Interconvertibility 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

January 31, 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.

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 [13, 15], interprocedural dataflow analysis [14], and shape analysis [22].

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 [7].

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

- 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.

[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).

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

- 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 [15] and interprocedural slicing [25, 13]. Set-constraints lead to natural formulations of shape analysis [17, 27]. 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. (I.e., 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 [4], this is not the correct explanation. We have long believed that the 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 a class of set-constraint problems.

- CFL-reachability is known to be log-space complete for polynomial time (or "PTIME-complete") [23]. 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 [22, 21]), 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 [22]. Set constraints 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 can be used to analyze lazy languages.

A different class of set constraints has been used by Heintze to formulate analysis problems for a higher-order language (ML) [8]. 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 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, 4.2, and 5.3 we examine this issue and show that, in fact, the asymptotic run-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 set-constraint problems, and describe dynamic-programming algorithms that can be used

$A ::= B\ C$	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.

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. Section 6 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 CFL-reachability problems.

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 labelled 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 the grammar *CF*. The (all-pairs) context-free-language reachability problem (CFL-reachability problem) is the *(all-pairs) S-path problem*: Determine all pairs of vertices v_1, v_2 such that there exists an S-path in G from v_1 to v_2. □

2.1.1 Solving CFL-Reachability Problems

We now give a dynamic-programming algorithm for solving 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. (For the rest of the paper, 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 $T \cup N$[1]. This can be done by introducing new non-terminal symbols. Thus, a production such as

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

[1]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 be appear.

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 ::= ε do
    for each node i in the graph do
        if the edge A⟨i,i⟩ is not in G then
            add A⟨i,i⟩ to G and to W
        fi
    od
od
```

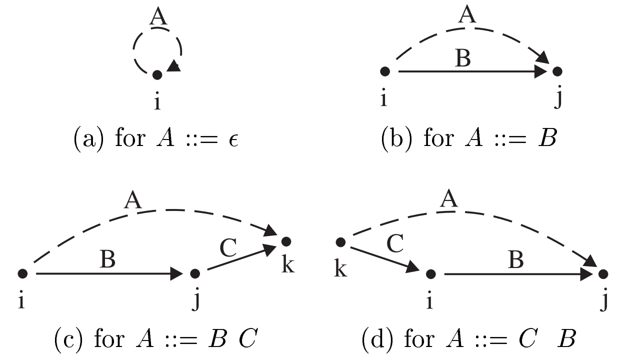
4. **Add edges for other productions:** To determine where to add other edges to the graph, the current edges must be examined.

```
while W is not empty do
    Select and remove an edge B⟨i,j⟩ 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⟨i,j⟩ is not in G then
            add A⟨i,j⟩ 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⟨j,k⟩, add A⟨i,k⟩ (see Figure 1(c)).                                                */
    for each production of the form A ::= B C do
        for each outgoing edge C⟨j,k⟩ from node j do
            if the edge A⟨i,k⟩ is not in G then
                add A⟨i,k⟩ 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⟨k,i⟩, add A⟨k,j⟩ (see Figure 1(d)).                                                */
    for each production of the form A ::= C B do
        for each incoming edge C⟨k,i⟩ into node i do
            if the edge A⟨k,j⟩ is not in G then
                add A⟨k,j⟩ to G and to W
```



(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).

```
                fi
            od
        od
    od
```

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

□

We now show that the running time of this 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, 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 [7, 8, 9].)

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

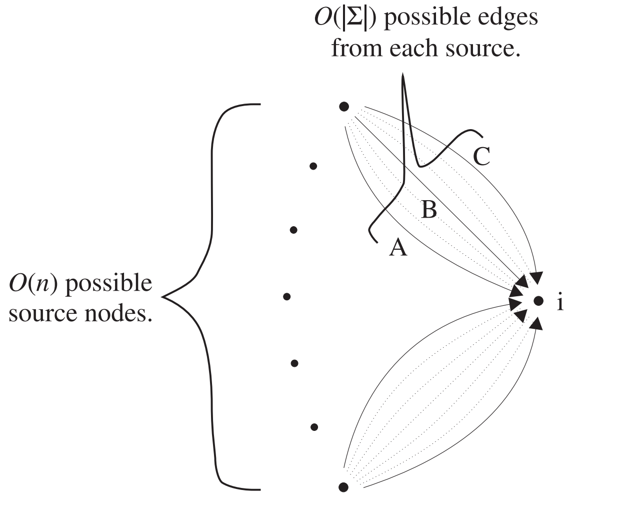


Figure 2: In a graph from a CFL-reachability problem, the number of edges into any given node is bounded by $O(|\Sigma|n)$, where Σ is the alphabet of the grammar, and n is the number of nodes in the graph.

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

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

$\mathcal{I}$ 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:

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

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 [7].

The solution to a collection of set constraints can be written as a *regular term grammar* [5], 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) [5].

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 $term_2$ ($term_1 \Rightarrow term_2$) 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 solution to Example 2.3 above has these productions:

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

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 [7]: 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 [7]. □

The SC-Reduction Algorithm never generates 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 $O(t)$, where t is the original number of constraints. The total number of constraints in $\mathcal{C}$ of the form $Y \supseteq c(V_1, V_2, \ldots, V_r)$ is bounded by $O(tv)$, where v is the number of set variables used in $\mathcal{C}$. 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 first reduction step is ever applied is bounded by $O(vt)$, and the number of times the second step is applied is bounded by $O(v^2t)$. In the worst case, v is 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 ⊇ sexp from W
    if X ⊇ sexp is of the form X ⊇ c(V_1, V_2, ..., V_r) then
        for each constraint of the form Y ⊇ c_i^{-1}(X) in C do
            if Y ⊇ V_i is not in C then
                Insert Y ⊇ V_i into C and W
            fi
        od
        for each constraint of the form Y ⊇ X in C do
            if Y ⊇ c(V_1, V_2, ..., V_r) is not in C then
                Insert Y ⊇ c(V_1, V_2, ..., V_r)
                into C and W
            fi
        od
    else if X ⊇ sexp is of the form X ⊇ Y then
        for each constraint of the form Y ⊇ c(V_1, V_2, ..., V_r) in C such that V_1, ..., V_r are all ground
        do
            if X ⊇ c(V_1, V_2, ..., V_r) is not in C then
                Insert X ⊇ c(V_1, V_2, ..., V_r)
                into 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. Also, ground information need not be propagated to generated constraints because generated constraints can only be created if their right-hand sides are ground. This means that ground information need only be propagated to the original constraints, of which there are only $O(t)$. Therefore, propagating ground information takes no more than time $O(t)$, and the entire algorithm runs in time $O(t^3)$.

3 Transforming CFL-Reachability 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.

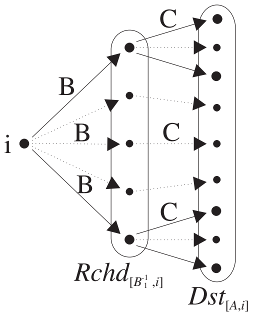


Figure 3: Use of $Dst_{[A,i]}$ and $Rchd_{[B_1^{-1},i]}$ to encode production $A ::= B\ C$. The variable $Rchd_{[B_1^{-1},i]}$ represents the set of nodes reached by following B-edges from i. The variable $Dst_{[A,i]}$ represents the set of nodes to which there should be an A-edge from node i.

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 $node_i$. For each node i in the graph, we introduce a unique set variable X_i and a unique, nullary constructor $node_i$. These are linked by constraints of the form

$X_i \supseteq node_i$, 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 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.

3.3 Correctness of the Construction

We claim that the solution to the set-constraint problem gives a solution to the original CFL-reachability problem. More precisely, let G be the regular term grammar that results from solving the set-constraint problem. Then there is an A-edge from node i to node j in the solution to the all-pairs problem iff $X_i \Rightarrow^* A(node_j)$ under G.

This can be proven by contradiction. The form of the argument is as follows: if solving the CFL-reachability problem introduces an edge that the solution to the constructed set-constraint problem misses, then there must be a first such edge. This leads to a contradiction. A similar argument works in the other direction.

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 program. A *log-space Turning 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 program $\mathcal{P}_1$ 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 program $\mathcal{P}_1$ 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 program $\mathcal{P}_2$ that, given a CFL-reachability problem with a context-free grammar in normal form, performs the construction of this 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 work-tape. 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 machine programs $\mathcal{Q}$ and $\mathcal{R}$, there is a log-space Turing machine program that is equivalent to the composition $\mathcal{Q} \circ \mathcal{R}$ [16]. This means that there is a log-space Turing machine program $\mathcal{P}$ 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) [23], this means that the given class of set-constraint problems are also PTIME-complete [16].

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 $O(pn)$ constraints, where p 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. We argue that the set-constraint problem can be solved in the same asymptotic time as the original CFL-reachability problem (i.e., $O(n^3)$). The initial constraints in a set-constraint problem constructed from a CFL-reachability problem must be in 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)

Selected Constraint form	Num. of possible constraints	Matching constraint form	Num. of possible matching constraints	Produced constraint	Total work
$Rchd_{[A_1^{-1},i]} \supseteq A_1^{-1}(X_i)$	sn	$X_i \supseteq A(X_j)$	n	$Rchd_{[A_1^{-1},i]} \supseteq X_j$	sn^2
		$X_i \supseteq A(Dst_{[A,i]})$	1	$Rchd_{[A_1^{-1},i]} \supseteq Dst_{[A,i]}$	sn
$Dst_{[A,i]} \supseteq B_1^{-1}(Rchd_{[C_1^{-1},i]})$	s^3n	$Rchd_{[C_1^{-1},i]} \supseteq B(X_j)$	n	$Dst_{[A,i]} \supseteq X_j$	s^3n^2
		$Rchd_{[C_1^{-1},i]} \supseteq B(Dst_{[B,j]})$	n	$Dst_{[A,i]} \supseteq Dst_{[B,j]}$	s^3n^2
$Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$	s^2n	$X_i \supseteq B(X_j)$	n	$Dst_{[A,i]} \supseteq X_j$	s^2n^2
		$X_i \supseteq B(Dst_{[B,i]})$	1	$Dst_{[A,i]} \supseteq Dst_{[B,i]}$	s^2n
$Rchd_{[A_1^{-1},i]} \supseteq X_j$	sn^2	$X_j \supseteq node_j$	1	$Rchd_{[A_1^{-1},i]} \supseteq node_j$	sn^2
		$X_j \supseteq B(X_k)$	sn	$Rchd_{[A_1^{-1},i]} \supseteq B(X_k)$	s^2n^3
		$X_j \supseteq B(Dst_{[B,k]})$	sn	$Rchd_{[A_1^{-1},i]} \supseteq B(Dst_{[B,k]})$	s^2n^3
$Rchd_{[A_1^{-1},i]} \supseteq Dst_{[A,i]}$	sn	$Dst_{[A,i]} \supseteq B(X_j)$	sn	$Rchd_{[A_1^{-1},i]} \supseteq B(X_j)$	s^2n^2
		$Dst_{[A,i]} \supseteq B(Dst_{[B,j]})$	sn	$Rchd_{[A_1^{-1},i]} \supseteq B(Dst_{[B,j]})$	s^2n^2
$Dst_{[A,i]} \supseteq X_j$	sn^2	$X_j \supseteq node_j$	1	$Dst_{[A,i]} \supseteq node_j$	sn^2
		$X_j \supseteq B(X_k)$	sn	$Dst_{[A,i]} \supseteq B(X_k)$	s^2n^3
		$X_j \supseteq B(Dst_{[B,j]})$	s	$Dst_{[A,i]} \supseteq B(Dst_{[B,j]})$	s^2n^2
$Dst_{[A,i]} \supseteq Dst_{[B,j]}$	s^2n^2	$Dst_{[B,j]} \supseteq C(X_k)$	sn	$Dst_{[A,i]} \supseteq C(X_k)$	s^3n^3
		$Dst_{[B,j]} \supseteq C(Dst_{[C,j]})$	s	$Dst_{[A,i]} \supseteq C(Dst_{[C,j]})$	s^3n^2

Table 2: Cost of the steps performed in solving a set-constraint problem that encodes a context-free reachability problem, where n is the number of nodes in the graph, and $s = |\Sigma|$. Columns 1 and 3 show a pair of constraints that the SC-Reduction Algorithm will reduce. Column 2 shows how many constraints of the form in column 1 may occur. Column 4 shows how many constraints of the form in column 3 may pair with a given constraint of the form in column 1. Column 5 shows the produced constraint, and column 6 shows how many pairings may occur between constraints of the forms in columns 1 and 3.

$$X_i \supseteq A(X_j) \qquad \text{(Encode an } A \text{ edge from } i \text{ to } j\text{)}$$

Following the rules of the SC-Reduction Algorithm, these constraints will give rise to constraints of the following additional forms:

$$\begin{array}{ll} Rchd_{[A_1^{-1},i]} \supseteq X_j & Dst_{[A,i]} \supseteq X_j \\ Rchd_{[A_1^{-1},i]} \supseteq Dst_{[A,i]} & Dst_{[A,i]} \supseteq Dst_{[B,j]} \\ Rchd_{[C_1^{-1},i]} \supseteq B(X_j) & Dst_{[B,j]} \supseteq C(X_k) \\ Rchd_{[C_1^{-1},i]} \supseteq B(Dst_{[B,j]}) & Dst_{[B,j]} \supseteq C(Dst_{[C,j]}) \end{array}$$

Table 2 summarizes the reductions that may take place and the cost of the work performed. Overall, the dominant term is s^3n^3, where $s = |\Sigma|$ is the size of the grammar's alphabet. 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 [22]. 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:

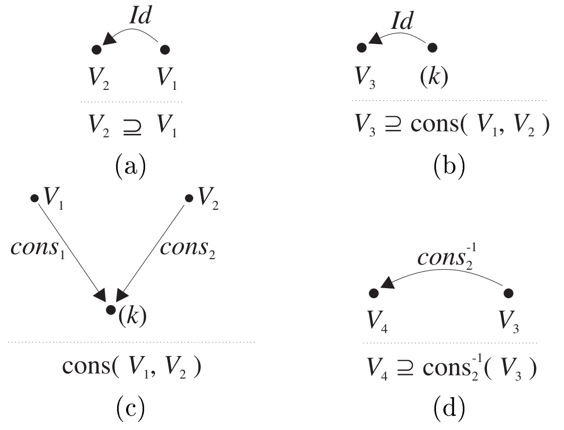


Figure 4: Edges inserted in the constructed graph to model terms and constraints.

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 $Id\langle V_1, V_2\rangle$[2]. 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 "$Id ::= cons_2 \;\; Id \;\; cons_2^{-1}$". This yields the edge $Id\langle V_2, V_4\rangle$, which models the constraint $V_4 \supseteq V_2$. □

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 4.1.3):

- For each set variable V_i, the graph contains a node labelled V_i.

[2]Technically, we should label these edges with a terminal symbol, say id, to get edges $id\langle a, V_1\rangle$ and $id\langle V_1, V_2\rangle$, and introduce the production $Id ::= id$. For convenience, we use only one symbol, Id.

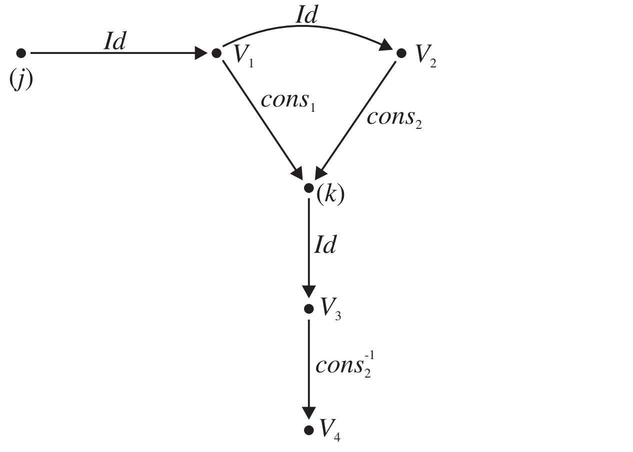


Figure 5: The graph built to encode the constraints in Example 4.1. The nodes (j) and (k) represent the atomic expressions a and $cons(V_1, V_2)$, respectively.

- 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 $c_2\langle V_j, (k)\rangle$. An edge $cons_m\langle V_i, (k)\rangle$ indicates that the values that reach V_i are wrapped in the m^{th} 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 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 k^{th} position of *cons* values at node j. (See Figure 4(d).)

Figure 5 shows the graph that is constructed to represent the set constraints of Example 4.1.

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:

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

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:

$$Id ::= Id\ \ Id$$

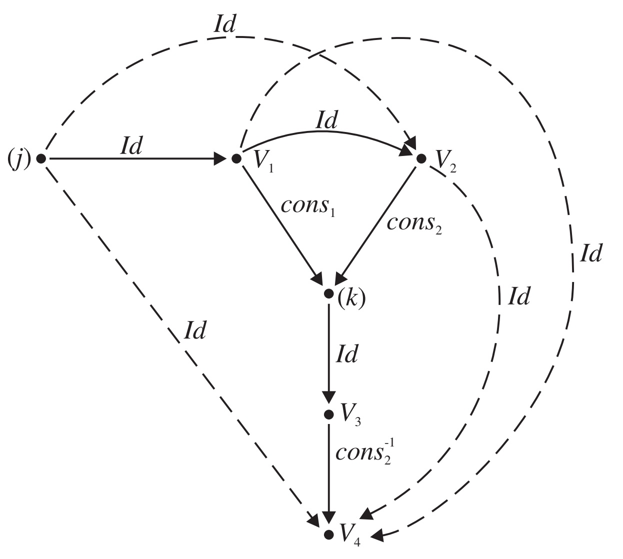


Figure 6: The graph for Example 4.1 after the CFL-reachability Algorithm has been run. Dashed lines represent edges inserted by the algorithm. The nodes (j) and (k) represent the atomic expressions a and $cons(V_1, V_2)$, respectively.

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 $Id ::= cons_1 \;\; Id \;\; cons_1^{-1}$ allows identity paths though *cons* expressions that are not ground. This is at odds with the simplification steps of the SC-Reduction Algorithm.

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

$$
\begin{aligned}
&V_1 \supseteq a \\
&V_3 \supseteq cons(V_1, V_2) \\
&V_4 \supseteq cons_1^{-1}(V_3) \\
&\qquad \vdots
\end{aligned}
$$

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 of set constraints to CFL-reachability problems. 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 can be used for the analysis of lazy languages (even though they were originally designed for use on strict languages).

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. $\square$

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 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:

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

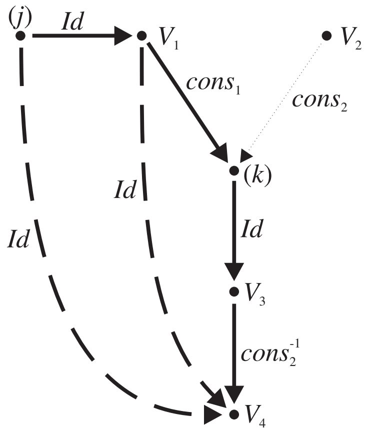


Figure 7: The edge $Id\langle V_1, V_4\rangle$ should be induced if and only if $cons(V_1, V_2)$ is ground. If the edge $Id\langle V_1, V_4\rangle$ is added when $cons(V_1, V_2)$ is not ground, it may incorrectly cause the edge $Id\langle a, V_4\rangle$ to be added, and the production $V_4 \Rightarrow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

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(b).

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[3].

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:

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

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(a).

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:

$$Ground ::= edgeV_3toV_3 \quad Rev\_Id \quad Ground \quad Id \quad edgeV_3toV_3$$

[3]Technically, we should label the edge with a terminal symbol *ground* (i.e. $ground\langle (j), (j)\rangle$) and use a production $Ground ::= ground$. For convenience, we use only one symbol, *Ground*.

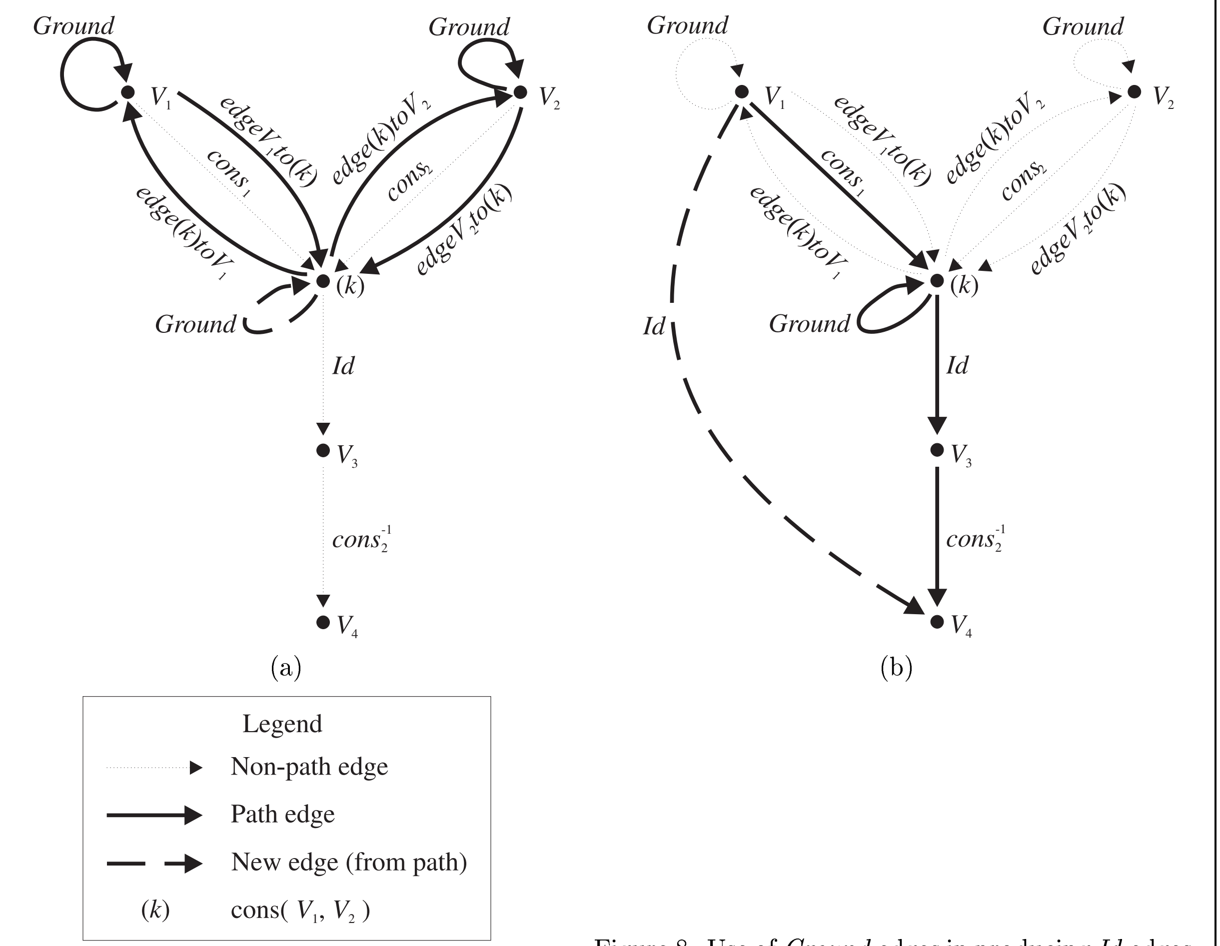


Figure 8: Use of *Ground* edges in producing *Id* edges.

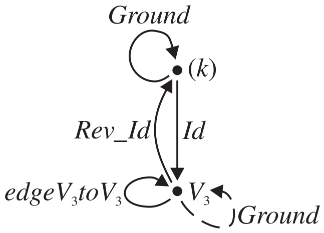


Figure 9: Propagation of *Ground* edges from (k) to V_3. This is accomplished using the production "*Ground* ::= *edge*V_3*to*V_3 *Rev_Id* *Ground* *Id* *edge*V_3*to*V_3".

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

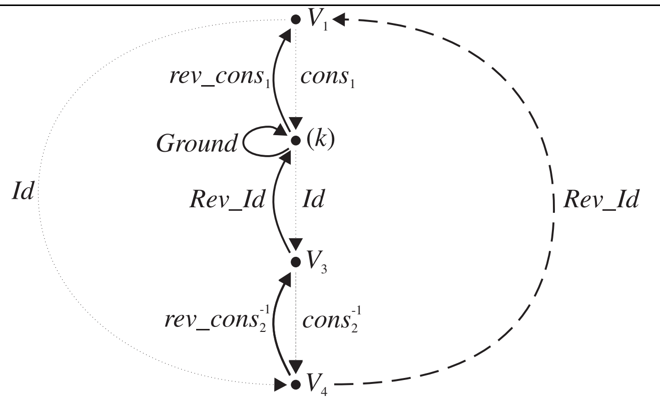


Figure 10: The production $\mathit{Rev\_Id} ::= \mathit{rev\_cons}_1^{-1}\ \ \mathit{Rev\_Id}\ \ \mathit{Ground}\ \ \mathit{rev\_cons}_1$ causes the CFL-reachability Algorithm to produce *Rev_Id* edges. (This production is the counterpart of the production $\mathit{Id} ::= \mathit{cons}_1\ \ \mathit{Ground}\ \ \mathit{Id}\ \ \mathit{cons}_1^{-1}$.)

example, the following production indicates that the CFL-reachability Algorithm should induce an *Id* edge (assuming an appropriate path exists in the graph):

$$\mathit{Id} ::= \mathit{cons}_1\ \ \mathit{Ground}\ \ \mathit{Id}\ \ \mathit{cons}_1^{-1}$$

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

$$\mathit{Rev\_Id} ::= \mathit{rev\_cons}_1^{-1}\ \ \mathit{Rev\_Id}\ \ \mathit{Ground}\ \ \mathit{rev\_cons}_1$$

Figure 10 illustrates the use of this reverse production.

For this production to work, we need additional reverse edges: For every edge $\mathit{cons}_1\langle V_i, V_j\rangle$ in the graph, we want the edge $\mathit{rev\_cons}_1\langle V_j, V_i\rangle$ to be in the graph; for every edge $\mathit{cons}_1^{-1}\langle V_i, V_j\rangle$, we want the edge $\mathit{rev\_cons}_1^{-1}\langle V_j, V_i\rangle$ 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*.

4.1.3 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

   $$\mathit{Id} ::= \mathit{Id}\ \ \mathit{Id}$$
   $$\mathit{Rev\_Id} ::= \mathit{Rev\_Id}\ \ \mathit{Rev\_Id}$$

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

   $$\mathit{Ground} ::= \mathit{edgeV_i toV_i}\ \ \mathit{Rev\_Id}\ \ \mathit{Ground}\ \ \mathit{Id}\ \ \mathit{edgeV_i toV_i}$$

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). If c is a nullary constructor (i.e., $r = 0$), then the graph contains the edge $\mathit{Ground}\langle(k),(k)\rangle$. 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

$$\begin{aligned} Ground \ ::= \ & edge(k)toV_1 \quad Ground \quad edgeV_1 to(k) \qquad edge(k)toV_2 \quad Ground \quad edgeV_2 to(k) \\ & \ldots \quad edge(k)toV_r \quad Ground \quad edgeV_r to(k) \end{aligned}$$

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 $rev\_c_k^{-1}\langle V_i, V_j\rangle$.

4.2 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.

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 4.2.1. In Section 4.2.2, we present Table 3 which summarizes all of the different types of edge labels that may be used in a constructed 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

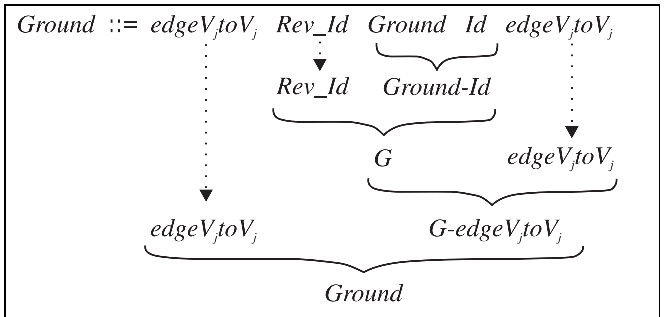


Figure 11: Normalization of the production $Ground ::= edgeV_j toV_j\ RevId\ Ground\ Id\ edgeV_j toV_j$.

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 ($n = v + t$), and r to refer to the maximum arity of a constructor.

4.2.1 Normalization of a Constructed Grammar

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

$$Ground ::= edgeV_j toV_j \quad RevId \quad Ground \quad Id \quad edgeV_j toV_j$$

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:

$$\begin{aligned} Ground &::= edgeV_j toV_j \quad G\text{-}edgeV_j toV_j \\ G\text{-}edgeV_j toV_j &::= G \quad edgeV_j toV_j \\ G &::= RevId \quad Ground\text{-}Id \\ Ground\text{-}Id &::= Ground \quad Id \end{aligned}$$

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-Id* and G anywhere in the graph. However, for any given V_j, there is only one edge labelled $edgeV_j toV_j$ in the graph; this is the edge $edgeV_j toV_j\langle V_j, V_j\rangle$. This means that for a fixed V_j, if the CFL-reachability Algorithm adds an edge $G\text{-}edgeV_j toV_j\langle V_i, V_k\rangle$, then it must use $edgeV_j toV_j\langle V_j, V_j\rangle$ to do so, and $k = j$. That is, all edges labelled $G\text{-}edgeV_j toV_j$ 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 $G\text{-}edgeV_j toV_j\langle V_i, V_j\rangle$ is bounded by $O(n)$, and the number of outgoing edges of the form $G\text{-}edgeV_k toV_k\langle V_j, V_k\rangle$ is bounded by $O(n)$. Also, of all the edges $G\text{-}edgeV_j toV_j\langle V_i, V_j\rangle$, only one, $G\text{-}edgeV_j toV_j\langle V_j, V_j\rangle$, can be combined with $edgeV_j toV_j\langle V_j, V_j\rangle$ to generate $Ground\langle V_j, V_j\rangle$.

Now we consider the following prototypical production:

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

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:

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

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 $O(trn)$.

We must also normalize productions having the following form:

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

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):

$$\begin{aligned} Ground &::= MarkV_1GrAt(k) \quad MarkV_2GrAt(k) \quad \ldots \quad MarkV_rGrAt(k) \\ MarkV_1GrAt(k) &::= edge(k)toV_1 \quad Ground \quad edgeV_2to(k) \\ MarkV_1GrAt(k) &::= edge(k)toV_2 \quad Ground \quad edgeV_2to(k) \\ &\;\vdots \\ MarkV_rGrAt(k) &::= edge(k)toV_r \quad Ground \quad edgeV_rto(k) \end{aligned}$$

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

$$MarkV_iGrAt(k) ::= edge(k)toV_i \quad Ground \quad edgeV_ito(k)$$

are normalized to the following productions:

$$\begin{aligned} MarkV_iGrAt(k) &::= edge(k)toV_i \quad Ground\text{-}edgeV_ito(k) \\ Ground\text{-}edgeV_ito(k) &::= Ground \quad edgeV_ito(k) \end{aligned}$$

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

$$Ground ::= MarkV_1GrAt(k) \quad MarkV_2GrAt(k) \quad \ldots \quad MarkV_rGrAt(k)$$

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

$$\begin{aligned} Ground &::= MarkV_1\text{-}V_rGrAt(k) \\ MarkV_1\text{-}V_2GrAt(k) &::= MarkV_1GrAt(k) \quad MarkV_2GrAt(k) \\ MarkV_1\text{-}V_3GrAt(k) &::= MarkV_1\text{-}V_2GrAt(k) \quad MarkV_3GrAt(k) \\ &\;\vdots \\ MarkV_1\text{-}V_rGrAt(k) &::= MarkV_1\text{-}V_{r-1}GrAt(k) \quad MarkV_rGrAt(k) \end{aligned}$$

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

4.2.2 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-Id*$\langle V_j, V_k\rangle$ from its worklist, it could potentially examine $O(tr)$ productions of the form *Ground-Id-*c_i^{-1} ::= *Ground-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-Id-*$c_i^{-1}\langle V_j, V_m\rangle$. 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 $O(tr)$ productions of the form *Id-*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-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 [8]. 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. 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 not ground terms of the form $c(v_1, \ldots, v_r)$. The presence of a value in Y_1 that is not a ground 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 are used to make set based analysis more accurate by preventing constraints that correspond to dead code from contributing to the solution. See [8] for details.

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:

- $\mathcal{I}(c(V_1, \ldots, V_r)) = \{c(v_1, \ldots, v_r) | v_1 \in \mathcal{I}(V_1), \ldots, v_r \in \mathcal{I}(V_r)\}$
- $\mathcal{I}(\lambda x.e) = \{\lambda x.e\}$
- $\mathcal{I}(\mathit{ifnonempty}(se_1, se_2)) =$
  if $\mathcal{I}(se_1) = \{\}$ then $\{\}$ else $\mathcal{I}(se_2)$;
- $\mathcal{I}(\mathit{apply}(se_1, se_2)) = \{v : \lambda x.e \in \mathcal{I}(se_1) \wedge \mathcal{I}(se_2) \neq \{\} \wedge v \in \mathcal{I}(\mathit{Range}_{\lambda x.e})\}$
  provided $\lambda x.e \in \mathcal{I}(se_1)$ implies $\mathcal{I}(se_2) \subseteq \mathcal{I}(V_x)$
- $\mathcal{I}(\mathit{case}(se_1, c(X_1, \ldots, X_n) \hookrightarrow se_2, Y \hookrightarrow se_3)) = S_1 \cup S_2$, 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. $S_2 = \{v : v \in \mathcal{I}(se_3) \wedge \exists c'(v_1, \ldots, v_n) \in \mathcal{I}(se_1) \text{ s.t. } c' \neq c\}$
  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 $\mathcal{C}$

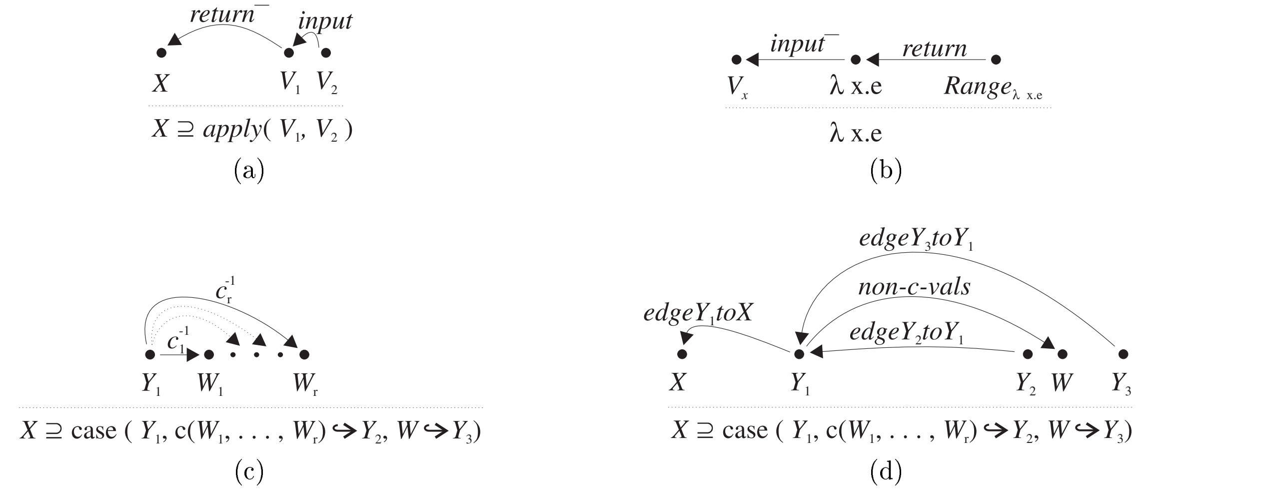


Figure 12: Edges inserted in the constructed graph to model ML set constraints.

(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 $X' \supseteq \mathit{se}$ both appear it $\mathcal{C}$, where $X' \supseteq \mathit{se}$ is in explicit form, then add $X \supseteq \mathit{se}$ 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 [8]. □

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 discuss 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.

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 $\mathit{edgeV_i toV_i}\langle V_i, V_i\rangle$.
- 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 an edge $\mathit{c\text{-}value}\langle (k), (k)\rangle$. This edge 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 $\mathit{c\text{-}value}\langle (k), (k)\rangle$ and $\mathit{Ground}\langle (k), (k)\rangle$ that indicates that (k) is known to be a ground c term.) If c is a nullary constructor, then the graph contains the edge $\mathit{Ground}\langle (k), (k)\rangle$. Otherwise, for each position j of the atomic expression (where $j = 1 \ldots r$), the graph contains the edges $c_j\langle V_j, (k)\rangle$, $\mathit{edge}(k)\mathit{toV_j}\langle (k), V_j\rangle$, and $\mathit{edgeV_j to}(k)\langle V_j, (k)\rangle$.
- 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 $\mathit{Id}\langle (k), V\rangle$ 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 $\mathit{Ground}\langle \lambda x.e, \lambda x.e\rangle$. The node $\lambda x.e$ is connected to the nodes representing the set variables V_x and $\mathit{Range}_{\lambda x.e}$ by the edges $\mathit{input}^{-}\langle \lambda x.e, V_x\rangle$ and $\mathit{return}\langle \mathit{Range}_{\lambda x.e}, \lambda x.e\rangle$. An edge $\mathit{input}^{-}\langle \lambda x.e, V_x\rangle$ indicates that values that are applied to the abstraction $\lambda x.e$ reach the variable V_x. An edge $\mathit{return}\langle \mathit{Range}_{\lambda x.e}, \lambda x.e\rangle$ indicates that $\mathit{Range}_{\lambda x.e}$ holds a superset of the values returned by the abstraction $\lambda x.e$ during program execution.
- For each constraint of the form $V \supseteq \mathit{apply}(V_1, V_2)$, the graph contains the following edges:
  - $\mathit{return}^{-}\langle V_1, V\rangle$. This edge indicates that V contains values that are returned by abstractions in V_1.
  - $\mathit{input}\langle V_2, V_1\rangle$. This edge indicates that values in V_2 are applied to abstractions in V_1.
- For each constraint of the form $V \supseteq \lambda x.e$, the graph contains an edge $\mathit{Id}\langle \lambda x.e, V\rangle$.
- For each constraint of the form $V_i \supseteq V_j$, the graph contains an edge $\mathit{Id}\langle V_j, V_i\rangle$.
- For each constraint of the form $X \supseteq \mathit{case}(Y_1, c(W_1, \ldots, W_r) \hookrightarrow Y_2,\ W \hookrightarrow Y_3)$, the graph contains the following edges:
  1. $c_i^{-1}\langle Y_1, W_i\rangle$ where $i = 1 \ldots r$
  2. $\mathit{non\text{-}c\text{-}vals}\langle y_1, W\rangle$
  3. $\mathit{edgeY_2 toY_1}\langle Y_2, Y_1\rangle$
  4. $\mathit{edgeY_3 toY_1}\langle Y_3, Y_1\rangle$
  5. $\mathit{edgeY_1 toX}\langle Y_1, X\rangle$
- For each constraint of the form $X \supseteq \mathit{ifnonempty}(Y_1, Y_2)$ the graph contains the edges $\mathit{edgeY_2 toY_1}\langle Y_2, Y_1\rangle$ and $\mathit{edgeY_1 toX}\langle Y_1, X\rangle$

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

$$\mathit{Ground} ::= \mathit{edgeV_i toV_i}\ \mathit{Rev\_Id}\ \mathit{Ground}\ \mathit{Id}\ \mathit{edgeV_i toV_i}$$

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

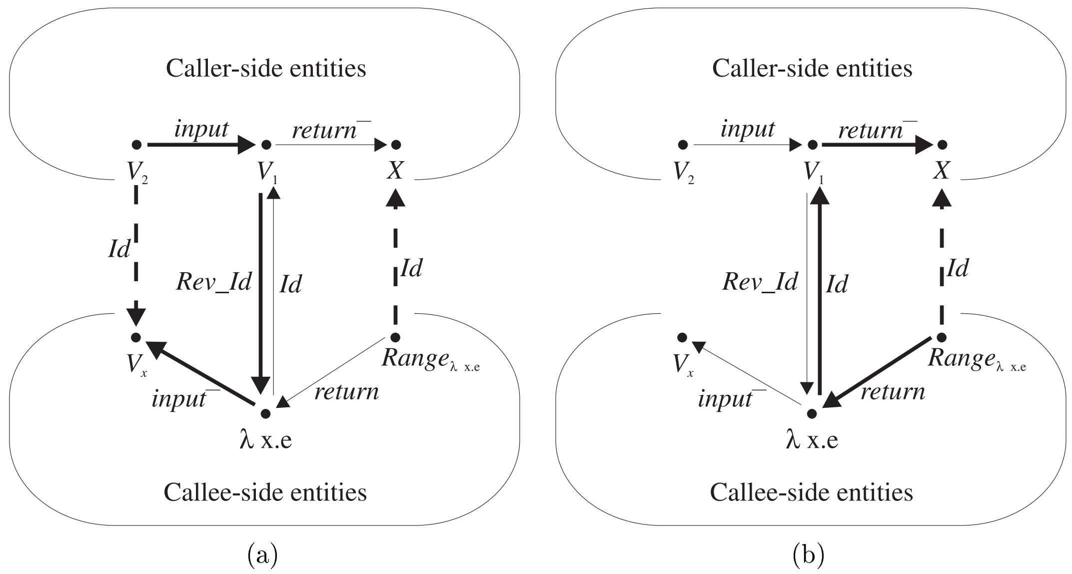


Figure 13: Graphs showing edge induction for constraints of the form $X \supseteq apply(V_1, V_2)$ and $V_1 \supseteq \lambda x.e$.

$$
\begin{aligned}
Ground ::= &\ edge(k)\,toV_1 \;\; Ground \;\; edgeV_1\,to(k) \\
&\ edge(k)\,toV_2 \;\; Ground \;\; edgeV_2\,to(k) \\
&\ \ldots \\
&\ edge(k)\,toV_r \;\; Ground \;\; edgeV_r\,to(k)
\end{aligned}
$$

In Section 3, the production $Id ::= Id \;\; Id$ 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(b).)

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:

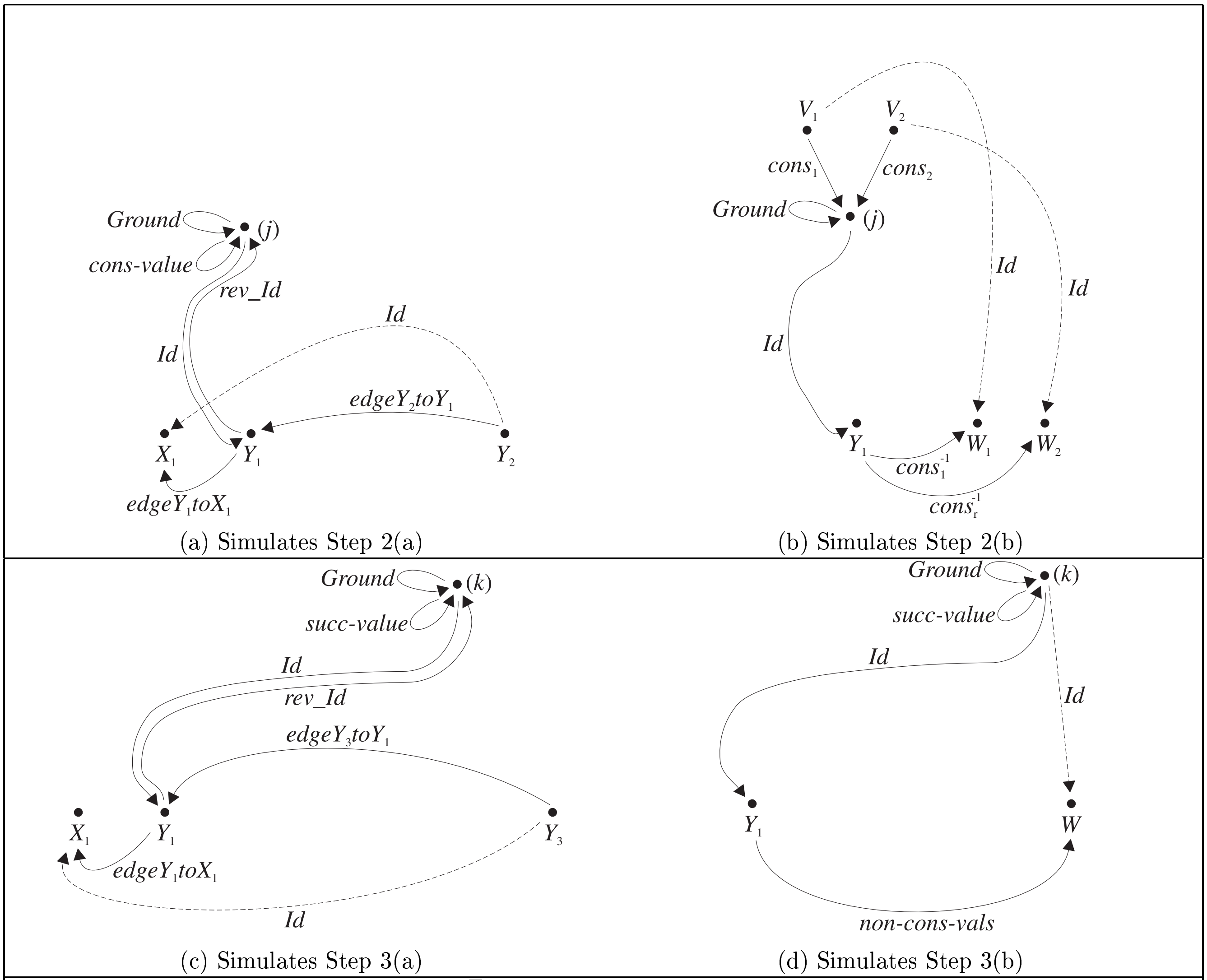
$$X \supseteq apply(V_1, V_2)$$
$$V_1 \supseteq \lambda x.e$$

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 this edge, 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 2(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:

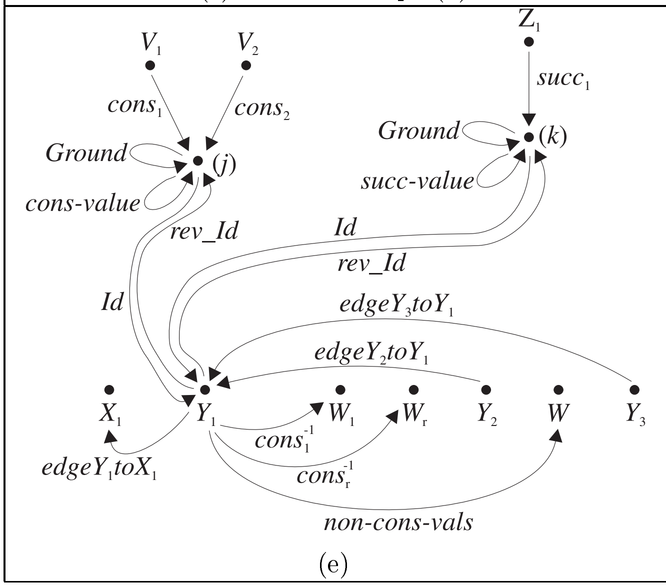


(a) Simulates Step 2(a)

(b) Simulates Step 2(b)

(c) Simulates Step 3(a)

(d) Simulates Step 3(b)



(e)

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*	::=	*edgeY*$_2$*toY*$_1$ *Rev_Id cons-value Ground cons-value Id edgeY*$_1$*toX*
(b)	*Id*	::=	*cons*$_1$ *Ground Id* $cons_1^{-1}$
	Id	::=	*cons*$_2$ *Ground Id* $cons_2^{-1}$
(c)	*Id*	::=	*edgeY*$_2$*toY*$_1$ *Ground edgeY*$_1$*toX*
(d)	*Id*	::=	*Ground succ-value Id non-cons-vals*

Part (e) shows the entire graph.

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(e) shows the graph constructed to represent the above constraints. 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 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(d).) 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

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

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

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 4.2, 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 4.2, 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

$$Id ::= edge Y_3 to Y_1\ Rev\_Id\ c\text{-}value\ Ground\ c\text{-}value\ Id\ edge Y_1 to X$$

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

$$Id ::= edge Y_3 to Y_1\ Rev\_Id\ Ground\ c\text{-}value\ Id\ edge Y_1 to X$$

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 $O(kn^2)$ edges and participate in $O(t)$ productions, or introduces a non-terminal that might appear on $O(tn)$ edges and participates in $O(kt)$ productions. Either way, if one considers k to be proportional to t, 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 a blowup. To see why, let us examine in more detail what goes wrong when the production

$$Id ::= edge Y_3 to Y_1\ Rev\_Id\ Ground\ c\text{-}value\ Id\ edge Y_1 to X$$

is normalized to the productions:

1. $Ground\text{-}c ::= Ground\ c\text{-}value$
2. $Ground\text{-}c\text{-}Id ::= Ground\text{-}c\ Id$
3. $Ground\text{-}c\text{-}Id\text{-}edge Y_1 to X ::= Ground\text{-}c\text{-}Id\ edge Y_1 to X$
4. $Edge Y_2 to Y_1\text{-}Rev\_Id ::= edge Y_2 to Y_1\ Rev\_Id$
5. $Id ::= Edge Y_2 to Y_1\text{-}Rev\_Id\ Ground\text{-}c\text{-}Id\text{-}edge Y_1 to X$

Notice that there are $O(t)$ productions of the form of the fifth production for each of k different constructor types. The problem with this normalization is with the non-terminal $edge Y_2 to Y_1\text{-}Rev\_Id$ and the fifth production. There may be $O(tn)$ edges labelled with this non-terminal, each involved in $O(kt)$ productions of the form of the fifth production above. Consider a particular edge $edge Y_2 to Y_1\text{-}Rev\_Id\langle i, j\rangle$ 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 $Ground\text{-}d'\text{-}Id\text{-}edge Y_1 to X$ 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 $Ground\text{-}d'\text{-}Id\text{-}edge Y_1 to X$. This implies that when the CFL-reachability Algorithm examines the edge $edge Y_2 to Y_1\text{-}Rev\_Id\langle i, j\rangle$ and looks at $O(tk)$ 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

$$Id ::= edge Y_3 to Y_1\ Rev\_Id\ c\text{-}value\ Ground\ c\text{-}value\ Id\ edge Y_1 to X$$

can be normalized to

1. *Ground-c* ::= *Ground c-value*
2. *Ground-c-Id* ::= *Ground-c Id*
3. *Ground-c-Id-edge*Y_1*to*X ::= *Ground-c-Id edge*Y_1*to*X
4. *Edge*Y_2*to*Y_1*-Rev_Id* ::= *edge*Y_2*to*Y_1 *Rev_Id*
5. *Edge*Y_2*to*Y_1*-Rev_Id-c* ::= *Edge*Y_2*to*Y_1*-Rev_Id c-value*
6. *Id* ::= *Edge*Y_2*to*Y_1*-Rev_Id-c Ground-c-Id-edge*Y_1*to*X

In this normalization, the nonterminal *Edge*Y_2*to*Y_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 k for a given edge of the form *Edge*Y_2*to*Y_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-edge*Y_1*to*X.

5.3.2 Normalization of the Constructed Grammar

Normalization of the context-free grammar in a constructed problem is done as in Section 4.2.1. 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 4.2.1 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* ::= *edge*Y_2*to*Y_1 *c-value Rev_Id Ground c-value Id edge*Y_1*to*X is normalized to

  Ground-c ::= *Ground c-value*
  Ground-c-Id ::= *Ground-c Id*
  Ground-c-Id-edgeY_1*to*X ::= *Ground-c-Id edge*Y_1*to*X
  EdgeY_2*to*Y_1*-Rev_Id* ::= *edge*Y_2*to*Y_1 *Rev_Id*
  EdgeY_2*to*Y_1*-Rev_Id*-c ::= *edge*Y_2*to*Y_1*-Rev_Id c-value*
  Id ::= *Edge*Y_2*to*Y_1*-Rev_Id*-c *Ground-c-Id-edge*Y_1*to*X

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

$$Ground\text{-}c' ::= Ground\ c'\text{-}value$$
$$Ground\text{-}c'\text{-}Id ::= Ground\text{-}c'\ Id$$
$$Id ::= Ground\text{-}c'\text{-}Id\ non\text{-}c\text{-}values$$

- $Id ::= edgeY_2toY_1\ Ground\ edgeY_1toX$ 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 **??**. 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 Related Work and Concluding Remarks

Heintze and McAllester have obtained results that have a bearing on the "$O(n^3)$ program-analysis bottleneck" by considering the problem of determining membership for languages defined by 2-way nondeterministic pushdown automata (2NPDA-recognition) [11]. The best known algorithm 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 [11] 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 [20, 10].

A variety of work exists that has applied graph reachability (of various forms) to analysis of imperative programs. Kou [19] and Hecht [6] 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 [18]. Cooper and Kennedy used reachability to give efficient algorithms for interprocedural side-effect analysis [2] and alias analysis [3].

The first uses of CFL-reachability for program analysis were in 1988, in Callahan's work on flow-sensitive side-effect analysis [1] and Horwitz et al.'s work on interprocedural slicing [12, 13]. 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 [28]; 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 et al. to be of utility for a wide variety of interprocedural program-analysis problems [26] (see also [24]). These ideas were elaborated on in a sequence of papers [15, 14, 25], and also applied to shape analysis of functional programs [22].

All of these papers use only very limited forms of CFL-reachability, namely variations on Dyck-language reachability. The second author became aware of the connection to the more general concept of CFL-reachability sometime in the fall of 1994. (Of the papers mentioned above, only [22] and [24] mention CFL-reachability explicitly and reference Yannakakis's paper [29].) 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.

It is also interesting to note another fact about CFL-reachability: every CFL-reachability problem can be stated as a *chain program* in DATALOG [29]; edges are represented as facts, and productions are encoded as Horn clauses. In fact, the CFL-reachability Algorithm presented here 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 [21]). Many parts of a constructed CFL-reachability problem are more easily expressed in a DATALOG program. In particular, the addition of reverse edges, and the tracking of ground information is easy to express. The program would not necessarily be a chain program, but it would still run in cubic time. Of course, this result also implies that set-constraints may be solved using an equivalent DATALOG program.

References

[1] 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.

[2] 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.

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

[4] 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.

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

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/fse95.ps).

[15] 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. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/fse94.ps).

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

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

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

			Work performed for a given edge	
Form of label	# of edges	Productions with label on the right-hand side	# examined productions	Total # of attempts to add an edge
Id	$O(n^2)$	$Id ::= Id \quad Id$ $Ground\text{-}Id ::= Ground \quad Id$	1 1	$O(n)$ 1
Rev_Id	$O(n^2)$	$Rev\_Id ::= Rev\_Id \quad Rev\_Id$ $G ::= Rev\_Id \quad Ground\text{-}Id$ $Rev\_Id\text{-}Ground ::= Rev\_Id \quad Ground$	1 1 1	$O(n)$ $O(n)$ 1
$Ground$	$O(v)$	$Ground\text{-}edgeV_i\,to(k) ::= Ground \quad edgeV_i\,to(k)$ $Ground\text{-}Id ::= Ground \quad Id$ $Rev\_Id\text{-}Ground ::= Rev\_Id \quad Ground$	$O(v)$ 1 1	$O(v)$ $O(n)$ $O(n)$
c_i	$O(tr)$	$Id ::= c_i \quad Ground\text{-}Id\text{-}c_i^{-1}$	1	$O(t)$
rev_c_i	$O(tr)$	$Rev\_Id ::= Rev\_c_i^{-1}\text{-}Rev\_Id\text{-}Ground \quad rev\_c_i$	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	$O(n)$
$edgeV_i\,to(k)$	$O(tr)$	$Ground\text{-}edgeV_i\,to(k) ::= Ground \quad edgeV_i\,to(k)$	1	1
$edge(k)toV_i$	$O(tr)$	$MarkV_i\,GrAt(k) ::= edge(k)toV_i \quad Ground\text{-}edgeV_i\,to(k)$	1	1
$edgeV_j\,toV_j$	$O(v)$	$G\text{-}edgeV_j\,toV_j ::= G \quad edgeV_j\,toV_j$ $Ground ::= edgeV_j\,toV_j \quad G\text{-}edgeV_j\,toV_j$	1 1	$O(n)$ 1
G	$O(n^2)$	$G\text{-}edgeV_j\,toV_j ::= G \quad edgeV_j\,toV_j$	$O(v)$	1
$Ground\text{-}Id$	$O(n^2)$	$G ::= Rev\_Id \quad Ground\text{-}Id$ $Ground\text{-}Id\text{-}c_i^{-1} ::= Ground\text{-}Id \quad c_i^{-1}$	1 $O(tr)$	$O(n)$ $O(t)$
$Rev\_Id\text{-}Ground$	$O(n^2)$	$Rev\_c_i^{-1}\text{-}Rev\_Id\text{-}Ground ::= rev\_c_i^{-1} \quad Rev\_Id\text{-}Ground$	$O(tr)$	$O(t)$
$Ground\text{-}Id\text{-}c_i^{-1}$	$O(tn)$	$Id ::= c_i \quad Ground\text{-}Id\text{-}c_i^{-1}$	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
$MarkV_{a_j}\,GrAt(k)$	$O(tr)$	$MarkV_{a_1}\text{-}V_{a_j}\,GrAt(k) ::= MarkV_{a_1}\text{-}V_{a_{j-1}}\,GrAt(k) \quad MarkV_{a_j}\,GrAt(k)$	r	r
$MarkV_{a_1}\text{-}V_{a_{j-1}}\,GrAt(k)$	$O(tr)$	$MarkV_{a_1}\text{-}V_{a_j}\,GrAt(k) ::= MarkV_{a_1}\text{-}V_{a_{j-1}}\,GrAt(k) \quad MarkV_{a_j}\,GrAt(k)$	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
$Ground\text{-}edgeV_i\,to(k)$	$O(tr)$	$MarkV_i\,GrAt(k) ::= edge(k)toV_i \quad Ground\text{-}edgeV_i\,to(k)$	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	Work performed for a given edge: Total # of attempts to add an edge
Id	$O(n^2)$	*Ground-c-Id* ::= *Ground-c* *Id*	$O(k)$	$O(n)$
		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	$O(v)$	*Ground-edgeV$_i$toV$_j$* ::= *Ground* *edgeV$_i$toV$_j$*	$O(v^2)$	$O(v)$
		Ground-c ::= *Ground* *c-value*	k	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$_j$*	k	$O(t)$
Ground-Id	$O(n^2)$	*c_i-Ground-Id* ::= c_i *Ground-Id*	$O(tr)$	$O(tr)$
Ground-edgeV$_j$toV$_i$	$O(t)$	*Id* ::= *edgeV$_k$toV$_j$* *Ground-edgeV$_j$toV$_i$*	t	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	$O(n^2)$	*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	$O(n^2)$	*Id* ::= *return-Id* *return$^-$*	1	$O(t)$
Ground-c-Id	$O(tv)$	*Ground-c-Id-edgeV$_i$toV$_j$* ::= *Ground-c-Id* *edgeV$_i$toV$_j$*	k	$O(t)$
		Id ::= *Ground-c-Id* *non-c'-values*	$O(t)$	$O(t)$
c-value	$O(k)$	*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	$O(v)$
non-c'-values	$O(t)$	*Id* ::= *Ground-c-Id* *non-c'-values*	$O(t)$	$O(t)$
edgeV$_i$toV$_j$-Rev_Id	$O(trn)$	*edgeV$_i$toV$_j$-Rev_Id-c* ::= *edgeV$_i$toV$_j$-Rev_Id* *c-value*	$O(k)$	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.

[24] 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.

[25] 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. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/popl95.ps).

[26] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report TR 94-14, Datalogisk Institut, University of Copenhagen, 1994. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/diku-tr94-14.ps).

[27] 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.

[28] 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.

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