Partial Online Cycle Elimination in Inclusion Constraint Graphs

Manuel Fähndrich* Jeffrey S. Foster* Zhendong Su* Alexander Aiken*

EECS Department
University of California, Berkeley
387 Soda Hall #1776
Berkeley, CA 94720-1776
{manuel,jfoster,zhendong,aiken}@cs.berkeley.edu

Abstract

Many program analyses are naturally formulated and implemented using inclusion constraints. We present new results on the scalable implementation of such analyses based on two insights: first, that online elimination of cyclic constraints yields orders-of-magnitude improvements in analysis time for large problems; second, that the choice of constraint representation affects the quality and efficiency of online cycle elimination. We present an analytical model that explains our design choices and show that the model’s predictions match well with results from a substantial experiment.

1 Introduction

Inclusion constraint systems have natural graph representations. For example, the constraints \( X \subseteq Y \subseteq Z \) are represented by nodes for the quantities \( X, Y, \) and \( Z \) and directed edges \( (X, Y) \) and \( (Y, Z) \) for the inclusions. Resolving the constraints corresponds to adding new edges to the graph to express relationships implied by, but not explicit in, the initial system. In this example, the transitive edge \( (X, Z) \) represents the implied constraint \( X \subseteq Z \).

The performance of constraint resolution can be improved by simplifying the constraint graph. Periodic simplification performed during resolution helps to scale to larger analysis problems [FA96, FF97, MW97], but performance is still unsatisfactory. One problem is deciding the frequency at which to perform simplifications to keep a well-balanced cost-benefit tradeoff. Simplification frequencies in past approaches range from once for an entire module to once for every program expression.

In this paper we show that cycle elimination in the constraint graph (a particular simplification) is one key to making inclusion constraint analyses scale to large problems with good performance. Cyclic constraints have the form \( X_1 \subseteq X_2 \subseteq X_3 \ldots \subseteq X_n \subseteq X_1 \) where the \( X_i \) are set variables. All variables on such a cycle are equal in all solutions of the constraints, and thus the cycle can be collapsed to a single variable.

We take an extreme approach to simplification frequency by performing cycle detection and elimination \\textit{online}, i.e., at every update of the constraint graph. At first glance, this approach seems overly expensive, since the best known algorithm for online cycle detection performs a full depth-first search for half of all edge additions [Shm83].

Our contribution is to show that \\textit{partial} online cycle detection can be performed cheaply by traversing only certain paths during the search for cycles. This approach is inspired by a non-standard graph representation called \\textit{inductive form} (IF) introduced in [AW93]. In practice, our approach requires constant time overhead on every edge addition and finds and eliminates about 80% of all variables involved in cycles. For our benchmarks, this approach radically improves the scaling behavior, making analysis of large programs practical. Furthermore, we provide an analytical model to explain the performance of particular graph representations.

Except ours, all implementations of inclusion constraint solvers are aware of employ a standard graph representation in which all edges are stored in adjacency lists and variable-variable edges always appear in successor lists. For example, the constraint \( X \subseteq Y \), between variables \( X \) and \( Y \), is represented as a successor edge from node \( X \) to node \( Y \).

Our measurements show that this \\textit{standard form} (SF), which is the one described in [Hei92] for use in set-based analysis (SBA), can also substantially benefit from partial online cycle elimination.

As our benchmark we study a points-to analysis for C [And94, SH97] implemented using both SF and IF. For large programs (more than 10000 lines), online cycle elimination reduces the execution time of our SF implementation by up to a factor of 13. Our implementation using IF and partial online cycle elimination outperforms SF with cycle elimination by up to a factor of 4, resulting in an overall speedup over standard implementations by up to 50.

Our measurement methodology uses a single well-engineered constraint solver to perform a number of exper-
iments using SF and IF with and without cycle elimination. We validate our results by comparing with Shapiro and Horwitz’s SF implementation (SH) of the same points-to analysis [SH97]. Experiments show that our implementation of points-to analysis using SF without cycle elimination closely matches SH on our benchmarks.

In Section 2, we define a language for set constraints, the particular constraint formalism we shall use. We also present the graph representations SF and IF and describe our cycle elimination algorithm. In Section 3 we describe the version of points-to analysis we study. Section 4 presents measurements illustrating the efficacy of our cycle elimination algorithm. Section 5 studies an analytical model that explains why IF can outperform SF. Finally, Section 6 presents related work, and Section 7 concludes.

2 Definitions

2.1 Set Constraints

In this paper we use a small subset of the full language of set constraints [HJ90, AW92]. Constraints in our constraint language are of the form $L \subseteq R$, where $L$ and $R$ are set expressions. Set expressions consist of set variables $X, Y, \ldots$ from a family of variables $\mathcal{V}$, terms constructed from unary constructors $c \in \text{Con}$, an empty set 0, and a universal set 1.

$$L, R \in se \quad ::= \quad X | c(se_1, \ldots, se_n) | 0 | 1$$

Each constructor $c$ is given a unique signature $S$, specifying the arity and variance of $c$. Intuitively, a constructor $c$ is covariant in an argument if the set denoted by a term $c(\ldots)$ becomes larger as the argument increases. Similarly, a constructor $c$ is contravariant in an argument if the set denoted by a term $c(\ldots)$ becomes smaller as the argument increases.

We define solutions to set constraints without restricting ourselves to a particular model\footnote{Standard models are the termset model [He92, Koz93] or the ideal model [AW93].} for set expressions. We simply assume that each constructor $c$ is also equipped with an interpretation $\phi_c$. Given a variable assignment $A$ of sets to variables, set expressions are interpreted as follows\footnote{The interpretation of 0 and 1 depends on the model and is not shown.}:

$$[X] A \equiv A(X)$$

$$[c(se_1, \ldots, se_n)] A \equiv \phi_c([se_1] A, \ldots, [se_n] A)$$

A solution to a system of constraints $\{L_i \subseteq R_i\}$ is a variable assignment $A$ such that $[L_i] A \subseteq [R_i] A$ for all $i$.

2.2 Constraint Graphs

Solving a system of constraints involves computing an explicit solved form of all solutions or of a particular solution. We study two distinct solved forms: Standard form SF represents the least solution explicitly and is commonly used for implementing SBA [He92]. Inductive form IF computes a representation of all solutions and is usually used with more expressive constraints and in type-based analyses [AW93, MW97]. As an aside, it is worth noting that for some analysis problems we require a representation of all solutions because no least solution exists. For the purposes of comparing the two forms we shall implicitly assume throughout that with respect to the variables of interest constraint systems have least solutions.

The solved form of a constraint system is a directed graph $G = (V, E)$ closed under a transitive closure rule, where the edges $E$ represent atomic constraints and the vertices $V$ are variables, sources, and sinks. Sources are constructed terms appearing to the left of an inclusion, and sinks are constructed terms appearing to the right of an inclusion. For the purposes of this paper, we treat 0 and 1 as constructors.

A constraint is atomic if it is one of the three forms

$$X \subseteq Y \quad \text{variable-variable constraint}$$

$$c(\ldots) \subseteq X \quad \text{source-variable constraint}$$

$$X \subseteq c(\ldots) \quad \text{variable-sink constraint}$$

We use the set of resolution rules $R$ shown in Figure 1 to transform constraints into atomic form. Each rule states that the system of constraints on the left has the same solutions as the system on the right. In a resolution engine these rules are used as left-to-right rewrite rules.

The next sections describe how constraint graphs are represented and closed by the two forms SF and IF. Both forms use adjacency lists to represent edges. Every edge $(X, Y)$ in a graph is represented exclusively either as a predecessor edge $(X \in \text{pred}(Y))$ or as a successor edge $(Y \in \text{succ}(X))$.

2.3 Standard Form

Standard form (SF) represents edges in constraint graphs as follows:

$$X \subseteq Y \quad X \rightarrow Y \quad \text{successor edge}$$

$$c(\ldots) \subseteq X \quad c(\ldots) \cdots \longrightarrow X \quad \text{predecessor edge}$$

$$X \subseteq c(\ldots) \quad X \longrightarrow c(\ldots) \quad \text{successor edge}$$

We draw predecessor edges in graphs using dotted arrows and successor edges using plain arrows. New edges are added by the transitive closure rule:

$$L \longrightarrow X \longrightarrow R \quad \Rightarrow \quad L \subseteq R$$

Given a predecessor edge $L \longrightarrow X$ and a successor edge at $X \longrightarrow R$, a new constraint $L \subseteq R$ is generated. We generate a constraint instead of an edge because rules in Figure 1
may apply. Note that in this case, \( L \) is always of the form 
\( c(\ldots) \). This closure rule combined with rules \( \mathcal{R} \) of Figure 1 produces a final graph containing an explicit form of the least solution \( LS \) of the constraints [Hei92].

SF makes the least solution explicit by propagating sources forward to all reachable variables via the closure rule. The particular choice of successor and predecessor representation is motivated by the need to implement the closure rule locally. Given a variable \( X \), the closure rule must be applied exactly to all combinations of predecessor and the successor edges of \( X \).

Figure 2 shows an example system of constraints, the initial SF graph, and the resulting closed SF graph (left). The example assumes that set expressions \( L_1 \ldots L_k \) are sources and \( R_1 \ldots R_m \) are sinks. The closure of the standard form adds transitive edges from each source \( L_i \) to all variables reachable from \( X \) i.e., \( Y \subseteq Z \). Note that the edges from \( L_1 \ldots L_k \) to \( Z \) are added \( l \) times each, namely along all \( l \) edges \( Y \rightarrow Z \). The total work of closing the graph is \( 2lk \) edge additions, of which \( k(l-1) \) additions are redundant, plus the work resulting from the \( km \) constraints \( L_i \subseteq R_j \) (not shown).

To see why cycle elimination can asymptotically reduce the amount of work to close a graph, suppose there is an extra edge \( Z \rightarrow X \) in Figure 2, forming a strongly connected component \( X, Y_1, \ldots, Y_j, Z \). If we collapse this component before adding the transitive edges \( L_i \rightarrow Y_j \), none of the 2\( kl \) transitive edge additions \( L_i \rightarrow Y_j \) are performed (the \( km \) constraints \( L_i \subseteq R_j \) are still produced of course).

2.4 Inductive Form

Inductive form (IF) exploits the fact that a variable-variable constraint \( X \subseteq Y \) can be represented either as a successor edge \( (Y \in \text{succ}(X)) \) or as a predecessor edge \( (X \in \text{pred}(Y)) \). The representation for a particular edge is chosen as a function of a fixed total order \( o : \text{Vars} 
\rightarrow \mathbb{N} \) on the variables. Edges in the constraint graph are represented as follows:

\[
\begin{align*}
X \subseteq Y & \quad \text{if } o(X) > o(Y) \\
X \rightarrow Y & \quad \text{a successor edge} \\
X \leftarrow Y & \quad \text{if } o(X) < o(Y) \\
& \quad \text{a predecessor edge}
\end{align*}
\]

The choice of the order \( o(\cdot) \) can have substantial impact on the size of the closed constraint graph and the amount of work required for the closure. We assume that the order \( o(\cdot) \) is randomly chosen. Choosing a good order is hard, and we have found that a random order performs as well or better than any other order we picked.

The other two kinds of edges are represented as in standard form, and the closure rule also remains unchanged:

\( L \cdots X \rightarrow R \Rightarrow L \subseteq R \)

Notice that \( L \) may be a source or a variable—unlike SF, where \( L \) is always a source. In IF the closure rule can therefore directly produce transitive edges between variables. (This is not to say that the closure of SF does not produce new edges between variables, but for SF such edges always involve the resolution rules \( \mathcal{R} \) of Figure 1.) The closure rule combined with the resolution rules \( \mathcal{R} \) produces a final graph in inductive form [AW93].

The least solution of the constraints is not explicit in the closed inductive form. However, it is easily computed as follows:

\[
LS(Y) = \{c(\ldots) \mid c(\ldots) \rightarrow Y \} \cup \bigcup_{X \rightarrow Y} LS(X) \tag{1}
\]
By the ordering \( o() \), we have \( o(X) < o(Y) \) for all \( X \rightarrow Y \). Thus there exists a variable \( Z_i \) with minimum index \( o(Z_i) \) that has no predecessor edges to any other variables and \( LS(Z_i) = \{ o(\ldots ) | o(\ldots ) \cdots \rightarrow Z_i \} \). Then \( LS(Z_i) \) is computed using \( LS(Z_j) \) for \( j < i \) and (1). The time to compute \( LS \) for all variables is \( O(pk) \) worst case, where \( p \) is the number of edges and \( k \) is the number of distinct sources in the final graph. In the rest of the paper, solving a system of constraints under IF always includes the computation of the least solution.

The right side of Figure 2 shows the initial and final graph for the example constraints using IF. Note that some variable-variable edges in IF are predecessor edges (dotted), whereas all variable-variable edges in SF are successor edges (solid). The ordering on the variables assumed in the example is \( o(X) < o(Z) < o(Y) \). Note the extra variable-variable edge \( X \rightarrow Z \) added by the closure rule for IF. As a result of this edge, the closure of IF adds edges from \( X \) to all \( R_i \). Each of the variables \( Y_1, \ldots, Y_l, Z \) has a single predecessor edge to \( X \), and thus their least solution is equal to \( LS(X) = \{ L_1, \ldots, L_k \} \). The total work of closing the graph is \( t + m \) edge additions, of which \( t - 1 \) additions are redundant, namely the addition of edge \( X \rightarrow Z \) through all \( Y_i \), plus the work for the \( km \) transitive constraints \( L_i \subseteq R_j \) (not shown). The work to compute the least solution is proportional to \( f \).

### 2.5 Cycle Detection

In this subsection we describe our cycle detection algorithm.

**Definition 2.1 (Path)** A path of length \( k \) from a vertex \( u \) to a vertex \( v \) in a constraint graph \( G = (V, E) \) is a sequence of vertices \( (v_0, \ldots, v_k) \), such that \( u = v_0, v_i = v_{i-1}, v_{i-1} \in E \) or \( v_{i-1} \rightarrow v_i \in E \) for \( i = 1, \ldots, k \). A path is simple if all vertices on the path are distinct.

**Definition 2.2 (Chain)** A chain in a constraint graph is a simple path \( (X_0, \ldots, X_k) \) consisting entirely of successor edges \( X_{i-1} \rightarrow X_i \) for \( i = 1, \ldots, k \) (a successor chain), or consisting entirely of predecessor edges \( X_i \rightarrow X_{i-1} \) for \( i = 1, \ldots, k \) (a predecessor chain).

A path \((X_0, \ldots, X_k)\) forms a cycle if \( X_0 = X_k \) and \( k \geq 1 \). As we show in Section 4, cycles in constraint graphs are a major contributor to constraint resolution times. It is thus important to detect and eliminate cycles. Cycles can always be replaced with a single variable, since all variables on a cycle must be equal in all solutions of the constraints.

**Figure 3:** Algorithms for cycle detection

```cpp
def insert_succeed_edge(vertex from, vertex to):
    // variable vertices: o(from) > o(to)
    if (result.chain(from, to)) { // Cycle found
        collapse_cycle(...);
    }
    else
        insert_into_successor_list(from, to);

def pred_chain(vertex from, vertex to):
    // TRUE if pred. chain to \( \rightarrow \) from
    if (from == to) return (TRUE);
    else {
        mark(from);
        for each v in predecessors of from
            if (!marked(v) && o(v) < o(from))
                if (pred_chain(v, to))
                    return (TRUE);
        return (FALSE);
    }
```

Figure 4: A cyclic graph in IF

Our algorithm (Figure 3) for online cycle elimination is a straightforward implementation of the following idea. When adding a successor edge \( X \rightarrow Y \), we search (using pred_chain) along all predecessor edges starting from \( X \) for a predecessor chain \( Y \rightarrow \cdots \rightarrow X \). Similarly, if we add a predecessor edge \( X \rightarrow \cdots \rightarrow Y \), we search (using succ_chain, not shown) along all successor edges starting from \( Y \) for a successor chain \( Y \rightarrow \cdots \rightarrow X \). If such a chain exists, then we have found a cycle that can be eliminated. The search algorithm on the right in Figure 3 differs from depth-first-search merely in that the next visited vertex must be less than the current vertex in the variable order \( o(\cdot) \). Note that for IF this condition is already implied by the graph representation; we include it for clarity and to make the algorithm work for SF. Detection for SF is slightly different since all variable-variable edges in SF are successors. Consequently, when adding a successor edge \( X \rightarrow Y \), we search (using succ_chain) along all successor edges starting from \( Y \) for a successor chain \( Y \rightarrow \cdots \rightarrow X \). The condition that we only follow successor edges if they point to lower indexed variables is crucial for SF. Without it, a full depth-first-search is performed at every graph update, which is impractical. Restricting the search to edges pointing to lower indexed variables reduces search time but results in only partial cycle detection.

For IF, cycle detection not only depends on the order \( o(\cdot) \) but also on the order in which edges are added to the graph. Consider the example in Figure 4. Our approach detects this cycle only if the successor edge \( X_3 \rightarrow X_1 \) is added last, since in this case, the predecessor chain \( X_1 \rightarrow \cdots \rightarrow X_3 \) is found. If the cycle is closed by adding either of the other edges the cycle is not detected. However, the closure of IF adds a transitive edge \( X_1 \rightarrow X_2 \) and the sub-cycle \((X_1, X_2)\) is detected in all cases. It is a theorem that for any ordering of variables, IF exposes at least a two-cycle for every non-trivial strongly connected component (SCC). Thus, using inductive form guarantees at least part of every non-trivial SCC is eliminated by our method; this result does not hold for SF.

\(^3\)A non-trivial strongly connected component consists of at least two vertices.
Once a cycle is found, we must collapse it to obtain any performance benefits in the subsequent constraint resolution. Collapsing a cycle involves choosing a witness variable on the cycle (we use the lowest indexed variable to preserve inductive form), redirecting the remaining variables on the cycle to the witness (through forwarding pointers), and combining the constraints of all variables on the cycle with those of the witness.

Finally, note that although some cycles may be found in the initial constraints, many cycles only arise during resolution through the application of the resolution rules \( R \). In the majority of our benchmarks, less than 20\% of the variables that are in strongly connected components in the final graph also appear in strongly connected components in the initial graph.

### 3 Case Study: Andersen’s Points-to Analysis

For a C program, points-to analysis computes a set of abstract memory locations (variables and heap) to which each expression could point. Andersen’s analysis computes a points-to graph [And94]. Graph nodes represent abstract memory locations, and there is an edge from a node \( x \) to a node \( y \) if \( x \) may contain a pointer to \( y \). Informally, Andersen’s analysis begins with some initial points-to relationships and closes the graph under the rule:

For an assignment \( e_1 = e_2 \), anything in the points-to set for \( e_2 \) must also be in the points-to set for \( e_1 \).

Figure 5 shows the points-to graph computed by Andersen’s analysis for a simple C program.

#### 3.1 Formulation using Set Constraints

Andersen’s set formulation of points-to graphs consists of a set of abstract locations \( \{l_1, \ldots, l_n\} \), together with set variables \( X_1, \ldots, X_n \), denoting the set of locations pointed to by \( l_1, \ldots, l_n \). The example in Figure 5 has the set formulation:

\[
\begin{align*}
X_a &= \{l_b, l_c\} \\
X_b &= \{l_d\} \\
X_c &= \{l_b\}
\end{align*}
\]

The association between a location \( l \) and its points-to set \( X_l \) is implicit in Andersen’s formulation and results in an ad-hoc resolution algorithm. We use a different formulation that makes this association explicit and enables us to use a generic set constraint solver. We model locations by pairing location names and points-to set variables with a constructor \( \text{ref}((l_x, X_l)) \) akin to reference types in languages like ML [MT90].

Unlike the type system of ML, which is equality-based, we need inclusion constraints. It is well known that subtyping of references is unsound in the presence of update operations (e.g., Java arrays [GJS96]). A sound approach is to turn inclusions between references into equality for their contents: \( \text{ref}(X) \subseteq \text{ref}(Y) \Leftrightarrow X = Y \).

We adapt this technique to a purely inclusion-based system using a novel approach. We intuitively treat a reference \( l_x \) as an object with a location name and two methods \( \text{get}: \text{void} \rightarrow X_l \) and \( \text{set}: X_l \rightarrow \text{void} \), where the points-to set of the location acts both as the range of the get function and the domain of the set function. Updating a location corresponds to applying the set function to the new value. Dereferencing a location corresponds to applying the get function.

Translating this intuition, we add a third argument to the \( \text{ref} \) constructor that corresponds to the domain of the set function, and is thus contravariant. A location \( l_x \) is then represented by \( \text{ref}(l_x, X_l, \overline{X}_l) \) (to improve readability we overline contravariant arguments). To update an unknown location \( \tau \) with a set \( T \), it suffices to add a constraint \( \tau \subseteq \text{ref}(1, 1, T) \). For example, if \( \text{ref}(l_x, X_l, \overline{X}_l) \subseteq \tau \), then the transitive constraint \( \text{ref}(l_x, X_l, \overline{X}_l) \subseteq \text{ref}(1, 1, \overline{T}) \) is equivalent to \( T \subseteq X_l \), (due to contravariance), which is the desired effect. Dereferencing is analogous, but involves the covariant points-to set of the \( \text{ref} \) constructor.

To formally express Andersen’s points-to graph, we must associate with each location \( l_x \) a set variable \( \overline{Y}_l \) for the set of abstract location names and a constraint \( X_l \subseteq \text{ref}(\overline{Y}_l, 1, 0) \) that constrains \( \overline{Y}_l \) to be a superset of all names of locations in the points-to set \( X_l \). The points-to graph is then defined by the least solution for \( \overline{Y}_l \). In our implementation we avoid using the location names \( l_x \) and the variables \( \overline{Y}_l \), and instead derive the points-to graph directly from the constraints.

#### 3.2 Constraint Generation

Figure 6 gives a subset of the constraint-generation rules for Andersen’s analysis. For the full set of rules, see [FA97]. The rules assign a set expression to each program expression and generate a system of set constraints as side conditions. The solution to the set constraints describes the points-to graph of the program. We write \( \tau \) for set expressions denoting locations. To avoid separate rules for L- and R-values, we infer sets denoting L-values for every expression. In (Var), the type \( \text{ref}(l_x, X_l, \overline{X}_l) \) associated with \( x \) therefore denotes the location of \( x \) and not its contents.

We briefly describe the other rules in Figure 6. The
address-of operator (Addr) adds a level of indirection to its operand by adding a ref constructor. The dereferencing operator (Deref) does the opposite, removing a ref and making the fresh variable T a superset of the points-to set of T. The second constraint in the assignment rule (Asst) transforms the right-hand side T0 from an L-value to an R-value T2, as in (Deref) (recall these rules infer sets representing L-values). The first constraint T1 ⊆ ref(1, T1) makes T1 a subset of the points-to set of T1. The final constraint T2 ⊆ T1 expresses exactly the intuitive meaning of assignment: the points-to set T1 of the left-hand side contains at least the points-to set T2 of the right-hand side. For example, the first statement of Figure 6, a = b, generates the constraints T1 = ref(l1, X1, X1), and so T1 ⊆ X1. and T2 = ref(0, ref(l1, X1, X1), ...) ∈ ref(1, T2, 0), and so ref(l1, X1, X1) ⊆ T2. The final constraint T2 ⊆ T1 implies the desired effect, namely ref(l1, X1, X1) ⊆ X1.

4 Measurements

In this section we compare the commonly used implementation strategy of set-based analysis [Hei92], which represents constraint graphs in standard form (SF), with the inductive form (IF) of [AW93]. We give empirical evidence that cycles in the constraint graph are the key inhibitors to scalability for both forms and that our online cycle elimination is cheap and improves the running times of both forms significantly. Using online cycle elimination, analysis times using inductive form come close to analysis times with perfect and zero-cost cycle elimination (measured using an oracle to predict cycles). Furthermore, on medium to large programs IF outperforms SF by factors of 2–4. This latter result is surprising, and we explore it on a more analytical level in Section 5.

Our measurements use the C benchmark programs shown in Table 1. For each benchmark, the table lists the number of abstract syntax tree (AST) nodes, the number of lines in the preprocessed source, the number of set variables, the total number of distinct nodes in the graph (sources, variables, and sinks), and the number of edges in the initial

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AST Nodes</th>
<th>LOC</th>
<th>Total Lines</th>
<th>Initial Nodes</th>
<th>Initial Edges</th>
<th>@Vars</th>
<th>@SCC max</th>
<th>@Vars</th>
<th>@SCC max</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallc</td>
<td>765</td>
<td>368</td>
<td>111</td>
<td>36</td>
<td>141</td>
<td>5</td>
<td>7</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>diffpush</td>
<td>923</td>
<td>310</td>
<td>137</td>
<td>303</td>
<td>207</td>
<td>10</td>
<td>2</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>smugmam</td>
<td>478</td>
<td>344</td>
<td>219</td>
<td>360</td>
<td>216</td>
<td>16</td>
<td>5</td>
<td>23</td>
<td>9</td>
</tr>
<tr>
<td>generic</td>
<td>1464</td>
<td>374</td>
<td>264</td>
<td>345</td>
<td>340</td>
<td>6</td>
<td>3</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>ioe</td>
<td>57</td>
<td>574</td>
<td>335</td>
<td>315</td>
<td>329</td>
<td>6</td>
<td>3</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>oil</td>
<td>3227</td>
<td>445</td>
<td>207</td>
<td>293</td>
<td>176</td>
<td>4</td>
<td>1</td>
<td>27</td>
<td>9</td>
</tr>
<tr>
<td>f0</td>
<td>2077</td>
<td>1179</td>
<td>510</td>
<td>859</td>
<td>487</td>
<td>0</td>
<td>0</td>
<td>70</td>
<td>5</td>
</tr>
<tr>
<td>compare</td>
<td>2693</td>
<td>1250</td>
<td>1024</td>
<td>1804</td>
<td>1026</td>
<td>26</td>
<td>7</td>
<td>164</td>
<td>8</td>
</tr>
<tr>
<td>printf</td>
<td>2565</td>
<td>1260</td>
<td>1024</td>
<td>1804</td>
<td>1026</td>
<td>26</td>
<td>7</td>
<td>164</td>
<td>8</td>
</tr>
<tr>
<td>compiler</td>
<td>2574</td>
<td>1593</td>
<td>1278</td>
<td>3293</td>
<td>1352</td>
<td>12</td>
<td>6</td>
<td>29</td>
<td>9</td>
</tr>
<tr>
<td>assembler</td>
<td>6246</td>
<td>2887</td>
<td>1811</td>
<td>2885</td>
<td>1497</td>
<td>9</td>
<td>4</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>MLtypecheck</td>
<td>6774</td>
<td>4003</td>
<td>1823</td>
<td>3096</td>
<td>1908</td>
<td>29</td>
<td>0</td>
<td>290</td>
<td>15</td>
</tr>
<tr>
<td>express</td>
<td>4817</td>
<td>2216</td>
<td>1373</td>
<td>2069</td>
<td>1445</td>
<td>36</td>
<td>15</td>
<td>236</td>
<td>21</td>
</tr>
<tr>
<td>simulator</td>
<td>10946</td>
<td>4230</td>
<td>3764</td>
<td>4578</td>
<td>2317</td>
<td>6</td>
<td>2</td>
<td>290</td>
<td>9</td>
</tr>
<tr>
<td>le = 177</td>
<td>10179</td>
<td>1246</td>
<td>3067</td>
<td>6113</td>
<td>3389</td>
<td>26</td>
<td>12</td>
<td>357</td>
<td>14</td>
</tr>
<tr>
<td>flex 0.4.7</td>
<td>18628</td>
<td>5761</td>
<td>7131</td>
<td>12313</td>
<td>6283</td>
<td>24</td>
<td>10</td>
<td>44</td>
<td>9</td>
</tr>
<tr>
<td>prime0</td>
<td>31240</td>
<td>25129</td>
<td>7009</td>
<td>11033</td>
<td>6007</td>
<td>63</td>
<td>24</td>
<td>775</td>
<td>23</td>
</tr>
<tr>
<td>make-3.71</td>
<td>26889</td>
<td>15319</td>
<td>8399</td>
<td>15314</td>
<td>6994</td>
<td>156</td>
<td>63</td>
<td>466</td>
<td>77</td>
</tr>
<tr>
<td>inform-6</td>
<td>28874</td>
<td>12456</td>
<td>5956</td>
<td>15058</td>
<td>6887</td>
<td>34</td>
<td>16</td>
<td>505</td>
<td>19</td>
</tr>
<tr>
<td>tax-1.11.2</td>
<td>41407</td>
<td>18312</td>
<td>6095</td>
<td>9733</td>
<td>6459</td>
<td>347</td>
<td>89</td>
<td>833</td>
<td>19</td>
</tr>
<tr>
<td>acme-3.0.5</td>
<td>42925</td>
<td>20423</td>
<td>12806</td>
<td>18706</td>
<td>8621</td>
<td>108</td>
<td>46</td>
<td>864</td>
<td>21</td>
</tr>
<tr>
<td>cws-1.3</td>
<td>51223</td>
<td>31195</td>
<td>13848</td>
<td>20773</td>
<td>10497</td>
<td>246</td>
<td>82</td>
<td>750</td>
<td>42</td>
</tr>
<tr>
<td>eqn-1.2</td>
<td>58774</td>
<td>35105</td>
<td>18329</td>
<td>13273</td>
<td>9740</td>
<td>54</td>
<td>37</td>
<td>1106</td>
<td>43</td>
</tr>
<tr>
<td>expresso</td>
<td>56938</td>
<td>27083</td>
<td>14806</td>
<td>19067</td>
<td>12371</td>
<td>233</td>
<td>140</td>
<td>1772</td>
<td>170</td>
</tr>
<tr>
<td>gawk-3.3.0</td>
<td>71051</td>
<td>27083</td>
<td>14806</td>
<td>19067</td>
<td>12371</td>
<td>233</td>
<td>140</td>
<td>1772</td>
<td>170</td>
</tr>
<tr>
<td>gnu-cp-2.0.5</td>
<td>87291</td>
<td>50699</td>
<td>13461</td>
<td>21837</td>
<td>11057</td>
<td>108</td>
<td>91</td>
<td>1576</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 1: Benchmark data common to all experiments

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF-Plain</td>
<td>Standard form, no cycle elimination</td>
</tr>
<tr>
<td>IF-Plain</td>
<td>Inductive form, no cycle elimination</td>
</tr>
<tr>
<td>SF-Oracle</td>
<td>Standard form, with full (oracle) cycle elimination</td>
</tr>
<tr>
<td>IF-Oracle</td>
<td>Inductive form, with full (oracle) cycle elimination</td>
</tr>
<tr>
<td>SF-Online</td>
<td>Standard from, using IF online cycle elimination</td>
</tr>
<tr>
<td>IF-Online</td>
<td>Inductive form, with online cycle elimination</td>
</tr>
</tbody>
</table>

Table 4: Experiments

4Not all benchmarks ran through SH.
Table 2: Benchmark data for I-F, SF-Plain, SF-Oracle, and SF-Oracle

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AS: I</th>
<th>SF: I</th>
<th>SF-O: I</th>
<th>SF-O: Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>abalone</td>
<td>10</td>
<td>405</td>
<td>16</td>
<td>216</td>
</tr>
<tr>
<td>diff.maid</td>
<td>925</td>
<td>133</td>
<td>7</td>
<td>685</td>
</tr>
<tr>
<td>congress</td>
<td>148</td>
<td>14</td>
<td>96</td>
<td>44</td>
</tr>
<tr>
<td>genecl</td>
<td>31</td>
<td>26</td>
<td>31</td>
<td>21</td>
</tr>
<tr>
<td>leu</td>
<td>32</td>
<td>28</td>
<td>43</td>
<td>26</td>
</tr>
<tr>
<td>sis</td>
<td>39</td>
<td>39</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>uil</td>
<td>22</td>
<td>29</td>
<td>30</td>
<td>26</td>
</tr>
<tr>
<td>comp</td>
<td>33</td>
<td>29</td>
<td>31</td>
<td>21</td>
</tr>
<tr>
<td>compiler</td>
<td>32</td>
<td>32</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>equinst</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>simulator</td>
<td>29</td>
<td>29</td>
<td>43</td>
<td>26</td>
</tr>
<tr>
<td>mem-177</td>
<td>74</td>
<td>74</td>
<td>43</td>
<td>26</td>
</tr>
<tr>
<td>mem-1312</td>
<td>74</td>
<td>74</td>
<td>43</td>
<td>26</td>
</tr>
<tr>
<td>flex-2.47</td>
<td>98</td>
<td>98</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>prime</td>
<td>356</td>
<td>356</td>
<td>43</td>
<td>26</td>
</tr>
<tr>
<td>mem-3.2</td>
<td>74</td>
<td>74</td>
<td>43</td>
<td>26</td>
</tr>
<tr>
<td>mem-3.1</td>
<td>74</td>
<td>74</td>
<td>43</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 3: Benchmark data for I-Online and SF-Oracle

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>I-Online</th>
<th>SF-Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>abalone</td>
<td>10</td>
<td>405</td>
</tr>
<tr>
<td>diff.maid</td>
<td>925</td>
<td>133</td>
</tr>
<tr>
<td>congress</td>
<td>148</td>
<td>14</td>
</tr>
<tr>
<td>genecl</td>
<td>31</td>
<td>26</td>
</tr>
<tr>
<td>leu</td>
<td>32</td>
<td>28</td>
</tr>
<tr>
<td>sis</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>uil</td>
<td>22</td>
<td>29</td>
</tr>
<tr>
<td>comp</td>
<td>33</td>
<td>29</td>
</tr>
<tr>
<td>compiler</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>equinst</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>simulator</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>mem-177</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>mem-1312</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>flex-2.47</td>
<td>98</td>
<td>98</td>
</tr>
<tr>
<td>prime</td>
<td>356</td>
<td>356</td>
</tr>
<tr>
<td>mem-3.2</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>mem-3.1</td>
<td>74</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 3: Benchmark data for I-Online and SF-Oracle

On an oracle. Whenever a fresh set variable is created, the oracle predicts to which strongly connected component the variable will eventually belong. We substitute the witness variable of that component for the fresh variable. As a result, the oracle experiment uses only a single variable (witness) for each strongly connected component, and thus the graphs are acyclic.

Since the oracle experiments avoid all unnecessary work related to cycles in the constraint graph (perfect cycle elimination), they provide lower bounds for the last two experiments, I-Online and SF-Oracle, which use the online cycle detection and elimination algorithm described in Section 2.5. Furthermore, the oracle experiments directly compare the graph representations of I-F and SF, independently of cycle elimination.

Table 2 shows the results for the first four experiments. For each benchmark and experiment, we report the number of edges in the final graph, the total number of edge additions (Work) including redundant ones, and the execution time in seconds. Note the large number of redundant edge additions for SF-Plain and I-F. All experiments were performed using a single processor on a SPARC Enterprise 5000. The reported CPU times are best out of three runs. As mentioned in Section 2.4, all reported times for I-F include the time to compute the least solution. Figure 7 plots the analysis time for both SF-Plain and I-F without cycle elimination against the number of AST nodes of the parsed program. As the size exceeds 15000 AST nodes there are many benchmarks where the analysis becomes impractical. Without cycle elimination, SF generally outperforms I-F because cycles add many redundant variable-variable edges in I-F that lead to redundant work.

The low numbers for the oracle runs I-F, Oracle, and SF-Oracle are shown in Table 2 to show that the bulk of work and execution time is attributable to strongly connected components in the constraint graph. Without cycles, the points-to analysis scales very well for both I-F and SF. Our oracle approach failed for the three programs, screen, gawk, and povray, hence the missing points.

Table 3 reports the measurement results for the online cycle elimination experiments. In addition to the information shown for the plain and oracle experiments, the table contains the number of variables that were eliminated.
through cycle detection. Online cycle elimination is very effective for medium and large programs. Figure 8 plots the analysis times for online cycle elimination and the oracle experiments (note the scale change). The fastest analysis times are achieved by IF-Oracle, followed by SF-Oracle, IF-Online, and then SF-Online. IF-Online stays relatively close to the oracle times, while SF-Online performs somewhat worse. This indicates that while our cycle detection algorithm is not perfect, it comes close.

Figure 9 shows the total speedup of our approach over standard implementations (IF-Online over SF-Plain), and the speedup obtained solely through online cycle elimination (SF-Online over SF-Plain). To show that our techniques help scaling, we plot the speedups vs. the absolute execution time of SF-Plain. As the execution time of the standard implementation grows, our speedup also grows. For very small programs, the cost of cycle elimination outweighs the benefits, but for medium and large programs, online cycle elimination improves analysis times substantially, for large programs by more than an order of magnitude.

The performance benefit of inductive over standard form is illustrated more clearly in Figure 10. In this plot, we can see that IF-Online is consistently faster for medium and large-sized programs (at least 10,000 AST nodes) than SF-Online. For large programs the difference is significant, with IF-Online outperforming SF-Online by over a factor of 3.8 for the largest program. For very small programs, IF is at most 90% slower than SF, which in absolute times means only fractions of seconds.

We can explain the performance difference of IF and SF by comparing the fraction of variables on cycles found by IF-Online and SF-Online (Figure 11). Throughout, SF finds only about half as many variables on cycles as IF, and the remaining cycles slow down SF. One reason for this difference is that for SF, the cycle detection only searches successor chains. The analog to predecessor chains in SF are increasing chains. Searching increasing chains in SF results in a higher detection rate (57%), but the much higher cost outweighs any benefits.

Our model in Section 5 explains why SF finds fewer cycles. The probability of finding chains of length greater than

---

\footnote{The outlier is the program \texttt{flex}; although \texttt{flex} is a large program, it contains large initialized arrays. Thus as far as points-to analysis is concerned, it actually behaves like a small program.}
2 is small. Thus cycles of larger size are detected with small probability. SF counters this trend by adding transitive variable-variable edges, thereby shortening cycle lengths.

5 An Analytical Model

In the last section, we saw that |F-Online and |F-Oracle both outperform SF-Online and SF-Oracle respectively, and that the simple online cycle elimination strategy is very effective, especially for |F. In this section we analytically compare the two different representations and answer three questions:

(Q1) Why is |F a better representation than SF?

(Q2) Why is partial online cycle elimination fast?

(Q3) Why is the cycle elimination strategy more effective for |F?

To answer these questions analytically, we need a tractable model of constraint graphs. We use the following simplifications and assumptions:

- We assume that graph closure adds no edges through the resolution rules $R$. That is, we only consider edges added directly through the graph closure rule.
- We consider random graphs $G = (V, E)$ with $n$ variable nodes, $m$ source or sink nodes, and we assume for all pairs of distinct nodes $u$ and $v$ there is an edge $(u, v) \in E$ with probability $p$, for some constant $p$.
- We consider only edges added through simple paths. Thus, the results correspond to the cases where we have perfect cycle detection i.e., |F-Oracle and SF-Oracle.

These are strong assumptions. Nevertheless, this model predicts our measurements quite well. The following two theorems summarize the results in this section.

**Theorem 5.1** For random graphs with $p = \frac{1}{n}$ and ratio $\frac{m}{n} = \frac{2}{3}$, the expected number of edge additions for SF is approximately 2.5 times more than that for |F.

**Theorem 5.2** For random graphs with $p = \frac{2}{n}$, the expected number of variable nodes reachable through predecessor or successor chains in |F from any given node is no more than 2.2.

The parameters $p$ and $\frac{m}{n}$ are taken from our experiments described in Section 4. The probabilities $p = \frac{1}{n}$ and $p = \frac{2}{n}$ are the approximate densities of the initial and final |F graphs, respectively.

Theorem 5.1 answers the first question (Q1). It explains why SF-Oracle does on average 4.1 times more work than |F-Oracle. The second question (Q2) is answered by Theorem 5.2. We expect partial online cycle detection to follow very few edges. We observe empirically that the number of reachable variables is close to two. To answer the third question (Q3), notice that since cycle detection searches chains in order of variable index, the probability of detecting a long cycle is exponentially small. However, in |F edges between variables are added to the constraint graph, thus shortening some long cycles and increasing the probability of detecting cycles. Although the same idea for detecting cycles can be applied to SF, it does not work as well since SF adds no transitive edges between variables. Figure 11 shows that for |F our simple strategy finds on average 80% of the variables involved in cycles, whereas the same strategy finds only 40% when used with SF.

In the rest of the section, we establish Theorem 5.1 and Theorem 5.2. We introduce some notation and terminology used in the following discussion. We use $u$ and $v$ to denote either variable nodes or source and sink nodes, $X$ or $X'$ to denote variable nodes, and $c$ or $c'$ to denote source and sink nodes. A total order on the $n$ variables is chosen uniformly at random from among all $n!$ possible permutations. Finally, we say a graph edge $(u, v)$ is added through a path $\rho$ if $(u, v)$ would be added by the graph closure rule considering only the nodes and edges of $\rho$.

5.1 Edge Additions in Standard Form

During the graph closure process edges may be added more than once because an edge may be implied by more than one path in the constraint graph (cf. Figure 2). Thus, a
constraint solver does work proportional to the number of edge additions, including redundant additions along different paths.

Define the random variables $X_{SF}^{uv}$ to be the number of additions of the edge $(u, v)$ through simple paths from $u$ to $v$ for the standard form. To calculate the total expected number of edge additions, it suffices to calculate the expected number of additions $E(X_{SF}^{uv})$ of a given edge $(u, v)$ and sum over all possible edges.

For the standard form we consider two kinds of edges, $(c, X)$ and $(c, c')$. We now calculate $E(X_{SF}^{(c, X)})$ and $E(X_{SF}^{(c, c')})$. Notice that the edge $(c, X)$ must be added through a simple path from $c$ to $X$. For edges of the form $(c, c')$, we also need only consider the simple paths from $c$ to $c'$.

For each simple path from $c$ to $X$ of length $i + 1$, there are $\binom{n-1}{i}$ choices of intermediate variable nodes. For each simple path from $c$ to $c'$ of length $i + 1$, there are $\binom{n}{i}$ choices of intermediate variable nodes. In both cases, each combination of variable nodes may appear in $i!$ possible orders. The probability that any particular sequence of the $i+2$ nodes (including $c$ and $X$ or $c$ and $c'$) is a path is $i^{i+1}$. We obtain the following:

$$E(X_{SF}^{(c, X)}) = \sum_{i=1}^{n-1} \binom{n-1}{i} i! p^{i+1}$$

$$E(X_{SF}^{(c, c')}) = \sum_{i=1}^{n} \binom{n}{i} i! p^{i+1}$$

Since there are $mn$ possible edges of the form $(c, X)$ and $m(m-1)$ possible edges of the form $(c, c')$, the expected number of edge additions for the standard form is given by

$$E(X_{SF}) = mn E(X_{SF}^{(c, X)}) + m(m-1) E(X_{SF}^{(c, c')})$$

5.2 Edge Additions in Inductive Form

Define the random variables $X_{IF}^{uv}$ to be the number of additions of the edge $(u, v)$ through simple paths from $u$ to $v$ for the inductive form. We need to consider four kinds of edges: $(X_1, X_2), (X, c), (c, c')$, and $(c, c_2)$. Notice that the probability that a given edge $(u, v)$ is added through a simple path $\rho$ of $l \geq 3$ nodes from $u$ to $v$ depends only on $l$. Thus we let $P_l(u, v)$ denote the probability that the edge $(u, v)$ is added through a simple path from $u$ to $v$ with $l$ nodes. We have the following equations:

$$E(X_{IF}^{(X_1, X_2)}) = \sum_{i=1}^{n-2} \binom{n-2}{i} i! P_{i+2}(X_1, X_2)$$

$$E(X_{IF}^{(X, c)}) = E(X_{IF}^{(X, c)})$$

$$E(X_{IF}^{(c, c')}) = \sum_{i=1}^{n-1} \binom{n-1}{i} i! P_{i+2}(X, c)$$

$$E(X_{IF}^{(c, c')}) = \sum_{i=1}^{n} \binom{n}{i} i! P_{i+2}(c, c')$$

We next calculate for any $l \geq 3$ the probability $P_l(u, v)$ for any nodes $u$ and $v$.

Lemma 5.3 Let $o(\cdot)$ be a random total order on the variables. Given a simple path $\rho$ from $u$ to $v$ with $l$ nodes, the following holds:

1. $P(u, v) = \frac{2}{n!}$ if $u$ and $v$ are variable nodes;
2. $P(u, v) = \frac{1}{l-1}$ if one of $u$ and $v$ is a variable node and the other is a constructed node;
3. $P(u, v) = 1$ if both $u$ and $v$ are constructed nodes.

Proof. We prove the first case. Similar arguments apply to the other two cases.

We first show $P(u, v) \leq \frac{2}{n!}$. Recall that $o(X)$ is the index of variable $X$. Assume the edge $(u, v)$ is added through a path $(u, X_1, \ldots, X_{l-2}, v)$, we claim that $o(u)$ and $o(v)$ are the smallest indices on the path, i.e., $o(u) < o(X_i)$ and $o(v) < o(X_i)$ for all $1 \leq i \leq l - 2$. For paths with three nodes, this claim is true by the closure rule, since the edge is only added if $u \cdots \rightarrow X_1$ and $X_1 \rightarrow v$ are in the graph and these edges imply that $o(u)$ and $o(v)$ are less than $o(X_1)$.

Suppose the claim is true for paths with at most $k \geq 3$ nodes. Consider a path $(u, X_1, \ldots, X_{k-1}, v)$ with $k+1$ nodes such that the edge $(u, v)$ is added through the path. Notice there must exist a $X_i$ with $1 \leq i \leq k-1$ such that the edges $(u, X_i)$ and $(X_i, v)$ are added and $o(u) < o(X_i)$ and $o(v) < o(X_i)$. By induction, the claim holds for the shorter paths $(u, \ldots, X_i)$ and $(X_i, \ldots, v)$. Thus, $o(u)$ and $o(v)$ must be the smallest indices on the path. There are $l!$ possible orderings on the $n$ variables and we claim that there are $\binom{n}{i}(2(l-2))!(n-l)!$ of them satisfying the above condition.

There are $l!$ possible ways of choosing the indices for the $l$ variables on the path. There are 2 ways of ordering $u$ and $v$, and $(l-2)!$ ways of ordering the rest of the variables on the path. For the other $(n-l)$ variables we can order them in $(n-l)!$ ways. Thus we have

$$P_l(u, v) \leq \frac{\binom{n}{i}(2(l-2))!(n-l)!}{l!}$$

$$= \frac{2}{l(l-1)}$$

We now show $P_l(u, v) \geq \frac{2}{n!}$. Let $o(\cdot)$ be an ordering such that $o(u)$ and $o(v)$ are the smallest indices on the path $(u, X_1, \ldots, X_{l-2}, v)$. We show that the edge $(u, v)$ is added through the path. The claim is clearly true for paths with three nodes. Suppose the claim holds for paths with at most $k$ nodes. Consider a path $(u, X_1, \ldots, X_{k-1}, v)$ with $k+1$ nodes such that $o(u)$ and $o(v)$ have the smallest indices. Let $X_i$ be the node such that $o(X_i) < o(X_j)$ for all $1 \leq j \leq k-1$ with $i \neq j$. By induction, the claim holds for the two subpaths $(u, \ldots, X_i)$ and $(X_i, \ldots, v)$, i.e., the edges $(u, X_i)$ and $(X_i, v)$ are added through the respective subpaths. Thus, the edge $(u, v)$ is added through the given path. Therefore, $P_l(u, v) \geq \frac{2}{m!}$.

Since there are $m(m-1)$ edges of the form $(c, c')$, $2mn$ edges of the form $(X, c)$ or $(c, X)$, and $n(n-1)$ edges of the form $(X_1, X_2)$, the expected number of edge additions for the inductive form is given by

$$E(X_{IF}) = \frac{m(n-1)}{2} E(X_{IF}^{(c, c')}) + 2mn E(X_{IF}^{(X, c)}) + n(n-1) E(X_{IF}^{(X_1, X_2)})$$

5.3 Comparison

To directly compare $SF$ and $IF$ it is necessary to make an additional assumption about the density of the initial graph.
In the following calculation, we assume $p = \frac{1}{n}$, which says that a typical initial graph has $\frac{(n + m)^2}{2}$ edges. In practice, initial constraint graphs are sparse; all our benchmark programs produce initial graphs of approximately this density.

We have the following approximation [Knu79]

$$\sum_{i=1}^{n} \binom{n}{i} \frac{1}{i!} \frac{1}{n}^i \approx \frac{\sqrt{\pi n}}{2}$$  \hspace{1cm} (2)

Using equation (2) we simplify $E(X^{SF})$ and $E(X^{RF})$ as follows.

$$E(X^{SF}) \approx m \left( \frac{\sqrt{\pi n}}{2} - 1 \right) + \frac{m(m - 1)}{n} \sqrt{\frac{\pi n}{2}}$$

$$E(X^{RF}) \approx \frac{m(m - 1)}{n} \sqrt{\frac{\pi n}{2}} + 2m \ln n + n$$

To obtain Theorem 5.1, we relate the expected edge additions to the amount of work done to close the constraint graphs. Since we consider only simple paths, the expected number of edge additions corresponds to the case where there are no cycles (i.e., the oracle runs in Section 4). For our benchmark programs, the typical ratio of $\frac{\theta}{\phi}$ is about $\frac{2}{3}$ (See Table 1). Thus, asymptotically, $E(X^{SF})/E(X^{RF})$ is about 2.5, i.e., using the standard form, we expect to do 2.5 times as much work as using the inductive form. On our benchmarks we have measured an average of 4.1 times more work for SF.

5.4 Cost of Online Cycle Elimination

Next we establish that the expected number of reachable nodes from any given node is small. This result explains why the simple heuristic for detecting cycles is very cheap.

Let $X$ be any variable node and let $R_X$ be the random variable denoting the number of nodes reachable from $X$ through a predecessor chain. Using the same method for calculating the expected number of edge additions, we consider all simple paths starting with $X$ involving only variable nodes. We thus have

$$E(R_X) \leq \sum_{i=1}^{n-1} \binom{n-1}{i} \frac{1}{(i+1)!} \frac{1}{n}^i$$

Next, we approximate $E(R_X)$. Let $p = \frac{k}{n}$ for some constant $k$. Then

$$E(R_X) \leq \sum_{i=1}^{n-1} \binom{n-1}{i} \frac{k}{n}^i \frac{1}{(i+1)!}$$

$$< \frac{1}{k} (e^k - 1 - k)$$

The value of $p$ here is the probability of an edge being present in the final constraint graph, not the initial one. If $p = \frac{2}{3} n$, i.e., $k = 2$ (which holds roughly for our benchmarks) we have

$$E(R_X) < \frac{1}{2} (e^2 - 1 - 2) \approx 2.2$$

completing the proof of Theorem 5.2. Note that for graphs denser than $p = \frac{2}{3}$ the value $E(R_X)$ climbs sharply—our method relies on sparse graphs.

6 Related Work

There are three strands of related work: constraint simplification, points-to analysis, and sub-cubic time analyses.

The importance of simplifications on constraint graphs has been recognized before. In contrast to our online approach, prior work has focused on periodic simplification. In [FA96] the authors describe several simplifications to reduce the heap requirements of graphs for a more complex constraint language. They give performance results obtained through simplifications at regular depths in the abstract syntax tree traversal. Simplification cost outweighs potential benefits when simplifications are performed frequently.

Several papers explore the theoretical foundations of constraint simplification [TS96, Pot96, FF97]. Among these, [FF97] implemented several simplifications in the context of a static debugger for Scheme. Constraint graphs are generated separately for each module, simplified, and finally merged. They report substantial reduction in constraint graph sizes and speedups of analysis times.

Marlow and Wadler use set constraints in a type system for Erlang [MW97]. Their system performs simplifications similar to [FA96, FF97] for every function declaration. They report that performance is poor for large sets of mutually recursive functions, which must be analyzed together.

Points-to analysis with set constraints is in Andersen’s thesis [And94]. Recent work by Shapiro and Horwitz [SH97] contrasts Andersen’s set based points-to analysis with the unification based points-to analysis of Steensgaard [Ste96]. They conclude that while Andersen’s analysis is substantially more precise than Steensgaard’s, its running time is impractical. However, our implementation of Andersen’s points-to analysis is generally competitive with [SH97]’s implementation of Steensgaard’s algorithm.

Inclusion constraint resolution algorithms usually have at least $O(n^3)$ time complexity. The lack of progress in achieving scalable implementations of these algorithms has encouraged interest in asymptotically faster algorithms that are either less precise or designed for special cases. Steensgaard’s system is an example of the former; the linear time closure-analysis algorithm for functional programs with bounded type size is an example of the latter [Mos96, HM97]. We plan to study the impact of online cycle elimination on the performance of closure analysis in future work.

7 Conclusions

We have shown that online elimination of cyclic constraints in inclusion constraint based program analyses yields orders-of-magnitude improvements in execution time. Our partial online cycle detection algorithm is cheap but effective and works best on a non-standard representation of constraint graphs.

Acknowledgments

We would like to thank David Gay, Raph Levien, and the anonymous referees for helpful comments on improving the paper. Special thanks go to Mark Shapiro and Susan Horwitz for providing their Points-to implementations for comparison.
References


