Interconvertibility of a Class of Set Constraints and Context-Free-Language Reachability[1]

David Melski[2]
Computer Sciences Department
University of Wisconsin

Thomas Reps[2]
Computer Sciences Department
University of Wisconsin

December 4, 1998

Abstract

We show the interconvertibility of context-free-language reachability problems and a class of set-constraint problems: given a context-free-language reachability problem, we show how to construct a set-constraint problem whose answer gives a solution to the reachability problem; given a set-constraint problem, we show how to construct a context-free-language reachability problem whose answer gives a solution to the set-constraint problem. The interconvertibility of these two formalisms offers an conceptual advantage akin to the advantage gained from the interconvertibility of finite-state automata and regular expressions in formal language theory, namely, a problem can be formulated in whichever formalism is most natural. It also offers some insight into the "$O(n^3)$ bottleneck" for different types of program-analysis problems and allows results previously obtained for context-free-language reachability problems to be applied to set-constraint problems and vice versa.

Key Words: *definite set constraints, context-free-language reachability, path problem, program analysis, complexity of program-analysis problems.*

1 Introduction

This paper concerns algorithms for converting between two techniques for formalizing program-analysis problems: context-free-language reachability and a class of set constraints. Context-free-language reachability (CFL-reachability) is a generalization of ordinary graph reachability (i.e., transitive closure). It has been used for a number of program-analysis applications, including interprocedural slicing [23, 25], interprocedural dataflow analysis [24], and shape analysis [37].

Set constraints have been applied to program analysis by using them 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 then 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 for program analysis, including [2, 17, 19, 29, 43], and type inference, including [3, 4].

[1] This work was supported in part by the National Science Foundation under grants CCR-9100424 and CCR-9625667, and by the Defense Advanced Research Projects Agency (monitored by the Office of Naval Research under contracts N00014-92-J-1937 and N00014-97-1-0114) and in part by a grant from IBM.

[2] 1210 West Dayton Street, Madison, WI 53706; {melski,reps}@cs.wisc.edu

Numerous classes of set constraints have been identified and studied. Except for Section 5, the class of set constraints considered in this paper is a subclass of what have been called *definite* set constraints [18]; throughout the paper, the term "set constraints" refers to the class of set constraints defined in Section 2.2.

The principal contribution of this paper is to relate these two formalisms:

- We give a construction for converting a CFL-reachability problem into a set-constraint problem. This construction can be carried out in $O(n + e)$ time, where n is the number of nodes in the graph, and e is the number of edges in the graph.
- We give a second construction for converting a set-constraint problem into a CFL-reachability problem. Again the construction can be carried out in time linear in the size of the set-constraint problem.

We gain several benefits from knowing that these two program-analysis formalisms are interconvertible:

- There is an advantage from the conceptual standpoint: When confronted with a program-analysis problem, one can think and reason in terms of whichever paradigm is most appropriate. (This is analogous to the situation one has in formal language theory with finite-state automata and regular expressions, or with pushdown automata and context-free grammars.) For example, CFL-reachability leads to natural formulations of interprocedural dataflow analysis [25] and interprocedural slicing [40, 23]. Set-constraints lead to natural formulations of shape analysis [30, 43]. Each of these problems could be formulated using the (respective) opposite formalisms—our interconvertibility result formulates this idea precisely—but it would be awkward.
- These constructions also offer some insight into the "$O(n^3)$ bottleneck" for program-analysis problems. That is, a number of program-analysis problems are known to be solvable in time $O(n^3)$, but no sub-cubic-time algorithm is known. This is sometimes (erroneously) attributed to the need to perform transitive closure when a problem is solved. However, because transitive closure can be performed in sub-cubic time [13], this is not the correct explanation. We have long believed that, in many cases, real source of the $O(n^3)$ bottleneck is that a CFL-reachability problem needs to be solved. This paper shows this to be the case for the class of definite set-constraint problems.[1]
- CFL-reachability is known to be log-space complete for polynomial time (or PTIME-complete) [1, 38, 48]. Because the CFL-reachability to set-constraint construction can be performed in log-space, this paper demonstrates that a class of set-constraint problems are also PTIME-complete. Because PTIME-complete problems are believed not to be efficiently parallelizable (i.e., cannot be solved in polylog time on a polynomial number of processors), this paper extends the class of program-analysis problems that are unlikely to have efficient parallel algorithms.
- A demand algorithm computes a partial solution to a problem, when only part of the full answer is needed. For example, a demand algorithm might be used to compute the results of a program analysis only for points in the innermost loops of a given program. Because CFL-reachability problems can be solved in a demand-driven fashion (e.g., see [37, 36]), this paper shows that (in principle) set-constraint problems can also be solved in a demand-driven fashion. To our knowledge, this has not been investigated before in the literature on set constraints.
- CFL-reachability lends itself to analysis of languages with a lazy semantics [37]. Set constraints with strict semantics are more readily used to analyze languages with a strict semantics. However, our interconvertibility results show that CFL-reachability can be used to analyze strict languages, and set constraints with strict semantics can be used to analyze lazy languages.

[1] The source of the $O(n^3)$ bottleneck has also been attributed to the need to solve a *dynamic transitive-closure problem*. The basis for this statement is that several cubic-time algorithms for solving program-analysis problems maintain the transitive closure of a relation in an on-line fashion (i.e., as a sequence of insertions into the relation is performed). At the present time, no sub-cubic-time algorithm is known for this version of the dynamic transitive-closure problem.

In a CFL-reachability problem, new base facts (in the form of graph edges or grammar productions) are not added to the problem in an on-line fashion. (When dynamic programming is used to solve CFL-reachability problems, additional edges *are* inserted in the graph; however, in this case, the edges are added by the algorithm and not inserted by an outside agent.) Thus, we feel that the statement "a CFL-reachability problem needs to be solved" offers a declarative characterization of the source of the $O(n^3)$ bottleneck.

$A ::= B\ C$	A production of a context free grammar
$A\langle V_i, V_j\rangle$	An edge labelled A from node V_i to node V_j
$c(V_1, \ldots, V_r)$	An atomic expression of arity r used in set constraints
$X \supseteq c(V_1, \ldots, V_r)$	A set constraint
$X \Rightarrow a$	A production of a regular term grammar

Table 1: Notation used throughout this paper.

A different class of set constraints has been used by Heintze to formulate analysis problems for a higher-order language (ML) [17]. In Section 5, we show how set-constraint problems of this class can be converted to CFL-reachability problems while preserving cubic-time solvability (i.e., cubic in the size of the original problem). A notable aspect of this result is that it demonstrates that the CFL-reachability framework is capable of expressing analysis problems, such as program slicing and shape analysis, for higher-order languages. All previous applications of CFL-reachability to program analysis have been limited to first-order languages.

For all three constructions there is a thorny issue that we must address: When we plug the various parameters that characterize the size of the transformed problems into the standard formulas for the worst-case asymptotic running time in which the transformed problems can be solved, it appears that both of our constructions cause a blowup in the time required to solve the problem. That is, from the standpoint of worst-case asymptotic running time, it appears that we do worse by performing the transformation and solving the transformed problem. If this were true, it would not be a satisfactory demonstration of interconvertibility. In Sections 3.5, 4.4, and 5.3 we examine this issue and show that, in fact, the asymptotic running time of the constructed problems is the same as the problems they were constructed from.

We assume that the reader is familiar with context-free grammars. In Section 2, we define CFL-reachability and a class of set-constraint problems, and describe dynamic-programming algorithms that can be used to solve them. Section 2 also defines regular term grammars, which are used to give finite presentations of solutions to set-constraint problems. In Section 3, we show how to express CFL-reachability using set constraints and discuss the running time of the dynamic-programming algorithm on the resulting problem. In Section 4, we discuss how to restate set-constraint problems as CFL-reachability problems and again examine the running time of the dynamic-programming algorithm. In Section 5, we show how to encode a second class of set-constraint problems as CFL-reachability problems. In Section 6 we show how to express CFL-reachability problems with this second class of set-constraints. Section 7 offers some concluding remarks.

2 Background

To understand the interconvertibility result, it is necessary to have a grasp of the problem domains that we are working with and the algorithms for solving these types of problems. Table 1 summarizes some of the notational conventions we will use in the paper.

2.1 CFL-Reachability

In this section, we define CFL-reachability and describe a dynamic-programming algorithm for solving the all-pairs CFL-reachability problem.

Definition 2.1 Let CF be a context-free grammar over an alphabet of terminal symbols T and non-terminal symbols N. (Unless explicitly noted, we will follow the convention of starting terminal symbols with lowercase letters, and starting non-terminal symbols with uppercase letters.) Let G be a directed graph whose edges are labeled with members of $\Sigma = T \cup N$. Each path in G defines a word over Σ, namely, the word obtained by concatenating, in order, the labels of the edges on the path. A path in G is an *S-path* if its word is derived from the start symbol S of grammar CF. We define four varieties of *context-free-language reachability problems* (CFL-reachability problems), as follows:

- The *all-pairs S-path problem* is to determine all pairs of nodes n_1 and n_2 such that there exists an S-path in G from n_1 to n_2.
- The *single-source S-path problem* is to determine all nodes n_2 such that there exists an S-path in G from a given source node n_1 to n_2.
- The *single-target S-path problem* is to determine all node n_1 such that there exists an S-path in G from n_1 to a given target node n_2.
- The *single-source/single-target S-path problem* is to determine whether there exists an S-path in G from a given source node n_1 to a given target node n_2.

□

2.1.1 Solving CFL-Reachability Problems

We now give a dynamic-programming algorithm for solving all-pairs CFL-reachability problems. We are given a graph G whose edges are labelled with terminal symbols from a context-free grammar. To find the S-paths in this graph, we go through a process of "filling in" the graph with new edges, which are labelled with non-terminal symbols. A new edge labelled A from node i to node j indicates that there is an A-path from node i to node j. (As indicated in Table 1, we use the notation $A\langle i, j\rangle$ to represent an edge labelled A from node i to node j.) When this process is completed, there will be an edge labelled S between any two nodes connected by an S-path. This idea is formalized in the following algorithm:

Algorithm 2.1 (CFL-Reachability Algorithm)

1. **Normalize the grammar:** In order for this process to work efficiently, we first convert the grammar to a normal form in which the right-hand side of each production has at most two symbols from $T \cup N$[2]. This can be done by introducing new non-terminal symbols. Thus, a production such as

   $A ::= a\ B\ C\ d$

   might be converted into these productions:

   $A ::= A^{'}\ A^{''}$
   $A^{'} ::= a\ B$
   $A^{''} ::= C\ d$

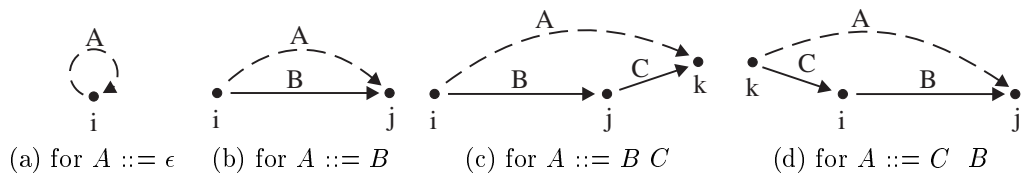
   This transformation can be done in time linear in the size of the grammar and causes a linear blowup in the size of the grammar. When the grammar is in normal form, each production will have one of the forms $A ::= M\ N$, $B ::= P$, or $C ::= \epsilon$, where A, B, and C are nonterminals, M, N, P are terminals or nonterminals, and ϵ represents the empty string.

2. **Create the initial worklist:** Let W be a worklist of edges. Initialize W with all of the edges in the original graph.

3. **Add edges for ϵ-productions:** The production $A ::= \epsilon$ indicates that there is a length-0 A-path from each node i to itself (see Figure 1(a)). Hence:

   for each production of the form $A ::= \epsilon$ **do**
       for each node i in the graph **do**
           if the edge $A\langle i, i\rangle$ is not in G **then** add $A\langle i, i\rangle$ to G and to W **fi**
       od
   od

[2] The normal form used is similar to Chomsky Normal Form, except that epsilon productions are allowed, and there are no restrictions on where terminal symbols may appear.



(a) for $A ::= \epsilon$ (b) for $A ::= B$ (c) for $A ::= B\ C$ (d) for $A ::= C\ \ B$

Figure 1: Edge induction in the CFL-reachability Algorithm (Algorithm 2.1). The figures show how a production of the context-free grammar causes the algorithm to add, or *induce*, an edge in the graph (dashed lines show induced edges).

4. **Add edges for other productions:** To determine where to add other edges to the graph, the current edges must be examined.

   while W is not empty **do**
   Select and remove an edge $B\langle i,j\rangle$ from W

   /* **Step 4.1:** look for productions of the form $A ::= B$ (see Figure 1(b)). */
   for each production of the form $A ::= B$ **do**
   if the edge $A\langle i,j\rangle$ is not in G **then** add $A\langle i,j\rangle$ to G and to W **fi**
   od

   /* **Step 4.2:** look for productions of the form $A ::= B\ C$. For each such production, for each */
   /* edge $C\langle j,k\rangle$, add $A\langle i,k\rangle$ (see Figure 1(c)). */
   for each production of the form $A ::= B\ C$ **do**
   for each outgoing edge $C\langle j,k\rangle$ from node j **do**
   if the edge $A\langle i,k\rangle$ is not in G **then** add $A\langle i,k\rangle$ to G and to W **fi**
   od
   od

   /* **Step 4.3:** look for productions of the form $A ::= C\ B$. For each such production, for each */
   /* edge $C\langle k,i\rangle$, add $A\langle k,j\rangle$ (see Figure 1(d)). */
   for each production of the form $A ::= C\ B$ **do**
   for each incoming edge $C\langle k,i\rangle$ into node i **do**
   if the edge $A\langle k,j\rangle$ is not in G **then** add $A\langle k,j\rangle$ to G and to W **fi**
   od
   od
   od

5. Return the set $\{(i,j)|S\langle i,j\rangle \in G\}$.

□

Note that the other varieties of CFL-reachability problems—single-source, single-target, and single-source/single-target problems—can be solved by solving the corresponding all-pairs problem. [24] describes a *demand* version of the CFL-Reachability Algorithm (tailored for a certain matched-parenthesis reachability problem) that is usually more efficient for the single-source, single-target, and single-source/single-target CFL-reachability problems. (In some cases, the demand algorithm in [24] performs the same amount of work as the CFL-Reachability Algorithm given here.) Section 7.6 gives a more detailed discussion of demand algorithms.

We now show that the running time of the CFL-Reachability Algorithm is bounded by $O(|\Sigma|^3 n^3)$, where Σ is the set of terminals and nonterminals in the normalized grammar, and n is the number of nodes in the graph. The running time is dominated by the amount of work performed in steps 4.2 and 4.3. In these steps,

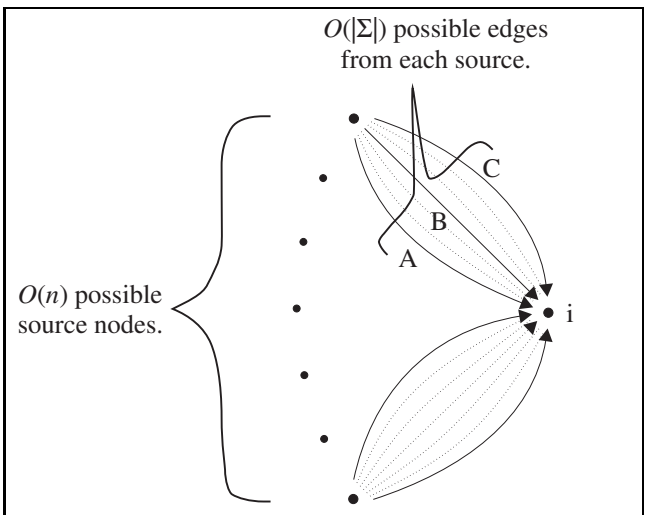


Figure 2: In a graph from a CFL-reachability problem, the number of edges into any given node is bounded by $O(|\Sigma|n)$, where Σ is the alphabet of the grammar, and n is the number of nodes in the graph.

each edge added to the graph is potentially paired with each of its neighboring edges. This is equivalent to saying that each pair of neighboring edges is considered; that is, for each node j, each incoming edge $A\langle i, j\rangle$ is potentially paired with each outgoing edge $B\langle j, k\rangle$.

For any given node j, the number of incoming edges is bounded by $|\Sigma|n$ (see Figure 2). Similarly, the number of outgoing edges from j is bounded by $|\Sigma|n$. This means that the total number of edge pairings that j ever participates in is bounded by $|\Sigma|^2 n^2$. For any given edge pair $B\langle i, j\rangle$ and $C\langle j, k\rangle$, the number of productions that may have "$B\ C$" as the body of the production is bounded by $|\Sigma|$. Node j is one of n nodes; consequently the total amount of work performed during any run of the algorithm is bounded by $O(|\Sigma|^3 n^3)$.

For a fixed grammar, $|\Sigma|$ is constant, and therefore an all-pairs CFL-reachability problem can be solved in time $O(n^3)$ (where the constant of proportionality is cubic in $|\Sigma|$).

2.2 Set Constraints

In this section, we define a class of set constraints. The material in this section is a summary of work done by Heintze and Jaffar [16, 17, 18].

2.2.1 Set Expressions and Set Constraints

In the class of set constraints we deal with, a *set expression* is either a set variable (denoted by V, W, X, etc.) or has one of the following forms:

- $c(V_1, \ldots, V_r)$. An expression of this form is called an *atomic expression*, and c is called a *constructor* or a *function symbol*. When set constraints are used for program analysis, atomic expressions are typically used to model data constructors of the language being analyzed (e.g., cons). All constructors have a fixed arity greater than or equal to zero. We will follow the convention of abbreviating nullary constructors as c, rather than writing $c()$.

- $c_i^{-1}(V)$. An expression of this form is called a *projection*. Projections can be used to model selection operators (such as car and cdr). The subscript of a projection indicates which field of the corresponding constructor is selected.

In the class of problems we consider, all *set constraints* are of the form $V \supseteq sexp$, where $sexp$ is a set expression.

The following example should clarify how set constraints can be used for program analysis:

Example 2.2 Suppose a program contains the following bindings:

x = cons(y,z) w = cdr(x)

This would generate the constraints $X \supseteq cons(Y, Z)$ and $W \supseteq cons_2^{-1}(X)$. In the second constraint, the projection $cons_2^{-1}(X)$ models cdr, asking for the second element of each cons value in X. □

2.2.2 Solutions to Set Constraints

A *ground term* over a set of constructors is either a nullary constructor or has the form $c(v_1 \ldots v_r)$ where $v_1 \ldots v_r$ are ground terms. Thus, given the nullary constructor a and the unary constructor $succ$, examples of ground terms include a, $succ(a)$, $succ(succ(a))$, *etc.*

A solution to a collection of set constraints is a mapping from set variables to sets of ground terms of constructors such that the constraints are satisfied. If we have a mapping $\mathcal{I}$ from set variables to sets of ground terms, then the mapping can be extended to map set expressions to sets of values:

- $\mathcal{I}(c(V_1, \ldots, V_r)) = \{c(v_1, \ldots, v_r) | v_1 \in \mathcal{I}(V_1), \ldots, v_r \in \mathcal{I}(V_r)\}$
- $\mathcal{I}(c_i^{-1}(V)) = \{v_i | c(v_1, \ldots, v_r) \in \mathcal{I}(V)\}$

(Note that this definition of $\mathcal{I}$ is strict with regards to the arguments of constructors; the expression $c(V_1, \ldots, V_r)$ is mapped to a nonempty value if and only if $V_1, \ldots, V_r$ are all mapped to nonempty values.) $\mathcal{I}$ is said to *satisfy* a constraint $X \supseteq sexp$ if $\mathcal{I}(X) \supseteq \mathcal{I}(sexp)$. $\mathcal{I}$ is said to be a *solution* to a collection of constraints if $\mathcal{I}$ satisfies each of the constraints.

An issue of how to represent a solution to a collection of set constraints arises because a solution may consist of an infinite set. Furthermore, a collection of set constraints may have multiple solutions.

Example 2.3 Consider the following constraints:

$X \supseteq a$ $X \supseteq succ(X)$

One solution to these constraints maps X to the infinite set $\{a, succ(a), succ(succ(a)), \ldots\}$. Another solution maps X to the infinite set $\{cons(a, a), succ(cons(a, a)), \ldots, a, succ(a), succ(succ(a)), \ldots\}$. □

We will always be interested in *least solutions* (under the subset ordering), e.g., the first of the two solutions listed in the above example. Heintze formalizes this idea in [16]. Note that a collection of set constraints must always have a solution. In particular, the map that sends each variable to the set of all ground terms is a trivial solution to any collection of constraints. (This is not generally true of all classes of constraints; it holds here because our constraints never restrict a ground term from appearing in the solution set of any variable.)

The solution to a collection of set constraints can be written as a *regular term grammar* [14], which is a formalism that allows certain infinite sets of terms to be represented in a finite manner. There are standard algorithms for dealing with regular term grammars (e.g., for determining membership) [14].

A regular term grammar consists of a finite, non-empty set of non-terminals, a set of function symbols, and a finite set of productions. Each function symbol has a fixed arity. Productions are of the form $N \Rightarrow term$ where N is a non-terminal. A $term$ is a non-terminal or of the form $c(term_1, \ldots, term_r)$, where c is a function symbol of arity r. As with other grammars, a derivability relation is defined. Given a production $N \Rightarrow term$, $term_1$ derives $term_2$ (denoted by $term_1 \Rightarrow term_2$) if $term_2$ is obtained from $term_1$ by replacing an occurrence of N in $term_1$ with $term$. The reflexive, transitive closure $\Rightarrow^*$ is defined as usual.

The regular term grammar that describes the least solution of Example 2.3 above has these productions:

$X \Rightarrow a$ $X \Rightarrow succ(X)$

2.2.3 Solving Set Constraints

The reader may notice that in Example 2.3 the set constraints $X \supseteq a$ and $X \supseteq succ(X)$ look very similar to the productions $X \Rightarrow a$ and $X \Rightarrow succ(X)$ of the regular term grammar specifying the solution. Such constraints are said to be in *explicit* form [16]: A constraint is in explicit form if it is of the form $V \supseteq c(V_1, \ldots, V_r)$. A collection of constraints in explicit form is converted to a regular term grammar by taking the variables to be non-terminals and converting each $\supseteq$ into $\Rightarrow$.

For any collection of constraints $\mathcal{C}$, we say that a variable X is *ground* if the least solution to the constraints of $\mathcal{C}$ that are in explicit form does not map X to the empty set (i.e., X is mapped to some ground term in the least solution). We say that $c(V_1, \ldots, V_r)$ is ground if $V_1 \ldots V_r$ are all ground.

The algorithm for solving set constraints involves augmenting the collection of set constraints with constraints in explicit form until no more can be added:

Algorithm 2.2 (SC-Reduction Algorithm) Given a collection of set constraints $\mathcal{C}$, the following steps are repeated until neither step causes $\mathcal{C}$ to change:

1. If $X \supseteq c_i^{-1}(Y)$ and $Y \supseteq c(V_1, \ldots, V_r)$ both appear in $\mathcal{C}$ and the expression $c(V_1, \ldots, V_r)$ is ground, then add the constraint $X \supseteq V_i$ to $\mathcal{C}$, if it is not already there.

2. If $X \supseteq Y$ and $Y \supseteq c(V_1, \ldots, V_r)$ both appear in $\mathcal{C}$, and $c(V_1, \ldots, V_r)$ is ground, then add the constraint $X \supseteq c(V_1, \ldots, V_r)$ to $\mathcal{C}$, if it is not already there.

When no more constraints can be added, the constraints in explicit form are converted to a regular term grammar; this describes the least solution [16]. □

The SC-Reduction Algorithm does not generate new atomic expressions; this means that when the algorithm finishes, for a fixed variable Y, the number of constraints of the form $Y \supseteq c(V_1, V_2, \ldots, V_r)$ in $\mathcal{C}$ is bound by $O(k)$, where k is the number of atomic expressions used in $\mathcal{C}$. The total number of constraints in $\mathcal{C}$ of the form $Y \supseteq c(V_1, V_2, \ldots, V_r)$ is bounded by $O(vk)$, where v is the number of set variables used in $\mathcal{C}$. Thus, the total number of times the first reduction step is ever applied is bounded by $O(pkv)$, where p is the maximum number of projection constraints that can match with a given constraint of the form $Y \supseteq c(V_1, V_2, \ldots, V_r)$.

The total number of constraints in $\mathcal{C}$ of the form $Y \supseteq X$ is bounded by $O(v^2)$. Thus, the total number of times the second step is applied is bounded by $O(v^2k)$. Let t be the total number of constraints in the original problem. In the worst case, v, k, and p are proportional to $O(t)$, and the total number of steps is bounded by $O(t^3)$.

The SC-Reduction Algorithm can be made to run in time $O(t^3)$ by using a worklist and a mark on each variable to track groundness information:

1. Let W be a worklist of constraints. Initialize W to $\{X \supseteq a \in \mathcal{C} | a \text{ is a nullary constructor}\}$.

2. Mark all set variables as having the property "not ground."

3. Perform the reduction steps:

while W is not empty **do**
  Select and remove a constraint $X \supseteq sexp$ from W
  if $X \supseteq sexp$ is of the form $X \supseteq c(V_1, V_2, \ldots, V_r)$ **then**
    for each constraint of the form $Y \supseteq c_i^{-1}(X)$ in $\mathcal{C}$ **do**
      if $Y \supseteq V_i$ is not in $\mathcal{C}$ **then** Insert $Y \supseteq V_i$ into $\mathcal{C}$ and W **fi**
    od
    for each constraint of the form $Y \supseteq X$ in $\mathcal{C}$ **do**
      if $Y \supseteq c(V_1, V_2, \ldots, V_r)$ is not in $\mathcal{C}$ **then** Insert $Y \supseteq c(V_1, V_2, \ldots, V_r)$ into $\mathcal{C}$ and W **fi**
    od
  else if $X \supseteq sexp$ is of the form $X \supseteq Y$ **then**
    for each constraint of the form $Y \supseteq c(V_1, V_2, \ldots, V_r)$ in $\mathcal{C}$ such that $V_1, \ldots, V_r$ are all ground **do**
      if $X \supseteq c(V_1, V_2, \ldots, V_r)$ is not in $\mathcal{C}$ **then** Insert $X \supseteq c(V_1, V_2, \ldots, V_r)$ into $\mathcal{C}$ and W **fi**

```
        od
    fi
    if X is not marked as ground then
        mark X as ground
        for each constraint of the form Y ⊇ c(...X...) in the original collection of constraints do
            if all set variables used in c(...X...) are ground then
                Insert Y ⊇ c(...X...) into W
            fi
        od
        for each constraint of the form Y ⊇ X in the original collection of constraints do
            Insert Y ⊇ X into W
        od
    fi
od
```

To make this run in time $O(t^3)$, constant-time access is needed to certain subsets of $\mathcal{C}$ in different parts of the algorithm; this can be achieved with a constant amount of overhead if the proper data structures (*e.g.*, matrices) are maintained for storing the subsets. The number of constraints of the form $X \supseteq c(V_1, V_2, \ldots, V_r)$ that may appear on the worklist is bounded by $O(kv)$; the number of reductions performed on a given constraint of this form is bounded by $O(p + v)$. The number of constraints of the form $X \supseteq Y$ that may appear on the worklist is bounded by $O(v^2)$; the number of reductions performed on a given constraint of this form is bounded by $O(k)$.

For each constraint $X \supseteq sexp$ that appears on the worklist, a check is performed to see if X is marked ground; these checks may contribute $O(kv + v^2)$ steps to the total running time. When X is first marked as ground, an attempt is made to propagate the new groundness information to all of the original constraints that use X in their right-hand side; note that groundness information need not be propagated to generated constraints because generated constraints can only be created if their right-hand sides are ground. The total number of attempts to propagate groundness information to an original constraint of the form $Y \supseteq c(V_1, V_2, \ldots, V_r)$ is bounded by r. The total number of attempts to propagate groundness information to an original constraint of the form $Y \supseteq X$ is 1. Since r is constant, the total amount of work done to propagate groundness information is bounded by $O(t)$.

Thus, the entire algorithm runs in time $O(pvk + kv^2 + t)$, which in the worst case is proportional to $O(t^3)$.

3 Transforming CFL-Reachability Problems into Set-Constraint Problems

We now turn to the method for expressing a CFL-reachability problem as a set-constraint problem. We first address how to encode the graph using set constraints. We then address how to encode the productions of the context-free grammar. Finally, we examine the time needed to solve the resulting collection of constraints.

3.1 Encoding the Graph

The construction is based on the idea of representing each node i with one variable X_i and one nullary constructor $node_i$. These are linked by constraints of the form

$$X_i \supseteq node_i, \text{ for } i = 1 \ldots n$$

In essence, $node_i$ serves as a label identifying the node to which X_i belongs.

We now need a way to associate a node with a set of edges to other nodes. (As in Section 2.1.1, "edges" also means the A-edges that may be added to a graph to represent A-paths.) In the final solution, an edge from node i to node j labelled A (where A is a terminal or nonterminal) is represented by the fact that the term $A(node_j)$ is in the solution set for variable X_i. In accordance with this goal, we use constraints involving X_i to indicate the set of targets of outgoing edges from node i, using unary constructors to encode

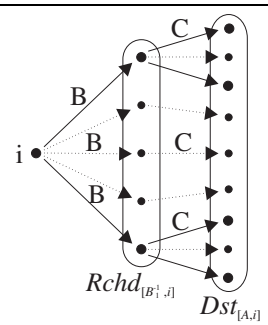


Figure 3: Use of $Dst_{[A,i]}$ and $Rchd_{[B_1^{-1},i]}$ to encode production $A ::= B\ C$. The variable $Rchd_{[B_1^{-1},i]}$ represents the set of nodes reached by following B-edges from i. The variable $Dst_{[A,i]}$ represents the set of nodes to which there should be an A-edge from node i.

the labels of edges. The argument to a constructor c is the target of an encoded c-edge. For example, if the initial graph contains an edge from node i to node j with label a, then the initial collection of constraints includes

$$X_i \supseteq a(X_j)$$

The set of constraints constructed in this manner completely encodes the initial graph.

3.2 Encoding the Productions

To encode the productions, we first convert the context-free grammar to the normal form discussed in Section 2.1.1. Thus, we assume that the right-hand side of each production has no more than two symbols. Now consider a production of the form $A ::= B\ C$, where A is a nonterminal, and B and C are either nonterminals or terminals. This production indicates that there is an A-path from node i to node k when there exists a node j such that there is an B-path from node i to node j, and a C-path from node j to node k.

Consider a fixed node i. To what nodes should node i have an A-edge (i.e., to what nodes is there an A-path)? The production $A ::= B\ C$ indicates that we should add an A-edge from node i to any nodes reached by following B edges from node i and then following C edges. In our representation of the graph, edges are represented as constructors, and "following an edge" can be encoded using projection: in particular, the production $A ::= B\ C$ can be encoded for node i by the following compound set constraint:

$$X_i \supseteq A(C_1^{-1}(B_1^{-1}(X_i)))$$

Note that this constraint does not belong to the class of set constraints introduced in Section 2.2; however, by introducing some additional set variables and constraints, it can be rewritten into the proper form: We introduce two set variables

$Dst_{[A,i]}$, which represents the "destinations" of A-edges from node i, and

$Rchd_{[B_1^{-1},i]}$, which represents the nodes reached by following B-edges from node i.

We also generate the following constraints to encode $A ::= B\ C$:

$Rchd_{[B_1^{-1},i]} \supseteq B_1^{-1}(X_i)$	(Follow B edges from node i)
$Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$	(Follow C edges from those nodes)
$X_i \supseteq A(Dst_{[A,i]})$	(Add A edges to the reached nodes)

Figure 3 depicts the use of the set variables $Rchd_{[B_1^{-1},i]}$ and $Dst_{[A,i]}$ in this encoding.

These constraints encode the production $A ::= B\ C$, but only "locally" for node i. That is, the solution to these constraints will give the A-paths starting at node i (assuming that the B-paths and C-paths are also solved for). To find all A-paths in the graph, similar constraints are generated for all other nodes of the graph.

We note that the set variables introduced to encode this production (i.e., $Dst_{[A,i]}$ and $Rchd_{[B_1^{-1},i]}$) may also be used in encoding other productions. For example, to encode $A ::= B\ D$, we need to generate only one new constraint: $Dst_{[A,i]} \supseteq D_1^{-1}(Rchd_{[B_1^{-1},i]})$.

The above discussion shows how to encode a production of the form $A ::= B\ C$. In a normalized CFL grammar, productions may also have the form $A ::= B$ and $A ::= \epsilon$. To encode a constraint of the form $A ::= B$ at node i, we generate the constraints $X_i \supseteq A(Dst_{[A,i]})$ and $Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$. To encode a constraint of the form $A ::= \epsilon$, we generate the constraint $X_i \supseteq A(X_i)$.

This completes the construction of the set-constraint problem. As we show in the next section, the solution to a constructed set-constraint problem $\mathcal{C}$ can be used to obtain the solution to the original CFL-reachability problem $\mathcal{P}$. In particular, let H be the regular term grammar that gives the solution to $\mathcal{C}$. Then there is an S-path from n to m in the solution to $\mathcal{P}$ if and only if $X_n \Rightarrow^* node_m$ under H. We give a formal proof of this in the next section.

3.3 Correctness of the Construction

We now formally prove that the solution to a constructed set-constraint problem gives a solution to the original CFL-reachability problem. More precisely, we have the following theorem:

Theorem 3.1 *Let $\mathcal{C}$ be the collection of set constraints constructed to represent the context-free reachability problem $\mathcal{P}$. Let G be the graph that results from running the CFL-reachability Algorithm on $\mathcal{P}$. Let H be the regular term grammar that results from solving $\mathcal{C}$. Then there is an edge $A\langle i,j\rangle$ in G if and only if $X_i \Rightarrow^* A(node_j)$ under H.*

To prove this theorem, we employ one lemma. In this lemma, $\mathcal{C}$, $\mathcal{P}$, H, and G are defined as in Theorem 3.1. The key lemma, which is proved in Appendix A, is as follows:

Lemma 3.2 *Let $\mathcal{C}'$ be the collection of constraints that results from running the SC-Reduction Algorithm on $\mathcal{C}$ (i.e., $\mathcal{C}'$ is $\mathcal{C}$ unioned with the collection of constraints generated by the SC-Reduction Algorithm). Then there is an edge $A\langle i,j\rangle$ in G if and only if $\mathcal{C}'$ contains $X_i \supseteq A(X_j)$ and/or $Dst_{[A,i]} \supseteq node_j$.*

Theorem 3.1 follows immediately from Lemma 3.2. Note that H contains no productions of the form $U \Rightarrow V$. This means that $X_i \Rightarrow^* A(node_j)$ under H if and only if H contains productions of the form $X_i \Rightarrow A(V)$ and $V \Rightarrow node_j$. H contains productions of this form if and only if $\mathcal{C}'$ contains $X_i \supseteq A(X_j)$ and $X_j \supseteq node_j$ or $\mathcal{C}'$ contains $X_i \supseteq A(Dst_{[A,i]})$ and $Dst_{[A,i]} \supseteq node_j$ (where A is a nonterminal). Since $\mathcal{C}$ (and hence $\mathcal{C}'$) must contain $X_j \supseteq node_j$ and $X_i \supseteq A(Dst_{[A,i]})$ (for each nonterminal A, it follows that H contains the required productions if and only if $\mathcal{C}'$ contains $X_i \supseteq A(X_j)$ or $Dst_{[A,i]} \supseteq node_j$.

3.4 Performing the Construction in Log-Space

It is also easily shown that the construction given in this section can be carried out by a log-space Turing machine. A *log-space Turing machine* has a read-only input tape, a read-write work tape with $O(\log x)$ cells, where x is the size of the input, and a write-only output tape.

We claim that there exists a log-space Turing machine $\mathcal{P}_1$ that does the following: given an arbitrary context-free grammar CF on the input tape, $\mathcal{P}_1$ outputs an equivalent context-free grammar CF' that is in normal form. Consider the following typical context-free production q:

$$N ::= a\ b\ C\ d\ E$$

This production can be replaced with the following productions (which are in normal form):

$N ::= a\ T_1$

$T_1 ::= b\ T_2$

$T_2 ::= C\ T_3$

$T_3 ::= d\ T_4$

$T_4 ::= E$

A Turing machine $\mathcal{P}_1$ can be written that processes each production q as follows: it scans q left to right; for each position i of the right-hand side of production q (except the first and last positions), a production $T_{i-1} ::= a_i\ T_i$ is output, where a_i is the symbol at position i, and T_i is a new non-terminal symbol. $\mathcal{P}_1$ requires space on the work tape for one counter *cnt*, which it uses to generate new non-terminal symbols. Since the number of non-terminals introduced is proportional to the length x of the context-free grammar, $\mathcal{P}_1$ needs at most $O(\log x)$ bits on the work tape for *cnt*. Let $\mathcal{P}_1'$ be the log-space Turing machine that takes a CFL-reachability problem as input, and outputs the same CFL-reachability problem but with a normalized grammar.

We also claim that there exists a log-space Turing machine $\mathcal{P}_2$ that, given a CFL-reachability problem with a context-free grammar in normal form, performs the construction of Sections 3.1 and 3.2. $\mathcal{P}_2$ operates in two phases: in phase I, it scans each edge e of the graph G of the CFL-reachability problem and outputs a corresponding constraint; in phase II, it encodes each production of the context-free grammar for each node of the graph G. Phase I requires no space on the worktape. Phase II requires space on the work tape for the following items:

1. an index $idx1$ into the input tape that points to the current production.
2. an index $idx2$ into the input tape that points to the current node.
3. a counter *cnt* for producing unique set variables for each constraint introduced during phase II.

The indices $idx1$ and $idx2$ can be represented with $O(\log x)$ bits, where x is the size of the input problem. The counter *cnt* requires $O(\log p \cdot n)$ bits, where p is the number of productions in the context-free grammar CF, and n is the number of nodes in the graph G. Note that $O(\log p \cdot n) \leq O(\log x^2) = O(2 \cdot \log x)$.

For any two log-space Turing machines $\mathcal{Q}$ and $\mathcal{R}$, there is a log-space Turing machine that is equivalent to the composition $\mathcal{Q} \circ \mathcal{R}$ [28]. This means that there is a log-space Turing machine $\mathcal{P}$ that is equivalent to $\mathcal{P}_2 \circ \mathcal{P}_1'$ and performs the construction of this section for an arbitrary context-free grammar. Since CFL-reachability problems are PTIME-complete (i.e., complete for PTIME under log-space reductions) [1, 38, 48], this means that the given class of set-constraint problems are also PTIME-complete [28].

3.5 Analysis of the Running Time

In general, an all-pairs CFL-reachability problem can be solved in time $O(n^3)$, where n is the number of nodes in the graph. The class of set constraints considered can be solved in time $O(t^3)$ where t is the number of constraints. However, for a set-constraint problem constructed from a CFL-reachability problem, this does not yield a satisfactory time bound—at least from the standpoint of showing that the two classes of problems are interconvertible: encoding the graph potentially creates n constraints of the form $X_i \supseteq node_i$ and e constraints of the form $X_i \supseteq a(X_j)$, where e is the number of edges in the graph. Encoding the productions may create $O(dn)$ constraints, where d is the number of productions. Because e can be as large as n^2, this would give a bound of $O(n^6)$ on the running time to solve the set-constraint problem.

However, as we now show, a sharper analysis yields a better bound on the running time for the constructed set-constraint problem. In the discussion below, we use the values defined in the following table:

k	the number of atomic expressions in $\mathcal{C}$
v	the number of variables in $\mathcal{C}$
p	the maximum number of projection constraints that can match with a given constraint in explicit form.
t	the total number of constraints in $\mathcal{C}$
d	the number of productions in the context-free grammar of the original problem.
s	the number of symbols in the context-free grammar of the original problem.
n	the number of nodes in the graph of the original problem.

Recall that in Section 2.2.3, we gave a tighter bound of $O(pkv + kv^2 + t)$ for the running time of the SC-Reduction Algorithm on a collection $\mathcal{C}$ of set constraints.

Let $\mathcal{C}$ be a constructed set-constraint problem. Then the atomic expressions in $\mathcal{C}$ have one of the forms $A(Dst_{[A,i]})$ and $A(X_i)$. This means that k is bounded by $O(sn)$. Each variable in $\mathcal{C}$ has one of the forms $Dst_{[A,i]}$, $Rchd_{[A_1^{-1},i]}$, and X_i. Thus, v is bounded by $O(sn)$. A given constraint of the form $Rchd_{[B_1^{-1},i]} \supseteq C(V)$ matches with projection constraints in $\mathcal{C}$ of the form $Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$. A given constraint of the form $X_i \supseteq B(X_j)$ matches with projection constraints in $\mathcal{C}$ with one of the forms $Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$ and $Rchd_{[B_1^{-1},i]} \supseteq B_1^{-1}(X_i)$. This means that p is bound by $O(s)$.

Thus, the total time needed to solve $\mathcal{C}$ is bounded by $O(s \cdot sn \cdot sn + sn \cdot (sn)^2 + dn + n^2)$. Since d is bounded by s^3, it follows that the run time is bounded by $O(s^3 n^3)$. Since s is a constant independent of the input, this gives a bound on the running time of $O(n^3)$.

4 Solving Set-Constraint Problems Using CFL-Reachability

4.1 Encoding Set Constraints as Graphs

4.1.1 The Idea Behind the Construction

We now turn to the problem of encoding set-constraint problems as CFL-reachability problems. The basic technique is a modification of work done by Reps in using CFL-reachability to do shape analysis [37]. In essence, our encoding involves simulating the steps of the SC-Reduction Algorithm with the productions of a reachability problem. In the following example, we show how the SC-Reduction Algorithm computes what atomic expressions reach each set variable and consider how this can be simulated with a CFL-reachability problem:

Example 4.1 Consider the following constraints:

$V_1 \supseteq a$
$V_2 \supseteq V_1$
$V_3 \supseteq cons(V_1, V_2)$
$V_4 \supseteq cons_2^{-1}(V_3)$

The SC-Reduction Algorithm reduces the constraints $V_1 \supseteq a$ and $V_2 \supseteq V_1$ by adding the constraint $V_2 \supseteq a$, which indicates that the atomic expression a reaches V_2. This will be simulated in the CFL-reachability problem by nodes for a, V_1, and V_2, together with edges $Id\langle a, V_1\rangle$ and $Id\langle V_1, V_2\rangle$. The counterpart of the reduction step is reachability in the graph: the path made of edges $Id\langle a, V_1\rangle$ and $Id\langle V_1, V_2\rangle$, together with the production "$Id ::= Id \;\; Id$", yields an edge $Id\langle a, V_2\rangle$. Just as the SC-Reduction Algorithm outputs the regular term grammar production $V_2 \Rightarrow a$ because of the constraint $V_2 \supseteq a$, we output the regular term grammar production $V_2 \Rightarrow a$ because of the edge $Id\langle a, V_2\rangle$.

The SC-Reduction Algorithm also reduces the constraints $V_3 \supseteq cons(V_1, V_2)$ and $V_4 \supseteq cons_2^{-1}(V_3)$ by adding the constraint $V_4 \supseteq V_2$. In the CFL-reachability problem, this will (roughly) be simulated by the edges $cons_2\langle V_2, cons(V_1, V_2)\rangle$, $Id\langle cons(V_1, V_2), V_3\rangle$ and $cons_2^{-1}\langle V_3, V_4\rangle$ and the production "$Id ::= cons_2 \;\; Id \;\; cons_2^{-1}$".

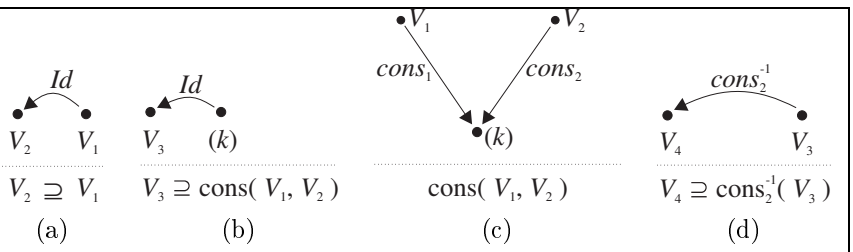


Figure 4: Edges inserted in the constructed graph to model terms and constraints.

This yields the edge $Id\langle V_2, V_4\rangle$, which models the constraint $V_4 \supseteq V_2$. Figure 5 shows the graph that is constructed to represent the set constraints used in this example; the construction of this graph is explained below.

□

With this intuition in mind, we make our first attempt to construct a CFL-reachability problem that will give the solution to a set-constraint problem. (For now, we ignore the clauses about ground expressions in the SC-Reduction Algorithm. Section 4.1.2 covers the modifications needed to account for ground expressions.)

The CFL-reachability framework uses a graph and context-free grammar and finds paths in the graph. We want to use this framework to find what atomic expressions reach each set variable; we construct a graph containing a node for each atomic expression and each set variable. This graph will contain edges that encode the set constraints. We construct a context-free grammar such that the CFL-reachability Algorithm will find *identity paths* from nodes representing atomic expressions to nodes representing set variables.

The solution to the set-constraint problem (in the form of a regular term grammar) is obtained from the reachability relations that hold in the graph. If node a represents an atomic expression, node V represents a variable, and there is an identity path from a to V, then the production $V \Rightarrow a$ is in the regular term grammar.

More precisely, the graph for Example 4.1 is constructed as follows (the general construction is given in Section 4.2):

- For each set variable V_i, the graph contains a node labelled V_i.
- Each atomic expression $cons(V_i, V_j)$ used in the constraints is associated with a unique index. This is for notational convenience; it is easier to refer to an expression by its index than by writing out the expression.

  For each atomic expression $cons(V_i, V_j)$ with index k, the graph contains a node labelled (k) and the edges $cons_1\langle V_i, (k)\rangle$ and $cons_2\langle V_j, (k)\rangle$. An edge $cons_m\langle V_i, (k)\rangle$ indicates that the values that reach V_i are wrapped in the mth position of the *cons* value represented by node (k). (See Figure 4(c)).
- For each constraint of the form $V_i \supseteq V_j$, the graph contains an edge $Id\langle V_j, V_i\rangle$. An edge labelled Id indicates an *identity path* in the graph. An identity path from node j to node i indicates that the values that reach node j also reach node i. (See Figure 4(a).)
- For each constraint of the form $V_l \supseteq cons(V_i, V_j)$, where the atomic expression $cons(V_i, V_j)$ has index k, the graph contains an edge $Id\langle (k), V_l\rangle$. This indicates that the atomic expression $cons(V_i, V_j)$ reaches V_l. (See Figure 4(b).)
- For each constraint of the form $V_i \supseteq cons_k^{-1}(V_j)$, the graph contains an edge $cons_k^{-1}\langle V_j, V_i\rangle$. An edge $cons_k^{-1}\langle V_j, V_i\rangle$ indicates that values at node i are taken from the kth position of *cons* values at node j. (See Figure 4(d).)

Productions are introduced in the context-free grammar to encode the simplification steps of the SC-Reduction Algorithm. The first reduction step of the SC-Reduction Algorithm is encoded via productions that indicate the fact that values can pass through *cons* values by being wrapped in a *cons* and then unwrapped by a projection:

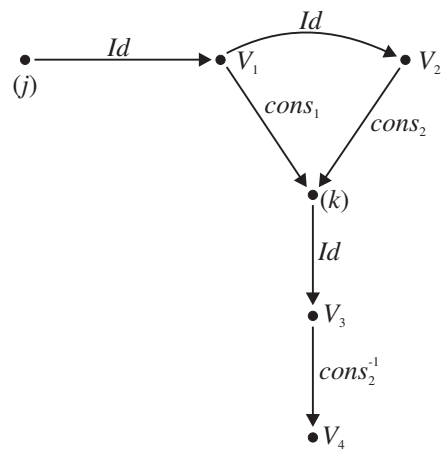


Figure 5: The graph built to encode the constraints in Example 4.1. The nodes (j) and (k) represent the atomic expressions a and $cons(V_1, V_2)$, respectively.

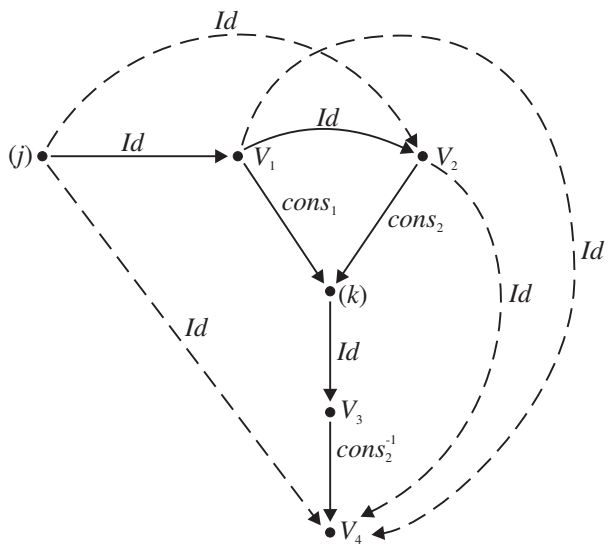


Figure 6: The graph for Example 4.1 after the CFL-reachability Algorithm has been run. Dashed lines represent edges inserted by the algorithm. The nodes (j) and (k) represent the atomic expressions a and $cons(V_1, V_2)$, respectively.

$Id ::= cons_1\ Id\ cons_1^{-1}$
$Id ::= cons_2\ Id\ cons_2^{-1}$

In Example 4.1, the SC-Reduction Algorithm adds the constraint $V_4 \supseteq V_2$ because of the constraints $V_3 \supseteq cons(V_1, V_2)$ and $V_4 \supseteq cons_2^{-1}(V_3)$. Let $cons(V_1, V_2)$ have index k. Then, in the constructed graph, the CFL-reachability algorithm adds the edge $Id\langle V_2, V_4\rangle$ because of the edges $cons_2\langle V_2, (k)\rangle$, $Id\langle (k), V_3\rangle$, and $cons_2^{-1}\langle V_3, V_4\rangle$ (see Figure 6).

To encode the second reduction step of the SC-Reduction Algorithm, the following production is put in the context-free grammar:

$Id ::= Id\ \ Id$

In Example 4.1, the SC-Reduction Algorithm adds the constraint $V_2 \supseteq a$ because of the constraints $V_2 \supseteq V_1$ and $V_2 \supseteq a$. Given that the atomic expression a has index j, the CFL-reachability algorithm adds the edge $Id\langle (j), V_2\rangle$ because of the edges $Id\langle (j), V_1\rangle$ and $Id\langle V_1, V_2\rangle$ (see Figure 6).

Figure 6 shows the graph constructed from Example 4.1 after the CFL-reachability Algorithm is run. The regular term grammar that is the solution to the set-constraint problem can be obtained from this graph by examining Id edges from nodes representing atomic expressions. Thus, the edges $Id\langle (j), V_1\rangle$, $Id\langle (j), V_2\rangle$, and $Id\langle (j), V_4\rangle$ indicate that the atomic expression a reaches set variables V_1, V_2, and V_4; this indicates that the regular term grammar that represents a solution to the set constraints should contain the following productions:

$V_1 \Rightarrow a$

$V_2 \Rightarrow a$

$V_4 \Rightarrow a$

The edge $Id\langle (k), V_3\rangle$ indicates that the following production should be in the regular term grammar as well:

$V_3 \Rightarrow cons(V_1, V_2)$

4.1.2 Accounting for Ground Expressions

For any given set-constraint problem, the construction of Section 4.1.1 does yield a regular term grammar that describes a solution to the problem. However, this regular term grammar does not necessarily describe the least solution.

The problem is that a production of the form "$Id ::= cons_i \;\; Id \;\; cons_i^{-1}$" allows identity paths though *cons* expressions that are not ground, and the production "$Id ::= Id \;\; Id$" propagates non-ground atomic expressions. This is at odds with the simplification steps of the SC-Reduction Algorithm. We consider the problem with productions of the form "$Id ::= cons_i \;\; Id \;\; cons_i^{-1}$" first.

Example 4.2 Let $\mathcal{C}$ be a collection of constraints. Suppose that $\mathcal{C}$ is a superset of the following constraints:

$$\begin{array}{l} V_1 \supseteq a \\ V_3 \supseteq cons(V_1, V_2) \\ V_4 \supseteq cons_1^{-1}(V_3) \\ \quad\vdots \end{array}$$

In the least solution to $\mathcal{C}$, V_2 may or may not be ground. If V_2 is ground, then $cons(V_1, V_2)$ is ground (since V_1 must be ground because of the constraint $V_1 \supseteq a$), and the SC-Reduction Algorithm would perform the following steps:

- Add the constraint $V_4 \supseteq V_1$ (because of the constraints $V_3 \supseteq cons(V_1, V_2)$ and $V_4 \supseteq cons_1^{-1}(V_3)$).
- Add the constraint $V_4 \supseteq a$ (because of the new constraint $V_4 \supseteq V_1$ and the constraint $V_1 \supseteq a$).
- Output the production $V_4 \Rightarrow a$ (because of the new constraint $V_4 \supseteq a$).

If V_2 ultimately is not ground, then the expression $cons(V_1, V_2)$ is not ground, and the SC-Reduction Algorithm does not perform the first two of these steps and might not generate the production $V_4 \Rightarrow a$. (The SC-Reduction Algorithm may still generate $V_4 \Rightarrow a$ as a result of reducing other constraints in $\mathcal{C}$; but it would not generate $V_4 \Rightarrow a$ as a result of reducing the particular constraints discussed above.)

Figure 7 shows a fragment of the graph created to represent these constraints by the construction from Section 4.1.1. (In Figure 7 and the following discussion, we assume that the atomic expression a has index j, and the atomic expression $cons(V_1, V_2)$ has index k.) The CFL-reachability algorithm will add the edge $Id\langle V_1, V_4\rangle$ to this graph regardless of whether or not the expression $cons(V_1, V_2)$ is ground. This is because of the production $Id ::= cons_1 \; Id \; cons_1^{-1}$ and the edges $cons_1\langle V_1, (k)\rangle$, $Id\langle (k), V_3\rangle$, and $cons_1^{-1}\langle V_3, V_4\rangle$. Adding edge $Id\langle V_1, V_4\rangle$ when the expression $cons(V_1, V_2)$ is not ground may lead to a non-minimal solution. In the remainder of the section, we give a modified construction for transforming a set-constraint problem to a CFL-reachability problem. With the modified construction, the edge $Id\langle V_1, V_4\rangle$ would be added if and only if the expression $cons(V_1, V_2)$ is ground.

Remark: Example 4.2 illustrates why it is natural to use CFL-reachability for the analysis of lazy languages: for these languages it is proper to infer that V_4 receives the value a. Because Section 3 gives a construction for converting CFL-reachability problems to set-constraint problems, this shows that set-constraints with strict semantics can be used for the analysis of lazy languages. The latter is not surprising; it is easy to get strict constraints to behave as if they have lazy semantics by artificially grounding each set variable V by adding the constraint $V \supseteq dummy$, where *dummy* is an otherwise unused nullary constructor. For alternative treatments of lazy languages using set constraints see [26, 27].

Example 4.2 suggests that CFL-reachability might not be powerful enough to express analysis problems for strict languages. The construction given in the remainder of this section shows that this is not the case. □

We now give a modified construction in which the production $Id ::= cons_1 \;\; Id \;\; cons_1^{-1}$ is replaced with productions that capture the groundness conditions. To do this we need a technique for tracking additional Boolean information about set variables. (For example, we need to keep track of whether or not a set variable is ground.) In the constructed CFL-reachability problem, set variables are represented by nodes, and we will use cyclic edges to mark Boolean information: the value of a Boolean property of a variable will be indicated by the presence or absence of a cyclic edge at a node. Some of these cyclic edges are generated

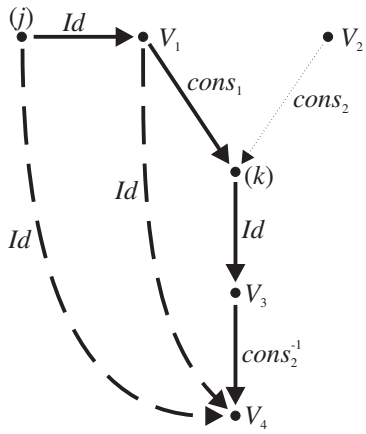


Figure 7: The edge $Id\langle V_1, V_4\rangle$ should be induced if and only if $cons(V_1, V_2)$ is ground. If the edge $Id\langle V_1, V_4\rangle$ is added when $cons(V_1, V_2)$ is not ground, it may incorrectly cause the edge $Id\langle a, V_4\rangle$ to be added, and the production $V_4 \Rightarrow a$ to be output.
The nodes (j) and (k) represent the atomic expressions a and $cons(V_1, V_2)$, respectively.

during the construction of the graph; others are induced by the CFL-reachability Algorithm. The graph and context-free grammar must be constructed properly for this to happen.

In particular, we introduce a new kind of edge label, *Ground*, which will be used to indicate that a variable or atomic expression is ground: an edge $Ground\langle V_i, V_i\rangle$ indicates that the variable V_i is known to be ground, while an edge $Ground\langle (j), (j)\rangle$ indicates that the atomic expression with index j is known to be ground. In Figure 7, the edges $Ground\langle V_1, V_1\rangle$ and $Ground\langle V_2, V_2\rangle$ will be added to the graph if and only if V_1 and V_2 are ground, respectively. The edge $Ground\langle (k), (k)\rangle$ will be added to the graph if and only if $cons(V_1, V_2)$ is known to be ground (i.e., if both V_1 and V_2 are ground).

We now illustrate the use of the *Ground* edges by means of Example 4.2. In Example 4.2, we want the graph to contain the cyclic edge $Ground\langle (k), (k)\rangle$ if and only if $cons(V_1, V_2)$ is ground. In place of the production $Id ::= cons_1\ Id\ cons_1^{-1}$, we use the following production:

$$Id ::= cons_1 \quad Ground \quad Id \quad cons_1^{-1}$$

With this production, the CFL-reachability Algorithm will add the edge $Id\langle V_1, V_4\rangle$ if and only if the edge $Ground\langle (k), (k)\rangle$ is present (i.e., if and only if $cons(V_1, V_2)$ is ground); see Figure 8(a).

We now show how to modify the graph and the productions to deal with *Ground* edges. Some *Ground* edges are generated when constructing the graph. In particular, for every atomic expression of the form a with index j, we generate the edge $Ground\langle (j), (j)\rangle$, because a nullary constructor is always ground.

Other *Ground* edges are induced during the running of the CFL-reachability Algorithm. In Example 4.2, the atomic expression $cons(V_1, V_2)$ is ground if and only if V_1 and V_2 are both ground. We modify the construction so that the following edges are also introduced in the original graph:

$$edge(k)toV_1\langle (k), V_1\rangle$$
$$edgeV_1to(k)\langle V_1, (k)\rangle$$
$$edge(k)toV_2\langle (k), V_2\rangle$$
$$edgeV_2to(k)\langle V_2, (k)\rangle$$

These edges simply connect nodes V_1, V_2, and (k), and allow us to introduce the following production:

$$Ground ::= edge(k)toV_1 \quad Ground \quad edgeV_1to(k) \quad edge(k)toV_2 \quad Ground \quad edgeV_2to(k)$$

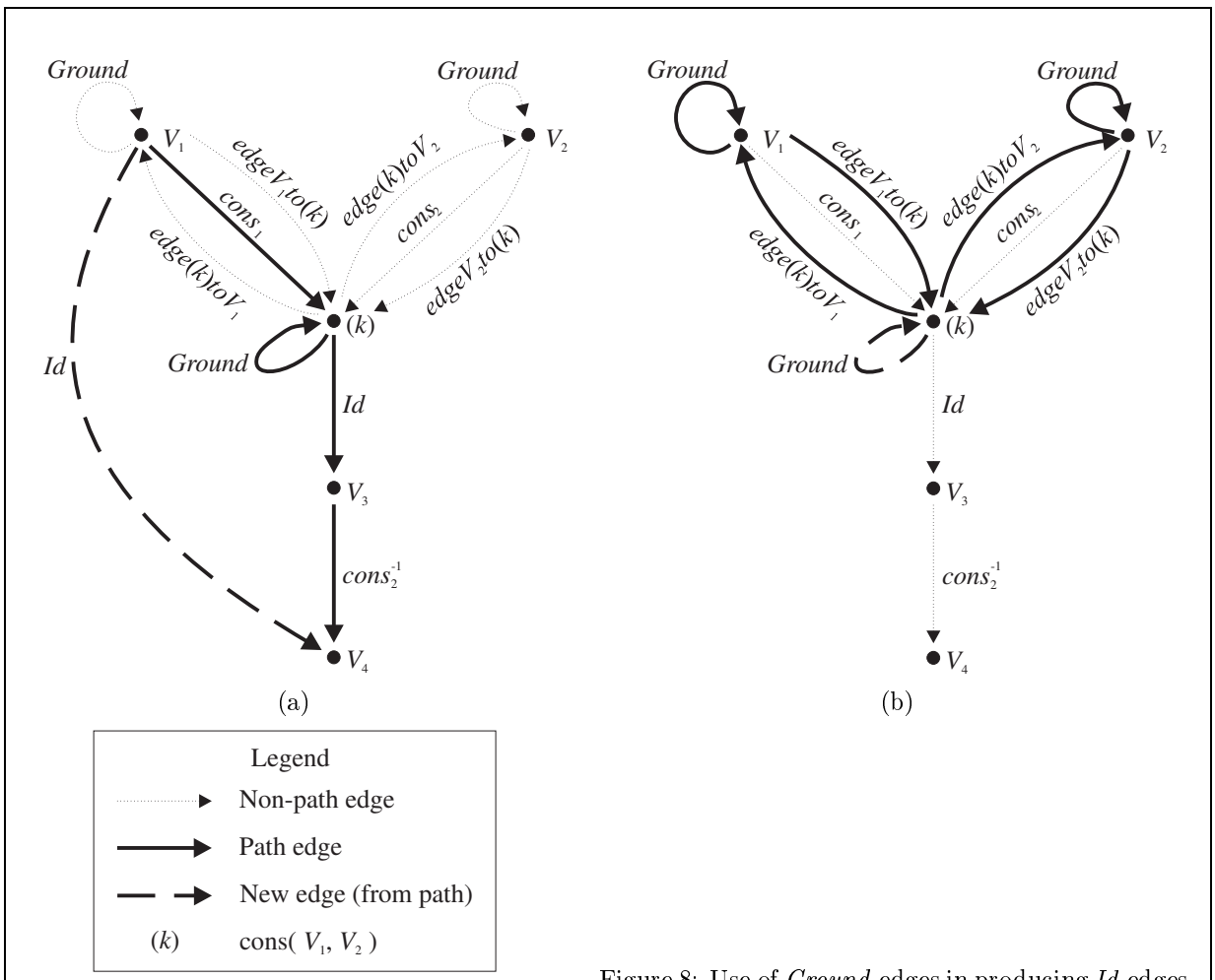


Figure 8: Use of *Ground* edges in producing *Id* edges.

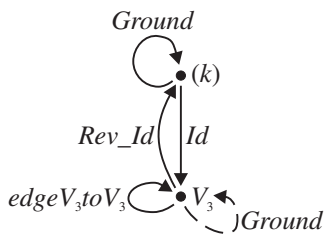


Figure 9: Propagation of *Ground* edges from (k) to V_3. This is accomplished using the production "$Ground ::= edgeV_3toV_3 \;\; Rev_Id \;\; Ground \;\; Id \;\; edgeV_3toV_3$".

With this production and the edges used in it, the CFL-reachability Algorithm will induce the edge $Ground\langle(k),(k)\rangle$ iff the edges $Ground\langle V_1, V_1\rangle$ and $Ground\langle V_2, V_2\rangle$ exist. See Figure 8(b).

There is one last situation we must take into account: Suppose that in Example 4.2 the atomic expression $cons(V_1, V_2)$ (with index k) is known to be ground, and consider the constraint $V_3 \supseteq cons(V_1, V_2)$; this implies that the variable V_3 is also ground. In the graph constructed for this situation, we have the edges $Ground\langle(k),(k)\rangle$ and $Id\langle(k), V_3\rangle$, and we want the edge $Ground\langle V_3, V_3\rangle$ to be added. In effect, we want the *Ground* information at (k) to be propagated along the *Id* edge. To accomplish this, we introduce the edges

$Rev_Id\langle V_3, (k)\rangle$ and $edgeV_3toV_3\langle V_3, V_3\rangle$, and the following production:

$$Ground ::= edgeV_3toV_3 \quad Rev_Id \quad Ground \quad Id \quad edgeV_3toV_3$$

With this production, the CFL-reachability Algorithm will add the edge $Ground\langle V_3, V_3\rangle$ to the graph (see Figure 9).

There is one more issue that is not well illustrated in Example 4.2. In order to propagate ground information along an Id edge, we need a corresponding Rev_Id edge. That is, for any edge $Id\langle V_i, V_j\rangle$ in the graph, we need an edge $Rev_Id\langle V_j, V_i\rangle$ in the reverse direction. We now show how these Rev_Id edges are created. Recall that some Id edges are induced by the CFL-reachability Algorithm. If the CFL-reachability Algorithm induces an edge $Id\langle V_i, V_j\rangle$, then we want it to induce an edge $Rev_Id\langle V_j, V_i\rangle$. To have this happen without changing the CFL-reachability Algorithm, we need to add more productions to the grammar. For example, the following production indicates that the CFL-reachability Algorithm should induce an Id edge (assuming an appropriate path exists in the graph):

$$Id ::= cons_1 \quad Ground \quad Id \quad cons_1^{-1}$$

Consequently, we need an equivalent "reverse" production to indicate that the corresponding Rev_Id edge should be induced:

$$Rev_Id ::= rev_cons_1^{-1} \quad Rev_Id \quad Ground \quad rev_cons_1$$

Figure 10 illustrates the use of this reverse production.

For this production to work, we need additional reverse edges: For every edge $cons_1\langle V_i, V_j\rangle$ in the graph, we want the edge $rev_cons_1\langle V_j, V_i\rangle$ to be in the graph; for every edge $cons_1^{-1}\langle V_i, V_j\rangle$, we want the edge $rev_cons_1^{-1}\langle V_j, V_i\rangle$ to be in the graph. Fortunately, these reverse edges can be added when we construct the graph. They do not require the introduction of new productions. Notice also that an edge labelled $Ground$ is always cyclic. Hence, it can serve as its own reverse edge and so we do not need an edge labelled Rev_Ground.

Now that we have addressed the problems with constraints of the form $Id ::= cons_i \quad Id \quad cons_i^{-1}$, we are ready to address the production $Id ::= Id \quad Id$. In fact there are two problems with this production:

1. Consider the constraints $X \supseteq Y$ and $Y \supseteq cons(Z, W)$ represented by the edges $Id\langle Y, X\rangle$ and $Id\langle (k), Y\rangle$. The production $Id ::= Id \quad Id$ causes the edge $Id\langle (k), X\rangle$ to be introduced, regardless of whether or not $cons(Z, W)$ is ground.

2. Consider the constraints $X \supseteq Y$ and $Y \supseteq Z$, represented by the edges $Id\langle Y, X\rangle$ and $Id\langle Z, Y\rangle$. The production $Id ::= Id \quad Id$ causes the edge $Id\langle Z, X\rangle$ to be introduced.

In both of these cases, the simplification steps of the SC-Reduction Algorithm are not accurately represented. To fix this, for each node (k) representing an atomic expression, we indicate that (k) represents an atomic expression by introducing the edge $ae\langle (k), (k)\rangle$. We replace the production $Id ::= Id \quad Id$ with the following production:

$$Id ::= Ground \quad ae \quad Id \quad Id$$

This production accurately encodes the second reduction step of the SC-Reduction Algorithm.

4.2 Summary of the Construction

Above, we presented the concepts of the construction in terms of a specific example. In this section, we present it for an arbitrary set-constraint problem. In general, the CFL-reachability problem—which consists of a graph and a context-free grammar—is constructed as follows:

1. The context-free grammar contains the productions

$$Id ::= Ground \quad ae \quad Id \quad Id$$
$$Rev_Id ::= Rev_Id \quad Rev_Id \quad Ground \quad ae$$

2. For each set variable V_i, the graph contains a node named V_i, and a uniquely labelled edge $edgeV_itoV_i\langle V_i, V_i\rangle$. The context-free grammar contains the production

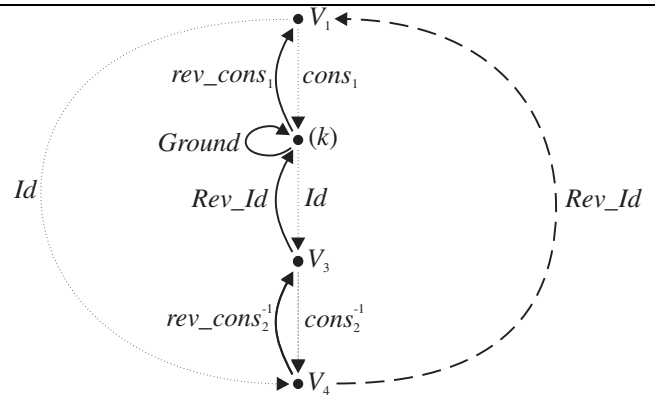


Figure 10: The production $Rev_Id ::= rev_cons_1^{-1} \quad Rev_Id \quad Ground \quad rev_cons_1$ causes the CFL-reachability Algorithm to produce Rev_Id edges. (This production is the counterpart of the production $Id ::= cons_1 \quad Ground \quad Id \quad cons_1^{-1}$.)

$$Ground ::= edgeV_i toV_i \quad Rev_Id \quad Ground \quad Id \quad edgeV_i toV_i$$

3. For each atomic expression $c(V_1, V_2, \ldots, V_r)$ with index k used in the set constraints the graph contains a node labelled (k) and an edge $ae\langle(k),(k)\rangle$. If c is a nullary constructor (i.e., $r = 0$), then the graph contains the edge $Ground\langle(k),(k)\rangle$. Otherwise, for each position j of this atomic expression, the graph contains the edges

   $c_j\langle V_j,(k)\rangle$
   $rev_c_j\langle(k),V_j\rangle$
   $edge(k)toV_j\langle(k),V_j\rangle$
   $edgeV_j to(k)\langle V_j,(k)\rangle$

   and the context-free grammar contains the productions

   $Id ::= c_j \quad Ground \quad Id \quad c_j^{-1}$
   $Rev_Id ::= rev_c_j^{-1} \quad Rev_Id \quad Ground \quad rev_c_j$

   The context free grammar also contains the production

   $$Ground ::= edge(k)toV_1 \quad Ground \quad edgeV_1 to(k) \quad edge(k)toV_2 \quad Ground \quad edgeV_2 to(k) \quad \ldots \quad edge(k)toV_r \quad Ground \quad edgeV_r to(k)$$

4. For each constraint of the form $V_i \supseteq V_j$, the graph contains edges $Id\langle V_j, V_i\rangle$ and $Rev_Id\langle V_i, V_j\rangle$.

5. For each constraint of the form $V \supseteq c(V_1, V_2, \ldots, V_r)$, where $c(V_1, V_2, \ldots, V_r)$ has index k, the graph contains edges $Id\langle(k), V\rangle$ and $Rev_Id\langle V,(k)\rangle$.

6. For each constraint of the form $V_i \supseteq c_k^{-1}(V_j)$, the graph contains edges $c_k^{-1}\langle V_j, V_i\rangle$ and $rev_c_k^{-1}\langle V_i, V_j\rangle$.

After the CFL-reachability Algorithm is run on a constructed problem, a tree grammar representing the solution to the original set-constraint problem is generated as follows: For each edge $Id\langle(k), V_i\rangle$, where k is the index of the atomic expression $c(V_1, V_2, \ldots, V_r)$ output the regular tree production $V_i \Rightarrow c(V_1, V_2, \ldots, V_r)$.

4.3 Correctness of the Construction

We claim that the solution to the CFL-reachability problem gives the solution to the original set constraint problem. Specifically, we have the following theorem:

Theorem 4.3 *Let $\mathcal{C}$ be a collection of set constraints, and let $\mathcal{P}$ be the CFL-reachability problem constructed to represent $\mathcal{C}$. Let H be the regular-tree grammar produced by the SC-Reduction Algorithm when run on $\mathcal{C}$. Let J be the regular-tree grammar produced from the solution to $\mathcal{P}$ (i.e., the grammar produced by outputting a production for each edge of the form $Id\langle(k), V\rangle$ in the solution to $\mathcal{P}$). Then, $H = J$.*

To prove this theorem, we enlist the help of several lemmas, which are proved in Appendix B. In the following lemmas, $\mathcal{C}$ and $\mathcal{P}$ are defined as in Theorem 4.3. We also have the following definitions:

$\mathcal{C}'$ is the collection of set constraints that results from running the SC-Reduction Algorithm on $\mathcal{C}$ (*i.e.*, $\mathcal{C}'$ is $\mathcal{C}$ unioned with the constraints generated by the SC-Reduction Algorithm).

G is the original graph of the CFL-reachability problem $\mathcal{P}$.

G' is the graph that results from running the CFL-reachability algorithm on $\mathcal{P}$ (*i.e.*, G' is G augmented with the edges added by the CFL-reachability Algorithm).

Lemma 4.4 *If $\mathcal{C}'$ contains the constraint $V \supseteq c(V_1, V_2, \ldots, V_r)$, then G' contains the edge $Id\langle(k), V\rangle$, where k is the index of $c(V_1, V_2, \ldots, V_r)$.*

Lemma 4.5 *If G' contains the edge $Id\langle(k), V\rangle$, then $\mathcal{C}'$ contains the constraint $V \supseteq c(V_1, V_2, \ldots, V_r)$ where $c(V_1, V_2, \ldots, V_r)$ is the atomic expression with index k.*

By Lemma 4.4 and Lemma 4.5, we have that $\mathcal{C}'$ contains $V \supseteq c(V_1, V_2, \ldots, V_r)$ iff G' contains $Id\langle(k), V\rangle$. Theorem 4.3 follows immediately.

4.4 Cost of Solving the Constructed CFL-Reachability Problem

A CFL-reachability problem can be solved in time $O(|\Sigma|^3 n^3)$, where n is the number of nodes in the graph and Σ is the alphabet of the grammar. Ordinarily, $|\Sigma|$ is considered to be a constant and is ignored; however, in a constructed CFL-reachability problem, $|\Sigma|$ is $O(t)$, where t is the number of constraints and the constant of proportionality depends on the maximum arity of the constructors. Since n is also $O(t)$, this gives us a bound on the running time to solve the context-free reachability problem of $O(t^6)$, which is worse than the bound of $O(t^3)$ of the SC-Reduction Algorithm.

However, a closer examination of the CFL-reachability Algorithm shows that the worst-case time bound is not realized on constructed CFL-reachability problems. We will focus our analysis on step 4 of the CFL-reachability Algorithm (Algorithm 2.1). In this step, the algorithm processes each edge that appears in the (final) graph. For each edge, it examines the productions in which that edge's label appears on the right-hand side, and attempts to add edges to the graph when it can complete the right-hand side of a production by matching the edge with neighboring edges in the graph. Recall that the CFL-reachability Algorithm will not add an edge to the graph if the edge already exists.

The cost accounting argument presented in this section goes as follows: We show that for each type of label used in the graph, the number of edges with a label of that type is bounded by $O(t^2)$ (this gives an upper bound on the number of edges that the CFL-reachability Algorithm must examine). Also, for any given edge $B\langle i, j\rangle$ in a constructed graph, the amount of work performed can be broken down into two categories:

1. The number of productions examined by the Algorithm: for a given edge $B\langle i, j\rangle$, this is the number of productions in which B appears on the right-hand side of the production. In a constructed CFL-reachability problem, this is bounded by $O(t)$.

2. The number of edges that the CFL-reachability Algorithm attempts to add to the graph: in a constructed CFL-reachability problem, this is bounded by $O(t)$ over all of the productions examined when processing a given edge $B\langle i, j\rangle$.

Thus, the total amount of work performed by the CFL-reachability Algorithm on a constructed problem is $O(t^2) * (O(t) + O(t)) = O(t^3)$.

We start by showing how a constructed grammar can be normalized in Section 4.4.1. In Section 4.4.2, we present Table 3 which summarizes all of the different types of edge labels that may be used in a constructed

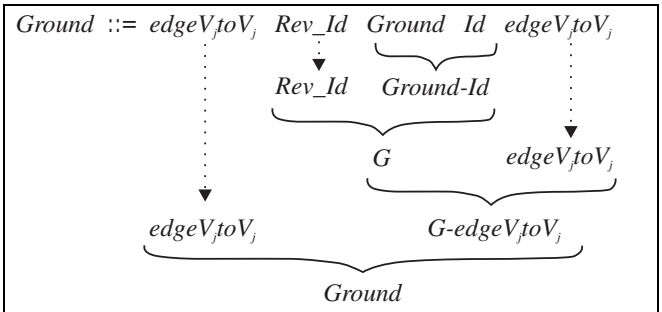


Figure 11: Normalization of the production $Ground ::= edgeV_jtoV_j\ RevId\ Ground\ Id\ edgeV_jtoV_j$.

CFL-reachability problem, including those introduced by the normalization of the grammar. For every given type of edge label, Table 3 also shows a bound on the number of edges with a label of that type, and a bound on the number of steps the CFL-reachability Algorithm performs on any given edge with a label of that type.

Throughout the rest of the section, we use v to refer to the number of variables in the set constraint problem, t to refer to the number of constraints, n to refer the number of nodes in the graph ($n = O(v+t)$), and r to refer to the maximum arity of a constructor.

4.4.1 Normalization of a Constructed Grammar

We start by converting the productions of the grammar to normal form. Consider the following prototypical production:

$$Ground ::= edgeV_jtoV_j\ \ RevId\ \ Ground\ \ Id\ \ edgeV_jtoV_j$$

There are v productions of this form, one for each node V_j. To normalize the production, we introduce several new non-terminals and productions to replace the original production:

$$\begin{array}{rcl} Ground & ::= & edgeV_jtoV_j\ \ G\text{-}edgeV_jtoV_j \\ G\text{-}edgeV_jtoV_j & ::= & G\ \ edgeV_jtoV_j \\ G & ::= & RevId\ \ Ground\text{-}Id \\ Ground\text{-}Id & ::= & Ground\ \ Id \end{array}$$

Figure 11 depicts this normalization. Note that edges labelled Id and Rev_Id may be ubiquitous; they may occur anywhere in the graph. This means that the CFL-reachability Algorithm may use the above productions and put edges labelled $Ground\text{-}Id$ and G anywhere in the graph. However, for any given V_j, there is only one edge labelled $edgeV_jtoV_j$ in the graph; this is the edge $edgeV_jtoV_j\langle V_j, V_j\rangle$. This means that for a fixed V_j, if the CFL-reachability Algorithm adds an edge $G\text{-}edgeV_jtoV_j\langle V_i, V_k\rangle$, then it must use $edgeV_jtoV_j\langle V_j, V_j\rangle$ to do so, and $k = j$. That is, all edges labelled $G\text{-}edgeV_jtoV_j$ must have node V_j as their destination, although they may have any node as their source. This in turn implies that for a fixed node V_j, the number of incoming edges of the form $G\text{-}edgeV_jtoV_j\langle V_i, V_j\rangle$ is bounded by $O(n)$, and the number of outgoing edges of the form $G\text{-}edgeV_ktoV_k\langle V_j, V_k\rangle$ is bounded by $O(n)$. Also, of all the edges $G\text{-}edgeV_jtoV_j\langle V_i, V_j\rangle$, only one, $G\text{-}edgeV_jtoV_j\langle V_j, V_j\rangle$, can be combined with $edgeV_jtoV_j\langle V_j, V_j\rangle$ to generate $Ground\langle V_j, V_j\rangle$.

Now we consider the following prototypical production:

$$Id ::= c_i\ \ \ Ground\ \ \ Id\ \ \ c_i^{-1}$$

There are $O(tr)$ productions of this form, one for each position of each different constructor type used in the constraints. It is normalized to the following productions:

$$\begin{array}{rcl} Id & ::= & c_i\ \ Ground\text{-}Id\text{-}c_i^{-1} \\ Ground\text{-}Id\text{-}c_i^{-1} & ::= & Ground\text{-}Id\ \ \ c_i^{-1} \\ Ground\text{-}Id & ::= & Ground\ \ \ Id \end{array}$$

The corresponding "reverse" production

$$Rev_Id ::= rev_c_i^{-1} \quad Rev_Id \quad Ground \quad rev_c_i$$

is normalized in a similar fashion:

$$\begin{aligned} Rev_Id &::= Rev_c_i^{-1}\text{-}Rev_Id\text{-}Ground \quad rev_c_i \\ Rev_c_i^{-1}\text{-}Rev_Id\text{-}Ground &::= rev_c_i^{-1} \quad Rev_Id\text{-}Ground \\ Rev_Id\text{-}Ground &::= Rev_Id \quad Ground \end{aligned}$$

Recall that *Ground* edges are always cyclic. This means that there are at most $O(n)$ edges with the label *Ground*, and at most $O(n^2)$ edges with the labels *Ground-Id* or *Rev_Id-Ground*. The number of edges with labels of the form c_i, c_i^{-1}, rev_c_i, or $rev_c_i^{-1}$ is bounded by $O(tr)$ (these edges are introduced only when constructing the original graph). This means that the number of edges with a label of the form $Ground\text{-}Id\text{-}c_i^{-1}$ or $rev_c_i^{-1}\text{-}Rev_Id\text{-}Ground$ is bounded by $O(trn)$.

The production

$$Id ::= Ground \quad ae \quad Id \quad Id$$

is normalized to

$$\begin{aligned} Ground\text{-}ae &::= Ground \quad ae \\ Ground\text{-}ae\text{-}Id &::= Ground\text{-}ae \quad Id \\ Id &::= Ground\text{-}ae\text{-}Id \quad Id \end{aligned}$$

The corresponding "reverse" production

$$Rev_Id ::= Rev_Id \quad Rev_Id \quad Ground \quad ae$$

is normalized to

$$\begin{aligned} Ground\text{-}ae &::= Ground \quad ae \\ Rev_Id\text{-}Ground\text{-}ae &::= Rev_Id \quad Ground\text{-}ae \\ Rev_Id &::= Rev_Id\text{-}Ground\text{-}ae \end{aligned}$$

Since *Ground* and *ae* edges are always cyclic, it follows that *Ground-ae* edges are always cyclic. This means the number of edges with the label *Ground-ae* is bounded by $O(t)$, which implies that the number of edges with the labels *Ground-ae-Id* and *Rev_Id-Ground-ae* are bound by $O(tv)$.

We must also normalize productions having the following form:

$$\begin{aligned} Ground \quad ::= \quad & edge(k)toV_1 \quad Ground \quad edgeV_1to(k) \quad edge(k)toV_2 \quad Ground \quad edgeV_2to(k) \\ & \ldots \quad edge(k)toV_r \quad Ground \quad edgeV_rto(k) \end{aligned}$$

There are $O(t)$ productions of this form, one for each atomic expression used in each constraint. This production is replaced by the following productions (which are not in normal form):

$$\begin{aligned} Ground \quad &::= \quad MarkV_1\,GrAt(k) \quad MarkV_2\,GrAt(k) \quad \ldots \quad MarkV_r\,GrAt(k) \\ MarkV_1\,GrAt(k) \quad &::= \quad edge(k)toV_1 \quad Ground \quad edgeV_2to(k) \\ MarkV_1\,GrAt(k) \quad &::= \quad edge(k)toV_2 \quad Ground \quad edgeV_2to(k) \\ &\vdots \\ MarkV_r\,GrAt(k) \quad &::= \quad edge(k)toV_r \quad Ground \quad edgeV_rto(k) \end{aligned}$$

An edge label of the form $MarkV_i\,GrAt(k)$ can only appear on a cyclic edge $MarkV_i\,GrAt(k)\langle(k),(k)\rangle$ at node (k). Such an edge has the effect of "Marking V_i ground at node (k)." Productions of the form

$$MarkV_i\,GrAt(k) ::= edge(k)toV_i \quad Ground \quad edgeV_ito(k)$$

are normalized to the following productions:

$$
\begin{array}{rcl}
MarkV_i\,GrAt(k) & ::= & edge(k)toV_i \quad Ground\text{-}edgeV_i to(k) \\
Ground\text{-}edgeV_i to(k) & ::= & Ground \quad edgeV_i to(k)
\end{array}
$$

Finally, productions of the following form must also be normalized:

$$
Ground ::= MarkV_1\,GrAt(k) \quad MarkV_2\,GrAt(k) \quad \ldots \quad MarkV_r\,GrAt(k)
$$

There are $O(t)$ productions of this form. It is normalized to the following productions:

$$
\begin{array}{rcl}
Ground & ::= & MarkV_1\text{-}V_r\,GrAt(k) \\
MarkV_1\text{-}V_2\,GrAt(k) & ::= & MarkV_1\,GrAt(k) \quad MarkV_2\,GrAt(k) \\
MarkV_1\text{-}V_3\,GrAt(k) & ::= & MarkV_1\text{-}V_2\,GrAt(k) \quad MarkV_3\,GrAt(k) \\
 & \vdots & \\
MarkV_1\text{-}V_r\,GrAt(k) & ::= & MarkV_1\text{-}V_{r-1}\,GrAt(k) \quad MarkV_r\,GrAt(k)
\end{array}
$$

With these normalized productions, the CFL-reachability Algorithm will add at most $O(tr)$ edges with labels of the form $MarkV_1\text{-}V_j\,GrAt(k)$ ($O(r)$ for each of $O(t)$ productions). All of these edges will be cyclic.

4.4.2 Counting Steps

Table 3 lists the various forms of labels that may appear in a constructed graph. For each form of label, it gives a bound on the number of edges with a label of that form (column 2), and shows the productions in which a label of that form appears on the right-hand side (column 3). Also, for each kind of label, Table 3 shows how many productions the CFL-reachability Algorithm may use with a given edge with that kind of label (column 4), and how many new edges the CFL-reachability Algorithm may attempt to produce as a result of examining that edge (column 5). (The latter is the total for all the productions the CFL-reachability Algorithm will examine.)

For example, consider the edge label *Ground-Id*. There may be $O(n^2)$ edges labelled *Ground-Id* in the graph. When the CFL-reachability Algorithm takes a given edge of the form $Ground\text{-}Id\langle V_j, V_k\rangle$ from its worklist, it could potentially examine $O(tr) = O(t)$ productions of the form $Ground\text{-}Id\text{-}c_i^{-1} ::= Ground\text{-}Id \;\; c_i^{-1}$, in which *Ground-Id* appears on the right-hand side. There is one production of this form for every position of every different kind of constructor used in the set-constraint problem. When the algorithm considers one of these productions, it will look for an edge of the form $c_i^{-1}\langle V_k, V_m\rangle$, in an attempt to add the edge $Ground\text{-}Id\text{-}c_i^{-1}\langle V_j, V_m\rangle$. However, edges of the form $c_i^{-1}\langle V_k, V_m\rangle$ are introduced in the graph to encode projection constraints; this means that their number is bounded by $O(t)$. Thus, over all of the $O(t)$ productions of the form $Id\text{-}c_i^{-1} ::= Id \;\; c_i^{-1}$, the CFL-reachability Algorithm will find no more than $O(t)$ matching edges of the form $c_i^{-1}\langle V_k, V_m\rangle$, and so it will add no more than $O(t)$ new edges as a result of processing any given edge of the form $Ground\text{-}Id\langle V_j, V_k\rangle$.

The accounting is more straightforward in most other cases. Table 3 summarizes the results. A bound on the amount of work performed is found by summing column 4 and column 5 and then multiplying by column 2. Since r is constant, and v and n are in the worst case proportional to t, the total running time of the algorithm is bounded by $O(t^3)$.

5 Solving ML Set-Constraint Problems Using CFL-reachability

Heintze has used a modified class of set constraints for set-based analysis of ML programs [17]. We refer to this class of set constraints as *ML set constraints* to distinguish them from the set constraints discussed in the earlier part of the paper. (This class of set constraints can be used to express closure analysis—the problem of determining the set of abstractions that can reach an application—and hence related work includes [35, 47, 45].) In this section, we define ML set constraints and then show how to encode an ML set-constraint problem as a CFL-reachability problem.

5.1 ML Set Constraints

Similar to set expressions defined in Section 2.2.1, an *ML set expression* (se) may be a set variable or a constructor of the form $c(V_1, \ldots, V_r)$. However, ML set expressions do not have explicit projections, but instead may also have the following forms:

- $case(Y_1, c(W_1, \ldots, W_r) \hookrightarrow Y_2,\ W \hookrightarrow Y_3)$. Expressions of this form are used to model case statements. The values in Y_1 are matched against the expression $c(V_1, \ldots, V_r)$. The presence of a ground term of the form $c(v_1, \ldots, v_r)$ in Y_1 indicates that $v_i \in V_i$ for $i = 1 \ldots r$ and that the values of the entire *case* expression are a superset of the values in Y_2. W represents the default branch of the *case* expression. It contains a superset of the values in Y_1 that are ground terms not of the form $c(v_1, \ldots, v_r)$. The presence of a ground value in Y_1 that is not a c term indicates that the values of the entire *case* expression are a superset of the values in Y_3.

  Note that the value decomposition feature of *case* expressions serves as a replacement for the projection operators of the set expressions described is Section 2.2.1.

- "Abstraction constants" of the form $\lambda x.e$. In program-analysis problems, such constants typically play a role in modeling function abstractions: Each abstraction constant is manufactured from a function abstraction in the program (e.g., the x and e in abstraction constant $\lambda x.e$ are derived in some fashion from the textual definition of a function abstraction in the program). The e part of abstraction constant $\lambda x.e$ serves as a tag to distinguish this abstraction constant from other abstraction constants of the set-constraint problem. The x part of abstraction-constant $\lambda x.e$ serves to link $\lambda x.e$ to two associated set variables:

  - V_x, which holds a superset of all the values that may bind to x during program execution.
  - $Range_{\lambda x.e}$, which represents the range of $\lambda x.e$. It holds a superset of all the values that $\lambda x.e$ may return during program execution.

  In program-analysis problems, one would typically standardize names apart, so that each two different abstraction constants $\lambda x.e$ and $\lambda y.e'$ of the set-constraint problem would use different variable names (x, y, etc.).

- $apply(se_1, se_2)$. Expressions of this form are used to model function application.

- $ifnonempty(se_1, se_2)$. Expressions of this form do not directly correspond to any language construct. They are used to make set based analysis more accurate by preventing constraints that correspond to certain infeasible execution configurations from contributing to the solution [17, 43].

ML set constraints are of the form $V \supseteq se$, where se is an ML set expression. A solution to a collection of ML set constraints is a mapping from set variables to a set of values such that the constraints are satisfied. In this case a "value" may be an abstraction $\lambda x.e$ as well as a ground term composed of constructors. Given a mapping $\mathcal{I}$ from set variables to sets of values, the mapping can be extended to map set expressions to sets of values as follows:

- $\mathcal{I}(c(V_1, \ldots, V_r)) = \{c(v_1, \ldots, v_r) | v_1 \in \mathcal{I}(V_1), \ldots, v_r \in \mathcal{I}(V_r)\}$

- $\mathcal{I}(\lambda x.e) = \{\lambda x.e\}$

- $\mathcal{I}(ifnonempty(se_1, se_2)) =$
  if $\mathcal{I}(se_1) = \{\}$ then $\{\}$ else $\mathcal{I}(se_2)$;

- $\mathcal{I}(apply(se_1, se_2)) = \{v : \lambda x.e \in \mathcal{I}(se_1) \wedge \mathcal{I}(se_2) \neq \{\} \wedge v \in \mathcal{I}(Range_{\lambda x.e}))\}$
  provided $\lambda x.e \in \mathcal{I}(se_1)$ implies $\mathcal{I}(se_2) \subseteq \mathcal{I}(V_x)$

- $\mathcal{I}(case(se_1, c(X_1, \ldots, X_n) \hookrightarrow se_2, Y \hookrightarrow se_3)) = S_1 \cup S_2$, where

  1. $S_1 = \{v : v \in \mathcal{I}(se_2) \wedge \exists c(v_1, \ldots, v_n) \in \mathcal{I}(se_1)\}$
  2. $S_2 = \{v : v \in \mathcal{I}(se_3) \wedge \exists c'(v_1, \ldots, v_n) \in \mathcal{I}(se_1)$ s.t. $c' \neq c\}$
  3. For all $c' \neq c$, $c(v_1, \ldots, v_n) \in \mathcal{I}(se_1)$ implies $v_i \in \mathcal{I}(X_i), i = 1 \ldots n$
  4. $c'(v_1, \ldots, v_n) \in \mathcal{I}(se_1)$ implies $c'(v_1, \ldots, v_n) \in \mathcal{I}(Y)$

Note that it is possible for an expression to be undefined in a given mapping. This can happen if the mapping $\mathcal{I}$ does not meet the requirements for interpreting the expression. A solution $\mathcal{I}$ to a collection of constraints $\mathcal{C}$ must define each set expression used in $\mathcal{C}$.

5.1.1 Solving ML Set Constraints

ML set constraints with the following form are said to be in *explicit* form:

$$V \supseteq V_1$$

$$V \supseteq c(V_1, \ldots, V_r)$$

$$V \supseteq \lambda x.e$$

As before, a collection of ML set constraints $\mathcal{C}$ is solved by augmenting the collection with constraints in explicit form until no more can be added. The constraints in explicit form can then be taken to be a regular term grammar that represents the least solution to the constraints. The ML-SC-Reduction Algorithm is defined below. *Groundness* is defined as in Section 2.2.3.

Algorithm 5.1 (ML-SC-Reduction Algorithm) Given a collection of ML set constraints $\mathcal{C}$, the following steps are repeated until neither step causes $\mathcal{C}$ to change:

1. if $X \supseteq \mathit{apply}(X_1, X_2)$ and $X_1 \supseteq \lambda x.e$ both appear in $\mathcal{C}$ then
   (a) add the constraint $X \supseteq \mathit{Range}_{\lambda x.e}$ to $\mathcal{C}$
   (b) add the constraint $V_x \supseteq X_2$ to $\mathcal{C}$
2. if $X \supseteq \mathit{case}(Y_1, c(W_1, \ldots, W_r) \hookrightarrow Y_2,\ W \hookrightarrow Y_3)$ and $Y_1 \supseteq c(Z_1, \ldots, Z_r)$ both appear in $\mathcal{C}$ and the expression $c(Z_1, \ldots, Z_r)$ is ground then
   (a) add the constraint $X \supseteq Y_2$ to $\mathcal{C}$
   (b) for $i = 1 \ldots n$, add the constraint $W_i \supseteq Z_i$ to $\mathcal{C}$
3. if $X \supseteq \mathit{case}(Y_1, c(W_1, \ldots, W_r) \hookrightarrow Y_2,\ W \hookrightarrow Y_3)$ and $Y_1 \supseteq c'(Z_1, \ldots, Z_r)$ both appear in $\mathcal{C}$ where $c' \neq c$ and the expression $c'(Z_1, \ldots, Z_r)$ is ground then
   (a) add the constraint $X \supseteq Y_3$ to $\mathcal{C}$
   (b) add the constraint $W \supseteq c'(Z_1, \ldots, Z_r)$ to $\mathcal{C}$
4. if $X \supseteq \mathit{ifnonempty}(Y_1, Y_2)$ appears it $\mathcal{C}$ and Y_1 is ground then add $X \supseteq Y_2$ to $\mathcal{C}$
5. if $X \supseteq X'$ and $X' \supseteq se$ both appear it $\mathcal{C}$, where $X' \supseteq se$ is in explicit form, then add $X \supseteq se$ to $\mathcal{C}$

When no more constraints can be added, the constraints in explicit form are converted to a regular term grammar; this describes the least solution [17]. □

5.2 Solving ML Set-Constraint Problems Using CFL-reachability

The idea for encoding an ML set-constraint problem is the same as in Section 4.1: we view the ML SC-Reduction Algorithm as computing what atomic expressions reach each set variable and construct a CFL-reachability problem that computes the same information. The constructed graph contains a node for each atomic expression and a node for each set variable. Where the ML SC-Reduction Algorithm produces the explicit constraint $V \supseteq ae$, the constructed CFL-reachability problem induces an *identity path* from the node representing atomic expression ae to the node representing the set variable V.

In the rest of this section, we first describe how to construct a graph to encode a collection of ML set constraints. Then we show what productions are used to encode the steps of the ML SC-Reduction Algorithm for a given collection of ML set constraints. The techniques for handling groundness information in the problem constructed here is the same as in Section 4.1.2. As in Section 4.1.2, for every edge from node i to node j, we need a corresponding reverse edge from node j to node i. To simplify of presentation, we will not explicitly list the reverse edges (nor the productions that generate them), but we assume that they are also produced.

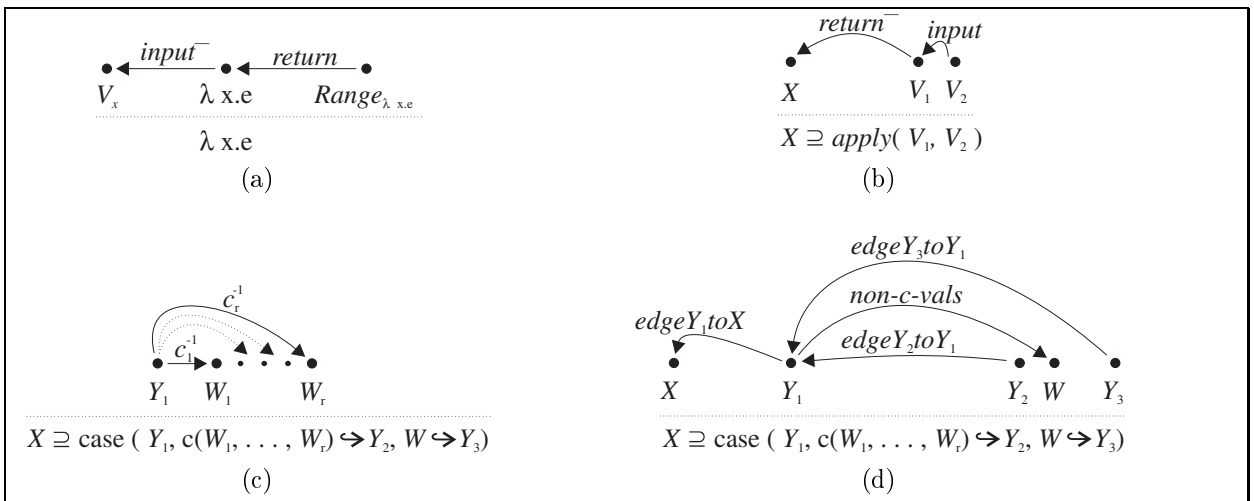


Figure 12: Edges inserted in the constructed graph to model ML set constraints.

5.2.1 Encoding ML Set Constraints

Given a collection of constraints $\mathcal{C}$, the graph encoding these constraints is constructed as follows:

- For each set variable V_i, the graph contains a node labelled V_i, and an edge $edgeV_i toV_i\langle V_i, V_i\rangle$.
- Each atomic expression $c(V_1, \ldots, V_r)$ used in a constraint of the form $V \supseteq c(V_1, \ldots, V_r)$ is associated with a unique index.

  Given an expression $c(V_1, \ldots, V_r)$ with index k, the graph contains a node labelled (k), and the edges $ae\langle(k), (k)\rangle$ and $c\text{-}value\langle(k), (k)\rangle$. The edge $c\text{-}value\langle(k), (k)\rangle$ indicates that the node (k) can represent a c value. (The node (k) actually represents a c-value iff $c(V_1, \ldots, V_r)$ is ground; thus it is really the presence of a pair of edges $c\text{-}value\langle(k), (k)\rangle$ and $Ground\langle(k), (k)\rangle$ that indicates that (k) is known to be a ground c term.) If c is a nullary constructor, then the graph contains the edge $Ground\langle(k), (k)\rangle$. Otherwise, for each position j of the atomic expression (where $j = 1 \ldots r$), the graph contains the edges $c_j\langle V_j, (k)\rangle$, $edge(k)toV_j\langle(k), V_j\rangle$, and $edgeV_j to(k)\langle V_j, (k)\rangle$.
- For each constraint of the form $V \supseteq c(V_1, \ldots, V_r)$, where the expression $c(V_1, \ldots, V_r)$ has index k, the graph contains an edge $Id\langle(k), V\rangle$ indicating that the atomic expression $c(V_1, \ldots, V_r)$ reaches V.
- For each atomic expression $\lambda x.e$, the graph contains a node labelled $\lambda x.e$ and an edge $Ground\langle\lambda x.e, \lambda x.e\rangle$. The node $\lambda x.e$ is connected to the nodes representing the set variables V_x and $Range_{\lambda x.e}$ by the edges $input^-\langle\lambda x.e, V_x\rangle$ and $return\langle Range_{\lambda x.e}, \lambda x.e\rangle$ (See Figure 12(a)). An edge $input^-\langle\lambda x.e, V_x\rangle$ indicates that values to which the abstraction $\lambda x.e$ is applied reach the variable V_x. An edge $return\langle Range_{\lambda x.e}, \lambda x.e\rangle$ indicates that $Range_{\lambda x.e}$ holds a superset of the values returned by the abstraction $\lambda x.e$ during program execution.
- For each constraint of the form $V \supseteq apply(V_1, V_2)$, the graph contains the following edges:
  - $return^-\langle V_1, V\rangle$. This edge indicates that V contains values that are returned by abstractions in V_1.
  - $input\langle V_2, V_1\rangle$. This edge indicates that values in V_2 are potential arguments of abstractions in V_1.

  See Figure 12(b).
- For each constraint of the form $V \supseteq \lambda x.e$, the graph contains an edge $Id\langle\lambda x.e, V\rangle$.
- For each constraint of the form $V_i \supseteq V_j$, the graph contains an edge $Id\langle V_j, V_i\rangle$.
- For each constraint of the form $X \supseteq case(Y_1, c(W_1, \ldots, W_r) \hookrightarrow Y_2,\ W \hookrightarrow Y_3)$, the graph contains the following edges:

1. $c_i^{-1}\langle Y_1, W_i\rangle$ where $i = 1 \ldots r$
2. $non\text{-}c\text{-}vals\langle Y_1, W\rangle$
3. $edgeY_2toY_1\langle Y_2, Y_1\rangle$
4. $edgeY_3toY_1\langle Y_3, Y_1\rangle$
5. $edgeY_1toX\langle Y_1, X\rangle$

Figure 12(c) illustrates point 1 above, and Figure 12(d) illustrates points 2 thru 5.

- For each constraint of the form $X \supseteq ifnonempty(Y_1, Y_2)$ the graph contains the edges $edgeY_2toY_1\langle Y_2, Y_1\rangle$ and $edgeY_1toX\langle Y_1, X\rangle$

5.2.2 Encoding the ML SC-Reduction Algorithm

The productions used to encode the ML SC-Reduction Algorithm are a superset of the productions used to encode the SC-Reduction Algorithm. The productions introduced in Section 4.1.2 are again used to propagate groundness information. For each node representing a variable V_i, there is a production

$$Ground ::= edgeV_itoV_i\ \ Rev_Id\ \ Ground\ \ Id\ \ edgeV_itoV_i$$

For each atomic expression of the form $c(V_1, \ldots, V_r)$ with index k, the context-free grammar contains the following production:

$$\begin{aligned} Ground ::= \ & edge(k)toV_1\ \ Ground\ \ edgeV_1to(k) \\ & edge(k)toV_2\ \ Ground\ \ edgeV_2to(k) \\ & \ldots \\ & edge(k)toV_r\ \ Ground\ \ edgeV_rto(k) \end{aligned}$$

In Section 3, the production $Id ::= Ground\ \ ae\ \ Id\ \ Id$ encodes Step 2 of the SC-Reduction Algorithm; now we use it to encode Step 5 of the ML SC-Reduction Algorithm. Similarly, productions of the form

$$Id ::= c_i\ \ Ground\ \ Id\ \ c_i^{-1}$$

were used earlier to encode Step 1 of the SC-Reduction Algorithm; now they encode the actions taken by Step 2(b) of the ML SC-Reduction Algorithm. (See Figure 14(a).)

New productions are needed to encode Steps 1, 2(a), 3, and 4 of the ML SC-Reduction Algorithm. In the following examples, we introduce the productions used to encode these steps.

Example 5.1 Consider the following constraints:

$$X \supseteq apply(V_1, V_2)$$
$$V_1 \supseteq \lambda x.e$$

Given these constraints, Step 1(a) of the ML SC-Reduction Algorithm introduces $X \supseteq Range_{\lambda x.e}$. This constraint is added because the result of the *apply* expression includes values in the range of any abstraction that reaches V_1. In the constructed graph, the presence of the edge $Id\langle\lambda x.e, V_1\rangle$ indicates that the abstraction $\lambda x.e$ reaches V_1. To simulate the actions of Step 1(a), the CFL-reachability algorithm uses the edges $Id\langle\lambda x.e, V_1\rangle$ and $return^-\langle V_1, X\rangle$, and the production $Id ::= return\ Id\ return^-$. (See Figure 13(a).)

Given the above constraints, Step 1(b) of the ML SC-Reduction Algorithm introduces the constraint $V_x \supseteq V_2$; the semantics of the *apply* expression demand that for any abstraction $\lambda x.e$ that reaches V_1, the values in V_2 should reach X. This is simulated in the CFL-reachability Algorithm by the edges $input\langle V_2, V_1\rangle$, $Rev_Id\langle V_1, \lambda x.e\rangle$, and $input^-\langle\lambda x.e, V_x\rangle$ and the production $Id ::= input\ Rev_Id\ input^-$. (See Figure 13(b).) □

In the following example, we introduce productions for encoding Step 2(a) and Step 3 of the ML SC-Reduction Algorithm.

Example 5.2 Consider the following constraints:

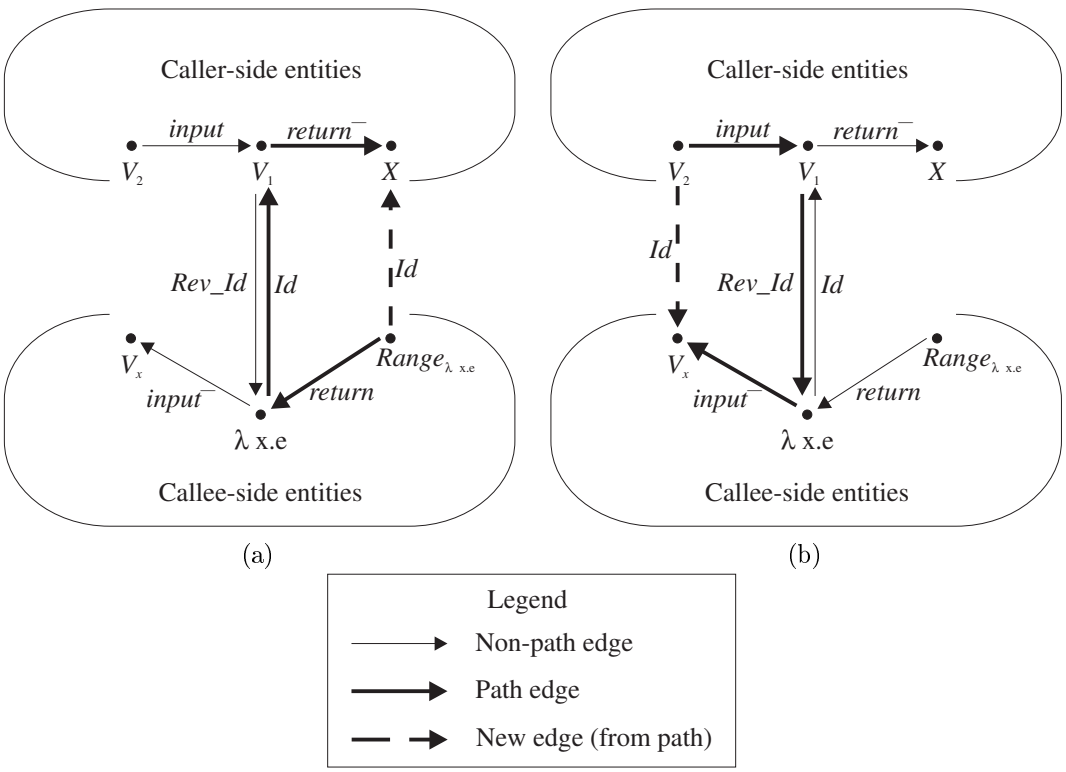


Figure 13: Graphs showing edge induction for constraints of the form $X \supseteq apply(V_1, V_2)$ and $V_1 \supseteq \lambda x.e$.

1. $X \supseteq case(Y_1, cons(W_1, W_2) \hookrightarrow Y_2, W \hookrightarrow Y_3)$
2. $Y_1 \supseteq cons(V_1, V_2)$
3. $Y_1 \supseteq succ(Z_1)$

Let $cons(V_1, V_2)$ and $succ(Z_1)$ have indices j and k, respectively, and suppose both expressions are ground. Figure 14 shows the graph constructed to represent the above constraints (and many subgraphs of this graph). The features of this graph are explained below.

Step 2(a) of the ML SC-Reduction Algorithm introduces the constraint $X \supseteq Y_2$ iff a ground *cons* expression reaches Y_1; in this example, $X \supseteq Y_2$ is introduced because of the constraint $Y_1 \supseteq cons(V_1, V_2)$, and the assumption that $cons(V_1, V_2)$ is ground. In the constructed graph, a node (m) represents a ground *cons* expression iff the graph contains both of the edges $Ground\langle(m),(m)\rangle$ and $cons\text{-}value\langle(m),(m)\rangle$. To encode the actions of Step 2(a) on the above *case* constraint, the constructed graph contains the edges $edgeY_2toY_1\langle Y_2, Y_1\rangle$ and $edgeY_1toX\langle Y_1, X\rangle$ and the grammar contains the following production:

$$Id ::= edgeY_2toY_1\ Rev_Id\ cons\text{-}value\ Ground\ cons\text{-}value\ Id\ edgeY_1toX$$

(See Figure 14(b).) (Note, the reason this production has two occurrences of the terminal symbol *cons-value* has to do with limiting the possible blow-up in the running time required to solve the constructed CFL-reachability problem. This feature will be explained in Section 5.3. The production is still correct if either of these terminals is removed.)

Given the above constraints, Step 3(a) of ML SC-Reduction Algorithm introduces the constraint $X \supseteq Y_3$ iff a ground expression of the form $c(V_1, \ldots, V_r)$ reaches Y_1, where $c \neq cons$; in this case, the constraint $Y_1 \supseteq succ(Z_1)$ and the assumption that $succ(Z_1)$ is ground mean that $X \supseteq Y_3$ is generated. In the constructed graph, the edges $Ground\langle(m),(m)\rangle$ and $c\text{-}value\langle(m),(m)\rangle$ indicate that node (m) represents a

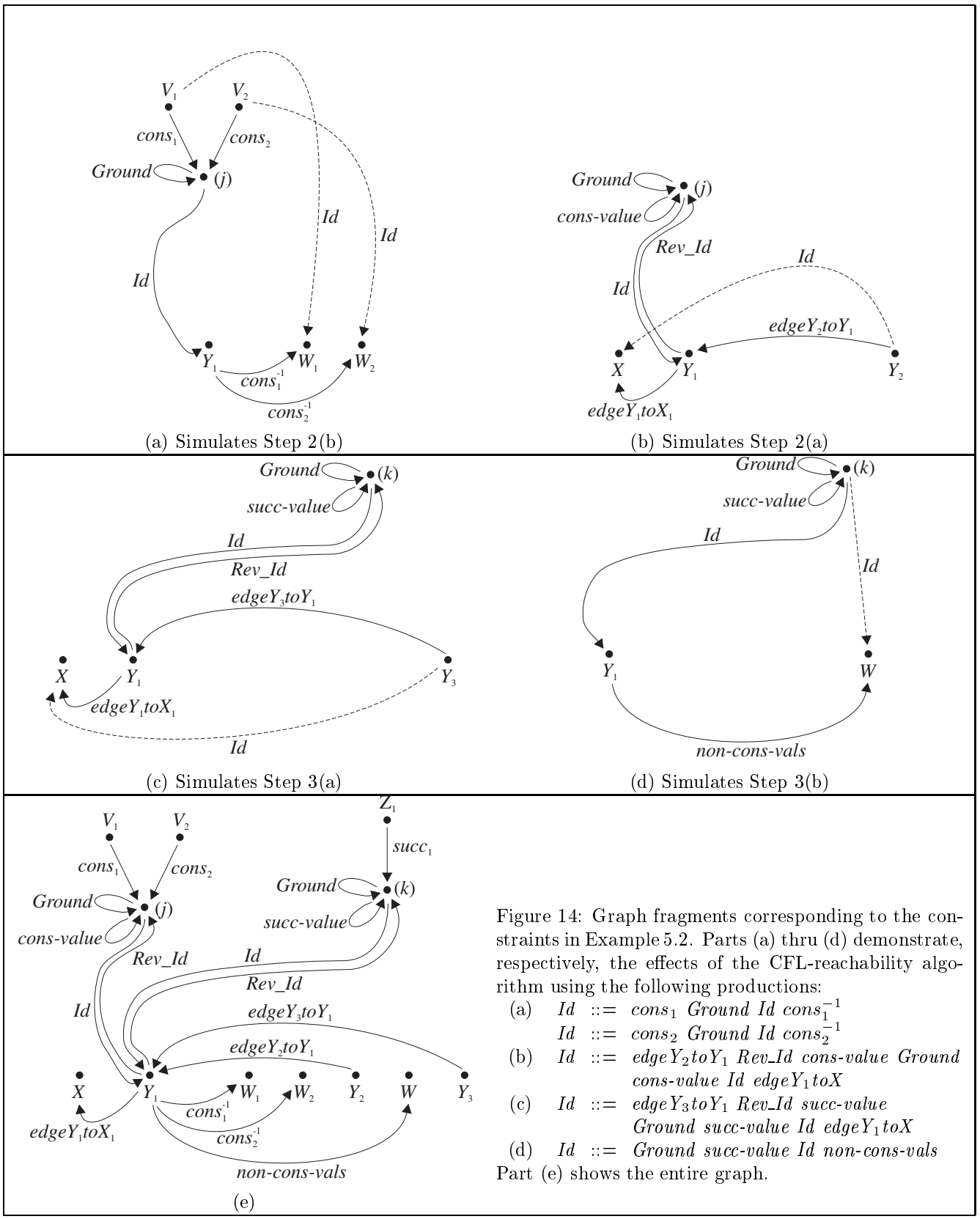


Figure 14: Graph fragments corresponding to the constraints in Example 5.2. Parts (a) thru (d) demonstrate, respectively, the effects of the CFL-reachability algorithm using the following productions:

(a) $Id ::= cons_1\ Ground\ Id\ cons_1^{-1}$
$Id ::= cons_2\ Ground\ Id\ cons_2^{-1}$

(b) $Id ::= edgeY_2toY_1\ Rev_Id\ cons\text{-}value\ Ground\ cons\text{-}value\ Id\ edgeY_1toX$

(c) $Id ::= edgeY_3toY_1\ Rev_Id\ succ\text{-}value\ Ground\ succ\text{-}value\ Id\ edgeY_1toX$

(d) $Id ::= Ground\ succ\text{-}value\ Id\ non\text{-}cons\text{-}vals$

Part (e) shows the entire graph.

ground expression of the form $c(V_1, \ldots, V_r)$. To encode the actions of Step 3(a) on the above *case* constraint, the graph contains the edges $edgeY_3toY_1\langle Y_3, Y_1\rangle$ and $edgeY_1toX\langle Y_1, X\rangle$ and the grammar contains the following productions:

$$Id ::= edgeY_2toY_1\ Rev_Id\ c\text{-}value\ Ground\ c\text{-}value\ Id\ edgeY_1toX$$

for each constructor c such that $c \neq cons$. (See Figure 14(c).) Note that there is one production of this form for each constructor type for each *case* constraint. This means that the construction is no longer linear in time, but its running time is bounded by $O(t^3)$. (See Section 5.3.)

Step 3(b) of the ML SC-Reduction Algorithm allows ground atomic expressions of the form $c(V_1, \ldots, V_r)$ to pass from Y_1 to W iff $c \neq cons$. In the current example, the constraint $W \supseteq succ(Z_1)$ is introduced. To encode Step 3(b), we use the edge $non\text{-}cons\text{-}vals\langle Y_1, W\rangle$, and constraints of the form

$$Id ::= Ground\ \ c\text{-}value\ \ Id\ \ non\text{-}cons\text{-}vals$$

for each constructor type c such that $c \neq cons$. (See Figure 14(d).) □

Finally, we must encode the action taken by Step 4 of the ML SC-Reduction Algorithm on a constraint of the form $X \supseteq ifnonempty(Y_1, Y_2)$. This is done using the edges $edgeY_2toY_1$ and $edgeY_1toX$ and the following production:

$$Id ::= edgeY_2toY_1\ Ground\ edgeY_1toX$$

As in Section 4, the regular term grammar that is the solution to the ML set-constraint problem can be obtained from the solution to the constructed CFL-reachability problem by examining *Id* edges. For each *Id* edge from a node representing an atomic expression ae, to a node representing a variable V, the regular term grammar contains a production of the form $V \Rightarrow ae$.

5.3 Cost of Solving the Constructed CFL-Reachability Problem

As with the construction described in Section 4.4, when we plug the various parameters that characterize the size of the constructed CFL-reachability problem into the standard formula for the worst-case asymptotic running time of CFL-reachability, we have not preserved the $O(t^3)$ bound on the time to solve ML set-constraint problems. In this section, by an argument similar to that used in Section 4.4, we show that the constructed CFL-reachability problem can indeed be solved in $O(t^3)$.

Below, we first discuss why it is necessary to repeat terminal symbols in some of the productions presented in the Section 5.2. In Section 5.3.2, we list the normalizations of the productions that are new to Section 5. Finally, Table 4 summarizes the work done for each edge added by the CFL-reachability Algorithm while solving a problem constructed from an ML set-constraint problem.

5.3.1 Repeating Terminal Symbols

In Section 5.2, we introduced some productions that have seemingly unnecessary repetitions of some terminal symbols. In particular, a production of the form

$$Id ::= edgeY_3toY_1\ Rev_Id\ c\text{-}value\ Ground\ c\text{-}value\ Id\ edgeY_1toX$$

causes the CFL-reachability Algorithm to induce an *Id* edge exactly when the production

$$Id ::= edgeY_3toY_1\ Rev_Id\ Ground\ c\text{-}value\ Id\ edgeY_1toX$$

causes the CFL-reachability Algorithm to induce an *Id* edge. This follows from the fact that the labels *c-value* and *Ground* always appear on cyclic edges. However, while the productions are functionally equivalent, every normalization of the latter production either introduces a non-terminal that might label $O(tn^2)$ edges and participate in $O(t)$ productions, or introduces a non-terminal that might appear on $O(tn)$ edges and participates in $O(t^2)$ productions. Either way, the bound on the running time of the CFL-reachability Algorithm increases to $O(t^4)$.

Adding the second *c-value* allows us to find a normalization that avoids this blowup. To see why, let us examine in more detail what goes wrong when the production

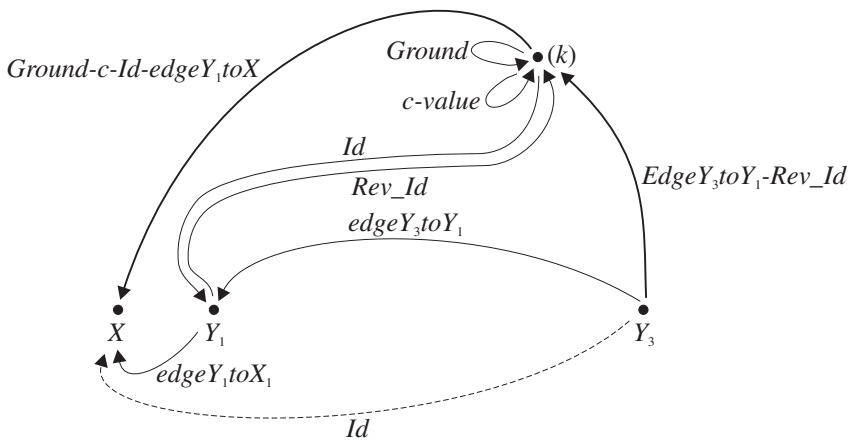


Figure 15: Graph showing the need to double terminals in some productions. The bold edges are used by the normalized production $Id ::= EdgeY_2toY_1\text{-}Rev_Id\ \ Ground\text{-}c\text{-}Id\text{-}edgeY_1toX$; note that the edge $EdgeY_2toY_1\text{-}Rev_Id\langle Y_3,(k)\rangle$ may be used in $O(t^2)$ productions of this form. However, suppose the edge $EdgeY_2toY_1\text{-}Rev_Id\langle Y_3,(k)\rangle$ is first paired with the edge $c-value\langle(k),(k)\rangle$ to generate the edge $EdgeY_2toY_1\text{-}Rev_Id\text{-}c\langle Y_3,(k)\rangle$. An edge of this form can be used with only $O(t)$ productions of the form $Id ::= EdgeY_2toY_1\text{-}Rev_Id\text{-}c\ \ Ground\text{-}c\text{-}Id\text{-}edgeY_1toX$ to generate the same Id edges.

$$Id ::= edgeY_3toY_1\ \ Rev_Id\ \ Ground\ \ c\text{-}value\ \ Id\ \ edgeY_1toX$$

is normalized to the productions:

1. $Ground\text{-}c ::= Ground\ \ c\text{-}value$
2. $Ground\text{-}c\text{-}Id ::= Ground\text{-}c\ \ Id$
3. $Ground\text{-}c\text{-}Id\text{-}edgeY_1toX ::= Ground\text{-}c\text{-}Id\ \ edgeY_1toX$
4. $EdgeY_2toY_1\text{-}Rev_Id ::= edgeY_2toY_1\ \ Rev_Id$
5. $Id ::= EdgeY_2toY_1\text{-}Rev_Id\ \ Ground\text{-}c\text{-}Id\text{-}edgeY_1toX$

Notice that there are $O(t)$ productions of the form of the fifth production for each of $O(t)$ different constructor types. The problem with this normalization is with the non-terminal $edgeY_2toY_1\text{-}Rev_Id$ and the fifth production. There may be $O(tn)$ edges labelled with this non-terminal, each involved in $O(t^2)$ productions of the form of the fifth production above. Consider a particular edge $edgeY_2toY_1\text{-}Rev_Id\langle i,j\rangle$ that has node j as its target. There can be at most one edge of the form $d\text{-}value\langle j,j\rangle$ for at most one constructor type d. For all edges labelled $Ground\text{-}d'\text{-}Id\text{-}edgeY_1toX$ that leave node j, it must be the case that $d = d'$. This means that there can be a maximum of $O(t)$ edges which leave node j and have a label of the form $Ground\text{-}d'\text{-}Id\text{-}edgeY_1toX$. This implies that when the CFL-reachability Algorithm examines the edge $edgeY_2toY_1\text{-}Rev_Id\langle i,j\rangle$ and looks at $O(t^2)$ productions, all but $O(t)$ of these productions cause the CFL-reachability Algorithm to search for a second edge that cannot exist.

In contrast, the production

$$Id ::= edgeY_3toY_1\ \ Rev_Id\ \ c\text{-}value\ \ Ground\ \ c\text{-}value\ \ Id\ \ edgeY_1toX$$

can be normalized to

1. *Ground-c* ::= *Ground c-value*
2. *Ground-c-Id* ::= *Ground-c Id*
3. *Ground-c-Id-edgeY*$_1$*toX* ::= *Ground-c-Id edgeY*$_1$*toX*
4. *EdgeY*$_2$*toY*$_1$*-Rev_Id* ::= *edgeY*$_2$*toY*$_1$ *Rev_Id*
5. *EdgeY*$_2$*toY*$_1$*-Rev_Id-c* ::= *EdgeY*$_2$*toY*$_1$*-Rev_Id c-value*
6. *Id* ::= *EdgeY*$_2$*toY*$_1$*-Rev_Id-c Ground-c-Id-edgeY*$_1$*toX*

In this normalization, the nonterminal *EdgeY*$_2$*toY*$_1$*-Rev_Id-c* may appear on $O(t^2)$ edges, but it participates in only $O(t)$ productions of the sixth form. In effect, production five, of which there are only $O(t)$ for a given edge of the form *EdgeY*$_2$*toY*$_1$*-Rev_Id*$\langle i, j\rangle$, forces the CFL-reachability Algorithm to determine what constructor type, if any, is represented at node j before it starts to consider productions that include the non-terminal *Ground-c-Id-edgeY*$_1$*toX*. See Figure 15.

5.3.2 Normalization of the Constructed Grammar

Normalization of the context-free grammar in a constructed problem is done as in Section 4.4.1. In fact, since the productions used to encode the ML SC-Reduction Algorithm are a superset of the productions used to encode the SC-Reduction Algorithm, all of the normalizations from Section 4.4.1 are needed for a CFL-reachability problem constructed from an ML set-constraint problem; these normalizations are not repeated here. We also do not show the normalization of "reverse" productions that have *Rev_Id* on their left-hand side; the normalization of a reverse production is the reverse of the normalization for the corresponding forward production.

The normalizations of the productions new to this section are as follows:

- *Id* ::= *input Rev_Id input*$^-$ is normalized to

  Input-Rev_Id ::= *input Rev_Id*
  Id ::= *Input-Rev_Id input*$^-$

- *Id* ::= *return Id return*$^-$ is normalized to

  Return-Id ::= *return Id*
  Id ::= *Return-Id return*$^-$

- *Id* ::= c_i *Ground Id* c_i^{-1} is normalized to

  Ground-Id ::= *Ground Id*
  C_i*-Ground-Id* ::= c_i *Ground-Id*
  Id ::= C_i*-Ground-Id* c_i^{-1}

- *Id* ::= *edgeY*$_2$*toY*$_1$ *c-value Rev_Id Ground c-value Id edgeY*$_1$*toX* is normalized to

  Ground-c ::= *Ground c-value*
  Ground-c-Id ::= *Ground-c Id*
  Ground-c-Id-edgeY$_1$*toX* ::= *Ground-c-Id edgeY*$_1$*toX*
  EdgeY$_2$*toY*$_1$*-Rev_Id* ::= *edgeY*$_2$*toY*$_1$ *Rev_Id*
  EdgeY$_2$*toY*$_1$*-Rev_Id*-c ::= *edgeY*$_2$*toY*$_1$*-Rev_Id c-value*
  Id ::= *EdgeY*$_2$*toY*$_1$*-Rev_Id*-c *Ground-c-Id-edgeY*$_1$*toX*

- *Id* ::= *Ground c'-value Id non-c-value* is normalized to

  Ground-c' ::= *Ground c'-value*
  Ground-c'-Id ::= *Ground-c' Id*
  Id ::= *Ground-c'-Id non-c-values*

- $Id ::= edgeY_2toY_1\ Ground\ edgeY_1toX$ is normalized to

  $Ground\text{-}edgeY_1toX ::= Ground\ edgeY_1toX$
  $Id ::= edgeY_2toY_1\ Ground\text{-}edgeY_1toX$

Table 4 together with Table 3 lists the costs entailed by the processing steps of the algorithm for solving CFL-reachability problems from Section 2.1.1. A bound on the amount of work performed is found by summing column 4 and column 5 and then multiplying by column 2. Since r is constant, and v, k, and n are in the worst case proportional to t, the total running time of the algorithm is bounded by $O(t^3)$.

6 Solving CFL-Reachability Problems Using ML Set Constraints

In this section, we discuss how ML set constraints can be used to solve CFL-reachability problems. First note that a projection constraint of the form

$$U \supseteq c_i^{-1}(V)$$

from the class of set constraints presented in Section 2.2 can be modelled by the ML set constraint

$$U \supseteq case(V, c(T_1, \ldots, T_i, \ldots, T_r) \hookrightarrow T_i, X \hookrightarrow Y)$$

where $T_1 \ldots T_r$, X, and Y are new variables. Note that to have the same semantics as the projection constraint, it is important that Y map to the empty set; otherwise, values from Y may reach U, which is not part of the semantics of the projection constraint.

By replacing projections with *case* expressions in this fashion, the construction in Section 3 becomes a transformation from CFL-reachability problems to ML set-constraint problems. The run time for an ML set-constraint problem constructed in this way has a higher constant of proportionality than a constructed set-constraint problem from Section 3, although the asymptotic run time is the same. In particular, the construction from Section 3, a constraint of the form $Rchd_{[B_1^{-1},i]} \supseteq C(V)$ may pair with at most $O(s)$ projection constraints of the form $Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$, where s is the number of symbols of the context-free grammar of the original CFL-reachability problem.

In a constructed ML set-constraint problem, a constraint of the form $Rchd_{[B_1^{-1},i]} \supseteq C(V)$ may match at most $O(s)$ *case* constraints of the following form:

$$Dst_{[A,i]} \supseteq case(Rchd_{[B_1^{-1},i]}, C(T) \hookrightarrow T, X \hookrightarrow Y)$$

However, the constraint $Rchd_{[B_1^{-1},i]} \supseteq C(V)$ may also match the "default" case of as many as $O(s^2)$ *case* constraints of the form

$$Dst_{[A,i]} \supseteq case(Rchd_{[B_1^{-1},i]}, D(T) \hookrightarrow T, X \hookrightarrow Y)$$

where $D \neq C$. This means that the time needed to solve a constructed ML set-constraint problem may be $O(s^4n^3)$, where n is the number of nodes in the original CFL-reachability problem. (The time needed to solve a constructed set-constraint problem from Section 3 is $O(s^3n^3)$.) Since s is a constant independent of the input, the total run time is still bounded by $O(n^3)$.

Of course, it is also possible to optimize ML set constraints to allow "don't care" defaults that will not match anything. If this is done, the runtime for a constructed ML set-constraint problem is the same as the runtime for a constructed set-constraint problem.

7 Related Work and Concluding Remarks

7.1 Broader Classes of Set Constraints

This paper has presented interconvertibity results for context-free reachability problems and two classes set-constraints. However, the problem of satisfiability for some classes of set constraints is NEXPTIME-complete [49, 7]. Since CFL-reachability is PTIME-complete [1, 38, 48], it is impossible to use CFL-reachability to cover these classes of set constraints (and it is unclear whether one can develop a more

powerful graph-reachability techniques that would handle them). It is also not clear that CFL-reachability can be used to model classes of set constraints in which intersection or negation is allowed.

7.1.1 Contravariant Set Constraints

We now sketch how the construction given in Section 4 can be modified to handle constructors that have *contravariant* fields. In a class of set constraints that uses contravariance, each constructor has a signature that indicates whether each field of the constructor is contravariant or covariant. In place of projections, there is a reduction rule that reduces a constraint of the form

$$c(U_1, \ldots, U_r) \supseteq c(V_1, \ldots, V_r)$$

to the following constraints:

$$\begin{array}{ll} U_i \supseteq V_i & \text{for all } i \text{ such that } c \text{ is covariant in field } i \\ V_j \supseteq U_j & \text{for all } j \text{ such that } c \text{ is contravariant in field } j \end{array}$$

(Note, that in the class of set constraints discussed in this paper, constraints of the form $c(U_1, \ldots, U_r) \supseteq c(V_1, \ldots, V_r)$ are not permitted; a system that uses contravariant constraints should allow constraints of this form, and might also allow constraints of the form $c(U_1, \ldots, U_r) \supseteq X$.)

Contravariance can be modeled by CFL-reachability by including the following elements in the construction of the CFL-reachability problem:

- Each atomic expression $c(V_1, \ldots, V_r)$ used in the constraints is associated with a unique index. As in Sections 4 and 5, we refer to refer to an atomic expression by its index rather than by writing out the expression.

  For each atomic expression $c(V_1, \ldots, V_r)$ with index k, the graph contains a node labeled (k) and the graph contains the following edges:

  $$\begin{array}{ll} c_i\langle V_i, (k)\rangle & \text{for all } i \text{ such that } c \text{ is covariant in field } i \\ c_i^{-1}\langle (k), V_i\rangle & \text{for all } i \text{ such that } c \text{ is covariant in field } i \\ \mathit{contra_c_j}\langle V_j, (k)\rangle & \text{for all } j \text{ such that } c \text{ is contravariant in field } j. \\ \mathit{contra_c_j}^{-1}\langle (k), V_j\rangle & \text{for all } j \text{ such that } c \text{ is contravariant in field } j. \end{array}$$

  In addition, for each of the above edges, the graph contains the corresponding reverse edge. For example, if the graph contains the edge $c_i\langle V_i, (k)\rangle$, then the graph also contains $\mathit{rev_c_i}\langle (k), V_i\rangle$. (Depending on the constraint system being modeled and the other aspects of the constructed CFL-reachability problem, some of these reverse edges may be unnecessary. For example, it may be possible to use the edge $c_i^{-1}\langle (k), V_i\rangle$ in place of the edge $\mathit{rev_c_i}\langle (k), V_i\rangle$.)

- For any constraint of the form $c(U_1, \ldots, U_r) \supseteq c(V_1, \ldots, V_r)$, where the expression $c(U_1, \ldots, U_r)$ has index j and the expression $c(V_1, \ldots, V_r)$ has index k, the graph contains the edges $\mathit{Id}\langle (k), (j)\rangle$ and $\mathit{Rev_Id}\langle (j), (k)\rangle$.

- For each constructor c, the grammar contains the following productions:

  $$\begin{array}{lll} \mathit{Id} ::= c_i \;\; \mathit{Id} \;\; c_i^{-1} & & \text{for all } i \text{ such that } c \text{ is covariant in field } i. \\ \mathit{Id} ::= \mathit{contra_c_j} \;\; \mathit{Rev_Id} \;\; \mathit{contra_c_j}^{-1} & & \text{for all } j \text{ such that } c \text{ is contravariant in field } j. \end{array}$$

  In addition, the grammar should contain the corresponding "reverse" productions that have *Rev_Id* on their left-hand side.

Example 7.1 Let the binary constructor *abs* be contravariant in its first field, and covariant in its second field.

The constructor *abs* can be used in set expressions to represent a functional abstraction $\lambda x.e$ (in the program that is being analyzed): let the set variable X represent (a superset of) the values that the program variable x may bind to at runtime and let the set variable $\mathit{Range}_{\lambda x.e}$ represents (a superset of) the values that are returned by $\lambda x.e$ during program execution. Then we use the set expression $\mathit{abs}(X, \mathit{Range}_{\lambda x.e})$ to represent the functional abstraction $\lambda x.e$.

To represent the application $(\lambda x.e)(y)$, we use the set variables Y and App and the set constraint $abs(Y, App) \supseteq abs(X, Range_{\lambda x.e})$. Here, the set expression $abs(X, Range_{\lambda x.e})$ represents $\lambda x.e$, the set variable Y represents (a superset of) the values y may bind to, and the set variable App represents a superset of the values that the expression $(\lambda x.e)(y)$ may return. Recall that the constraint

$$abs(Y, App) \supseteq abs(X, Range_{\lambda x.e})$$

reduces to the following constraints:

$X \supseteq Y$	(which indicates that the values in y bind to the values in x as a result of the application $(\lambda x.e)(y)$.)
$App \supseteq Range_{\lambda x.e}$	(which indicates that the set of values that $(\lambda x.e)(y)$ evaluates to is a superset of the values returned by $\lambda x.e$.

Now let us consider the CFL-reachability problem constructed to represent the constraint $abs(Y, App) \supseteq abs(X, Range_{\lambda x.e})$. The graph constructed to represent $abs(Y, App) \supseteq abs(X, Range_{\lambda x.e})$ contains the edges $contra_abs_1\langle Y, (j)\rangle$, $Rev_Id\langle (j), (k)\rangle$, and $contra_abs_1^{-1}\langle (k), X\rangle$ (where j is the index of the expression $abs(Y, App)$ and k is the index of the expression $abs(X, Range_{\lambda x.e})$). These edges, together with the production

$$Id ::= contra_abs_1 \quad Rev_Id \quad contra_abs_1^{-1}$$

cause the CFL-Reachability Algorithm to add the edge $Id\langle Y, X\rangle$, which encodes the constraint $X \supseteq Y$.

The constructed graph also contains the edges $abs_2\langle Range_{\lambda x.e}, (k)\rangle$, $Id\langle (k), (j)\rangle$, and $abs_2^{-1}\langle (j), App\rangle$. These edges, together with the production

$$Id ::= abs_2 \quad Id \quad abs_2^{-1}$$

cause the CFL-Reachability Algorithm to add the edge $Id\langle Range_{\lambda x.e}, App\rangle$, which encodes the constraint $App \supseteq Range_{\lambda x.e}$.

Thus the constructed CFL-reachability problem correctly captures the effects of reducing the constraint $abs(Y, App) \supseteq abs(X, Range_{\lambda x.e})$. $\square$

7.2 Insight Into the Cubic-Time Bottleneck for Program Analysis

As pointed out in the Introduction, the results presented in this paper offer some insight into the source of the cubic-time bottleneck for program analysis problems. Heintze and McAllester have also obtained results that have a bearing on this issue by considering the problem of determining membership for languages defined by 2-way nondeterministic pushdown automata (2NPDA-recognition) [21]. The asymptotically best algorithm known for solving the 2NPDA-recognition problem runs in $O(n^3)$ time, and they observe that if there is a linear-time reduction from 2NPDA-recognition to a given problem, then that problem is unlikely to be solvable in better than $O(n^3)$ time. In [21] reductions are given from 2NPDA-recognition to problems of flow analysis and typability in the Amadio-Cardelli type system. (This is consistent with something we had observed in unpublished work, where we gave a linear-time reduction from the 2NPDA-recognition problem to CFL-reachability.) Heintze and McAllester have also examined the complexity of set-based analysis with data constructors [33, 20].

7.3 Applications of CFL-reachability

Dolev, Even, and Karp used CFL-reachability to devise a formal model for studying the vulnerability to intrusion by a third party of a class of two-party ("ping-pong") protocols in distributed systems to intrusion by a third party [11]. In particular, they reduce the security-validation problem to a (single-source/single-target) CFL-reachability problem in which labeled edges represent possible encoding and decoding operations and the context-free language captures the interactions between possible actions that can take place during the protocol.

Yannakakis surveys the literature up to 1990 on applications of graph-theoretic methods in database theory [51]. He discusses many types of graph-reachability problems, including CFL-reachability.

A variety of work exists that has applied graph reachability (of various forms) to analysis of imperative programs. Kou [32] and Hecht [15] gave linear-time graph-reachability algorithms for solving intraprocedural "bit-vector" dataflow-analysis problems. This approach was later applied to intraprocedural bi-directional bit-vector problems [31]. Cooper and Kennedy used reachability to give efficient algorithms for interprocedural side-effect analysis [9] and alias analysis [10].

The first uses of CFL-reachability for program analysis were in 1988, in Callahan's work on flow-sensitive side-effect analysis [8] and Horwitz, Reps, and Binkley's work on interprocedural slicing [22, 23]. Both papers use only limited forms of CFL-reachability, namely various kinds of matched-parenthesis (Dyck) languages, and neither paper relates the work to the more general concept of CFL-reachability. (Dyck languages had been used in earlier work on interprocedural dataflow analysis by Sharir and Pnueli to specify that the contributions of certain kinds of nonexecutable paths should be filtered out [46]; however, the dataflow-analysis algorithms given by Sharir and Pnueli are based on machinery other than pure graph reachability.)

Dyck-language reachability was shown by Reps, Sagiv, and Horwitz to be of utility for a wide variety of interprocedural program-analysis problems [41]. These ideas were elaborated on in a sequence of papers [25, 24, 40], and also applied to shape analysis of functional programs [37]. (See also [39] for a survey of this work.)

The second author became aware of the connection between program analysis and the general concept of CFL-reachability sometime in the fall of 1994. (Of the papers mentioned above, only [37] and [39] mention CFL-reachability explicitly and reference Yannakakis's paper [51].) The constructions of the present paper for converting set-constraint problems to CFL-reachability problems—together with the fact that set constraints have been used for program analysis—show that CFL-reachability using path languages other than Dyck languages is also of utility for program analysis.

7.4 Slicing Higher-Order Functional Languages

Program slicing is an operation that identifies semantically meaningful decompositions of programs, where the decompositions consist of elements that are not necessarily textually contiguous [50, 34, 23]. CFL-reachability has been applied to the problem of slicing programs written in imperative Algol-like languages [23]. Regular-tree grammars have been applied to the problem of slicing programs written in a first-order functional language (that manipulates heap-allocated data structures) [42].

We now sketch how the technique developed in the construction given in Section 5 allows CFL-reachability to be applied to the problem of slicing programs written in a higher-order functional language (again that manipulates heap-allocated data structures). The latter problem has not been previously addressed in the literature on program slicing.

Specifically the slicing algorithm will be formulated for a higher-order LISP-like functional language that has the constructor and selector operations NIL, CONS, CAR, and CDR for manipulating heap-allocated data (*i.e.*, lists and dotted pairs), together with appropriate predicates (EQUAL, ATOM, and NULL), but no operations for destructive updating (*e.g.*, RPLACA and RPLACD). The constructs of the language are

x_i	(ATOM e_1)	(CONS e_1 e_2)	(OP op e_1 e_2)
'c	(NULL e_1)	(IF e_1 e_2 e_3)	(DEFINE (f $x_1 \cdots x_k$) e_f)
(CAR e_1)	(CDR e_1)	(EQUAL e_1 e_2)	(CALL f $e_1 \cdots e_k$)

A program is a list of function definitions, with a distinguished top-level goal function, named *main*. We assume that the distinguished atom "NIL" is used for terminating lists, and that there is also a special empty-tree value (different from NIL) denoted by "?".

Following Reps and Turnidge, we consider the problem of slicing a functional program $P(x)$ in terms of symbolically composing $P(x)$ with an appropriate projection function $\pi(y)$ [42]. Projection function $\pi(y)$ characterizes what information should be retained and what information should be discarded from the value that $P(x)$ computes. We consider projection functions that can be represented as regular language of *access paths*, where an access path represents a sequence of CAR and CDR operations. We require that the set of access paths defined by projection function $\pi(y)$ be prefix-closed. (In order to access a part of $P(x)$'s return value along an access path p, it is also necessary to access every part of $P(x)$'s return value that is reached along a prefix of p; requiring that $\pi(y)$ be prefix-closed is not strictly necessary, but it simplifies the presentation below.)

subexpression	corresponding graph	subexpression	corresponding graph
x_i	x_i	'c	'c
(CAR e_1)	CAR, $cons_1^{-1}$, e_1	(CDR e_1)	CDR, $cons_2^{-1}$, e_1
(CONS e_1 e_2)	CONS, $cons_1$, $cons_2$, e_1, e_2	(IF e_1 e_2 e_3)	IF, *ctrlOrAtomicUse*, *Id*, *Id*, e_1, e_2, e_3
(ATOM e_1)	ATOM, *ctrlOrAtomicUse*, e_1	(EQUAL e_1 e_2)	EQUAL, $cons_1^{-1}$, $cons_2^{-1}$, *Id*, *Id*, e_1, e_2
(NULL e_1)	NULL, *ctrlOrAtomicUse*, e_1	(OP op e_1 e_2)	OP, *ctrlOrAtomicUse*, *ctrlOrAtomicUse*, e_1, e_2
(CALL f e_1 ... e_k)	CALL, $return^{-}$, *ctrlOrAtomicUse*, f, $input_1$, $input_k$, ..., e_1, e_k	(DEFINE (f x_1 ... x_k) e_f)	f, $input_1^{-}$, $input_k^{-}$, *return*, x_1, ..., x_k, e_f

Legend	
•	expression node
○	incomplete expression

Table 2: Summary of the construction of a value-flow graph from each subexpression of a program. Reverse edges (*e.g.*, *Rev_Id*) are not shown. Each occurrence of a variable x generates a new node in the value-flow graph. In addition to the edges shown above, there is an *Id* edge from each function parameter x_i to each use of x_i and a *Rev_Id* edge from the use to the parameter. There is also an *Id* edge from each function definition to a use of the function and a *Rev_Id* edge from each function use to the function definition. See Figures 17 and 19 for complete examples of value-flow graphs.

$\pi(P(x))$'s return value is a pruned copy of $P(x)$'s return value in which every substructure that cannot be reached by an access path in $\pi(y)$ has been replaced by "?". The slicing problem becomes one of understanding what parts of $P(x)$ affect the return value of $\pi(P(x))$. The slicing algorithm should therefore identify the subexpressions of $P(x)$ that could not affect a portion of $P(x)$'s return value that will be accessed by $\pi(y)$ (via an access path in $\pi(y)$), and replace these subexpressions by '?. As long as the client of the sliced program abides by the access "contract" given by $\pi(y)$, the values that can be inspected will be the same as those generated by $P(x)$.

We define a graph, called a *value-flow* graph, whose nodes represent the subexpressions of $P(x)$ and whose edges represent dependences among subexpressions, the passing of parameters and return values, etc. Table 2 summarizes the construction of the value-flow graph from the subexpressions of $P(x)$. With the exception of *ctrlOrAtomicUse* edges, the edges in a value-flow graph are similar in function to the analogous edges in Section 5.2.1. An edge $ctrlOrAtomicUse\langle v, w\rangle$ indicates one of the following facts: (i) the expression

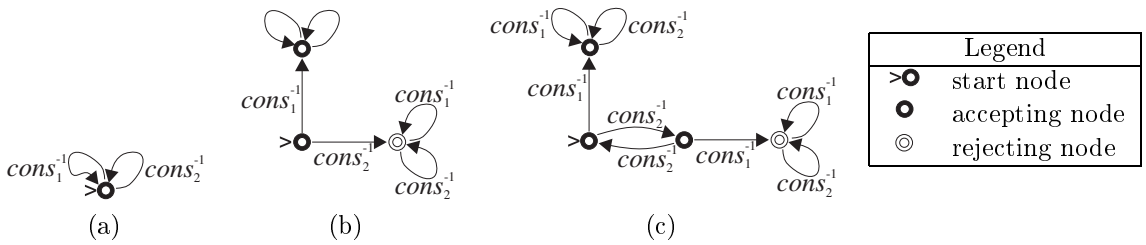


Figure 16: Example projection graphs: (a) shows the projection graph for the identity function; (b) shows the projection graph for the projection function that accesses everything in the CAR of the return value and discards everything in the CDR; and (c) shows the projection graph for the projection function that accesses every odd element of a list and discards every even element.

w makes a *control use* of the values returned by v (*i.e.*, either w calls a function value returned by v, or w makes a branch decision based on a boolean value returned by v); (ii) the expression w makes an *atomic use* of the values returned by v (*i.e.*, w uses the values returned by v but never performs a CAR or a CDR on those values). For purposes of slicing, an edge $ctrlOrAtomicUse\langle v, w\rangle$ indicates that if the values returned by w can affect $\pi(P(x))$, then the values returned by v can affect $\pi(P(x))$, although the CAR and the CDR of values returned by v cannot affect $\pi(P(x))$.

In adition to the value-flow graph, we define a *projection* graph that represents the deterministic-finite automaton (DFA) that accepts the language of access paths defined by $\pi(y)$. The projection graph has a unique *start* node, one or more *accepting* nodes, and at most one *rejecting* node. Each transition in the DFA is represented in the projection graph by a $cons_1^{-1}$ edge (representing a CAR operation) or a $cons_2^{-1}$ edge (representing a CDR operation). Figure 16 shows some example projection graphs.

To apply CFL-reachability to the slicing problem, a composite graph is created by connecting the value-flow graph and the projection graph with the edges $return^{-}\langle main, start\rangle$ and $ctrlOrAtomicUse\langle main, start\rangle$, where *main* is the node in the value-flow graph that represents the definition of main and *start* is the start node of the projection graph. (The edge $return^{-}\langle main, start\rangle$ indicates that we are interested in the values returned by main. The edge $ctrlOrAtomicUse\langle main, start\rangle$ indicates that any execution of the program makes a control use of the function *main*.) We define a language *Slice*, such that a *Slice*-path from a node v in the value-flow graph to an accepting node in the projection graph indicates that the value computed by subexpression v may affect the value returned by $\pi(P(x))$:

$$
\begin{array}{rcl}
Id & ::= & Id\ Id \\
 & | & cons_1\ Id\ cons_1^{-1} \\
 & | & cons_2\ Id\ cons_2^{-1} \\
 & | & input_i\ Rev_Id\ input_i^{-} \text{(for } 1 \leq i \leq \text{maximum number of function parameters)} \\
 & | & return\ Id\ return^{-} \\
 & | & \epsilon \\
UnbalRight & ::= & UnbalRight\ cons_1^{-1}\ Id \\
 & | & UnbalRight\ cons_2^{-1}\ Id \\
 & | & Id \\
CtrlOrAtomicSlice & ::= & UnbalRight\ ctrlOrAtomicUse \\
 & | & CtrlOrAtomicSlice\ UnbalRight\ ctrlOrAtomicUse \\
Slice & ::= & UnbalRight \\
 & | & CtrlOrAtomicSlice\ UnbalRight
\end{array}
$$

Issues of groundness are ignored in this grammar. Furthermore, the productions for *Rev_Id*—which correspond exactly to the productions for *Id* but in the "reverse" direction—have not been shown (see Section 4.1.2 for a discussion of reversing productions).

In this grammar, the nonterminal *UnbalRight* represents an *unbalanced right* path. An unbalanced right path includes an excess of selection operators; an edge $UnbalRight\langle v, w\rangle$ indicates that the values returned by the expression w may include substructures of the values returned by the expression v. The nonterminal *CtrlOrAtomicSlice* represents a *control or atomic slice* path. An edge $CtrlOrAtomicSlice\langle v, w\rangle$ indicates that

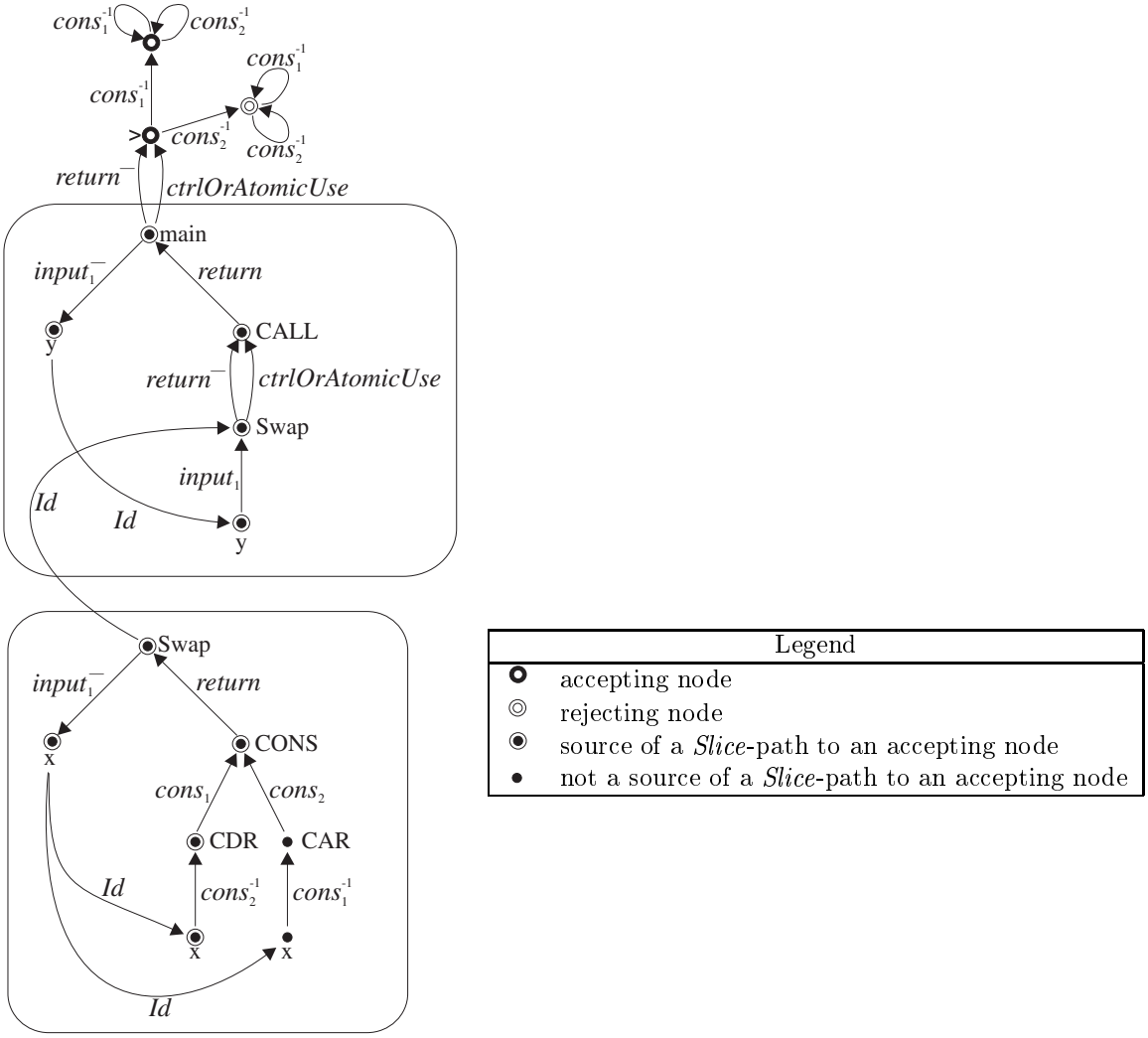


Figure 17: Value-flow graph and projection graph for Example 7.2. Reverse edges are not shown.

the values returned by the expression w are affected (*e.g.*, via a control dependence) by a substructure of the values returned by the expression v.

Value-flow is performed by determining all sub-terms w for which there is no *Slice*-path from the node that represents w to an accepting node and replacing them by '?; this is done with one exception: formal parameters are never replaced with '?.

Example 7.2 Consider the following program:

```
(DEFINE (main y) (CALL Swap y))
(DEFINE (Swap x) (CONS (CDR x) (CAR x)))
```

Suppose that we are only interested in the CAR of the value returned by this program. Figure 17 shows the value-flow graph for this program together with the projection graph for the CAR projection function.

The results of running the CFL-Reachability Algorithm on the graph in Figure 17 indicate that there is no *Slice*-path from the node that represents the expression (CAR x) to an accepting node. There are *Slice*-paths from all other expression nodes. For example, from the expression (CDR x), there is a path to an accepting node that spells out the string

$$cons_1 \; return \; Id \; return^- \; return \; return^- \; cons_1^{-1}.$$

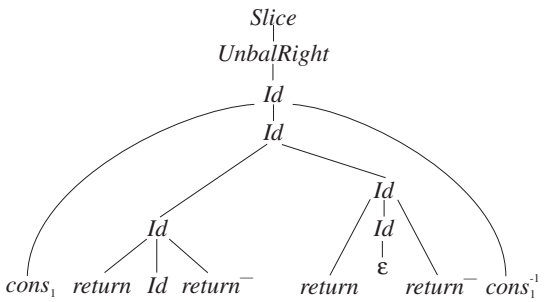


Figure 18: Derivation tree for the string $cons_1$ *return Id* $return^-$ *return* $return^-$ $cons_1^{-1}$.

This string can be derived from the nonterminal *Slice* as shown in Figure 18.

The sliced version of the above program is

```
(DEFINE (main y) (CALL Swap y))
(DEFINE (Swap x) (CONS (CDR x) ('?)))
```

□

Example 7.3 To illustrate slicing of a higher-order function, consider the following program:

```
(DEFINE (main y) (CALL Swap y MyCons))
(DEFINE (Swap x pairfn) (CALL pairfn (CDR x) (CAR x)))
(DEFINE (MyCons z w) (CONS z w))
```

This program is very similar to the the program in Example 7.2 except that the function *MyCons* is passed as a parameter to the function *Swap*. As in the previous example, suppose we are interested in the CAR of the value returned by this program. Figure 19 shows the value-flow graph together with the projection graph. The results of running the CFL-Reachability Algorithm on the graph in Figure 19 indicate that there are no *Slice*-paths from the expression (CAR x) nor from the second argument of the function MyCons. There are slice paths from all other expressions. For example, there is path from the expression (CDR x) to an accepting node that spells out the string

$$input_1\ Rev_Id\ rev_input_2^-\ Id\ rev_input_2\ Rev_Id\ input_1^-\ Id\ cons_1\ return\ Id\ input_2\ Rev_Id\ input_2^-\ Id\ return^-\ return\ Id\ return^-\ return\ return^-\ cons_1^-. \tag{1}$$

This string can be derived from the nonterminal *Slice*.

We observe that the slice path from (CDR x) to the accepting node contains a *Rev_Id*-path from the variable *pairfn* to the function *MyCons* and an *Id*-path from *MyCons* to *pairfn*. These paths mean that pairfn can take on the value MyCons. The *Rev_Id*-path spells out the string *Rev_Id* $rev_input_2^-$ *Id* rev_input_2 *Rev_Id* and the *Id*-path spells out the string *Id* $input_2$ *Rev_Id* $input_2^-$ *Id*; both of these strings are substrings of (1).

The sliced version of the above program is

```
(DEFINE (main y) (CALL Swap y MyCons))
(DEFINE (Swap x pairfn) (CALL pairfn (CDR x) '?))
(DEFINE (MyCons z w) (CONS z '?))
```

□

The method described above yields executable slices. We now briefly discuss the relationship between the semantics of a slice and the semantics of the orignal program. Let $Q(x)$ be the program that results from slicing $P(x)$ with projection $\pi(y)$. There are two important points:

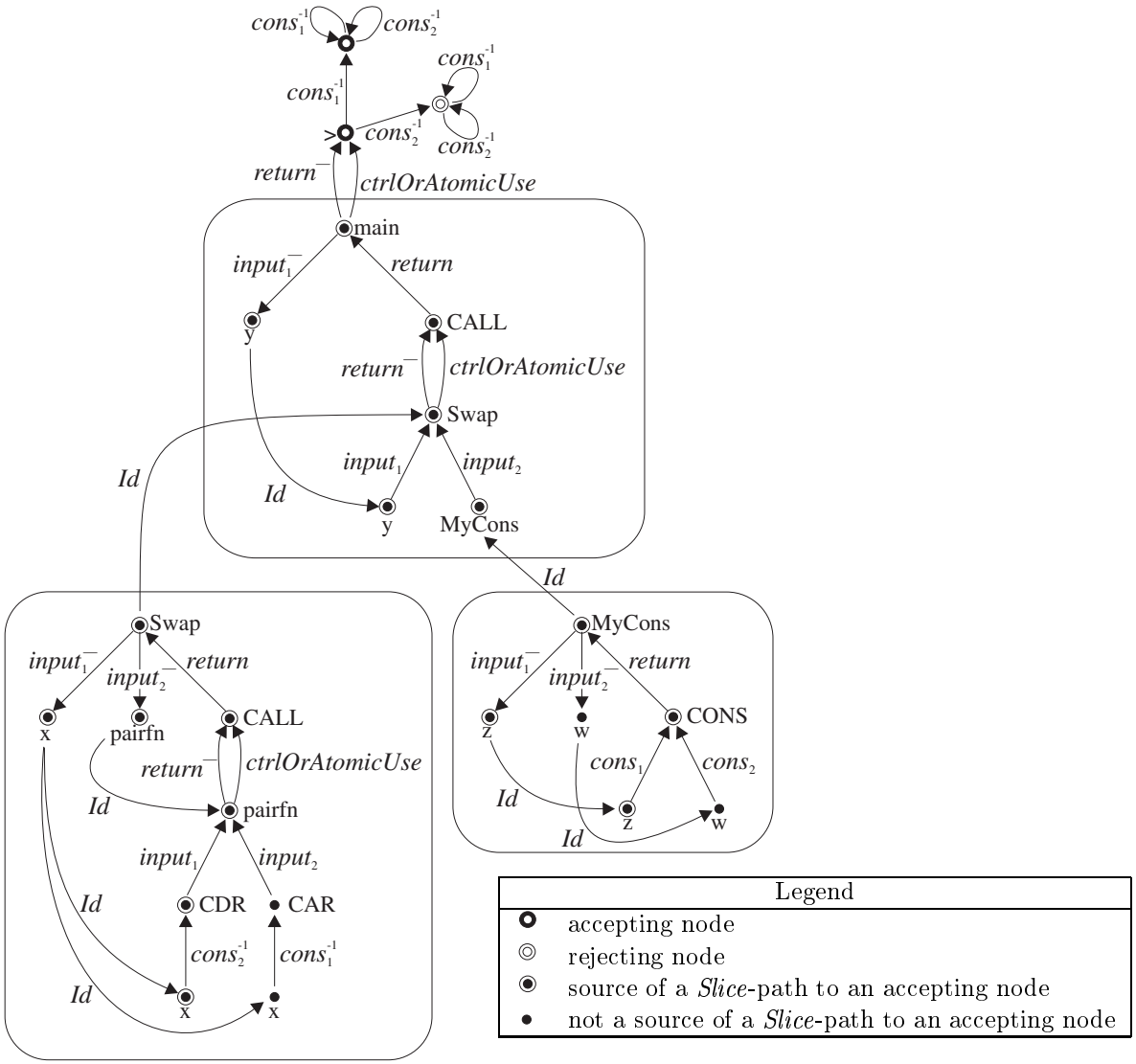


Figure 19: Value-flow graph and projection graph for Example 7.3. Reverse edges are not shown.

- In a call-by-value language, it is possible that $Q(x)$ may terminate on inputs for which $\pi(P(x))$ diverges. Slicing can never introduce *divergence*; it can only introduce *termination*, which, from a pragmatic standpoint, is quite reasonable. If $\pi(P(x))$ does terminate, then $\pi(Q(x)) = \pi(P(x))$.

- It is possible that $Q(x) \neq \pi(P(x))$. In particular, $Q(x)$ may contain additional material that is not in $\pi(P(x))$. The reason that such extra information may exist is that slicing is a monovariant analysis. Because different portions of a the result of a function may be needed at different call sites, a function in a slice may return more information than is needed at a specific call site. In addition more information may be present in a variable than is needed at all uses of that variable. For these reasons, a sliced program may return more information than is actually needed. However, the information returned by a sliced program is safe with respect to $\pi(y)$. In particular, $\pi(Q(x)) = \pi(P(x))$.

[42] contains a more detailed discussion of the semantic relationship between a slice and its original program.

7.5 Connection to DATALOG

It is also interesting to note another fact about CFL-reachability problems: every CFL-reachability problem can be stated as a *chain program* in DATALOG [51]; edges are represented as facts, and productions are encoded as Horn clauses. In fact, the CFL-reachability Algorithm presented in Section 2.1.1 in effect emulates semi-naive bottom-up evaluation of the equivalent DATALOG program. This suggests that the class of DATALOG programs that run in cubic time may be useful for program analysis (see also [36, 5]). The construction described in Section 4 also implies that the class of set-constraints studied in this paper may also be solved by converting them to equivalent DATALOG programs. In fact, many parts of the set-constraint-to-CFL-reachability-problem constructions are more easily expressed in DATALOG. In particular, the addition of reverse edges, and the tracking of ground information is easy to express. The resulting DATALOG program would not necessarily be a chain program, but it would still run in cubic time.

7.6 Demand Analysis

An *exhaustive* program-analysis algorithm associates with each point in a program a set of "facts" that characterize (in some fashion) the execution state that holds whenever that point is reached during execution. By contrast, a *demand* program-analysis algorithm computes a partial solution to a problem, when only part of the full answer is needed — e.g., whether a particular fact (or set of facts) holds at a single specific point [6, 52, 36, 12, 37, 24, 44]. Demand analysis can sometimes be preferable to exhaustive analysis for the following reasons:

Narrowing the focus to specific points of interest. In program optimization, most of the gains are obtained from making improvements at a program's "hot spots", such as the innermost loops, which means that information obtained from program analysis is really only needed for selected locations in the program. Thus, the use of a demand algorithm has the potential to reduce greatly the amount of extraneous information computed. Similarly, software-engineering tools that analyze programs often require information only at a certain set of program points. Because it is unlikely that a programmer will ask questions about all program points, solving just the user's sequence of demands is likely to be significantly less costly than performing an exhaustive analysis.

Narrowing the focus to specific facts of interest. Even when information is desired for every program point p, the full set of facts at p may not be required. For example, in a closure-analysis problem, we may be interested in determining which abstractions reach a certain specific application, rather than determining that information for all applications.

Sidestepping incremental-updating problems. A transformation performed at one point in the program can affect the validity of program-analysis information for other points in the program: In many cases, the old information at such points is no longer safe; the information needs to be updated before it is possible to perform further transformations at such points. An incremental updating algorithm could be used to maintain complete information at all program points; however, updating all invalidated information can be expensive. An alternative is to demand only the information needed to validate a proposed transformation; each demand would be solved using the current program, thereby ensuring that the answer is up-to-date.

Of course, determining whether a given fact holds at a given point may require determining whether other, related facts hold at other points (and those other facts may not be "facts of interest" in the sense of the second bullet-point above). It is desirable, therefore, for a demand-driven program-analysis algorithm to minimize the amount of such auxiliary information computed.

For program-analysis problems that have been transformed into CFL-reachability problems, demand algorithms are obtained for free, typically by solving a single-target or multi-target CFL-reachability problem [24]. Because an algorithm for solving single-target (or multi-target) CFL-reachability problems focuses on the nodes that reach the specific target(s), it minimizes the amount of extraneous information computed.

The construction described in Sections 4.1 and 4.2 shows that set-constraint problems can also be solved in a demand-driven fashion: apply the construction to convert the system of set constraints to a CFL-reachability problem; convert each query to an appropriate single-target (or single-source) CFL-reachability

query, and solve accordingly; finally, convert the answer back to the form that would be expected from solving a set-constraint problem.

It is likely that demand algorithms could be designed that operate on the set constraints directly; however, to our knowledge, this has not been investigated before in the literature on set constraints.

8 Acknowledgements

We are grateful to the referees for their careful reading of, and extensive comments on, the paper. Jon Kleinberg pointed out to us the use of CFL-reachability in reference [11].

A Correctness of the CFL-Reachability to Set-Constraint Construction

Lemma A.1 *Let $\mathcal{C}$ be a collection of set constraints containing the constraint $V \supseteq ae_1$, where ae_1 is an atomic expression that does not appear in any other constraint. Let $\mathcal{C}'$ be $\mathcal{C}$ unioned with the collection of set constraints generated by running the SC-Reduction Algorithm on $\mathcal{C}$. Then for any atomic expression ae_2 that is ground in $\mathcal{C}'$, if $\mathcal{C}'$ contains the constraints $V \supseteq ae_2$ and $U \supseteq ae_1$, then $\mathcal{C}'$ also contains $U \supseteq ae_2$.*

Proof: The SC-Reduction Algorithm generates a constraint of the form $W \supseteq ae$ iff it is given constraints of the form $W \supseteq W'$ and $W' \supseteq ae$ and ae is ground. Thus, if $U \neq V$, then the SC-Reduction Algorithm generates the constraint $U \supseteq ae_1$ iff ae_1 is grounded and the following collection of constraints are present:

$$\begin{array}{l} U \supseteq W_1 \\ W_1 \supseteq W_2 \\ \vdots \\ W_n \supseteq V \end{array}$$

This implies that such a collection of constraints must appear in $\mathcal{C}'$ if $U \supseteq ae_1$ is in $\mathcal{C}'$. It follows that if $\mathcal{C}'$ also contains the constraint $V \supseteq ae_2$ where ae_2 is ground, then the SC-Reduction Algorithm must also have generated the constraint $U \supseteq ae_2$. $\Box$

Lemma 3.2 *Let $\mathcal{C}$ be the collection of set constraints constructed to represent the context-free reachability problem $\mathcal{P}$. Let G be the graph that results from running the CFL-reachability Algorithm on $\mathcal{P}$. Let $\mathcal{C}'$ be $\mathcal{C}$ unioned with the collection of set constraints generated by running the SC-Reduction Algorithm on $\mathcal{C}$. Then there is an edge $A\langle i,j\rangle$ in G if and only if $\mathcal{C}'$ contains $X_i \supseteq A(X_j)$ and/or $Dst_{[A,i]} \supseteq node_j$.*

Proof of the $\Rightarrow$ direction: First, we dispense with a technical detail that is the same in all parts of the proof. In many subcases, we will be able to show that $\mathcal{C}'$ contains constraints of the form $U \supseteq c_1^{-1}(W)$ and $W \supseteq c(Y)$ and need to argue that $\mathcal{C}'$ contains $U \supseteq Y$. In all the cases that arise in the proof, we can show that $\mathcal{C}'$ must contain a constraint of the form $Y \supseteq node_j$. This will follow either from the original construction of $\mathcal{C}$ (if Y is one of the variable X_j) or from the suppositions in effect at that point of the proof (if Y is of the form $Dst_{[c,k]}$). In either case, the groundness of Y will be assured. To avoid clutter in the following discussion, we will not mention the groundness properties explicitly when we perform reductions.

Assume, on the contrary, that there is an edge $A\langle i,j\rangle$ in G such that $\mathcal{C}'$ contains neither $X_i \supseteq A(X_j)$ nor $Dst_{[A,i]} \supseteq node_j$. Note that for each edge $B\langle u,v\rangle$ in the original graph of the context-free reachability problem, $\mathcal{C}$ (and hence $\mathcal{C}'$) contains the constraint $X_u \supseteq B(X_v)$. Thus $A\langle i,j\rangle$ must have been generated by the CFL-reachability Algorithm.

Without loss of generality, let $A\langle i,j\rangle$ be the first edge that the CFL-reachability Algorithm generates such that $\mathcal{C}'$ contains neither $X_i \supseteq A(X_j)$ nor $Dst_{[A,i]} \supseteq node_j$. There are three reasons that the CFL-reachability algorithm might have introduced the edge $A\langle i,j\rangle$:

case 1: The context-free grammar contains the production $A ::= \epsilon$. In this case $i = j$. However, for each production of the form $A ::= \epsilon$, for each node k, $\mathcal{C}$ (and hence $\mathcal{C}'$) contains the constraint $X_k \supseteq A(X_k)$. Thus in this case, $\mathcal{C}'$ must contain the constraint $X_i \supseteq A(X_i)$.

case 2: The context-free grammar contains the production $A ::= B$, and the edge $B\langle i,j\rangle$ is present. Since $B\langle i,j\rangle$ must be present before $A\langle i,j\rangle$, and $A\langle i,j\rangle$ is the first edge generated by the CFL-reachability Algorithm such that $\mathcal{C}'$ contains neither $X_i \supseteq A(X_j)$ nor $Dst_{[A,i]} \supseteq node_j$, we conclude that $\mathcal{C}'$ must contain $X_i \supseteq B(X_j)$ and/or $Dst_{[B,i]} \supseteq node_j$. The construction also guarantees that $\mathcal{C}'$ also contains the constraints $X_i \supseteq A(Dst_{[A,i]})$ and $Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$ (to encode the production $A ::= B$) and the constraint $X_j \supseteq node_j$ (to encode node j).

The constraints $Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$ and $X_i \supseteq B(X_j)$ combine to give the constraint $Dst_{[A,i]} \supseteq X_j$. The constraints $Dst_{[A,i]} \supseteq X_j$ and $X_j \supseteq node_j$ reduce to the constraint $Dst_{[A,i]} \supseteq node_j$. Thus, if $\mathcal{C}'$ contains $X_i \supseteq B(X_j)$, it must also contain $Dst_{[A,i]} \supseteq node_j$.

If $\mathcal{C}'$ contains $Dst_{[B,i]} \supseteq node_j$, then it must also contain the constraint $X_i \supseteq B(Dst_{[B,i]})$ (because the variable $Dst_{[B,i]}$ is introduced iff this constraint is introduced). The constraints $Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$ and $X_i \supseteq B(Dst_{[B,i]})$ combine to give $Dst_{[A,i]} \supseteq node_j$. Thus, if $\mathcal{C}'$ contains $Dst_{[B,i]} \supseteq node_j$, it must also contain $Dst_{[A,i]} \supseteq node_j$.

In either case $\mathcal{C}'$ must contain the constraint $Dst_{[A,i]} \supseteq node_j$.

case 3: The context-free grammar contains the production $A ::= B\ C$ and the edges $B\langle i,k\rangle$ and $C\langle k,j\rangle$ are present. $\mathcal{C}'$ must contain the constraints

$Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$ and
$Rchd_{[B_1^{-1},i]} \supseteq B_1^{-1}(X_i)$

to encode the production $A ::= B\ C$. Since the edge $B\langle i,k\rangle$ is present before $A\langle i,j\rangle$, $\mathcal{C}'$ must also contain $X_i \supseteq B(X_k)$ and/or $Dst_{[B,i]} \supseteq node_k$. This gives us two subcases:

case 3.a: Suppose $\mathcal{C}'$ contains $X_i \supseteq B(X_k)$. This constraint and the constraint $Rchd_{[B_1^{-1},i]} \supseteq B_1^{-1}(X_i)$ give the constraint $Rchd_{[B_1^{-1},i]} \supseteq X_k$. (Thus, in this case, $\mathcal{C}'$ must contain $Rchd_{[B_1^{-1},i]} \supseteq X_k$.) Since the edge $C\langle k,j\rangle$ was present before edge $A\langle i,j\rangle$, $\mathcal{C}'$ must also contain $X_k \supseteq C(X_j)$ and/or $Dst_{[C,k]} \supseteq node_j$. This gives two subcases:

case 3.a.i: Suppose $\mathcal{C}'$ contains $X_k \supseteq C(X_j)$. The constraints

$Rchd_{[B_1^{-1},i]} \supseteq X_k$ and
$X_k \supseteq C(X_j)$

combine to give the constraint $Rchd_{[B_1^{-1},i]} \supseteq C(X_j)$. The constraints

$Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$ and
$Rchd_{[B_1^{-1},i]} \supseteq C(X_j)$

reduce to the constraint $Dst_{[A,i]} \supseteq X_j$. This constraint combines with $X_j \supseteq node_j$ to give $Dst_{[A,i]} \supseteq node_j$. So in this case, $\mathcal{C}'$ must contain $Dst_{[A,i]} \supseteq node_j$.

case 3.a.ii: Suppose $\mathcal{C}'$ contains $Dst_{[C,k]} \supseteq node_j$. The construction introduces a variable of the form $Dst_{[C,k]}$ iff it also introduces the constraint $X_k \supseteq C(Dst_{[C,k]})$; thus $\mathcal{C}'$ must contain $X_k \supseteq C(Dst_{[C,k]})$. Given the constraints

$Rchd_{[B_1^{-1},i]} \supseteq X_k$ and
$X_k \supseteq C(Dst_{[C,k]})$

the SC-Reduction Algorithm produces the constraint $Rchd_{[B_1^{-1},i]} \supseteq C(Dst_{[C,k]})$. The constraints

$Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$ and
$Rchd_{[B_1^{-1},i]} \supseteq C(Dst_{[C,k]})$

reduce to $Dst_{[A,i]} \supseteq Dst_{[C,k]}$. This constraint and $Dst_{[C,k]} \supseteq node_j$ reduce to the constraint $Dst_{[A,i]} \supseteq node_j$. Thus, in this case, $\mathcal{C}'$ must contain $Dst_{[A,i]} \supseteq node_j$.

case 3.b: Suppose $\mathcal{C}'$ contains $Dst_{[B,i]} \supseteq node_k$. This implies that $\mathcal{C}'$ contains the constraint $X_i \supseteq B(Dst_{[B,i]})$ (since the variable $Dst_{[B,i]}$ is introduced iff this constraint is added to $\mathcal{C}$ during the original construction). The constraints

$Rchd_{[B_1^{-1},i]} \supseteq B_1^{-1}(X_i)$
$X_i \supseteq B(Dst_{[B,i]})$

reduce to the constraint $Rchd_{[B_1^{-1},i]} \supseteq Dst_{[B,i]}$. The constraints

$Rchd_{[B_1^{-1},i]} \supseteq Dst_{[B,i]}$ and
$Dst_{[B,i]} \supseteq node_k$

combine to give $Rchd_{[B_1^{-1},i]} \supseteq node_k$.

Again, we know that $\mathcal{C}'$ must contain $X_k \supseteq C(X_j)$ and/or $Dst_{[C,k]} \supseteq node_j$:

case 3.b.i: Suppose $\mathcal{C}'$ contains $X_k \supseteq C(X_j)$. Since the only occurrence of the atomic expression $node_k$ in $\mathcal{C}$ is in the constraint $X_k \supseteq node_k$, we can use Lemma A.1 and the presence of $X_k \supseteq C(X_j)$ and $Rchd_{[B_1^{-1},i]} \supseteq node_k$ in $\mathcal{C}'$ to conclude that $Rchd_{[B_1^{-1},i]} \supseteq C(X_j)$ is also in $\mathcal{C}'$. The constraints

$Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$ and
$Rchd_{[B_1^{-1},i]} \supseteq C(X_j)$

reduce to the constraint $Dst_{[A,i]} \supseteq X_j$. This constraint combines with the constraint $X_j \supseteq node_j$ to give $Dst_{[A,i]} \supseteq node_j$. Thus in this case, $\mathcal{C}'$ must contain $Dst_{[A,i]} \supseteq node_j$.

case 3.b.ii: Suppose $\mathcal{C}'$ contains $Dst_{[C,k]} \supseteq node_j$. Then $\mathcal{C}'$ must also contain $X_k \supseteq C(Dst_{[C,k]})$. Again by use of Lemma A.1, and the presence of the constraints $X_k \supseteq node_k$, $Dst_{[B,i]} \supseteq node_k$, and $X_k \supseteq C(Dst_{[C,k]})$, we conclude that $\mathcal{C}'$ contains the constraint $Dst_{[B,i]} \supseteq C(Dst_{[C,k]})$. The constraints

$Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$ and
$Dst_{[B,i]} \supseteq C(Dst_{[C,k]})$

combine to give $Dst_{[A,i]} \supseteq Dst_{[C,k]}$ which combines with $Dst_{[C,k]} \supseteq node_j$ to give $Dst_{[A,i]} \supseteq node_j$. Thus in this case, $\mathcal{C}'$ must contain $Dst_{[A,i]} \supseteq node_j$.

For all of the possible cases that may cause the CFL-reachability Algorithm to introduce the edge $A\langle i,j\rangle$, we have shown that $\mathcal{C}'$ contains $X_i \supseteq A(X_j)$ or $Dst_{[A,i]} \supseteq node_j$. This contradicts the assumption that $A\langle i,j\rangle$ is the first edge introduced by the CFL-reachability Algorithm such that $\mathcal{C}'$ contains neither $X_i \supseteq A(X_j)$ nor $Dst_{[A,i]} \supseteq node_j$, and implies that there can be no such edge $A\langle i,j\rangle$. □

Proof of the $\Leftarrow$ direction: We need to show that the presence of the constraint $X_i \supseteq A(X_j)$ or the constraint $Dst_{[A,i]} \supseteq node_j$ in $\mathcal{C}'$ allows us to assert that the edge $A\langle i,j\rangle$ appears in G.

The constraints in $\mathcal{C}$ (the initial collection of constraints constructed to represent the CFL-reachability problem) must have one of the following forms:

$Rchd_{[B_1^{-1},i]} \supseteq B_1^{-1}(X_i)$	(Follow B-edges from node i; used to encode $A ::= B\ C$)
$Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$	(Follow C-edges from those nodes; used to encode $A ::= B\ C$)
$X_i \supseteq A(Dst_{[A,i]})$	(Add A-edges to the reached nodes; used to encode $A ::= B\ C$ and $A ::= B$)
$Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$	(Follow B-edges from X_i; used to encode $A ::= B$)
$X_i \supseteq node_i$	(Encode X_i as representing node i)
$X_i \supseteq A(X_j)$	(Encode an A edge from i to j)
$X_i \supseteq X_i$	(Used to encode and $A ::= \epsilon$)

Following the rules of the SC-Reduction Algorithm, the constraints in $\mathcal{C}$ may give rise to constraints of the following additional forms (which may appear in $\mathcal{C}'$):

$Rchd_{[A_1^{-1},i]} \supseteq X_j$	$Dst_{[A,i]} \supseteq X_j$
$Rchd_{[A_1^{-1},i]} \supseteq Dst_{[A,i]}$	$Dst_{[A,i]} \supseteq Dst_{[B,j]}$
$Rchd_{[C_1^{-1},i]} \supseteq B(X_j)$	$Dst_{[B,j]} \supseteq C(X_k)$
$Rchd_{[C_1^{-1},i]} \supseteq B(Dst_{[B,j]})$	$Dst_{[B,j]} \supseteq C(Dst_{[C,k]})$
$Rchd_{[A_1^{-1},i]} \supseteq node_j$	$Dst_{[A,i]} \supseteq node_j$

Selected constraint form and associated assertion	Matching constraint form and associated assertion	Produced constraint and associated assertion
$Rchd_{[A_1^{-1},i]} \supseteq A_1^{-1}(X_i)$ Null assertion	$X_i \supseteq A(X_j)$ (Encodes $A\langle i,j\rangle$)	$Rchd_{[A_1^{-1},i]} \supseteq X_j$ "$A\langle i,j\rangle \in E_G$"
	$X_i \supseteq A(Dst_{[A,i]})$ Null assertion	$Rchd_{[A_1^{-1},i]} \supseteq Dst_{[A,i]}$ Null assertion
$Dst_{[A,i]} \supseteq B_1^{-1}(Rchd_{[C_1^{-1},i]})$ (Encodes $A ::= C \quad B$)	$Rchd_{[C_1^{-1},i]} \supseteq B(X_j)$ "$\exists k[C\langle i,k\rangle \in E_G$ and $B\langle k,j\rangle \in E_G]$"	$Dst_{[A,i]} \supseteq X_j$ "$A\langle i,j\rangle \in E_G$"
	$Rchd_{[C_1^{-1},i]} \supseteq B(Dst_{[B,j]})$ "$C\langle i,j\rangle \in E_G$"	$Dst_{[A,i]} \supseteq Dst_{[B,j]}$ "$\forall n[B\langle j,n\rangle \in E_G$ imp. $A\langle i,n\rangle \in E_G]$"
$Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$ (Encodes $A ::= B$)	$X_i \supseteq B(X_j)$ (Encodes $B\langle i,j\rangle \in E_G$)	$Dst_{[A,i]} \supseteq X_j$ "$A\langle i,j\rangle \in E_G$"
	$X_i \supseteq B(Dst_{[B,i]})$ Null assertion	$Dst_{[A,i]} \supseteq Dst_{[B,i]}$ "$\forall n[B\langle i,n\rangle \in E_G$ imp. $A\langle i,n\rangle \in E_G]$"
$Rchd_{[A_1^{-1},i]} \supseteq X_j$ "$A\langle i,j\rangle \in E_G$"	$X_j \supseteq node_j$ Null Assertion	$Rchd_{[A_1^{-1},i]} \supseteq node_j$ "$A\langle i,j\rangle \in E_G$"
	$X_j \supseteq B(X_k)$ (Encodes $B\langle j,k\rangle$)	$Rchd_{[A_1^{-1},i]} \supseteq B(X_k)$ "$\exists j[A\langle i,j\rangle \in E_G$ and $B\langle j,k\rangle \in E_G]$"
	$X_j \supseteq B(Dst_{[B,j]})$ Null assertion	$Rchd_{[A_1^{-1},i]} \supseteq B(Dst_{[B,j]})$ "$A\langle i,j\rangle \in E_G$"
$Rchd_{[A_1^{-1},i]} \supseteq Dst_{[A,i]}$ Null assertion	$Dst_{[A,i]} \supseteq B(X_j)$ "$\exists k[A\langle i,k\rangle \in E_G$ and $B\langle k,j\rangle \in E_G]$"	$Rchd_{[A_1^{-1},i]} \supseteq B(X_j)$ "$\exists k[A\langle i,k\rangle \in E_G$ and $B\langle k,j\rangle \in E_G]$"
	$Dst_{[A,i]} \supseteq B(Dst_{[B,j]})$ "$A\langle i,j\rangle \in E_G$"	$Rchd_{[A_1^{-1},i]} \supseteq B(Dst_{[B,j]})$ "$A\langle i,j\rangle \in E_G$"
$Dst_{[A,i]} \supseteq X_j$ "$A\langle i,j\rangle \in E_G$"	$X_j \supseteq node_j$ Null assertion	$Dst_{[A,i]} \supseteq node_j$ "$A\langle i,j\rangle \in E_G$"
	$X_j \supseteq B(X_k)$ (Encodes $B\langle j,k\rangle$)	$Dst_{[A,i]} \supseteq B(X_k)$ "$\exists j[A\langle i,j\rangle \in E_G$ and $B\langle j,k\rangle \in E_G]$"
	$X_j \supseteq B(Dst_{[B,j]})$ Null assertion	$Dst_{[A,i]} \supseteq B(Dst_{[B,j]})$ "$A\langle i,j\rangle \in E_G$"
$Dst_{[A,i]} \supseteq Dst_{[B,j]}$ "$\forall n[B\langle j,n\rangle \in E_G$ imp. $A\langle i,n\rangle \in E_G]$"	$Dst_{[B,j]} \supseteq node_k$ "$B\langle j,k\rangle \in E_G$"	$Dst_{[A,i]} \supseteq node_k$ "$A\langle i,k\rangle \in E_G$"
	$Dst_{[B,j]} \supseteq C(X_k)$ "$\exists n[B\langle j,n\rangle \in E_G$ and $C\langle n,k\rangle \in E_G]$"	$Dst_{[A,i]} \supseteq C(X_k)$ "$\exists n[A\langle j,n\rangle \in E_G$ and $C\langle n,k\rangle \in E_G]$"
	$Dst_{[B,j]} \supseteq C(Dst_{[C,k]})$ "$B\langle j,k\rangle \in E_G$"	$Dst_{[A,i]} \supseteq C(Dst_{[C,k]})$ "$A\langle j,k\rangle \in E_G$"

Table 3: Summary of the reductions that the SC-Reduction Algorithm may perform on a constructed set-constraint problem. For each line of the table, column 3 shows the constraint that results from reducing the constraints shown in columns 1 and 2. Each constraint is shown with its purpose in the original construction, or with its associated assertion in Lemma 3.2, where E_G denotes the set of edges in graph G. The highlighted entries indicate the key result for Lemma 3.2.

Note that a constraint of the form $X_i \supseteq A(X_j)$ cannot be generated by the SC-Reduction Algorithm; this means that if $X_i \supseteq A(X_j)$ appears in $\mathcal{C}'$, it must also appear in $\mathcal{C}$. This means that $X_i \supseteq A(X_j)$ either encodes $A\langle i,j\rangle$, or else $j = i$, and $X_i \supseteq A(X_i)$ encodes a result of the production "$A ::= \epsilon$" by representing the edge $A\langle i,i\rangle$. In either case, G contains the edge $A\langle i,j\rangle$.

It remains for us to show that if $\mathcal{C}'$ contains a constraint of the form $Dst_{[A,i]} \supseteq node_j$, then G contains the edge $A\langle i,j\rangle$. To do this, we associate as assertion about the graph G with every constraint generated by the SC-Reduction Algorithm as shown below (where E_G is the set of edges of the graph G):

Constraint form:	Associated assertion:
$Rchd_{[A_1^{-1},i]} \supseteq X_j$	"$A\langle i,j\rangle \in E_G$"
$Rchd_{[A_1^{-1},i]} \supseteq Dst_{[A,i]}$	Null assertion
$Rchd_{[A_1^{-1},i]} \supseteq B(X_j)$	"$\exists k[A\langle i,k\rangle \in E_G$ and $B\langle k,j\rangle \in E_G]$"
$Rchd_{[A_1^{-1},i]} \supseteq B(Dst_{[B,j]})$	"$A\langle i,j\rangle \in E_G$"
$Rchd_{[A_1^{-1},i]} \supseteq node_j$	"$A\langle i,j\rangle \in E_G$"
$Dst_{[A,i]} \supseteq X_j$	"$A\langle i,j\rangle \in E_G$"
$Dst_{[A,i]} \supseteq Dst_{[B,j]}$	"$\forall n[B\langle j,n\rangle \in E_G$ impl. $A\langle i,n\rangle \in E_G]$"
$Dst_{[A,i]} \supseteq B(X_j)$	"$\exists k[A\langle i,k\rangle \in E_G$ and $B\langle k,j\rangle \in E_G]$"
$Dst_{[A,i]} \supseteq node_j$	"$A\langle i,j\rangle \in E_G$"

Table 3 summarizes the reductions that may take place in a set-constraint problem created by our construction; each constraint is shown with its associated assertion. It is clear that for all lines of Table 3, the assertion A associated with a generated constraint $V \supseteq sexp$ (shown in column 3) is supported by the assertions associated with the constraints (shown in columns 1 and 3) that were reduced to $V \supseteq sexp$. Since the (implicit) assertions associated with the constraints in $\mathcal{C}$ follow from the original construction, it follows that for each constraint generated by the SC-Reduction Algorithm, the associated assertion is true. In particular, for any constraint of the form $Dst_{[A,i]} \supseteq node_j$ in $\mathcal{C}'$, it follows that G contains the edge $A\langle i,j\rangle$ (see the two highlighted boxes in Table 3). □

B Correctness of the Set-Constraint to CFL-reachability Construction

In this section we prove the lemmas used in Section 4.3. We use the following definitions:

$\mathcal{C}$ is a collection of set constraints.

$\mathcal{P}$ is the CFL-reachability problem constructed to represent $\mathcal{C}$.

$\mathcal{C}'$ is the collection of set constraints that results from running the SC-Reduction Algorithm on $\mathcal{C}$ (*i.e.*, $\mathcal{C}'$ is $\mathcal{C}$ unioned with the constraints generated by the SC-Reduction Algorithm).

G is the graph of the CFL-reachability problem $\mathcal{P}$.

G' is the graph that results from running the CFL-reachability algorithm on $\mathcal{P}$ (*i.e.*, G' is G augmented with the edges added by the CFL-reachability Algorithm).

To prove Lemmas 4.4 and 4.5, it is useful to have the following observation:

Observation B.1 *If G' contains the edges $Ground\langle V_1,V_1\rangle \ldots Ground\langle V_r,V_r\rangle$, and $c(V_1,V_2,\ldots,V_r)$ is an atomic expression used in $\mathcal{C}$ with index k, then G' contains the edge $Ground\langle(k),(k)\rangle$.*

This follows from the construction of $\mathcal{P}$. In particular, the CFL-reachability Algorithm will use the production

$$Ground ::= edge(k)toV_1 \quad Ground \quad edgeV_1to(k) \quad \ldots \quad edge(k)toV_r \quad Ground \quad edgeV_rto(k)$$

with the appropriate edges to induce the edge $Ground\langle(k),(k)\rangle$. (See Section 4.1.2 for details about how groundness information is handled in the constructed CFL-Reachability problem.)

Lemma 4.4 *If $\mathcal{C}'$ contains the constraint $V \supseteq c(V_1,V_2,\ldots,V_r)$, then G' contains the edge $Id\langle(k),V\rangle$, where k is the index of $c(V_1,V_2,\ldots,V_r)$.*
Proof: To show this, we must simultaneously prove the following:

(a) If $\mathcal{C}'$ contains $V_1 \supseteq V_2$, then G' contains $Id\langle V_2,V_1\rangle$.

(b) If V is ground in $\mathcal{C}'$, then G' contains $Ground\langle V_V,V_V\rangle$.

Observe that the construction of G guarantees the following:

- If $\mathcal{C}$ contains $V \supseteq c(V_1, V_2, \ldots, V_r)$, then G (and hence G') contains $Id\langle(k), V\rangle$.
- If $\mathcal{C}$ contains $V_1 \supseteq V_2$, then G (and hence G') contains $Id\langle V_2, V_1\rangle$.

To prove the lemma and goals (a) and (b), we show that the following conditions hold when the SC-Reduction Algorithm is run on $\mathcal{C}$:

(1) If the SC-Reduction Algorithm generates the constraint $V \supseteq c(V_1, V_2, \ldots, V_r)$, then G' contains the edge $Id\langle(k), V\rangle$, where k is the index of $c(V_1, V_2, \ldots, V_r)$.

(2) If the SC-Reduction Algorithm generates the constraint $V_1 \supseteq V_2$, then G' contains the edge $Id\langle V_2, V_1\rangle$.

(3) If the SC-Reduction Algorithm marks the variable V as ground, then G' contains the edge $Ground\langle V, V\rangle$.

The lemma follows immediately from condition (1).

Assume, on the contrary, that one or more conditions (1)-(3) fails. Then there must be some first action taken by the SC-Reduction Algorithm that causes the conditions to fail. There are three cases:

case 1: Suppose the SC-Reduction Algorithm generates the constraint $V \supseteq c(V_1, V_2, \ldots, V_r)$, and G' does not contain the edge $Id\langle(k), V\rangle$.

The only way for the SC-Reduction Algorithm to generate this constraint is from the constraints

$V \supseteq U$ and
$U \supseteq c(V_1, V_2, \ldots, V_r)$

where $c(V_1, V_2, \ldots, V_r)$ is ground. Since the SC-Reduction Algorithm has established that $c(V_1, V_2, \ldots, V_r)$ is ground, the variables $V_1 \ldots V_r$ must be marked as ground. Since this is the first failure of conditions (1)-(3), G' must contain the edges $Id\langle(k), U\rangle$ and $Id\langle U, V\rangle$ and the edges $Ground\langle V_1, V_1\rangle$ ... $Ground\langle V_r, V_r\rangle$. This allows us to use Observation B.1 to conclude that G' contains the edge $Ground\langle(k), (k)\rangle$. Finally, G' contains the edge $ae\langle(k), (k)\rangle$.

Since the context-free grammar of $\mathcal{P}$ contains the production "$Id ::= Ground\ ae\ Id\ Id$," it follows that G' must contain the edge $Id\langle(k), V\rangle$, which contradicts our supposition.

case 2: Suppose the SC-Reduction Algorithm generates the constraint $U \supseteq V_i$, and G' does not contain the edge $Id\langle V_i, U\rangle$.

The only way for the SC-Reduction Algorithm to generate this constraint is from constraints of the form

$U \supseteq c_i^{-1}(V)$ and
$V \supseteq c(V_1, V_2, \ldots, V_r)$

where $c(V_1, V_2, \ldots, V_r)$ is ground. The SC-Reduction Algorithm performs this reduction only if it has already marked the variables $V_1 \ldots V_r$ as ground. (This follows because the SC-Reduction Algorithm adds the constraint $c(V_1, V_2, \ldots, V_r)$ to its worklist only if $V_1 \ldots V_r$ have been marked ground.) Since this is the first failure of conditions (1)-(3), G' must contain the edge $Id\langle(k), V\rangle$ and the edges

$Ground\langle V_1, V_1\rangle$
$Ground\langle V_2, V_2\rangle$
$\vdots$
$Ground\langle V_r, V_r\rangle$

By Observation B.1, we conclude that G' also contains the edge $Ground\langle(k), (k)\rangle$. From the construction of G, it follows that G' contains the edges $c_i\langle V_i, (k)\rangle$ and $c_i^{-1}\langle V, U\rangle$, as well.

We also have that the context-free grammar of $\mathcal{P}$ contains the production "$Id ::= c_j\ Ground\ Id\ c_j^{-1}$." Given the above edges and this production, the CFL-reachability Algorithm generates the edge $Id\langle V_i, U\rangle$, which contradicts our supposition.

case 3: Suppose the SC-Reduction Algorithm marks the variable V ground and G' does not contain the edge $Ground\langle V, V\rangle$.

There are two reasons why the SC-Reduction Algorithm might mark V as ground, which are covered is the following subcases:

case 3.a: Suppose the SC-Reduction Algorithm marks V as ground because U is marked as ground and the constraint $V \supseteq U$ is present. Since this is the first failure of any of the conditions (1)-(3) above, we have that G' must contain the edges $Ground\langle U, U\rangle$ and $Id\langle U, V\rangle$. It follows from the construction of $\mathcal{P}$ that G' also contains the edges $edgeVtoV\langle V, V\rangle$ and $Rev_Id\langle V, U\rangle$. The context-free grammar of $\mathcal{P}$ contains the production

$$Ground ::= edgeVtoV \;\; RevId \;\; Ground \;\; Id \;\; edgeVtoV$$

This means that G' must contain the edge $Ground\langle V, V\rangle$, which is a contradiction.

case 3.b: Suppose the SC-Reduction Algorithm marks the variable V ground because $V_1 \ldots V_r$ are marked ground and the constraint $V \supseteq c(V_1, V_2, \ldots, V_r)$ is present. Since this is the first failure of any of the conditions (1)-(3), G' contains the edge $Id\langle (k), V\rangle$. We also have that G' contains the edge $Ground\langle (k), (k)\rangle$ (by the argument in case 1 above). By the construction of $\mathcal{P}$, it follows that G' also contains the edges $edgeVtoV\langle V, V\rangle$ and $Rev_Id\langle V, (k)\rangle$. This means that the production

$$Ground ::= edgeVtoV \;\; RevId \;\; Ground \;\; Id \;\; edgeVtoV$$

causes the CFL-reachability Algorithm to induce the edge $Ground\langle V, V\rangle$, which is a contradiction.

Thus, there can be no action taken by the SC-Reduction Algorithm that causes conditions (1)-(3) to be violated. □

Lemma 4.5 *If G' contains the edge $Id\langle (k), V\rangle$, then $\mathcal{C}'$ contains the constraint $V \supseteq c(V_1, V_2, \ldots, V_r)$ where $c(V_1, V_2, \ldots, V_r)$ is the atomic expression with index k.*

Proof: To show this, we need to prove a stronger property, namely that the following four conditions hold:

(1) If G' contains the edge $Id\langle (k), V\rangle$, then $\mathcal{C}'$ contains the constraint $V \supseteq c(V_1, V_2, \ldots, V_r)$, where $c(V_1, V_2, \ldots, V_r)$ has index k.

(2) If G' contains the edge $Id\langle V_i, V_j\rangle$, then $\mathcal{C}'$ contains the constraint $V_j \supseteq V_i$.

(3) If G' contains the edge $Ground\langle V, V\rangle$, then the variable V is ground in $\mathcal{C}'$.

(4) If G' contains the edge $Ground\langle (k), (k)\rangle$, then the atomic expression $c(V_1, V_2, \ldots, V_r)$ is ground in $\mathcal{C}'$, where $c(V_1, V_2, \ldots, V_r)$ has index k.

Note that edges from G satisfy the above conditions. Thus, if G' contains an edge e such that one or more of the above conditions is not satisfied, then e must have been added by the CFL-reachability Algorithm. Assume, on the contrary, that such an edge e exists in G'. Without loss of generality, let e be the first edge generated by the CFL-reachability Algorithm that causes one (or more) of the above conditions to fail.

case 1: Suppose e has the form $Id\langle (k), V\rangle$ and condition (1) is violated. The only way the CFL-Reachability Algorithm can generate this constraint is from the production "$Id ::= Ground \;\; ae \;\; Id \;\; Id$." This implies that the edges $Ground\langle (k), (k)\rangle$, $Id\langle (k), U\rangle$, and $Id\langle U, V\rangle$ are present before e. Since e is the first failure of conditions (1)-(4), it follows that $\mathcal{C}$ contains the constraints

$$U \supseteq c(V_1, V_2, \ldots, V_r) \text{ and } V \supseteq U$$

and $c(V_1, V_2, \ldots, V_r)$ is ground in $\mathcal{C}'$. This means that $\mathcal{C}'$ must contain $V \supseteq c(V_1, V_2, \ldots, V_r)$, which contradicts our assumption.

case 2: Suppose e has the form $Id\langle V_i, V_j\rangle$ and condition (2) is violated. To generate this edge, the CFL-Reachability Algorithm must use a production of the following form:

$$Id ::= c_i \;\; Ground \;\; Id \;\; c_i^{-1}$$

This implies that G' must contain the edges

$c_i\langle V_i, (k)\rangle$,
$Ground\langle (k), (k)\rangle$,
$Id\langle (k), U\rangle$, and
$c_i^{-1}\langle U, V_j\rangle$

where k is the index of an atomic expression of the form $c(\ldots V_i \ldots)$. Since this is the first failure of conditions (1)-(4), the edge $Ground\langle (k), (k)\rangle$ implies that $c(\ldots V_i \ldots)$ is ground in $\mathcal{C}'$, and the edge $Id\langle (k), U\rangle$ implies that $\mathcal{C}'$ contains the constraint $U \supseteq c(\ldots V_i \ldots)$. The edge $c_i^{-1}\langle U, V_j\rangle$ encodes the constraint $V_j \supseteq c_i^{-1}(U)$, which must be in $\mathcal{C}$. It follows that $\mathcal{C}'$ must contain the constraint $V_j \supseteq V_i$, which contradicts our supposition.

case 3: Suppose e has the form $Ground\langle V, V\rangle$ and condition (3) is violated. To generate this edge, the CFL-Reachability Algorithm uses the following production:

$$Ground ::= edgeVtoV \quad Rev_Id \quad Ground \quad Id \quad edgeVtoV$$

It follows that G' must contain either the edges $Ground\langle U, U\rangle$ and $Id\langle U, V\rangle$ or the edges $Ground\langle (k), (k)\rangle$ and $Id\langle (k), V\rangle$. In either case, since this is the first failure of conditions (1)-(4), it follows that V is ground in $\mathcal{C}'$, which contradicts our supposition.

case 4: Suppose e has the form $Ground\langle (k), (k)\rangle$ and condition (4) is violated. Let $c(V_1, V_2, \ldots, V_r)$ be the atomic expression with index k. The only way for the CFL-reachability Algorithm to generate the edge $Ground\langle (k), (k)\rangle$ is by using the following production:

$$Ground ::= edge(k)toV_1 \quad Ground \quad edgeV_1to(k) \quad \ldots \quad edge(k)toV_r \quad Ground \quad edgeV_rto(k)$$

This implies that the edges $Ground\langle V_1, V_1\rangle \ldots Ground\langle V_r, V_r\rangle$ are present before e is generated. Since the introduction of e is the first failure of conditions (1)-(4), this implies that the variables $V_1 \ldots V_r$ are all ground in $\mathcal{C}'$. But then $c(V_1, V_2, \ldots, V_r)$ must also be ground in $\mathcal{C}'$, which contradicts our supposition.

Thus, the CFL-reachability Algorithm does not generate any edge that causes conditions (1)-(4) to fail. The lemma is the same as condition (1). □

References

[1] F. Afrati and C. H. Papadimitriou. The parallel complexity of simple chain queries. In *Proceedings of the 6yd ACM Symposium on Principles of Database Systems*, pages 210–214, 1987.

[2] A. Aiken and B. Murphy. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, pages 427–447, August 1991.

[3] A. Aiken and B. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, January 1991.

[4] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[5] U. Assmann. On edge addition rewrite systems and their relevance to program analysis. In J. Cuny, editor, *5th Workshop on Graph Grammars and Their Application To Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, Williamsburg, Virginia, November 1994 1995. Springer.

[6] W.A. Babich and M. Jazayeri. The method of attributes for data flow analysis: Part ii. demand analysis. *Acta Inf.*, 10(3):265–272, October 1978.

[7] L. Bachmir, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Symposium on Logic in Computer Science*, pages 75–83, June 1993.

[8] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 47–56, 1988.

[9] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 57–66, 1988.

[10] K.D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *ACM Symposium on Principles of Programming Languages*, pages 49–59, 1989.

[11] D. Dolev, S. Even, and R.M. Karp. On the security of ping-pong protocols. *Information and Control*, 55:57–68, 1982.

[12] E. Duesterwald, R. Gupta, and M.L. Soffa. Demand-driven computation of interprocedural data flow. In *ACM Symposium on Principles of Programming Languages*, pages 37–48, 1995.

[13] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *Conference Record of the IEEE 12th Symposium on Switching and Automata Theory*, 1971.

[14] F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, 1984.

[15] M.S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, 1977.

[16] N. Heintze. *Set-based program analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1992.

[17] N. Heintze. Set based analyis of ML programs. Technical Report CMU-CS-93-193, Carnegie Mellon University, 1993.

[18] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints. Technical Report CMU-CS-91-110, Carnegie Mellon University, 1991.

[19] N. Heintze and J. Jaffar. Set constraints and set-based analysis. In *2nd Workshop on Priciples and Practice of Constraint Programming*, May 1994.

[20] N. Heintze and D. McAllester. Linear-time subtransitive control flow analysis. In *SIGPLAN Conference on Programming Languages Design and Implementation*, 1997.

[21] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *LICS '97: Proceedings of the IEEE Symposium on Logic in Computer Science*, 1997.

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 35–46, 1988.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[24] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, October 1995.

[25] S. Horwitz, T. Reps, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, December 1994.

[26] T. Jensen. Inference of polymorphic and conditional strictness properties. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1998.

[27] N. D. Jones. Flow analysis of lazy higher-order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 103–122. Ellis Horwood, Chichester, England, 1987.

[28] N. D. Jones and W. T. Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science 3*, pages 105–117, 1977.

[29] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of LISP-like structures. *Program Flow Analysis: Theory and Applications*, 1981.

[30] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[31] U.P. Khedker and D.M. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems*, 16(5):1472–1511, September 1994.

[32] L.T. Kou. On live-dead analysis for global data flow problems. *J. ACM*, 24(3):473–483, July 1977.

[33] D. McAllester and N. Heintze. On the complexity of set-based analysis. In *ICFP '97: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, 1997.

[34] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[35] J. Palsberg. Closure analysis in constraint form. *toplas*, 17(1):47–62, January 1995.

[36] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*. Kluwer Academic Publishers, 1994.

[37] T. Reps. Shape analysis as a generalized path problem. In *PEPM '95: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New York, NY, 1995. ACM.

[38] T. Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Inf.*, 33:739–757, 1996.

[39] T. Reps. Program analysis via graph reachability. In J. Maluszynski, editor, *Proceedings of ILPS '97: International Logic Programming Symposium*, pages 5–19. The M.I.T. Press, Cambridge, MA, 1997.

[40] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[41] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report TR 94-14, Datalogisk Institut, University of Copenhagen, 1994.

[42] T. Reps and T. Turnidge. Program specialization via program slicing. In O. Danvy, R. Glueck, and P. Thiemann, editors, *Proc. of the Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429, Schloss Dagstuhl, Wadern, Germany, February 1996. Springer-Verlag.

[43] J.C. Reynolds. Automatic computation of data set definitions. In *Information Processing 68: Proceedings of the IFIP Congress*, pages 456–461, New York, NY, 1968. North-Holland.

[44] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.

[45] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, University of Copenhagen, 1991.

[46] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[47] O. Shivers. Control flow analysis in scheme. In *ACM SIGPLAN'88, Conference on Programming Language Design and Implementation*, June 1988.

[48] J. D. Ullman and A. Van Gelder. Parallel complexity of logical query programs. In *Proceedings of the 27th IEEE Symposium on Foundation of Computer Science*, pages 438–454, 1986.

[49] Charatonik W and L. Pacholski. Set constraints with projections are in nexptime. In *Proceedings of 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 642–653, 1994.

[50] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[51] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Symposium on Principles of Database Systems*, pages 230–242, 1990.

[52] F.K. Zadeck. Incremental data flow analysis in a structured program editor. In *ACM Symposium on Compiler Construction*, pages 132–143, 1984.

Form of label	# of edges	Productions with label on the right-hand side	Work performed for a given edge	
			# examined productions	Total # of attempts to add an edge
Id	$O(n^2)$	$Id ::= Ground\text{-}ae\text{-}Id \quad Id$	1	$O(t)$
		$Ground\text{-}ae\text{-}Id ::= Ground\text{-}ae \quad Id$	1	1
		$Ground\text{-}Id ::= Ground \quad Id$	1	1
Rev_Id	$O(n^2)$	$Rev_Id ::= Rev_Id \quad Rev_Id\text{-}Ground\text{-}ae$	1	$O(t)$
		$Rev_Id\text{-}Ground\text{-}ae ::= Rev_Id \quad Ground\text{-}ae$	1	1
		$G ::= Rev_Id \quad Ground\text{-}Id$	1	$O(n)$
		$Rev_Id\text{-}Ground ::= Rev_Id \quad Ground$	1	1
$Ground$	$O(n)$	$Ground\text{-}edgeV_i to(k) ::= Ground \quad edgeV_i to(k)$	$O(t)$	$O(t)$
		$Ground\text{-}Id ::= Ground \quad Id$	1	$O(n)$
		$Rev_Id\text{-}Ground ::= Rev_Id \quad Ground$	1	$O(n)$
		$Ground\text{-}ae ::= Ground \quad ae$	1	1
$Ground\text{-}ae\text{-}Id$	$O(tv)$	$Id ::= Ground\text{-}ae\text{-}Id \quad Id$	1	$O(v)$
$Rev_Id\text{-}Ground\text{-}ae$	$O(vt)$	$Rev_Id ::= Rev_Id \quad Rev_Id\text{-}Ground\text{-}ae$	1	$O(v)$
$Ground\text{-}ae$	$O(t)$	$Ground\text{-}ae\text{-}Id ::= Ground\text{-}ae \quad Id$	1	$O(v)$
ae	$O(t)$	$Ground\text{-}ae ::= Ground \quad ae$	1	1
c_i	$O(t)$	$Id ::= c_i \quad Ground\text{-}Id\text{-}c_i^{-1}$	1	$O(t)$
rev_c_i	$O(t)$	$Rev_Id ::= Rev_c_i^{-1}\text{-}Rev_Id\text{-}Ground \quad rev_c_i$	1	$O(t)$
c_i^{-1}	$O(t)$	$Ground\text{-}Id\text{-}c_i^{-1} ::= Ground\text{-}Id \quad c_i^{-1}$	1	$O(n)$
$rev_c_i^{-1}$	$O(t)$	$Rev_c_i^{-1}\text{-}Rev_Id\text{-}Ground ::= rev_c_i^{-1} \quad Rev_Id\text{-}Ground$	1	$O(n)$
$edgeV_i to(k)$	$O(t)$	$Ground\text{-}edgeV_i to(k) ::= Ground \quad edgeV_i to(k)$	1	1
$edge(k) toV_i$	$O(t)$	$MarkV_i GrAt(k) ::= edge(k) toV_i \quad Ground\text{-}edgeV_i to(k)$	1	1
$edgeV_j toV_j$	$O(v)$	$G\text{-}edgeV_j toV_j ::= G \quad edgeV_j toV_j$	1	$O(n)$
		$Ground ::= edgeV_j toV_j \quad G\text{-}edgeV_j toV_j$	1	1
G	$O(n^2)$	$G\text{-}edgeV_j toV_j ::= G \quad edgeV_j toV_j$	$O(v)$	1
$Ground\text{-}Id$	$O(n^2)$	$G ::= Rev_Id \quad Ground\text{-}Id$	1	$O(n)$
		$Ground\text{-}Id\text{-}c_i^{-1} ::= Ground\text{-}Id \quad c_i^{-1}$	$O(t)$	$O(t)$
$Rev_Id\text{-}Ground$	$O(n^2)$	$Rev_c_i^{-1}\text{-}Rev_Id\text{-}Ground ::= rev_c_i^{-1} \quad Rev_Id\text{-}Ground$	$O(t)$	$O(t)$
$Ground\text{-}Id\text{-}c_i^{-1}$	$O(tn)$	$Id ::= c_i \quad Ground\text{-}Id\text{-}c_i^{-1}$	1	1
$Rev_c_i^{-1}\text{-}Rev_Id\text{-}Ground$	$O(tn)$	$Rev_Id ::= Rev_c_i^{-1}\text{-}Rev_Id\text{-}Ground \quad rev_c_i$	1	1
$G\text{-}edgeV_j toV_j$	$O(n^2)$	$Ground ::= edgeV_j toV_j \quad G\text{-}edgeV_j toV_j$	1	1
$MarkV_{a_j} GrAt(k)$	$O(t)$	$MarkV_{a_1}\text{-}V_{a_j} GrAt(k) ::= MarkV_{a_1}\text{-}V_{a_{j-1}} GrAt(k) \quad MarkV_{a_j} GrAt(k)$	r	r
$MarkV_{a_1}\text{-}V_{a_{j-1}} GrAt(k)$	$O(t)$	$MarkV_{a_1}\text{-}V_{a_j} GrAt(k) ::= MarkV_{a_1}\text{-}V_{a_{j-1}} GrAt(k) \quad MarkV_{a_j} GrAt(k)$	1	1
$MarkV_{a_1}\text{-}V_{a_r} GrAt(k)$	$O(t)$	$Ground ::= MarkV_{a_1}\text{-}V_{a_r} GrAt(k)$	1	1
$Ground\text{-}edgeV_i to(k)$	$O(t)$	$MarkV_i GrAt(k) ::= edge(k) toV_i \quad Ground\text{-}edgeV_i to(k)$	1	1

Table 3: Total work performed by the CFL-Reachability Algorithm on a constructed problem. Column 1 shows the forms of the labels used in a constructed problem. Column 2 gives a bound on the number of edges with labels of the form listed in column 1. Column 3 shows productions in which labels from column 1 appear on the right hand side. Column 4 shows the number of productions of the form in column 3 that will be examined when considering a fixed edge with a label of the form in column 1. Column 5 shows the number of new edges that may be produced in total for all of the productions counted in column 4. The total work performed is bounded by (column 4 + column 5) * column 2.

Form of label	# of edges	Productions with label on the right-hand side	Work performed for a given edge	
			# examined productions	Total # of attempts to add an edge
Id	$O(n^2)$	*Ground-c-Id* ::= *Ground-c* *Id*	$O(t)$	$O(1)$
		return-Id ::= *return* *Id*	1	$O(t)$
Rev_Id	$O(n^2)$	*input-Rev_Id* ::= *input* *Rev_Id*	1	$O(t)$
		$EdgeV_i toV_j$*-Rev_Id* ::= $edgeV_i toV_j$ *Rev_Id*	$O(t)$	$O(t)$
Ground	$O(n)$	*Ground-*$edgeV_i toV_j$ ::= *Ground* $edgeV_i toV_j$	$O(t)$	$O(t)$
		Ground-c ::= *Ground* *c-value*	$O(t)$	1
c_i	$O(t)$	C_i*-Ground-Id* ::= c_i *Ground-Id*	1	$O(n)$
c_i^{-1}	$O(t)$	*Id* ::= c_i*-Ground-Id* c_i^{-1}	1	$O(t)$
$edgeV_i toV_j$	$O(t)$	$EdgeV_i toV_j$*-Rev_Id* ::= $edgeV_i toV_j$ *Rev_Id*	1	$O(n)$
		Id ::= $edgeV_k toV_i$ *Ground-*$edgeV_i toV_j$	$O(t)$	$O(t)$
		Ground-c-Id-$edgeV_i toV_j$ ::= *Ground-c-Id* $edgeV_i toV_j$	$O(t)$	$O(t)$
Ground-Id	$O(n^2)$	c_i*-Ground-Id* ::= c_i *Ground-Id*	$O(t)$	$O(t)$
Ground-$edgeV_j toV_i$	$O(t)$	*Id* ::= $edgeV_k toV_j$ *Ground-*$edgeV_j toV_i$	$O(t)$	$O(t)$
input	$O(t)$	*input-Rev_Id* ::= *input* *Rev_Id*	1	$O(n)$
$input^-$	$O(t)$	*Id* ::= *input-Rev_Id* $input^-$	1	$O(t)$
input-Rev_Id	$O(tn)$	*Id* ::= *input-Rev_Id* $input^-$	1	$O(t)$
return	$O(t)$	*return-Id* ::= *return* *Id*	1	$O(n)$
$return^-$	$O(t)$	*Id* ::= *return-Id* $return^-$	1	$O(t)$
return-Id	$O(tn)$	*Id* ::= *return-Id* $return^-$	1	$O(t)$
Ground-c-Id	$O(tn)$	*Ground-c-Id-*$edgeV_i toV_j$ ::= *Ground-c-Id* $edgeV_i toV_j$	$O(t)$	$O(t)$
		Id ::= *Ground-c-Id* *non-c'-vals*	$O(t)$	$O(t)$
c-value	$O(t)$	*Ground-c* ::= *Ground* *c-value*	1	1
		$edgeV_i toV_j$*-Rev_Id-c* ::= $edgeV_i toV_j$*-Rev_Id* *c-value*	$O(t)$	$O(t)$
Ground-c	$O(t)$	*Ground-c-Id* ::= *Ground-c* *Id*	1	$O(n)$
non-c'-vals	$O(t)$	*Id* ::= *Ground-c-Id* *non-c'-vals*	$O(t)$	$O(t)$
$edgeV_i toV_j$*-Rev_Id*	$O(tn)$	$edgeV_i toV_j$*-Rev_Id-c* ::= $edgeV_i toV_j$*-Rev_Id* *c-value*	$O(t)$	1
$edgeV_i toV_j$*-Rev_Id-c*	$O(t^2)$	*Id* ::= $edgeV_i toV_j$*-Rev_Id-c* *Ground-c-Id-*$edgeV_j toV_k$	$O(t)$	$O(t)$
Ground-c-Id-$edgeV_j toV_k$	$O(t^2)$	*Id* ::= $edgeV_i toV_j$*-Rev_Id-c* *Ground-c-Id-*$edgeV_j toV_k$	$O(t)$	$O(t)$

Table 4: Work performed by the CFL-Reachability Algorithm on a problem constructed from an ML set-constraint problem. (See also Table 3). Column 1 shows the forms of the labels used in a constructed problem. Column 2 gives a bound on the number of edges with labels of the form listed in column 1. Column 3 shows productions in which labels from column 1 appear on the right-hand side. Column 4 shows the number of productions of the form in column 3 that will be examined when considering a fixed edge with a label of the form in column 1. Column 5 shows the number of new edges that may be produced in total for all of the productions counted in column 4. The total work performed is bounded by (column 4 + column 5) * column 2.