The Set Constraint/CFL Reachability Connection in Practice

John Kodumal
EECS Department
University of California, Berkeley
jkodumal@cs.berkeley.edu

Alex Aiken
Computer Science Department
Stanford University
aiken@cs.stanford.edu

ABSTRACT

Many program analyses can be reduced to graph reachability problems involving a limited form of context-free language reachability called Dyck-CFL reachability. We show a new reduction from Dyck-CFL reachability to set constraints that can be used in practice to solve these problems. Our reduction is much simpler than the general reduction from context-free language reachability to set constraints. We have implemented our reduction on top of a set constraints toolkit and tested its performance on a substantial polymorphic flow analysis application.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms

Algorithms, Design, Experimentation, Languages, Theory

Keywords

Set constraints, context-free language reachability, flow analysis, type qualifiers

1. INTRODUCTION

Many program analyses have been formulated and implemented using set constraints. Set constraint-based analyses have been shown to scale to large programs on applications such as points-to analysis, shape analysis, and receiver class prediction, though scalability requires highly optimized implementations [4, 8, 19, 21].

Recently, a number of program analyses have been formulated as context-free language (CFL) reachability problems. These include applications such as type-based polymorphic flow analysis, field-sensitive points-to analysis, and interprocedural dataflow analysis [15, 17, 18]. Getting CFL-based implementations to scale has proved as tricky as implementing set constraints, leading to new algorithms and optimization techniques.

Set constraints and CFL reachability were shown to be closely related in work by Melski and Reps [13]. While their result is useful for understanding the conceptual similarity between the two problems, it does not serve as an implementation strategy; as with many reductions, the cost of encoding one problem in the other formalism proves to be prohibitive in practice.

Furthermore, the Melski-Reps reduction does not show how to relate the state-of-the-art algorithms for set constraints to the state-of-the-art algorithms for CFL reachability. Optimizations in one formalism are not preserved across the reduction to the other formalism. Demand-driven algorithms for CFL reachability do not automatically lead to demand algorithms for set constraints (except by applying the reduction first).

Our insight is based on the observation that almost all of the applications of CFL reachability in program analysis are based on *Dyck languages*, which contain strings of matched parentheses. For Dyck languages, we show an alternative reduction from CFL reachability to set constraints that addresses the issues mentioned above. The principal contributions of this work are as follows:

- We give a novel construction for converting a Dyck-CFL reachability problem into a set constraint problem. The construction is simpler than the more general reduction described in [13]. In fact, the constraint graphs produced by our construction are nearly isomorphic to the original CFL graph.

- We show that on real polymorphic flow analysis problems, our implementation of Dyck-CFL reachability based on set constraints remains competitive with a highly tuned CFL reachability engine. This is somewhat surprising since that implementation contains optimizations that exploit the specific structure of the CFL graphs that arise in flow analysis.

These results have several consequences:

- Our results show that it is possible to use an off-the-shelf set constraint solver to solve Dyck-CFL reachability problems without suffering the penalties we normally expect when using a reduction. Furthermore, reasonable performance can be expected without having to tune the solver for each specific application.

$$\begin{array}{rcl} G \cup \{B\langle u,u'\rangle, C\langle u',v\rangle\} & \Leftrightarrow & \text{add } A\langle u,v\rangle \text{ for each production of the form } A \to B\ C \\ G \cup \{B\langle u,v\rangle\} & \Leftrightarrow & \text{add } A\langle u,v\rangle \text{ for each production of the form } A \to B \\ G & \Leftrightarrow & \text{add } A\langle u,u\rangle \text{ for each production of the form } A \to \epsilon \end{array}$$

Figure 1: Closure rules for CFL reachability

- We believe that our reduction bridges the gap between the various algorithms for Dyck-CFL reachability [10] and the algorithms for solving set constraints. In light of our reduction, it seems that the distinction between these problems is illusory: the manner in which a problem is specified (as Dyck-CFL reachability vs. set constraints) is orthogonal to other algorithmic issues (e.g. whether the solver is online vs. offline, demand-driven vs. exhaustive). This has some important consequences– for instance, we plan in the future to investigate an algorithm for incremental set constraints. Using our reduction, such an algorithm could be used immediately to solve Dyck-CFL reachability problems incrementally.

- Furthermore, our reduction shows how techniques used to solve set constraints (inductive form and cycle elimination [4]) can be applied to Dyck-CFL reachability problems.

The remainder of this paper is structured as follows. In Section 2 we briefly introduce CFL reachability and set constraints. In Section 3 we review the reduction from CFL reachability to set constraints. Section 4 presents our specialized reduction from Dyck-CFL reachability to set constraints. In Section 5 we use polymorphic flow analysis as a case study for our reduction. Section 6 covers related work, and Section 7 concludes.

2. PRELIMINARIES

We review basic material on CFL reachability and set constraints. Readers familiar with CFL reachability may wish to skip Section 2.1, which is standard. Section 2.2 is based on the framework of [21], which uses some nonstandard notation.

2.1 CFL and Dyck-CFL Reachability

In this subsection we define CFL reachability and Dyck-CFL reachability and describe an approach to solving these problems.

Let $CFG = (T, N, P, S)$ be a context free grammar with terminals T, nonterminals N, productions P and start symbol S. Let G be a directed graph with edges labeled by elements of T. The notation $A\langle u,v\rangle$ denotes an edge in G from node u to node v labeled with symbol A. Each path in G defines a word over T by concatenating, in order, the labels of the edges in the path. A path in G is an S-path if its word is in the language of CFG. The *all-pairs CFL reachability problem* determines the pairs of vertices (u, v) where there exists an S-path from u to v in G.

Now let $O = \{(_1, \ldots, (_n\}$ and $C = \{)_1, \ldots,)_n\}$ be two disjoint sets of terminals (interpreted as opening and closing symbols, respectively). The subscripts are the terminals' *indices*. The Dyck language over $O \cup C \cup \{s\}$ is generated by a context free grammar with productions of the following form:

$$S \to S\ S \mid (_1\ S\ )_1 \mid \cdots \mid (_n\ S\ )_n \mid \epsilon \mid s$$

(where s is a distinguished terminal not in O or C). The *all-pairs Dyck-CFL reachability problem* is the restriction of the all-pairs CFL reachability problem to Dyck languages.

$$\begin{tikzcd} x_1 \arrow[rrr, dashed, "S"] \arrow[dr, "(_1"] & & & z_1 \\ & y_1 \arrow[r, "s"] & y_2 \arrow[ur, ")_1"] \arrow[dr, ")_2"'] & \\ x_2 \arrow[ur, "(_2"'] \arrow[rrr, dashed, "S"'] & & & z_2 \end{tikzcd}$$

Figure 2: Example Dyck-CFL reachability problem

Figure 1 shows the closure rules for the CFL and Dyck-CFL reachability algorithms. The rules assume that the CFL grammar has been normalized such that no right-hand side of any production contains more than two symbols[1]. A naive CFL reachability algorithm might apply the rules as follows: whenever the CFL graph matches the form on the left-hand side of the rule, add the edges indicated by the right-hand side of the rule; iterate this process until no rule induces a new edge addition. The application of these rules produces a graph $closure(G)$ that contains all possible edges labeled by nonterminals in the grammar. To check whether there is an S-path from nodes u to v in G, we simply check for the existence of an edge $S\langle u,v\rangle$ in $closure(G)$.

In general, the all-pairs CFL-reachability problem can be solved in time $O(|T \cup N|^3 n^3)$ for a graph with n nodes.

Figure 2 shows an example of a Dyck-CFL reachability problem. The dashed lines show the edges added by the computation of $closure(G)$. The edges show that there are S-paths between nodes x_1 and z_1, and x_2 and z_2. Self-loop S-paths derived from the production $S \to \epsilon$ are not shown.

2.2 Set Constraints

In this subsection, we review the set constraints framework introduced in [21]. We define a language of set constraints (essentially a subset of the full language defined in [1, 6]) and discuss *inductive form*, a graph-based representation of set constraints.

2.2.1 A Language of Set Constraints

A *constraint system* is a finite collection of *set constraints*. A set constraint is a relation of the form $L \subseteq R$, where L and R are *set expressions*. Set expressions consist of set

[1]For instance, by converting the grammar to Chomsky normal form.

$$\begin{array}{rcl}
C \cup \{\mathcal{X} \subseteq \mathcal{X}\} & \Leftrightarrow & C \\
C \cup \{c(se_1, \ldots, se_{a(c)}) \subseteq c(se'_1, \ldots, se'_{a(c)})\} & \Leftrightarrow & C \cup \bigcup_i \{se_i \subseteq se'_i\} \\
C \cup \{c(se_1, \ldots, se_{a(c)}) \subseteq proj(c, i, se)\} & \Leftrightarrow & C \cup \{se_i \subseteq se\} \\
C \cup \{d(se_1, \ldots, se_{a(d)}) \subseteq proj(c, i, se)\} & \Leftrightarrow & C \\
 & & \text{if } c \neq d \\
C \cup \{c(\ldots) \subseteq d(\ldots)\} & \Leftrightarrow & \text{no solution} \\
 & & \text{if } c \neq d
\end{array}$$

Figure 3: Resolution rules for set constraints

variables $\mathcal{X}, \mathcal{Y}, \ldots$, terms constructed from n-ary constructors $c, d, \ldots$[2], and projection patterns:

$$L, R \in se ::= \mathcal{X} \mid c(se_1, \ldots, se_{a(c)}) \mid proj(c, i, se)$$

Each constructor c has an arity $a(c)$.

Given a constraint of the form $c(se_1, \ldots, se_{a(c)}) \subseteq proj(c, i, se)$, the projection pattern $proj(c, i, se)$ has the effect of selecting se_i, the *ith* component of the term on the left-hand side, and adding the constraint $se_i \subseteq se$. Projection patterns are closely related to the more standard projection notation $c^{-i}(se)$. We can express $c^{-i}(se)$ using the following equivalence:

$$\begin{array}{rcl}
c^{-i}(se) & \equiv & \mathcal{V} \\
 & & \text{where } \mathcal{V} \text{ is fresh and} \\
 & & se \subseteq proj(c, i, \mathcal{V})
\end{array}$$

The equivalence preserves least solutions under a compatibility restriction: the variable $\mathcal{V}$ used to represent $c^{-i}(se)$ can only occur on the left-hand side of any other inclusion constraint. We use the *proj* notation because it simplifies the presentation of the constraint re-write rules and elucidates the connection between Dyck-CFL reachability and set constraints. However, we also use $c^{-i}(se)$ wherever that notation is more natural. Our uses of $c^{-i}(se)$ always adhere to the compatibility restriction; therefore in this work it is always safe to replace that notation with the equivalent using *proj*.

2.2.2 Constraint Graphs

Algorithms for solving set constraints operate by applying a set of resolution rules to the constraint system until no rules apply; this is the *solved form*. The solved form makes the process of reading off a particular solution (or all solutions) simple. For our language of set constraints, we apply the resolution rules in Figure 3 as left-to-right re-write rules to reduce the constraint system to solved form.

Algorithmically, a system of set constraints C can be represented as a directed graph $G(C)$ where the nodes of the graph are set expressions and the edges denote *atomic constraints*. A constraint is atomic if either the left-hand side or the right-hand side is a set variable. Computing the solved form involves closing $G(C)$ under the resolution rules, which are descriptions of how to add new edges to the graph.

In general, a system of set constraints C with v variables can be reduced to solved form in time $O(v^2|C|)$.

Inductive form is a particular graph representation that exploits the fact that variable-variable constraints $\mathcal{X} \subseteq \mathcal{Y}$ can be represented as either a successor edge ($\mathcal{Y} \in succ(\mathcal{X})$) or a predecessor edge ($\mathcal{X} \in pred(\mathcal{Y})$). The choice is made based on a fixed total order o (generated randomly) on the variables. For a constraint $\mathcal{X} \subseteq \mathcal{Y}$, the edge is stored as a successor edge on $\mathcal{X}$ if $o(\mathcal{X}) > o(\mathcal{Y})$, otherwise, it is stored as a predecessor edge on $\mathcal{Y}$. Constraints of the form $c(\ldots) \subseteq \mathcal{X}$ are always stored as predecessor edges on $\mathcal{X}$, and constraints of the form $\mathcal{X} \subseteq proj(c, i, se)$ are always stored as successor edges on $\mathcal{X}$.

Given these representations, the transitive closure rule for inductive form is as follows:

$$L \in pred(\mathcal{X}) \wedge R \in succ(\mathcal{X}) \Rightarrow L \subseteq R$$

This rule, in conjunction with the resolution rules in Figure 3, produce a solved graph in inductive form. The advantage of inductive form is that many fewer transitive edges are added in comparison to other graph representations [4]. Inductive form does not explicitly compute the least solution, but the least solution is easily calculated by doing backwards-reachability on the constraint graph:

$$LS(\mathcal{Y}) = \{c(\ldots) | c(\ldots) \in pred(\mathcal{Y})\} \cup \bigcup_{\mathcal{X} \in pred(Y)} LS(\mathcal{X})$$

2.2.3 Solutions of Constraint Systems

We interpret set constraints under the term-set model [6]. In this model, the solution to a system of set constraints maps each set variable to a (possibly infinite) set of ground terms such that all inclusion relations are satisfied. It turns out that each set variable describes a *regular tree language* and the solved form of the constraint graph can be viewed as a collection of regular tree grammars. Each atomic constraint $c(\ldots) \subseteq \mathcal{X}$ in the solved form of the constraint system can be interpreted as a production in a tree grammar by treating it as a production $X \Rightarrow c(\ldots)$. For example, consider the solved constraint system

$$\begin{array}{c}
cons(zero, \mathcal{X}) \subseteq \mathcal{X} \\
nil \subseteq \mathcal{X}
\end{array}$$

where *zero* and *nil* are nullary constructors, and *cons* is a binary constructor. We see that $\mathcal{X}$ describes the set of all lists where every element is *zero*. The least solution for $\mathcal{X}$ can be viewed as a tree language $L(\mathcal{X})$ whose grammar is

$$\begin{array}{rcl}
X & \Rightarrow & cons(zero, X) \\
X & \Rightarrow & nil
\end{array}$$

3. THE MELSKI-REPS REDUCTION

In this section we review the reduction from CFL reachability to set constraints [12, 13]. Readers familiar with this reduction may wish to skip this section. Again, we assume that the CFL grammar is normalized so that the right-hand side of every production contains at most two symbols.

[2]We assume throughout that constructors are non-strict.

The construction encodes each node u in the initial CFL graph G with one set variable $\mathcal{U}$, one nullary constructor n_u and the constraint

$$n_u \subseteq \mathcal{U}$$

In encoding the graph, the goal is to represent an edge $A\,\langle u, v\rangle$ in the closed graph by the presence of a term $c_A(\mathcal{V})$, where c_A is a unary constructor corresponding to symbol A, in the least solution of $\mathcal{U}$. Accordingly, an initial edge of the form $a\,\langle u, v\rangle$ in G is captured by a constraint

$$c_a(\mathcal{V}) \subseteq \mathcal{U}$$

This completes the representation of the initial graph. The next step of the reduction is to encode the productions of the CFL grammar. Since edges are encoded as constructed terms, the notion of "following an edge" is captured by projection. The left-hand side of each production tells us which edge we should add if we follow edges labeled by the right-hand side. For instance, a binary production of the form $A \rightarrow B\ C$ says that we should add an A edge from node u to any nodes reached by following a B edge from u and then following a C edge. The set constraint that encodes this rule is

$$c_A(c_C^{-1}(c_B^{-1}(\mathcal{U}))) \subseteq \mathcal{U}$$

For example, the CFL graph

$$u \xrightarrow{B} u' \xrightarrow{C} v$$

and the production $A \rightarrow B\ C$ result in a system of set constraints containing the following inclusion relations:

$$\begin{gathered} n_u \subseteq \mathcal{U} \\ c_C(\mathcal{V}) \subseteq \mathcal{U}' \\ c_B(\mathcal{U}') \subseteq \mathcal{U} \\ c_A(c_C^{-1}(c_B^{-1}(\mathcal{U}))) \subseteq \mathcal{U} \end{gathered}$$

These constraints imply the desired relationship $c_A(n_v) \subseteq \mathcal{U}$, which represents the edge $A\,\langle u, v\rangle$. Note these constraints only encode each production locally: similar constraints must be generated for every node in the graph.

Besides binary productions, productions of the form $A \rightarrow B$ and $A \rightarrow \epsilon$ may occur in the grammar. Productions of the first form are encoded locally for each node u by the constraint

$$c_A(c_B^{-1}(\mathcal{U})) \subseteq \mathcal{U}$$

representing the fact that any B edge from node u is also an A edge. Finally, productions of the form $A \rightarrow \epsilon$ are encoded locally for each node u by the constraint

$$c_A(\mathcal{U}) \subseteq \mathcal{U}$$

Correctness of the reduction is proved by showing there is an S-path from node u to node v in $\mathit{closure}(G)$ if and only if $c_S(n_v) \subseteq \mathcal{U}$ is in the least solution of the constructed system of constraints.

4. A SPECIALIZED DYCK REDUCTION

This section presents our reduction, which is specialized to Dyck-CFL reachability problems. We first explain the intuition behind our approach. We then describe the specifics of the encoding, sketch a proof of correctness, and provide a complexity argument. Finally, we provide experimental

$$\begin{tikzcd}
O_1(\mathcal{X}_1) \arrow[dr] & & & \mathit{proj}(O_1, 1, \mathcal{Z}_1) \\
& Y_1 \arrow[r] & Y_2 \arrow[ur] \arrow[dr] & \\
O_2(\mathcal{X}_2) \arrow[ur] & & & \mathit{proj}(O_2, 1, \mathcal{Z}_2)
\end{tikzcd}$$

Figure 4: The constraint graph corresponding to the Dyck-CFL graph from Figure 2 using the reduction from Section 4

results suggesting that the specialized reduction is more efficient in practice than the Melski-Reps reduction applied to Dyck-CFL grammars.

The idea behind our reduction is that the set constraint closure rules can encode Dyck language productions. The intuition is straightforward: the rule $S \rightarrow S\ S$ corresponds to the transitive closure rule, and the rules $S \rightarrow (_i\ S\ )_i$ correspond to matching constructed terms with projection patterns. By encoding the original Dyck-CFL graph as a system of set constraints in a way that exploits this correspondence, we can avoid encoding the productions of the grammar at each node. This leads to a more natural and compact encoding of the input graph, and better performance in applications where the number of parenthesis kinds (hence, the number of productions) is not a constant.

The first step of the encoding is unchanged. We represent each node u with a variable $\mathcal{U}$ and a nullary constructor n_u. We connect these with constraints of the form

$$n_u \subseteq \mathcal{U}$$

Edges are encoded so that subset relationships $\mathcal{U} \subseteq \mathcal{V}$ between variables in the constraint system represent discovered $S\,\langle u, v\rangle$ edges in the input graph. Since we are only considering Dyck languages, there are only three kinds of labeled edges to consider:

1. For each $s\,\langle u, v\rangle$ edge in the input graph, we add the constraint $\mathcal{U} \subseteq \mathcal{V}$ (where $\mathcal{U}$ and $\mathcal{V}$ are the variables corresponding to nodes u and v).

2. For each $(_i\langle u, v\rangle$ edge in the input graph, we create a unary constructor O_i and add the constraint $O_i(\mathcal{U}) \subseteq \mathcal{V}$.

3. For each $)_i\,\langle u, v\rangle$ edge in the input graph, we add the constraint $\mathcal{U} \subseteq \mathit{proj}(O_i, 1, \mathcal{V})$.

Notice that there appears to be a slight asymmetry in the reduction: edges labeled with $(_i$ symbols are encoded using constructed terms, while edges labeled with $)_i$ symbols are encoded using projections. Naively, one might attempt to devise a "symmetric" reduction by encoding an edge $)_i\,\langle u, v\rangle$ with a constraint $\mathcal{U} \subseteq O_i(\mathcal{V})$, making a correspondence between the production $S \rightarrow (_i\ S\ )_i$ and the second rule (not the third) in Figure 3. However, this approach would yield an inconsistent system of set constraints for any interesting graph. For example, the graph in Figure 2 would result in a constraint system containing the constraints $O_1(\mathcal{X}_1) \subseteq \mathcal{Y}_1 \subseteq \mathcal{Y}_2 \subseteq O_2(\mathcal{Z}_2)$, which

Constraints	New Constraint	Production	Edges	New Edge
$\mathcal{X} \subseteq \mathcal{Y} \wedge \mathcal{Y} \subseteq \mathcal{Z}$	$\mathcal{X} \subseteq \mathcal{Z}$	$S \to S\ S$	$S\langle x,y\rangle \wedge S\langle y,z\rangle$	$S\langle x,z\rangle$
$O_i(\mathcal{X}) \subseteq \mathcal{Y} \wedge \mathcal{Y} \subseteq \mathcal{Z}$	$O_i(\mathcal{X}) \subseteq \mathcal{Z}$	$L_i \to L_i\ S$	$L_i\langle x,y\rangle \wedge S\langle y,z\rangle$	$L_i\langle x,z\rangle$
$\mathcal{X} \subseteq \mathcal{Y} \wedge \mathcal{Y} \subseteq proj(O_i,1,\mathcal{Z})$	$\mathcal{X} \subseteq proj(O_i,1,\mathcal{Z})$	$R_i \to S\ R_i$	$S\langle x,y\rangle \wedge R_i\langle y,z\rangle$	$R_i\langle x,z\rangle$
$O_i(\mathcal{X}) \subseteq \mathcal{Y} \wedge \mathcal{Y} \subseteq proj(O_i,1,\mathcal{Z})$	$O_i(\mathcal{X}) \subseteq proj(O_i,1,\mathcal{Z}) \wedge \mathcal{X} \subseteq \mathcal{Z}$	$S \to L_i\ R_i$	$L_i\langle x,y\rangle \wedge R_i\langle y,z\rangle$	$S\langle x,z\rangle$

Table 1: Constraints added to C' and the corresponding edges in $closure(G)$

are clearly inconsistent. We avoid this problem by using projection patterns to represent $)_i$ edges. Thus, the graph in Figure 2 results in a constraint system containing $O_1(\mathcal{X}_1) \subseteq \mathcal{Y}_1 \subseteq \mathcal{Y}_2 \subseteq proj(O_2, 1, \mathcal{Z}_2)$, which is consistent.

This construction only introduces one constraint per node and one constraint per edge of the original CFL graph. In contrast, the reduction described in Section 3 introduces $O(k)$ constraints per node (where k is the number of parenthesis kinds in the Dyck grammar) to encode the grammar productions locally, in addition to the cost of encoding the graph edges.

To solve the all-pairs Dyck-CFL reachability problem, we apply the rewrite rules in Section 2.2 to the resulting constraint system. As we will show, there is an S-path from nodes u to v in the initial graph if and only if $n_u \subseteq \mathcal{V}$ in the least solution of the constructed constraint system.

Figure 4 shows the constraint graph corresponding to the Dyck-CFL reachability problem shown in Figure 2 using this reduction. The nullary constructors are omitted for clarity.

4.1 Correctness

We sketch a proof of correctness for the specialized reduction. The reduction is correct if the solution to the constructed set constraint problem gives a solution to the original Dyck-CFL reachability problem.

THEOREM 1. *Let G be an instance of the Dyck-CFL reachability problem, and C be the collection of set constraints constructed as above to represent G. Then there is an edge $S\langle u,v\rangle$ in $closure(G)$ if and only if $n_u \subseteq \mathcal{V}$ is present in $LS(C)$.*

The theorem follows immediately from the following lemmas:

LEMMA 1. *Given C and G as in Theorem 1, let C' denote the conjunction of C with the system of constraints introduced by applying the resolution rules shown in Figure 3 along with the transitive closure rule $se \subseteq \mathcal{X} \wedge \mathcal{X} \subseteq se' \Rightarrow se \subseteq se'$. Then there is an edge $S\langle u,v\rangle$ in $closure(G)$ if and only if $\mathcal{U} \subseteq \mathcal{V}$ is present in C'.*

PROOF. We apply the set constraint resolution rules in lock-step with the CFL closure rules, and show that for every S-edge added to $closure(G)$, there is a corresponding variable-variable constraint that can be added to C'. This approach requires us to normalize the Dyck-CFL grammar in a very specific way:

$$\begin{array}{lcl} S & \to & S\ S \\ S & \to & L_i\ R_i \\ L_i & \to & (_i \\ L_i & \to & L_i\ S \\ R_i & \to &)_i \\ R_i & \to & S\ R_i \\ S & \to & s \\ S & \to & \epsilon \end{array}$$

It is easy to see that the above grammar derives the same set of strings as the original Dyck-CFL grammar. The intuition here is that normalizing the Dyck-CFL grammar this way causes the CFL closure rules to add edges in exactly the same way as the set constraint resolution rules.

We now formalize this intuition. Consider the following order for adding edges to $closure(G)$ and constraints to C':

1. Add all edges implied by the productions $S \to s$, $S \to \epsilon$, $R_i \to\ )_i$, and $L_i \to\ (_i$. Note that there are only finitely many edges that can be added, and that they are all added in this step.

2. Add all edges implied by the productions $S \to S\ S$, $S \to L_i\ R_i$, $L_i \to L_i\ S$, and $R_i \to S\ R_i$, and add the corresponding constraints to C' according to Table 1.

In the first step, notice that the added S edges correspond to variable-variable edges that already exist in the initial system of constraints C (refer to the first rule of the reduction in Section 4). In the second step, notice that for every S edge added to $closure(G)$, there are corresponding constraints that must exist in C' that lead to the appropriate variable-variable constraint.

□

LEMMA 2. *Given C and G as in Theorem 1, let C' denote the conjunction of C with the system of constraints introduced by applying the resolution rules shown in Figure 3 along with the transitive closure rule $se \subseteq \mathcal{X} \wedge \mathcal{X} \subseteq se' \Rightarrow se \subseteq se'$. Then the constraint $n_u \subseteq \mathcal{V}$ implies the existence of a constraint $\mathcal{U} \subseteq \mathcal{V}$ in C'.*

PROOF. Clearly, the rules in Figure 3 cannot add a constraint of the form $n_u \subseteq \mathcal{V}$, so such a constraint is either present initially in C or it was added to C' by transitive closure. In the former case, $\mathcal{U} = \mathcal{V}$ and the lemma holds trivially. In the latter case, we can show (by induction on the number of applications of the transitive closure rule) that there are variables $\mathcal{X}_1, \ldots, \mathcal{X}_n$ with $n_u \subseteq \mathcal{U} \subseteq \mathcal{X}_1 \subseteq \ldots \subseteq \mathcal{X}_n \subseteq \mathcal{V}$ present in C'. Then by the transitive closure rule, $\mathcal{U} \subseteq \mathcal{V}$ must be present in C'. □

We now prove Theorem 1 using the two lemmas. To show that $S\langle u,v\rangle \in closure(G)$ implies $n_u \subseteq \mathcal{V}$ is present in C', we note that Lemma 1 guarantees the existence of a constraint $\mathcal{U} \subseteq \mathcal{V}$ in C'. Our construction guarantees an initial constraint $n_u \subseteq \mathcal{U}$, so by the transitive closure rule, $n_u \subseteq \mathcal{V}$ exists in C'. To show that a constraint $n_u \subseteq \mathcal{V}$ in C' implies $S\langle u,v\rangle \in closure(G)$, we appeal to Lemma 2, which guarantees the existence of a constraint $\mathcal{U} \subseteq \mathcal{V}$ in C'. Appealing once again to Lemma 1, we have that $S\langle u,v\rangle \in closure(G)$.

4.2 Complexity

We first discuss the running time to solve the all-pairs Dyck-CFL reachability problem using the generic CFL reachability algorithm outlined in Section 2.1. In general, the generic algorithm has complexity $O(|T \cup N|^3 n^3)$. Applying this result directly to an instance of the Dyck-CFL problem with k parenthesis kinds and n graph nodes yields an $O(k^3 n^3)$ running time.

However, we can sharpen this result by specializing the complexity argument to the Dyck-CFL grammar. The running time of the generic CFL reachability algorithm is dominated by the first rule in Figure 1. In this step, each edge in the graph might pair with each of its neighboring edges. In other words, for a given node j, each of j's incoming edges might pair with each of j's outgoing edges. Each node in the graph might have $O(kn)$ edges labeled with $(_i$ or $)_i$ (there are k different kinds, and n potential target nodes), as well as $O(n)$ edges labeled with S. For node j, each of the $O(kn)$ incoming edges may pair with $O(n)$ outgoing edges, by a single production. Similarly, each of the $O(n)$ incoming S edges can match with $O(n)$ outgoing edges by the single $S \rightarrow S\ S$ production. Thus, the work for each node j is bounded by $O(kn^2)$. Since there are n nodes in the graph, the total work is $O(kn^3)$.

An analogous argument holds for our reduction. For the fragment of set constraints used in this paper, a constraint system C can be solved in time $O(v^2|C|)$ where v is the number of variables in C and $|C|$ is the number of constraints in C. Given an instance of the Dyck-CFL problem with k parenthesis kinds and n nodes, our reduction yields a constraint system with $O(n)$ variables and $O(kn^2)$ constraints, yielding an $O(kn^4)$ running time. Again, a more precise running time can be achieved by noting that for a given variable node i in the constraint graph, the rules in Figure 3 can apply only $O(kn^2)$ times to pairs of i's upper and lower bounds. In an n node graph, then, the total work is $O(kn^3)$.

4.3 Empirical Comparison

We have implemented both the Melski-Reps reduction and our reduction in order to compare the two approaches. Both implementations used the BANSHEE toolkit as the underlying set constraint solver [11]. Our implementation of the Melski-Reps reduction uses the following normalized form of the Dyck-CFL grammar:

$$\begin{array}{rcl} S & \rightarrow & S\ S \\ S & \rightarrow & L_i\ )_i \\ L_i & \rightarrow & (_i\ S \\ S & \rightarrow & s \\ S & \rightarrow & \epsilon \end{array}$$

We ran both implementations on randomly generated graphs. To test correctness, we checked that both implementations gave consistent answers to random reachability queries. We found that our implementation of the Melski-Reps reduction did not scale to large graphs, even when the number of parenthesis kinds was kept small. For example, on a 500 node graph with 7500 edges and 20 parenthesis kinds, the Melski-Reps implementation computed the graph closure in 45 seconds, while our reduction completed in less than 1 second. This prevented us from testing the Melski-Reps approach on real applications: we tried the implementation on the benchmarks in our case study (see Section 5) but found that none finished within 10 minutes.

5. CASE STUDY: POLYMORPHIC FLOW ANALYSIS

Flow analysis statically estimates creation, use, and flow of values in a program [14]. Problems such as computing may-alias relationships, determining receiver class information, and resolving control flow in the presence of indirect function calls can be solved using flow analysis. As with most static analyses, flow analysis can be made more precise by treating functions polymorphically. Polymorphic flow analysis eliminates spurious flow paths that arise from conflating all call sites of a function.

Recent work has established a connection between type-based polymorphic flow analysis and CFL reachability [15]. As an alternative to constraint copying, systems of instantiation constraints are reduced to a CFL graph. The flow relation is described by a Dyck-CFL grammar, hence, the problem of finding the flow of values in the input program reduces to an all-pairs Dyck-CFL reachability problem.

In this section, we apply our technique to the problem of polymorphic flow analysis. For concreteness, we consider the specific problem of polymorphic tainting analysis [20]. In tainting analysis, we are interested in checking whether any values possibly under adversarial control can flow to functions that expect trusted data. As a trivial example, consider the following code:

```
int main(void)
{
    char *buf;
    buf = read_from_network();
    exec(buf);
}
```

The call `exec(buf)` is probably not safe, since the buffer may come from a malicious or untrusted source. The particular tainting analysis we use employs *type qualifiers* to solve this problem [5]. Briefly, we can detect errors like the example shown above by introducing two type qualifiers *tainted* and *untainted*. Data that may come from an untrusted source is given the type qualifier *tainted*, and data that must be trusted is given the type qualifier *untainted*. A type error is produced whenever a *tainted* value flows to a place that must be *untainted*.

Figure 5 illustrates the connection between polymorphic tainting analysis and CFL reachability with another small program. There are two calls to the identity function `id`. At the first call site, `id` is passed *tainted* data. At the second call site, `id` is passed *untainted* data. The variable `z2` is required to be *untainted*. Monomorphic flow analysis would conflate the two call sites, leading to a spurious error (namely, that *tainted* data from variable `x1` reaches the *untainted* variable `z2`). Polymorphic flow analysis can distinguish these two call sites, eliminating the spurious error. The flow graph in Figure 5 shows the connection to Dyck-CFL reachability: each function call site is labeled with a distinct index. The flow from an actual parameter to a formal at call site i is denoted by an edge labeled $(_i$. The flow corresponding to the return value of the function is denoted by an edge labeled $)_i$. Intraprocedural assignment is denoted by s edges. In this example, the spurious flow path between x_1 and z_2 corresponds to a path whose word is $(_1\ s\ )_2$, which is not in the Dyck language.

In the remainder of this section, we discuss how to extend

```
int id(int y1) { int y2 = y1; return y2; }

int main(void)
{
    tainted int x1;
    int z1,x2;
    untainted int z2;
    z1 = id(x1); // call site 1
    z2 = id(x2); // call site 2
}
```

$$\begin{tikzcd}
\textit{tainted} \arrow[r, "s"] & x_1 \arrow[dr, "(_1"] & & & z_1 & \\
& & y_1 \arrow[r, "s"] & y_2 \arrow[ur, ")_1"] \arrow[dr, ")_2"] & & \\
& x_2 \arrow[ur, "(_2"'] & & & z_2 \arrow[r, "s"] & \textit{untainted}
\end{tikzcd}$$

Figure 5: Example C program and the corresponding Dyck-CFL reachability problem

our technique to solve polymorphic flow analysis problems. We also discuss some optimizations that improve the performance of our reduction on polymorphic flow graphs. We conclude with an experimental assessment of our approach on a set of C benchmarks.

5.1 Extensions

It turns out that simple Dyck-CFL reachability isn't quite sufficient for most polymorphic flow analysis applications. In this section, we discuss two additional aspects of the problem that required us to extend our technique.

5.1.1 PN Reachability

The first problem is that Dyck-CFL reachability fails to capture many of the valid flow paths. Besides the matched reachability paths represented by Dyck languages, certain kinds of partially matched reachability also represent valid flow. For example, in the graph shown in Figure 5, there should be a valid flow path from the *tainted* qualifier to the node y_2, since it is possible for `y2` to contain a *tainted* integer at runtime. However, there is no S path between *tainted* and y_2. This issue also arises in other applications such as interprocedural dataflow analysis.

The conceptual solution to this problem is to add productions to the Dyck grammar that admit these additional partially matched paths. For our application, it turns out that we should admit all paths generated by the following grammar (see [15]):

$$\begin{array}{rcl}
\mathit{Start} & \rightarrow & P\ N \\
P & \rightarrow & S\ P \\
& | &)_i\ P \\
& | & \epsilon \\
N & \rightarrow & S\ N \\
& | & (_i\ N \\
& | & \epsilon \\
S & \rightarrow & (_i\ S\ )_i \\
& | & S\ S \\
& | & s \\
& | & \epsilon
\end{array}$$

Intuitively, this grammar accepts any substring of a string in a Dyck language: P paths correspond to prefixes of Dyck strings, and N paths correspond to suffixes of Dyck strings.

Since our reduction (unlike the Melski-Reps reduction) is specialized to Dyck languages, it is not immediately obvious that we can modify our approach to accept exactly the strings in the above language. Fortunately, it turns out that we can handle these additional flow paths without fundamentally changing our reduction. We separate the problem into three subproblems. First, we tackle the problem of admitting N paths (open-paren suffixes) within our reduction. Second, we tackle the problem of admitting P paths (close-paren prefixes) within our reduction. Finally, we combine the two solutions and handle the above language in its full generality.

We first consider the problem of finding N paths in the system of constraints produced by our reduction. Suppose we ask whether there is an N path between nodes u and v. To answer this query, we first recall that the least solution of a set variable is a regular tree language. Let $L(\mathcal{V})$ denote the language corresponding to the least solution of $\mathcal{V}$. Now consider the following tree language LN_u:

$$\begin{array}{rcl}
N & \Rightarrow & n_u \\
& | & O_i(N)
\end{array}$$

We claim (without proof) that there is an N path in the closed graph between nodes u and v if and only if the intersection of LN_u and $L(\mathcal{V})$ is non-empty. To see why, recall that edges of the form $(_i\langle u, v\rangle$ are represented by constraints $O_i(\mathcal{U}) \subseteq \mathcal{V}$ in our reduction while edges of the form $S\,\langle u, v\rangle$ are represented by constraints $\mathcal{U} \subseteq \mathcal{V}$. Then an N path between nodes u and v is indicated by the presence of a term of the form $O_i(\ldots O_j(n_u))$ in the least solution of $\mathcal{V}$. The language LN_u generates exactly the terms of this form, so if $L(\mathcal{V}) \cap LN_u$ is non-empty, there is at least one such term in the least solution of $\mathcal{V}$, hence, there is an N path from u to v. As a simple example, consider the following CFL graph:

$$\begin{tikzcd}
u \arrow[r, "(_i"] & x \arrow[r, "s"] & y \arrow[r, "(_j"] & v
\end{tikzcd}$$

and suppose we ask whether there is an N path from node u to node v. Our reduction yields a constraint system which includes the following constraints:

$$\begin{array}{c}
n_u \subseteq \mathcal{U} \\
O_i(\mathcal{U}) \subseteq \mathcal{X} \\
\mathcal{X} \subseteq \mathcal{Y} \\
O_j(\mathcal{Y}) \subseteq \mathcal{V}
\end{array}$$

Note that $L(\mathcal{V})$ contains the term $O_j(O_i(n_u))$, which is also an element of LN_u, indicating that there is an N path from u to v.

We now consider the problem of finding P paths in the system of constraints produced by our reduction. The technique is essentially the same as with finding N paths. There is one wrinkle, however: the $)_i$ edges that form P paths are represented as projections in the constraint system. There is no term representation of a $)_i$ edge as there is with a $(_i$ edge. The solution is straightforward: we simply add a term representation for these edges. We modify the reduction so that for each edge of the form $)_i\ \langle u, v\rangle$, we also add the constraint $p(\mathcal{U}) \subseteq \mathcal{V}$ (where p is a new unary constructor) to the system. Note that there is only one p constructor used for all indexed $)_i$ symbols. This is because the indices of the $)_i$ symbols are irrelevant to the P paths. Now, to check for P paths between nodes u and v, we simply check the non-emptiness of the intersection of $L(\mathcal{V})$ with the following language LP_u:

$$\begin{array}{rcl} P & \Rightarrow & n_u \\ & | & p(P) \end{array}$$

Finally, to check for PN paths from u to v, we combine the above two solutions, and check the non-emptiness of $L(\mathcal{V}) \cap LPN_u$, where LPN_u is defined as follows:

$$\begin{array}{rcl} N & \Rightarrow & P \\ & | & O_i(N) \\ P & \Rightarrow & n_u \\ & | & p(P) \end{array}$$

Note that this language consists of terms of the form

$$O_i(\ldots O_j(p(\ldots p(n_u))))$$

which precisely characterize the PN paths.

In practice, we do not actually build a tree automata recognizing the language LPN_u to compute the intersection. Instead, to check for a PN path from node u to node v, we traverse the terms in the least solution of $\mathcal{V}$, searching for n_u. We prune the search when the expanded term is no longer in the language LPN_u or when a cycle is found. Since our implementation applies the rules in Figure 3 online, this divides our algorithm into two phases: an exhaustive phase where all S paths are discovered, and a demand-driven phase where PN paths are discovered as reachability queries are asked[3].

The approach outlined in this section can be generalized. The ideas apply to any regular language reachability problem composed with a Dyck language reachability problem. In general, let G be an instance of the Dyck-language reachability problem with alphabet $O \cup C$, and R be a regular language with alphabet $A \cup \epsilon$. Alphabet A need not be disjoint from $O \cup C$, but the symbols s and S in the Dyck language must be interpreted as ϵ in the regular language. The *all-pairs composed reachability problem* over D and G determines the set of vertices (u, v) such that there is a path in the language R from u to v in $closure(G)$. Viewing the PN grammar in this setting, the P productions correspond to the regular language $)_i^*$, the N productions correspond to the regular language $(_i^*$, and the regular language R is $)_i^*(_i^*$.

The approach outlined previously solves the all-pairs composed reachability problem: the constraint solver eliminates the Dyck language problem, leaving a regular language reachability problem that can be solved on demand by exploring the least solution of the constraints as described above.

[3]A completely demand-driven algorithm could be implemented using the approach of [7].

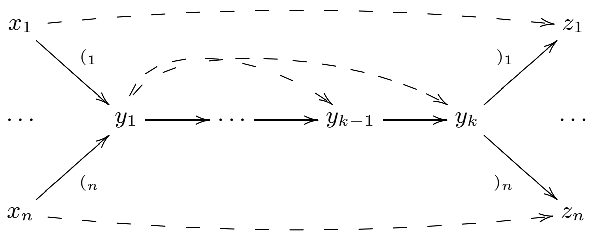

Figure 6: The summary edge optimization

5.1.2 Global Nodes

Another problem we face is that global variables in languages such as C must be treated specially to discover all the valid flow paths. The problem is that assignments to global variables can flow to any context, since global storage can be accessed at any program point. Conceptually, nodes corresponding to global variables should have self-loops labeled $)_i$ for every index i that may appear elsewhere in the graph. This solution essentially treats globals monomorphically (see [3] for a more complete discussion). While this solution is simple and easy to implement, in practice it is too expensive to represent these edges explicitly. With an explicit representation, each global variable's upper bounds may be proportional to the size of the input program. Explicitly constructing this list is prohibitively expensive: with an explicit representation of self-edges, our implementation did not finish after 10 minutes on even the smallest benchmark.

We solved this problem by adding a new feature to the constraint solver. The added feature has a simple interpretation, and may prove useful in other contexts. We introduce *constructor groups*, which are simply user-specified sets of constructors. The elements of a constructor group must all have the same arity. Once a constructor group g is defined, a special kind of projection pattern called a *group projection pattern* can be created. A group projection pattern $gproj(g, i, e)$ has essentially the same semantics as a projection pattern, except that instead of specifying a single constructor, an entire group is specified. The new rules for handling group projections are as follows:

$$\begin{array}{lcl} C \cup \{c(se_1, \ldots, se_{a(c)}) \subseteq gproj(g, i, se)\} & \Leftrightarrow & C \cup \{se_i \subseteq se\} \\ & & \text{if } c \in g \\ C \cup \{c(se_1, \ldots, se_{a(c)}) \subseteq gproj(g, i, se)\} & \Leftrightarrow & C \\ & & \text{if } c \notin g \end{array}$$

Constructor groups provide us with a way to concisely represent collections of similar projection patterns. To handle global nodes, we add each O_i constructor to a new group g_O. For each global node u, we add the constraint

$$\mathcal{U} \subseteq gproj(g_O, 1, \mathcal{U})$$

which simulates the addition of self loops $)_i$ for each index i in the graph.

5.2 Optimizations

In this subsection, we discuss some optimizations for polymorphic flow analysis applications.

5.2.1 Summary Edges

The CFL graphs that arise in this application have a particular structure that can be exploited to eliminate redun-

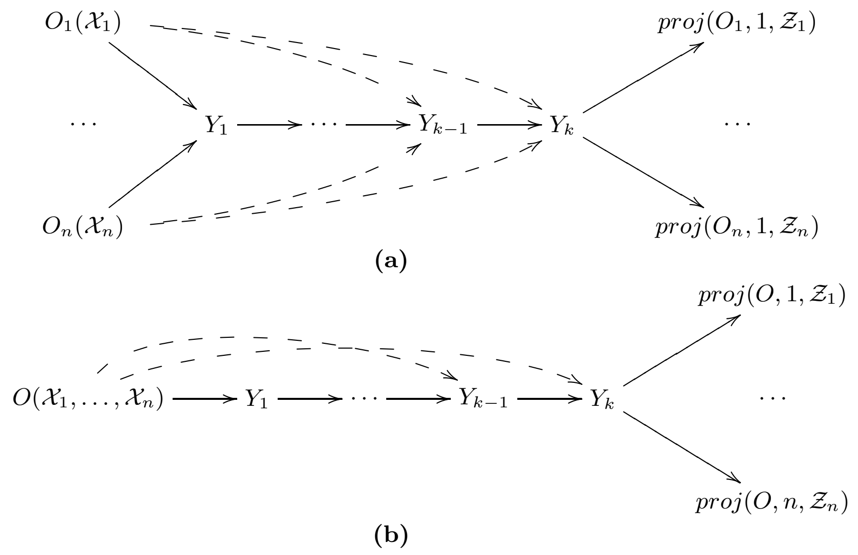
$O_1(\mathcal{X}_1)$
$\cdots$
$O_n(\mathcal{X}_n)$
Y_1
Y_{k-1}
Y_k
$proj(O_1, 1, \mathcal{Z}_1)$
$\cdots$
$proj(O_n, 1, \mathcal{Z}_n)$
(a)
$O(\mathcal{X}_1, \ldots, \mathcal{X}_n)$
Y_1
Y_{k-1}
Y_k
$proj(O, 1, \mathcal{Z}_1)$
$\cdots$
$proj(O, n, \mathcal{Z}_n)$
(b)

Figure 7: Edges added by the set constraint solver for the graph in Figure 6 using (a) the standard reduction and (b) the reduction with the clustering optimization

dant work during the computation of the graph closure. An exemplar of this structure is shown in Figure 6. There are two important features of this graph that we exploit. First, there is a long chain of nodes $y_2, \ldots, y_{k-1}$, none of which contain edges other than s-edges. Second, the nodes y_1 and y_k have a large number of predecessor $(_i$ edges and successor $)_i$ edges, respectively. This situation arises frequently in polymorphic flow analysis applications: the x_i nodes represent the inflow of actual arguments to the formal argument y_1 at call sites, and the z_i nodes represent the outflow of return values back to call sites from the returned value y_k.

The trick is to discover all the $S\langle x_i, z_i\rangle$ paths with as little redundant work as possible. To find these edges, the generic CFL algorithm discussed in Section 2.1 would add a total of nk new edges from each of the x_i's to each of the y_j's. Conceptually, this corresponds to analyzing the function body once per static call site.

One way to avoid this redundant work is to construct a so-called *summary edge* after analyzing the chain of y nodes once [10]. This optimization works as follows: the first path explored from an x_i node triggers a new search forward from node y_1. Analyzing the chain of y nodes causes $k-1$ edge additions from y_1 to each of the remaining y_i's. When the edge $S\langle y_1, y_k\rangle$ is finally discovered, it is marked as a summary edge, and y_1 is marked as having a summary edge. When searching forward for new paths from the other x_i nodes, the summary edge $S\langle y_1, y_k\rangle$ is used, avoiding repeated analysis of the y chain. In total, the summary edge approach adds a total of $n+k-1$ new edges to discover the $S\langle x_i, z_i\rangle$ edges. Conceptually, the summary edge approach corresponds to analyzing a function body once, and copying the summarized flow to each call site. The dashed edges in Figure 6 show the edges added by a closure algorithm with the summary edge optimization.

Our reduction as described may perform the same work as the naive CFL closure algorithm on this example. Figure 7(a) shows the worst-case edge additions performed on this example in inductive form. In inductive form, the number of edge additions depends on the ordering of the $\mathcal{Y}_i$ variables. In the worst case, the variable ordering might cause inductive form to represent all variable-variable edges as as successor edges (i.e. if $o(\mathcal{Y}_i) > o(\mathcal{Y}_{i+1})$ for every i in the chain). On the other hand, if the ordering is such that some of the variable-variable edges are represented as predecessor edges, inductive form can reduce the number of edge additions. Under the best variable ordering, inductive form adds the same number of edges as the summary edge optimization.

It is also possible to modify the reduction so that the number of edge additions is minimized regardless of the variable ordering. The key is to "cluster" the variables corresponding to the x_i nodes into a single n-ary constructed term instead of n unary constructed terms. The indices can be encoded by the position within the constructed term: for instance, the ith subterm represents the source end of an edge labeled $(_i$. Our representation of $)_i$ edges is modified as well, so that, e.g., the constraint $\mathcal{U} \subseteq proj(O, i, \mathcal{V})$ represents an edge $)_i\langle u, v\rangle$. Figure 7(b) shows the effect of the clustering optimization on the same example graph. The net effect of this optimization is the same as with summary edges: the edges $S\langle x_i, z_i\rangle$ are discovered with the addition of at most $n+k-1$ edges instead of nk edges. With some work, this reduction can be extended to handle PN-reachability and global nodes, though we omit the details here.

5.2.2 Cycle Elimination

One advantage of our reduction is that it preserves the structure of the input graph in a way that the Melski-Reps reduction does not. The constraint graphs produced by our reduction are isomorphic (modulo the representation of edges) to the original CFL graph. In fact, the connection is so close that cycle elimination, one of the key optimizations used in set constraint algorithms, is revealed to have an interpretation in the Dyck-CFL reachability problem.

Cycle elimination exploits the fact that variables involved in cyclic constraints (constraints of the form $\mathcal{X}_1 \subseteq \mathcal{X}_2 \subseteq \mathcal{X}_3 \ldots \subseteq \mathcal{X}_n \subseteq \mathcal{X}_1$) are equal in all solutions, and thus can be collapsed into a single variable. By reversing our reduction, we see that cyclic constraints correspond to cycles of the form:

$$x_1 \xrightarrow{s} x_2 \xrightarrow{s} \cdots \xrightarrow{s} x_n \xrightarrow{s} x_1$$

in the Dyck-CFL graph. The corresponding set variables are equivalent in all solutions; the CFL notion is that any node reaching one of the x_i nodes reaches every x_i node because we can always concatenate additional s terminals onto any word in the language and still derive a valid S-path. Hence, all the x_i's in the CFL graph can be collapsed and treated as a single node.

Note that the Melski-Reps reduction does not preserve such cycles: their reduction applied to CFL graph shown above produces constraints of the form $s(\mathcal{X}_i) \subseteq \mathcal{X}_{i-1}$ which do not expose a cyclic constraint for cycle elimination to simplify.

We validate these observations experimentally in Section 5.3 by showing the effect of partial online cycle elimination [4] on our implementation.

5.3 Experimental Results

In this section we compare an implementation of the reduction in Section 4 to a hand-written Dyck-CFL reachability implementation by Robert Johnson, which is based on an algorithm described in [10]. Johnson's implementation is customized to the polymorphic qualifier inference problem, and contains optimizations (including summary edges) that exploit the particular structure of the graphs that arise in this application.

Our implementation uses the BANSHEE analysis toolkit as the underlying set constraint solver [11]. We implemented the extensions for PN-reachability and global nodes as described in Section 5.1. In addition, all of the optimizations described in Section 5.2 are part of our implementation: we support clustering, and the BANSHEE toolkit uses inductive form and enables cycle elimination by default.

We used the C benchmark programs shown in Table 2 for our experiments. For each benchmark, the table lists the number of lines of code in the original source, the number of pre-processed lines of code, the number of distinct nodes in the CFL graph, the number of edges in the CFL graph, and the number of distinct indices (recall that this number corresponds to the number of function call sites). All experiments were performed on a dual-processor 550 MHz Pentium III with 2GB of memory running RedHat 9.0[4].

We ran CQUAL to generate a Dyck-CFL graph, and computed the closure of that graph using Johnson's Dyck-CFL reachability implementation and our own. We also ran our implementation with the clustering optimization enabled. Table 2 shows the analysis times (in seconds) for each experiment. The analysis times also include the time for CQUAL to parse the code and build the initial graph. Column 7 shows the time required for our implementation without cycle elimination. Column 8 shows times for the same implementation with partial online cycle elimination enabled. Column 9 shows times with cycle elimination and the clustering optimization. Finally, column 10 shows the times for Johnson's implementation, which acts as the gold standard. Figure 8 plots the analysis times for our implementation, normalized to Johnson's implementation. The results show that our implementation exhibits the same scaling behavior

[4]Though only one processor was actually used.

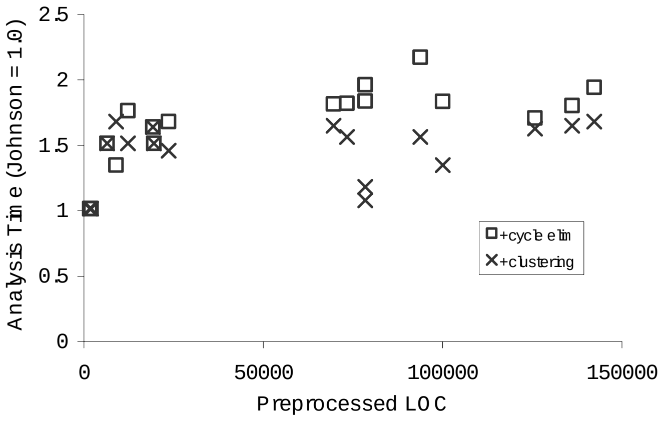

Figure 8: Results for the tainted/untainted experiment

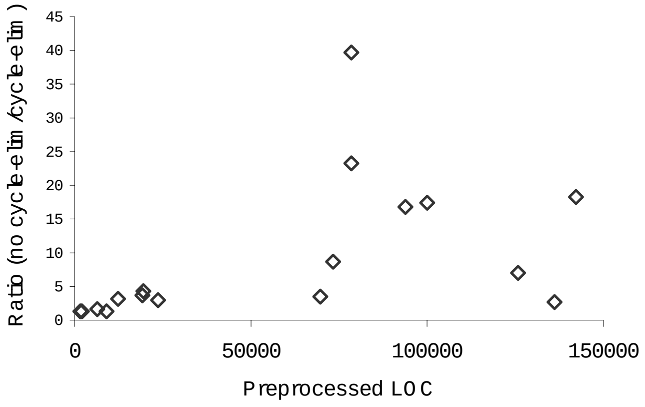

Figure 9: Speedup due to cycle elimination

as Johnson's. In all the benchmarks except one, the analysis time for our implementation remains within a factor of two of Johnson's implementation. Because parse time is a constant for all the implementations, factoring it out does have an affect on these ratios; however, we found that parse time accounts for a small fraction of the total analysis time. The clustering optimization improved a few benchmarks significantly, but did not seem to improve scalability overall.

Figure 9 compares the performance of our implementation (no clustering) with and without cycle elimination. We found that our benchmarks ran up to 40 times faster with cycle elimination enabled. These results suggest that online cycle elimination could be incorporated into the "standard" algorithms for Dyck-CFL reachability, an idea that has been implicitly suggested before [8].

6. RELATED WORK

We view our work as a practical follow-up to Melski and Reps' work, which shows that set constraints and CFL reachability are interconvertible [12, 13]. Our approach expands on the connection between Dyck-CFL reachability and set constraints, and illustrates the practical consequences of that connection.

Many researchers have formulated program analyses as CFL reachability problems. The authors first became aware of the connection from work by Reps et al., who formulated

Benchmark	LOC	Preproc	Nodes	Edges	Indices	BANSHEE(s)	+ cycle elim(s)	+ clusters(s)	johnson(s)
identd-1.0.0	385	1224	3281	1440	74	.25	.25	.25	.25
mingetty-0.9.4	441	1599	5421	1469	111	.25	.25	.25	.25
bftpd-1.0.11	964	6032	9088	37558	380	1	.75	.75	.5
woman-3.0a	2282	8611	10638	69171	450	1	1	1.25	.75
patch-2.5	7561	11862	20587	76121	899	5	1.75	1.5	1
m4-1.4	13881	18830	30460	268313	1187	11	3	3	2
muh-2.05d	4963	19083	19483	141550	684	9	2	2	1.5
diffutils-2.7	15135	23237	33462	281736	1191	10	4	3	2
uucp-1.04	32673	69238	91575	2007054	4725	43	14	12	8
mars_nwe-0.99	25789	72954	115876	1432531	4312	117	14	12	8
imapd-4.7c	31655	78049	388794	2402204	7714	1782	45	25	23
ipopd-4.7c	29762	78056	378085	2480718	7037	877	38	25	21
sendmail-8.8.7	44004	93383	126842	4173247	6076	454	28	20	13
proftpd-1.20pre10	23733	99604	184195	4048202	7206	561	33	24	18
backup-cffixed	39572	125350	139189	3657071	6339	139	21	20	12
apache-1.3.12	51057	135702	152937	6509818	5627	75	32	29	18
cfengine-1.5.4	39909	141863	146274	7008837	6339	597	33	29	17

Table 2: Benchmark data for all experiments

interprocedural dataflow analysis as a Dyck-CFL graph, and introduced an algorithm to answer the all-pairs reachability problem for these graphs [18]. Follow-up work introduced a demand-driven algorithm for solving this problem [10]. Many implementations of Dyck-CFL reachability are based on this algorithm. Rehof et al. show how to use CFL reachability as an implementation technique for polymorphic flow analysis [15]. CFL reachability is used as an alternative to repeated copying and simplification of systems of instantiation constraints. The first application of Rehof et al.'s work was to polymorphic points-to analysis [3]. Other applications of Dyck-CFL reachability include field-sensitive points-to analysis, shape analysis [16], interprocedural slicing [9], and debugging systems of unification constraints [2].

7. CONCLUSION

We have shown a reduction from Dyck-CFL reachability to set constraints, and shown that our technique can be applied in practice to the CFL reachability problems that arise in program analyses.

Acknowledgments

We would like to thank Robert Johnson for providing us with his implementation of CFL reachability, and David Melski, Tom Reps, and the anonymous reviewers for their helpful comments and suggestions.

8. REFERENCES

[1] A. Aiken and E. L. Wimmers. Solving Systems of Set Constraints. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, Santa Cruz, California, June 1992.

[2] V. Choppella and C. T. Haynes. Source-tracking Unification. In F. Baader, editor, *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 458–472. Springer, 2003.

[3] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *SAS '01: The 8th International Static Analysis Symposium*, Lecture Notes in Computer Science, Paris, France, July 16–18 2001. Springer-Verlag.

[4] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.

[5] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[6] N. Heintze. *Set Based Program Analysis.* PhD dissertation, Carnegie Mellon University, Department of Computer Science, Oct. 1992.

[7] N. Heintze and O. Tardieu. Demand-driven Pointer Analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[8] N. Heintze and O. Tardieu. Ultra-fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[10] S. Horwitz, T. Reps, and M. Sagiv. Demand Interprocedural Dataflow Analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 104–115. ACM Press, 1995.

[11] J. Kodumal. BANSHEE: A Toolkit for Building

Constraint-Based Analyses.
`http://bane.cs.berkeley.edu/banshee`.

[12] D. Melski and T. Reps. Interconvertbility of Set Constraints and Context-Free Language Reachability. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 74–89. ACM Press, 1997.

[13] D. Melski and T. Reps. Interconvertibility of a Class of Set Constraints and Context-Free-Language Reachability. *Theoretical Computer Science*, 248:29–98, 2000.

[14] C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[15] J. Rehof and M. Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, London, United Kingdom, Jan. 2001.

[16] T. Reps. Shape Analysis as a Generalized Path Problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program manipulation*, pages 1–11. ACM Press, 1995.

[17] T. Reps. Program Analysis Via Graph Reachability. In *Information and Software Technology*, pages 701–726, 1998.

[18] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, Jan. 1995.

[19] A. Rountev and S. Chandra. Off-line Variable Substitution for Scaling Points-to Analysis. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 47–56. ACM Press, 2000.

[20] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., Aug. 2001.

[21] Z. Su, M. Fähndrich, and A. Aiken. Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95. ACM Press, 2000.