "A basis for a mathematical theory of computation," pp. 33-70 in *Computer Programming and Formal Systems*, ed. Braffort and Hershberg, North-Holland, Amsterdam (1963).
22. Melski, D. and Reps, T., "Interconvertibility of a class of set constraints and context-free language reachability," *Theor. Comp. Sci.* **248**(1-2)(Nov. 2000). (To appear.)
23. Nielson, F. and Nielson, H.R., "Infinitary control flow analysis: A collecting semantics for closure analysis," pp. 332-345 in *Conf. Rec. of the Twenty-Fourth ACM Symp. on Princ. of Prog. Lang.,* (Paris, France, Jan. 15-17, 1997), ACM, New York, NY (1997).
24. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," *SIGSOFT 94: Proc. of the Second ACM SIGSOFT Symp. on the Found. of Softw. Eng.,* (New Orleans, LA, Dec. 7-9, 1994)*, ACM SIGSOFT Softw. Eng. Notes* **19**(5) pp. 11-20 (Dec. 1994).
25. Reps, T., "Shape analysis as a generalized path problem," pp. 1-11 in *Proc. of the ACM SIGPLAN Symp. on Part. Eval. and Sem.-Based Prog. Manip. (PEPM 95),* (La Jolla, California, June 21-23, 1995), ACM, New York, NY (1995).
26. Reps, T., Horwitz, S., and Sagiv, M., "Precise interprocedural dataflow analysis via graph reachability," pp. 49-61 in *Conf. Rec. of the Twenty-Second ACM Symp. on Princ. of Prog. Lang.,* (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995).
27. Reps, T., "Program analysis via graph reachability," *Information and Software Technology* **40**(11-12) pp. 701-726 Elsevier Science, (Nov./Dec. 1998).
28. Reynolds, J.C., "Automatic computation of data set definitions," pp. 456-461 in *Information Processing 68: Proc. of the IFIP Congress 68*, North-Holland, New York, NY (1968).
29. Sagiv, M., Reps, T., and Horwitz, S., "Precise interprocedural dataflow analysis with applications to constant propagation," *Theor. Comp. Sci.* **167** pp. 131-170 (1996).
30. Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones, Prentice-Hall, Englewood Cliffs, NJ (1981).
31. Shivers, O., "Control flow analysis in Scheme," *Proc. of the ACM SIGPLAN 88 Conf. on Prog. Lang. Design and Implementation,* (Atlanta, GA, June 22-24, 1988)*, SIGPLAN Not.* **23**(7) pp. 164-174 (July 1988).
32. Yannakakis, M., "Graph-theoretic methods in database theory," pp. 230-242 in *Proc. of the Ninth ACM Symp. on Princ. of Database Syst.*, (1990).