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

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.
- 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 [24, 13]. Set-constraints lead to natural formulations of shape analysis [17, 26]. 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

[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

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.

For both 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 3.3 and 4.2, 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 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. Finally, 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. Section 5 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.

$A ::= B\ C$	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.

Definition 2.1 Let CF be a context-free grammar over an alphabet of terminal symbols T and non-terminal symbols N. 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[1]. This can be done by introducing new non-terminal symbols. Thus, a production such as

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

   might be converted into these productions:

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

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

[1] The normal form used is similar to Chomsky Normal Form.

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. 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 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
            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

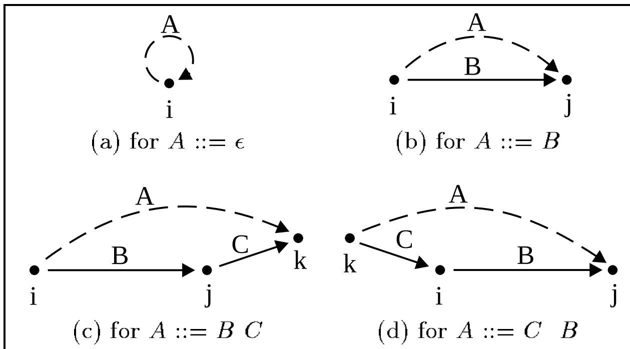

Figure 1: Edge induction in the CFL-reachability Algorithm (Algorithm 2.1). The figures show how a production of the context-free grammar causes the algorithm to add, or *induce*, an edge in the graph (dashed lines show induced edges).

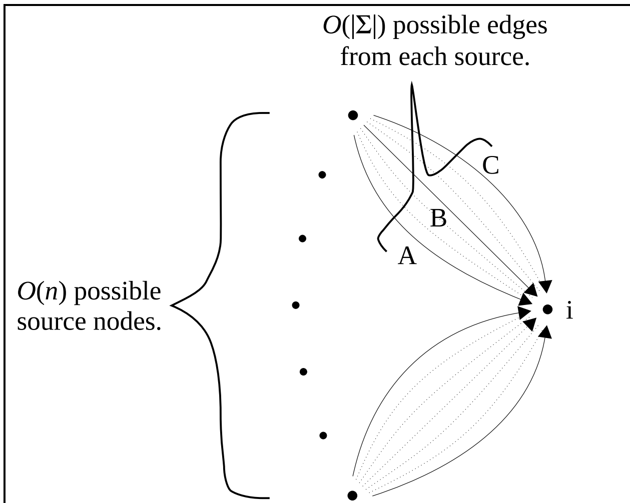

Figure 2: In a graph from a CFL-reachability problem, the number of edges into any given node is bounded by $O(|\Sigma|n)$, where Σ is the alphabet of the grammar, and n is the number of nodes in the graph.

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 the class of set constraints considered in this paper. (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 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:

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

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:

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

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 $Y \supseteq c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$ 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 $Y \supseteq c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$ 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 $O(v^2 t)$. 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 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_a1, V_a2, ..., V_ar)
    then
        for each constraint of the form Y⊇c_i^-1(X) in C do
            if Y⊇V_ai is not in C then
                Insert Y⊇V_ai into C and W
            fi
        od
        for each constraint of the form Y⊇X in C do
            if Y⊇c(V_a1, V_a2, ..., V_ar) is not in C then
                Insert Y⊇c(V_a1, V_a2, ..., V_ar)
                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_a1, V_a2, ..., V_ar) in C such that
        V_a1, ..., V_ar are all ground do
            if X⊇c(V_a1, V_a2, ..., V_ar) is not in C then
                Insert X⊇c(V_a1, V_a2, ..., V_ar)
                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.

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

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

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

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

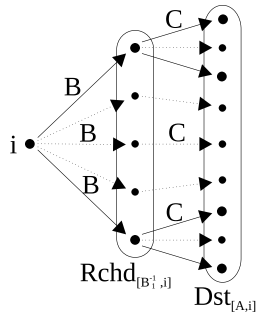

Figure 3: Use of $Dst_{[A,i]}$ and $Rchd_{[B_1^{-1},i]}$ to encode production $A ::= B\ C$. The variable $Rchd_{[B_1^{-1},i]}$ represents the set of nodes reached by following B-edges from i. The variable $Dst_{[A,i]}$ represents the set of nodes to which there should be an A-edge from node i.

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)? Let $Dst_{[A,i]}$ be a unique set variable for holding the set of nodes that answer this question. To specify that there is an A-edge from node i to the nodes in $Dst_{[A,i]}$, we generate the constraint $X_i \supseteq A(Dst_{[A,i]})$.

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. We introduce another unique variable $Rchd_{[B_1^{-1},i]}$ to hold the set of nodes reached by following B edges from node i. In our representation of the graph, edges are represented as constructors, and "following an edge" can be encoded using projection: in particular, we generate the constraint $Rchd_{[B_1^{-1},i]} \supseteq B_1^{-1}(X_i)$.

Finally, the set of nodes to which we want to add an A-edge from i is found by following C edges from the nodes in $Rchd_{[B_1^{-1},i]}$, and so we generate the constraint
$Dst_{[A,i]} \supseteq C_1^{-1}(Rchd_{[B_1^{-1},i]})$.

All told, we generate three 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. I.e., 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 generate 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.

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.

It is also easily shown that the construction given in this section can be carried out in log-space. 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].

3.3 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
$Dst_{[A,i]} \supseteq B_1^{-1}(Rchd_{[C_1^{-1},i]})$	$s^3 n$	$Rchd_{[C_1^{-1},i]} \supseteq B(X_j)$	n	$Dst_{[A,i]} \supseteq X_j$	$s^3 n^2$
		$Rchd_{[C_1^{-1},i]} \supseteq B(Dst_{[B,j]})$	n	$Dst_{[A,i]} \supseteq Dst_{[B,j]}$	$s^3 n^2$
$Dst_{[A,i]} \supseteq B_1^{-1}(X_i)$	$s^2 n$	$X_i \supseteq B(X_j)$	n	$Dst_{[A,i]} \supseteq X_j$	$s^2 n^2$
		$X_i \supseteq B(Dst_{[B,i]})$	1	$Dst_{[A,i]} \supseteq Dst_{[B,i]}$	$s^2 n$
$Rchd_{[A_1^{-1},i]} \supseteq X_j$	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	$Rchd_{[A_1^{-1},i]} \supseteq B(X_k)$	$s^2 n^3$
		$X_j \supseteq B(Dst_{[B,k]})$	sn	$Rchd_{[A_1^{-1},i]} \supseteq B(Dst_{[B,k]})$	$s^2 n^3$
$Rchd_{[A_1^{-1},i]} \supseteq Dst_{[A,i]}$	sn	$Dst_{[A,i]} \supseteq B(X_j)$	sn	$Rchd_{[A_1^{-1},i]} \supseteq B(X_j)$	$s^2 n^2$
		$Dst_{[A,i]} \supseteq B(Dst_{[B,j]})$	sn	$Rchd_{[A_1^{-1},i]} \supseteq B(Dst_{[B,j]})$	$s^2 n^2$
$Dst_{[A,i]} \supseteq X_j$	sn^2	$X_j \supseteq node_j$	1	$Dst_{[A,i]} \supseteq node_j$	sn^2
		$X_j \supseteq B(X_k)$	sn	$Dst_{[A,i]} \supseteq B(X_k)$	$s^2 n^3$
		$X_j \supseteq B(Dst_{[B,j]})$	s	$Dst_{[A,i]} \supseteq B(Dst_{[B,j]})$	$s^2 n^2$
$Dst_{[A,i]} \supseteq Dst_{[B,j]}$	$s^2 n^2$	$Dst_{[B,j]} \supseteq C(X_k)$	sn	$Dst_{[A,i]} \supseteq C(X_k)$	$s^3 n^3$
		$Dst_{[B,j]} \supseteq C(Dst_{[C,j]})$	s	$Dst_{[A,i]} \supseteq C(Dst_{[C,j]})$	$s^3 n^2$

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.

$X_i \supseteq A(X_j)$ (Encode an A edge from i to j)

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

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

Table 2 summarizes the reductions that may take place and the cost of the work performed. Overall, the dominant term is $s^3 n^3$, 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:

Example 4.1 Consider the following constraints:

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

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

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

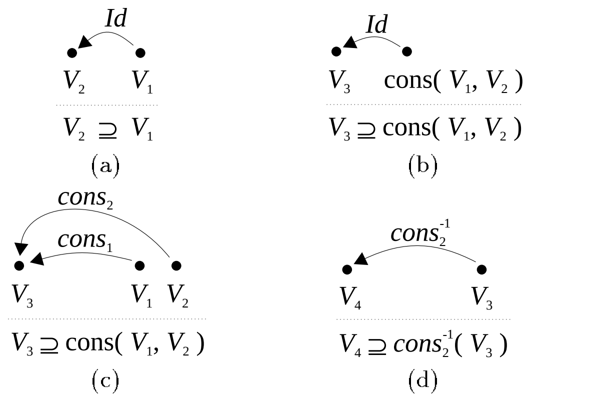

Figure 4: Edges inserted in the constructed graph to model constraints.

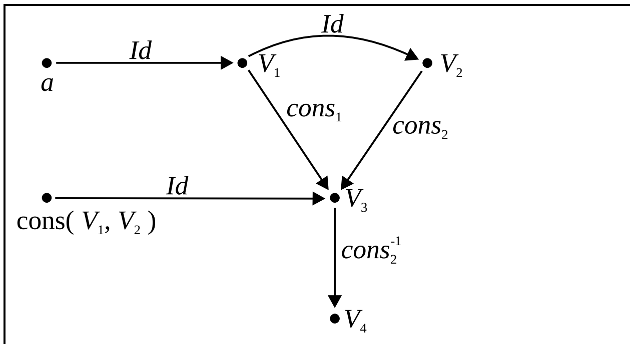

Figure 5: The graph built to encode the constraints in Example 4.1.

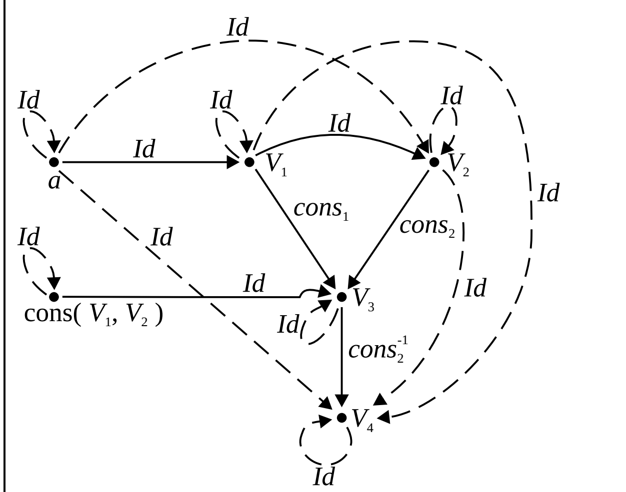

Figure 6: The graph for Example 4.1 after the CFL-reachability Algorithm has been run. Dashed lines represent edges inserted by the algorithm.

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 is constructed as follows:

- For each set variable V_i, the graph contains a node labelled V_i.
- For each atomic expression $cons(V_i, V_j)$ used in the constraints, the graph contains a node labelled $cons(V_i, V_j)$.
- 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 $V_k \supseteq cons(V_i, V_j)$, the graph contains an edge $Id\langle cons(V_i, V_j), V_k\rangle$. This indicates that the atomic expression $cons(V_i, V_j)$ reaches V_k. (See Figure 4(b).)
- For each constraint of the form $V_k \supseteq cons(V_i, V_j)$, the graph contains the edges $cons_1\langle V_i, V_k\rangle$ and $cons_2\langle V_j, V_k\rangle$. An edge $cons_m\langle V_i, V_k\rangle$ indicates that the values that reach node i are wrapped in the m^{th} position of a *cons* value at node k. (See Figure 4(c).)
- 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:

$$
\begin{aligned}
Id &::= cons_1 \; Id \; cons_1^{-1} \\
Id &::= cons_2 \; Id \; cons_2^{-1} \\
Id &::= \epsilon
\end{aligned}
$$

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 $V_4 \supseteq cons_2^{-1}(V_3)$. Similarly, in the constructed graph, the CFL-reachability algorithm adds the edge $Id\langle V_2, V_4\rangle$ because of the edges $cons_2\langle V_2, V_3\rangle$, $Id\langle V_3, V_3\rangle$, and $cons_2^{-1}\langle V_3, V_4\rangle$ (see Figure 6). ($Id\langle V_3, V_3\rangle$ is added to the graph because of production $Id ::= \epsilon$.)

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

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

In Example 4.1, the SC-Reduction Algorithm adds the constraint $V_2 \supseteq a$ because of the constraints $V_2 \supseteq V_1$ and $V_2 \supseteq a$. Similarly, the CFL-reachability algorithm adds the edge

$Id\langle a, V_2\rangle$ because of the edges $Id\langle a, V_1\rangle$ 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 $Id\langle a, V_1\rangle$, $Id\langle a, V_2\rangle$, and $Id\langle a, V_4\rangle$ 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:

$V_1 \Rightarrow a$ $\qquad V_2 \Rightarrow a$ $\qquad V_4 \Rightarrow a$

The edge $Id\langle cons(V_1, V_2), V_3\rangle$ 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:

$$\begin{array}{l} V_1 \supseteq a \\ V_3 \supseteq cons(V_1, V_2) \\ V_4 \supseteq V_3 \\ V_5 \supseteq cons_1^{-1}(V_4) \\ \quad \vdots \end{array}$$

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

- Add the constraint $V_4 \supseteq cons(V_1, V_2)$ (because of constraints $V_3 \supseteq cons(V_1, V_2)$ and $V_4 \supseteq V_3$).
- Add the constraint $V_5 \supseteq V_1$ (because of the new constraint $V_4 \supseteq cons(V_1, V_2)$ and the constraint $V_5 \supseteq cons_1^{-1}(V_4)$).
- Add the constraint $V_5 \supseteq a$ (because of the new constraint $V_5 \supseteq V_1$ and the constraint $V_1 \supseteq a$).
- Output the production $V_5 \Rightarrow a$ (because of the new constraint $V_5 \supseteq a$).

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 $V_5 \Rightarrow a$. (The SC-Reduction Algorithm may still generate $V_5 \Rightarrow a$ as a result of reducing other constraints in $\mathcal{C}$; but it would not generate $V_5 \Rightarrow a$ 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.

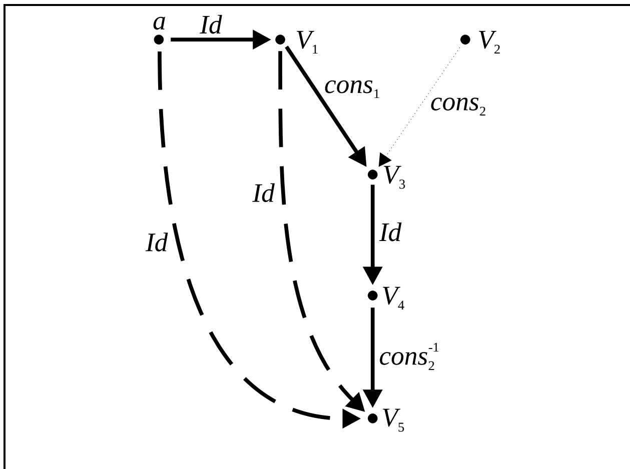

Figure 7: The edge $Id\langle V_1, V_5\rangle$ should be induced if and only if $cons(V_1, V_2)$ is ground. If the edge $Id\langle V_1, V_5\rangle$ is added when $cons(V_1, V_2)$ is not ground, it may incorrectly cause the edge $Id\langle a, V_5\rangle$ to be added, and the production $V_5 \Rightarrow a$ to be output.

The CFL-reachability algorithm will add the edge $Id\langle V_1, V_5\rangle$ 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 $cons_1\langle V_1, V_3\rangle$, $Id\langle V_3, V_4\rangle$, and $cons_1^{-1}\langle V_4, V_5\rangle$. Adding edge $Id\langle V_1, V_5\rangle$ 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 $Id\langle V_1, V_5\rangle$ 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_5 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. □

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 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.

We now illustrate the major elements of the construction by means of Example 4.2. In Example 4.2, we want the graph to contain the cyclic edge $MarkV_1GrAtV_3\langle V_3, V_3\rangle$

("Mark V_1 ground at V_3") if and only if V_1 is ground. Similarly, we want the cyclic edge $MarkV_2\,GrAtV_3\langle V_3, V_3\rangle$ if and only if V_2 is ground. In place of the production $Id ::= cons_1\ Id\ cons_1^{-1}$, we use the following production:

$$Id ::= cons_1\ \ MarkV_1\,GrAtV_3\ \ MarkV_2\,GrAtV_3\ \ Id\ \ cons_1^{-1}$$

With this production, the CFL-reachability Algorithm will add the edge $Id\langle V_1, V_5\rangle$ if and only if the edges $MarkV_1\,GrAtV_3\langle V_3, V_3\rangle$ and $MarkV_2\,GrAtV_3\langle V_3, V_3\rangle$ exist (i.e., if and only if V_1 and V_2 are ground); see Figure 8(c). In essence, these productions transfer knowledge about groundness at V_1 and V_2 to knowledge about groundness (of V_1 and V_2) at V_3.

We need still more edges and productions to ensure that the CFL-reachability Algorithm will induce the edges $MarkV_1\,GrAtV_3\langle V_3, V_3\rangle$ and $MarkV_2\,GrAtV_3\langle V_3, V_3\rangle$ when appropriate. In particular, we introduce a new kind of edge label, "*Ground*", which will be used to indicate that a variable is ground: edge $Ground\langle V_i, V_i\rangle$ indicates that variable V_i is known to be ground. In Figure 7, the edges $Ground\langle V_1, V_1\rangle$ and $Ground\langle V_3, V_3\rangle$ will be added to the graph if and only if V_1 and V_3 are ground, respectively.

We also introduce the following edges during the construction of the original graph:

$$EdgeV_3\,toV_1\langle V_3, V_1\rangle \qquad EdgeV_3\,toV_2\langle V_3, V_2\rangle$$
$$EdgeV_1\,toV_3\langle V_1, V_3\rangle \qquad EdgeV_2\,toV_3\langle V_2, V_3\rangle$$

These edges simply connect nodes V_1, V_2, and V_3, and allow us to introduce the following productions:

$$MarkV_1\,GrAtV_3 ::= EdgeV_3\,toV_1\ \ Ground\ \ EdgeV_1\,toV_3$$
$$MarkV_2\,GrAtV_3 ::= EdgeV_3\,toV_2\ \ Ground\ \ EdgeV_1\,toV_3$$

With these productions and the edges used in them, the CFL-reachability Algorithm will induce the edges $MarkV_1\,GrAtV_3\langle V_3, V_3\rangle$ and $MarkV_2\,GrAtV_3\langle V_3, V_3\rangle$ iff the respective edges $Ground\langle V_1, V_1\rangle$ and $Ground\langle V_2, V_2\rangle$ exist. See Figure 8(a-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 constraint of the form $V_j \supseteq a$, we generate the edge $Ground\langle V_j, V_j\rangle$, because a nullary constructor is always ground.

Other *Ground* edges are induced during the running of the CFL-reachability Algorithm. In Example 4.2, the variable V_3 is ground if V_1 and V_2 are both ground. This is captured by the following production:

$$Ground ::= MarkV_1\,GrAtV_3\ \ MarkV_2\,GrAtV_3$$

With this production, the CFL-reachability Algorithm will add the edge $Ground\langle V_3, V_3\rangle$ if and only if the edges $MarkV_1\,GrAtV_3\langle V_3, V_3\rangle$ and $MarkV_2\,GrAtV_3\langle V_3, V_3\rangle$ are both in the graph.

There is one last situation we must take into account: Suppose that in Example 4.2 the set variable V_3 is known to be ground, and consider the constraint $V_4 \supseteq V_3$; this implies that the variable V_4 is also ground. In the graph constructed for this situation, we have the edges $Ground\langle V_3, V_3\rangle$ and $Id\langle V_3, V_4\rangle$, and we want the edge $Ground\langle V_4, V_4\rangle$ to be added. In effect, we want the *Ground* information at V_3 to be propagated along the *Id* edge. To accomplish this, we introduce the edges $Rev\_Id\langle V_4, V_3\rangle$ and $EdgeV_4\,toV_4\langle V_4, V_4\rangle$, and the following production:

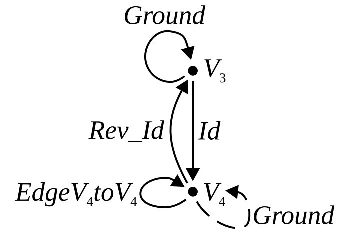

Figure 9: Propagation of *Ground* edges from V_3 to V_4. This is accomplished using the production "$Ground ::= EdgeV_4\,toV_4\ \ Rev\_Id\ \ Ground\ \ Id\ \ EdgeV_4\,toV_4$".

$$Ground ::= EdgeV_4\,toV_4\ \ Rev\_Id\ \ Ground\ \ Id\ \ EdgeV_4\,toV_4$$

With this production, the CFL-reachability Algorithm will add the edge $Ground\langle V_4, V_4\rangle$ to the graph (see Figure 9).

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

$$Id ::= cons_1\ \ MarkV_1\,GrAtV_3\ \ MarkV_2\,GrAtV_3\ \ Id\ \ cons_1^{-1}$$

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

$$Rev\_Id ::= Rev\_cons_1^{-1}\ \ Rev\_Id\ \ MarkV_2\,GrAtV_3\ \ MarkV_1\,GrAtV_3\ \ Rev\_cons_1$$

Figure 10 illustrates the use of this reverse production.

For this production to work, we need additional reverse edges: For every edge $cons_1\langle V_i, V_j\rangle$ in the graph, we want the edge $Rev\_cons_1\langle V_j, V_i\rangle$ to be in the graph; for every edge $cons_1^{-1}\langle V_i, V_j\rangle$, we want the edge $Rev\_cons_1^{-1}\langle V_j, V_i\rangle$ to be in the graph. Fortunately, these reverse edges can be added when we construct the graph. They do not require the introduction of new productions. Notice also that an edge like $MarkV_2\,GrAtV_3$ is always cyclic. Hence, it can serve as its own reverse edge and so we do not need an edge labelled $Rev\_MarkV_2\,GrAtV_3$.

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

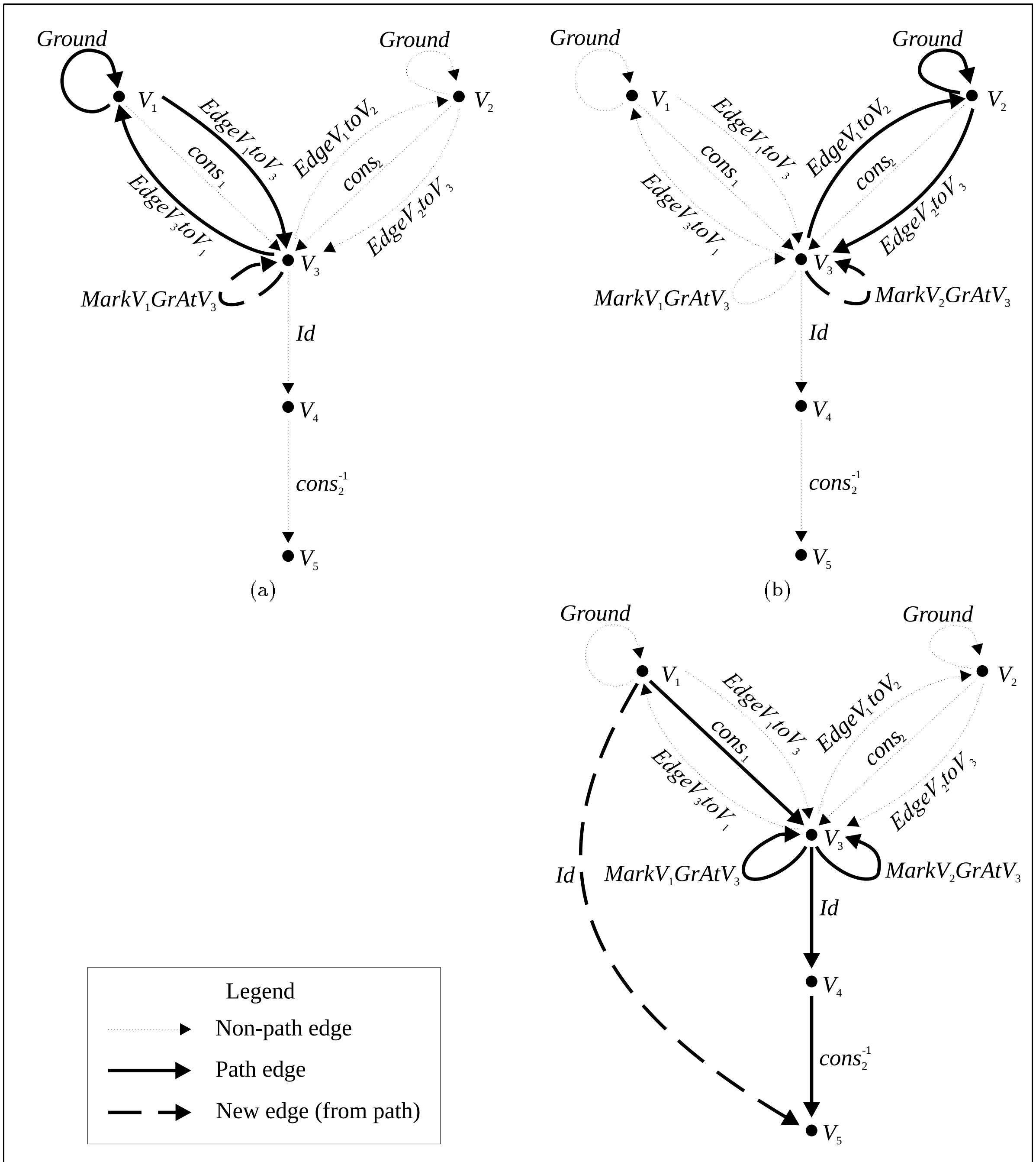

Figure 8: Use of *Ground* edges in producing *Id* edges.

$$Ground ::= EdgeV_i toV_i \quad Rev\_Id \quad Ground \quad Id \quad EdgeV_i toV_i$$

2. For each atomic expression $c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$ used in the set constraints the graph contains a node named $c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$.

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

4. For each constraint of the form $V_{a_0} \supseteq c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$, the graph contains an edge $Id\langle c(V_{a_1}, V_{a_2}, \ldots, V_{a_r}), a_0\rangle$.

This edge indicates that the value $c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$ reaches the variable V_{a_0}. For each position j of the atomic expression $(c(V_{a_1}, V_{a_2}, \ldots, V_{a_r}))$ used in this constraint (where $j = 1 \ldots r$), the graph contains the following edges:

(a) $c_j\langle V_{a_j}, V_{a_0}\rangle$

(b) $Rev\_c_j\langle V_{a_0}, V_{a_j}\rangle$

(c) $EdgeV_{a_j}toV_{a_0}\langle V_{a_j}, V_{a_0}\rangle$

(d) $EdgeV_{a_0}toV_{a_j}\langle V_{a_0}, V_{a_j}\rangle$

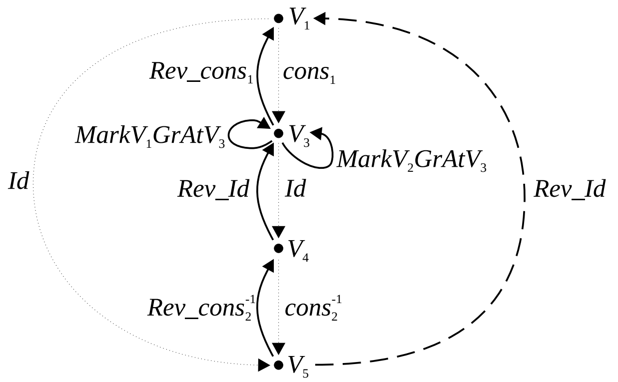

Figure 10: The production $Rev\_Id ::= Rev\_cons_1^{-1}\ \ Rev\_Id\ \ MarkV_2\,GrAtV_3\ \ MarkV_1\,GrAtV_3\ \ Rev\_cons_1$ causes the CFL-reachability Algorithm to produce *Rev_Id* edges. (This production is the counterpart of the production $Id ::= cons_1\ \ MarkV_1\,GrAtV_3\ \ MarkV_2\,GrAtV_3\ \ Id\ \ cons_1^{-1}$.)

For each position j of the atomic expression in this constraint, the context-free grammar contains the following productions:

(a) $MarkV_{a_j}\,GrAtV_{a_0} ::= EdgeV_{a_0}toV_{a_j}\ \ Ground\ \ EdgeV_{a_j}toV_{a_0}$

(b) $Id ::= c_j\ \ MarkV_{a_1}\,GrAtV_{a_0}\ \ MarkV_{a_2}\,GrAtV_{a_0} \ldots MarkV_{a_r}\,GrAtV_{a_0}\ \ Id\ \ c_j^{-1}$

(c) $Rev\_Id ::= Rev\_c_j\ \ Rev\_Id\ \ MarkV_{a_r}\,GrAtV_{a_0} \ldots MarkV_{a_1}\,GrAtV_{a_0}\ \ Rev\_c_j$

(d) $Ground ::= MarkV_{a_1}\,GrAtV_{a_0} \ldots MarkV_{a_r}\,GrAtV_{a_0}$

5. For each constraint of the form $V_i \supseteq c_k^{-1}(V_j)$, the graph contains an edge $c_k^{-1}\langle V_j, V_i\rangle$.

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 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:

$$Ground ::= EdgeV_jtoV_j\ \ RevId\ \ Ground\ \ Id\ \ EdgeV_jtoV_j$$

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

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

Figure 11 depicts this normalization. Note that edges labelled *Id* and *Rev_Id* may be ubiquitous; they may occur anywhere in the graph. This means that the CFL-reachability Algorithm may use the above productions and put edges labelled *Ground-Id* and G anywhere in the graph. However, for any given V_j, there is only one edge labelled $EdgeV_jtoV_j$ in the graph; this is the edge $EdgeV_jtoV_j\langle V_j, V_j\rangle$. This means that for a fixed V_j, if the CFL-reachability Algorithm adds an edge $G\text{-}EdgeV_jtoV_j\langle V_i, V_k\rangle$, then it must use $EdgeV_jtoV_j\langle V_j, V_j\rangle$ to do so, and $k = j$. That is, all edges

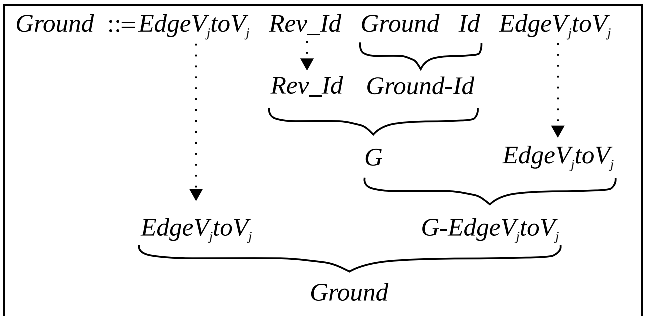

Figure 11: Normalization of the production $Ground ::= EdgeV_jtoV_j \;\; RevId \;\; Ground \;\; Id \;\; EdgeV_jtoV_j$.

labelled $G\text{-}EdgeV_jtoV_j$ must have node V_j as their destination, although they may have any node as their source. This in turn implies that for a fixed node V_j, the number of incoming edges of the form $G\text{-}EdgeV_jtoV_j\langle V_i, V_j\rangle$ is bounded by $O(n)$, and the number of outgoing edges of the form $G\text{-}EdgeV_ktoV_k\langle V_j, V_k\rangle$ is bounded by $O(n)$. Also, of all the edges $G\text{-}EdgeV_jtoV_j\langle V_i, V_j\rangle$, only one, $G\text{-}EdgeV_jtoV_j\langle V_j, V_j\rangle$, can be combined with $EdgeV_jtoV_j\langle V_j, V_j\rangle$ to generate $Ground\langle V_j, V_j\rangle$.

Now we consider the following prototypical production:

$$MarkV_iGrAtV_j ::= EdgeV_jtoV_i \quad Ground \quad EdgeV_itoV_j$$

There are $O(tr)$ productions of this form, one for each position of each atomic expression used in each constraint. It is normalized to the following productions:

$$\begin{array}{rcl} MarkV_iGrAtV_j & ::= & EdgeV_itoV_j \\ & & Ground\text{-}EdgeV_itoV_j \\ Ground\text{-}EdgeV_itoV_j & ::= & Ground \quad EdgeV_itoV_j \end{array}$$

$Ground$ edges are always cyclic, and for fixed i and j there is only one edge labelled $EdgeV_itoV_j$. This means that for fixed i and j, the CFL-reachability Algorithm will introduce at most one edge labelled $Ground\text{-}EdgeV_itoV_j$. Also, the Algorithm will introduce at most one edge labelled $MarkV_iGrAtV_j$, and this edge is cyclic.

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

$$\begin{array}{rcl} Id & ::= & c_i \quad MarkV_{a_1}GrAtV_{a_0} \quad MarkV_{a_2}GrAtV_{a_0} \\ & & MarkV_{a_3}GrAtV_{a_0} \ldots MarkV_{a_r}GrAtV_{a_0} \\ & & Id \quad c_i^{-1} \end{array}$$

This production is used to encode the second reduction step of the SC-Reduction Algorithm for a constraint of the form $V_{a_0} \supseteq c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$. The string

$$MarkV_{a_1}GrAtV_{a_0} \quad MarkV_{a_2}GrAtV_{a_0} \ldots MarkV_{a_r}GrAtV_{a_0}$$

in the right-hand side of this production accounts for whether or not the atomic expression $c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$ is ground. Thus, when we introduce non-terminals to normalize this part of the production, we must make sure that they are unique for the constraint; otherwise, confusion may occur with labels representing atomic expressions in other constraints. For example, suppose that the atomic expression $c(V_{b_1}, V_{b_2}, \ldots, V_{b_r})$ were also found at V_{a_0} and $b_1 = a_1$ and $b_r = a_r$. Then we do not want an edge that indicates that V_{a_1} thru V_{a_r} of $c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$ are ground to also (possibly incorrectly) indicate that V_{b_1} thru V_{b_r} are ground.

To avoid this, we assume that each constraint has been assigned a unique index. In the following productions, the superscript (k) on the introduced non-terminals refers to the index of the constraint that is encoded by the above production. The following productions are introduced:

$$\begin{array}{rcl} Id & ::= & c_i\text{-}MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} \\ & & Id\text{-}c_i^{-1} \\ c_i\text{-}MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} & ::= & c_i \quad MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} \\ Id\text{-}c_i^{-1} & ::= & Id \quad c_i^{-1} \\ MarkV_{a_1}\text{-}V_{a_2}GrAtV_{a_0}^{(k)} & ::= & MarkV_{a_1}GrAtV_{a_0} \\ & & MarkV_{a_2}GrAtV_{a_0} \\ MarkV_{a_1}\text{-}V_{a_3}GrAtV_{a_0}^{(k)} & ::= & MarkV_{a_1}\text{-}V_{a_2}GrAtV_{a_0}^{(k)} \\ & & MarkV_3GrAtV_0 \\ & \vdots & \\ MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} & ::= & MarkV_{a_1}\text{-}V_{a_{r-1}}GrAtV_{a_0}^{(k)} \\ & & MarkV_{a_r}GrAtV_{a_0} \end{array}$$

We can use the non-terminal $MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$ introduced here to normalize other productions associated with the constraint $V_{a_0} \supseteq c(V_{a_1}, V_{a_2}, \ldots, V_{a_r})$. For example, the production

$$\begin{array}{rcl} Rev\_Id & ::= & Rev\_c_i^{-1} \quad Rev\_Id \quad MarkV_{a_r}GrAtV_{a_0} \ldots \\ & & MarkV_{a_1}GrAtV_{a_0} \quad Rev\_c_i \end{array}$$

is normalized to the following productions:

$$\begin{array}{rcl} Rev\_Id & ::= & Rev\_c_i^{-1}\text{-}Rev\_Id \\ & & MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}\text{-}Rev\_c_i \\ Rev\_c_i^{-1}\text{-}Rev\_Id & ::= & Rev\_c_i^{-1} \quad Rev\_Id \\ MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}\text{-}Rev\_c_i & ::= & MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} \\ & & Rev\_c_i \end{array}$$

This works because $MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$ is cyclic; it is its own reverse edge. We can also normalize the production

$$Ground ::= MarkV_{a_1}GrAtV_{a_0} \ldots MarkV_{a_r}GrAtV_{a_0}$$

to the following production:

$$Ground ::= MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$$

With these normalized productions, the CFL-reachability Algorithm will add at most $O(tr)$ edges with labels of the form $MarkV_{a_1}\text{-}V_{a_j}GrAtV_{a_0}^{(k)}$ ($O(r)$ edges for each of $O(t)$ productions). All of these edges will be cyclic. 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 when constructing the original graph). This means that the number of edges with a label of the form $c_i\text{-}MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$ or $MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}\text{-}Rev_c_i$is bounded by $O(tr)$. Also, the number of edges with a label of the form $Id\text{-}c_i^{-1}$ or $Rev\_c_i^{-1}\text{-}Rev\_Id$ is bounded by $O(nt)$.

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 Id. There may be $O(n^2)$ edges labelled Id in the graph. When the CFL-reachability Algorithm takes a given edge of the form $Id\langle V_j, V_k\rangle$ from its worklist, it could potentially examine $O(tr)$ productions of the form $Id\text{-}c_i^{-1} ::= Id \;\; c_i^{-1}$, in which 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 $Id\text{-}c_i^{-1}\langle V_j, V_m\rangle$. However, edges of the form $c_i^{-1}\langle V_k, V_m\rangle$ are introduced in the graph to encode projection constraints; this means that their number is bounded by $O(t)$. Thus, over all of the $O(tr)$ productions of the form $Id\text{-}c_i^{-1} ::= Id \;\; c_i^{-1}$, the CFL-reachability Algorithm will find no more than $O(t)$ matching edges of the form $c_i^{-1}\langle V_k, V_m\rangle$, and so it will add no more than $O(t)$ new edges as a result of processing any given edge of the form $Id\langle V_j, V_k\rangle$.

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 Related Work and Concluding Remarks

The techniques described in this paper can be extended to apply to the class of set constraints used by Heintze to do set-based analysis of ML programs [8]. This class of set constraints is effectively a superset of the class of set constraints used in this paper. In particular, Heintze extends the set constraints to handle λ-terms and function applications. These can be modelled in the CFL-reachability framework using techniques that are similar to those used in tracking ground information in the construction given in this paper.

Heintze and McAllester have also 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 [27]; 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 [25]. These ideas were elaborated on in a sequence of papers [15, 14, 24], 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] mentions CFL-reachability explicitly and references Yannakakis's paper [28].) 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 [28]; 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.

Form of label	# of edges	Productions with label on the right-hand side	Work performed for a given edge	
			# examined productions	Total # of attempts to add an edge
Id	$O(n^2)$	$Id ::= Id \;\; Id$	1	$O(n)$
		$Id\text{-}c_i^{-1} ::= Id \;\; c_i^{-1}$	$O(tr)$	$O(t)$
		$Ground\text{-}Id ::= Ground \;\; Id$	1	1
Rev_Id	$O(n^2)$	$Rev\_Id ::= Rev\_Id \;\; Rev\_Id$	1	$O(n)$
		$G ::= Rev\_Id \;\; Ground\text{-}Id$	1	$O(n)$
		$Rev\_c_i^{-1}\text{-}Rev\_Id ::= Rev\_c_i^{-1} \;\; Rev\_Id$	$O(tr)$	$O(t)$
$Ground$	$O(v)$	$Ground\text{-}EdgeV_itoV_j ::= Ground \;\; EdgeV_itoV_j$	$O(v)$	$O(v)$
		$Ground\text{-}Id ::= Ground \;\; Id$	1	$O(n)$
c_i	$O(t)$	$c_i\text{-}MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} ::= c_i \;\; MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$	$O(t)$	$O(t)$
Rev_c_i	$O(t)$	$MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}\text{-}Rev\_c_i ::= MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} \;\; Rev\_c_i$	$O(t)$	$O(t)$
c_i^{-1}	$O(t)$	$Id\text{-}c_i^{-1} ::= Id \;\; c_i^{-1}$	1	$O(n)$
$Rev\_c_i^{-1}$	$O(t)$	$Rev\_c_i^{-1}\text{-}Rev\_Id ::= Rev\_c_i^{-1} \;\; Rev\_Id$	1	$O(n)$
$EdgeV_itoV_j$	$O(tr)$	$Ground\text{-}EdgeV_itoV_j ::= Ground \;\; EdgeV_itoV_j$	1	1
		$MarkV_iGrAtV_j ::= EdgeV_itoV_j \;\; Ground\text{-}EdgeV_jtoV_i$	1	1
$EdgeV_jtoV_j$	$O(v)$	$G\text{-}EdgeV_jtoV_j ::= G \;\; EdgeV_jtoV_j$	1	$O(n)$
		$Ground ::= EdgeV_jtoV_j \;\; G\text{-}EdgeV_jtoV_j$	1	1
G	$O(n^2)$	$G\text{-}EdgeV_jtoV_j ::= G \;\; EdgeV_jtoV_j$	$O(v)$	1
$Ground\text{-}Id$	$O(n^2)$	$G ::= Rev\_Id \;\; Ground\text{-}Id$	1	$O(n)$
$G\text{-}EdgeV_jtoV_j$	$O(n^2)$	$Ground ::= EdgeV_jtoV_j \;\; G\text{-}EdgeV_jtoV_j$	1	1
$MarkV_{a_j}GrAtV_{a_0}$	$O(tr)$	$MarkV_{a_1}\text{-}V_{a_j}GrAtV_{a_0}^{(k)} ::= MarkV_{a_1}\text{-}V_{a_{j-1}}GrAtV_{a_0}^{(k)} \;\; MarkV_{a_j}GrAtV_{a_0}$	$O(tr)$	$O(tr)$
$MarkV_{a_1}\text{-}V_{a_{j-1}}GrAtV_{a_0}^{(k)}$	$O(tr)$	$MarkV_{a_1}\text{-}V_{a_j}GrAtV_{a_0}^{(k)} ::= MarkV_{a_1}\text{-}V_{a_{j-1}}GrAtV_{a_0}^{(k)} \;\; MarkV_{a_j}GrAtV_{a_0}$	1	1
$MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$	$O(t)$	$c_i\text{-}MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} ::= c_i \;\; MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$	$O(r)$	$O(t)$
		$MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}\text{-}Rev\_c_i ::= MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} \;\; Rev\_c_i$	$O(r)$	$O(t)$
		$Ground ::= MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$	1	1
$Ground\text{-}EdgeV_jtoV_i$	$O(t)$	$MarkV_iGrAtV_j ::= EdgeV_itoV_j \;\; Ground\text{-}EdgeV_jtoV_i$	1	1
$c_i\text{-}MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}$	$O(t)$	$Id ::= c_i\text{-}MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} \;\; Id\text{-}c_i^{-1}$	1	$O(t)$
$MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}\text{-}Rev\_c_i$	$O(t)$	$Rev\_Id ::= Rev\_c_i^{-1}\text{-}Rev\_Id \;\; MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}\text{-}Rev\_c_i$	1	$O(t)$
$Id\text{-}c_i^{-1}$	$O(nt)$	$Id ::= c_i\text{-}MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)} \;\; Id\text{-}c_i^{-1}$	$O(t)$	$O(t)$
$Rev\_c_i^{-1}\text{-}Rev\_Id$	$O(nt)$	$Rev\_Id ::= Rev\_c_i^{-1}\text{-}Rev\_Id \;\; MarkV_{a_1}\text{-}V_{a_r}GrAtV_{a_0}^{(k)}\text{-}Rev\_c_i$	$O(t)$	$O(t)$

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.

[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.

[24] 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).

[25] 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).

[26] 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.

[27] 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.

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