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 constraints are a natural vehicle for expressing a wide range of program analyses including shape analysis, closure analysis, soft typing systems, receiver-class prediction for object-oriented programs, and points-to analysis for pointer-based programs, among others [Rey69, JM79, Shi88, PS91, AWL94, Hei94, And94, FFK+96, MW97]. Such analyses are efficient for small to medium size programs, but they are known to be impractical for large analysis problems.

Inclusion constraint systems have natural graph representations. For example, the constraints $\mathcal{X} \subseteq \mathcal{Y} \subseteq \mathcal{Z}$ are represented by nodes for the quantities $\mathcal{X}, \mathcal{Y}$, and $\mathcal{Z}$ and directed edges $(\mathcal{X}, \mathcal{Y})$ and $(\mathcal{Y}, \mathcal{Z})$ 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 $(\mathcal{X}, \mathcal{Z})$ represents the implied constraint $\mathcal{X} \subseteq \mathcal{Z}$.

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 $\mathcal{X}_1 \subseteq \mathcal{X}_2 \subseteq \mathcal{X}_3 \ldots \subseteq \mathcal{X}_n \subseteq \mathcal{X}_1$ where the $\mathcal{X}_i$ 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 *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 *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 *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 we 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 $\mathcal{X} \subseteq \mathcal{Y}$, between variables $\mathcal{X}$ and $\mathcal{Y}$, is represented as a *successor edge* from node $\mathcal{X}$ to node $\mathcal{Y}$. Our measurements show that this *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-

*Supported in part by an NDSEG fellowship, NSF Young Investigator Award CCR-9457812, NSF Grant CCR-9416973, and a gift from Rockwell Corporation.

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 $\mathcal{X}, \mathcal{Y}, \ldots$ from a family of variables *Vars*, terms constructed from n-ary constructors $c \in Con$, an empty set 0, and a universal set 1.

$$L, R \in se \quad ::= \quad \mathcal{X} \mid c(se_1, \ldots, se_n) \mid 0 \mid 1$$

Each constructor c is given a unique *signature* S_c 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[1] for set expressions. We simply assume that each constructor c is also equipped with an interpretation ϕ_c. Given a *variable assignment* A of sets to variables, set expressions are interpreted as follows[2]:

$$\begin{aligned} [\![\mathcal{X}]\!]\, A &= A(\mathcal{X}) \\ [\![c(se_1, \ldots, se_n)]\!]\, A &= \phi_c([\![se_1]\!]\, A, \ldots, [\![se_n]\!]\, A) \end{aligned}$$

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 [Hei92]. 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.

[1] Standard models are the termset model [Hei92, Koz93] or the ideal model [AW93].

[2] The interpretation of 0 and 1 depends on the model and is not shown.

$$\begin{aligned} S \cup \{\mathcal{X} \subseteq \mathcal{X}\} &\Leftrightarrow S \\ S \cup \{se \subseteq 1\} &\Leftrightarrow S \\ S \cup \{0 \subseteq se\} &\Leftrightarrow S \\ S \cup \{c(se_1, \ldots, se_n) \subseteq c(se'_1, \ldots, se'_n)\} &\Leftrightarrow \\ S \cup \bigcup_i \begin{cases} \{se_i \subseteq se'_i\} & c \text{ covariant in } i \\ \{se_i \supseteq se'_i\} & c \text{ contravariant in } i \end{cases} & \\ S \cup \{c(\ldots) \subseteq d(\ldots)\} &\Leftrightarrow \text{no solution if } d \neq c \\ S \cup \{c(\ldots) \subseteq 0\} &\Leftrightarrow \text{no solution} \\ S \cup \{1 \subseteq 0\} &\Leftrightarrow \text{no solution} \\ S \cup \{1 \subseteq d(\ldots)\} &\Leftrightarrow \text{no solution} \end{aligned}$$

Figure 1: Resolution rules $\mathcal{R}$ for SF and IF

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

$\mathcal{X} \subseteq \mathcal{Y}$	variable-variable constraint
$c(\ldots) \subseteq \mathcal{X}$	source-variable constraint
$\mathcal{X} \subseteq c(\ldots)$	variable-sink constraint

We use the set of resolution rules $\mathcal{R}$ 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 $(\mathcal{X}, \mathcal{Y})$ in a graph is represented exclusively either as a *predecessor edge* ($\mathcal{X} \in pred(\mathcal{Y})$) or as a *successor edge* ($\mathcal{Y} \in succ(\mathcal{X})$).

2.3 Standard Form

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

$\mathcal{X} \subseteq \mathcal{Y}$	$\mathcal{X} \longrightarrow \mathcal{Y}$	successor edge
$c(\ldots) \subseteq \mathcal{X}$	$c(\ldots) \cdots\!\!\rightarrow \mathcal{X}$	predecessor edge
$\mathcal{X} \subseteq c(\ldots)$	$\mathcal{X} \longrightarrow c(\ldots)$	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*:

$$L \cdots\!\!\rightarrow \mathcal{X} \longrightarrow R \quad \Leftrightarrow \quad L \subseteq R$$

Given a predecessor edge $L \cdots\!\!\rightarrow \mathcal{X}$ and a successor edge at $\mathcal{X} \longrightarrow R$, a new constraint $L \subseteq R$ is generated. We generate a constraint instead of an edge because rules in Figure 1

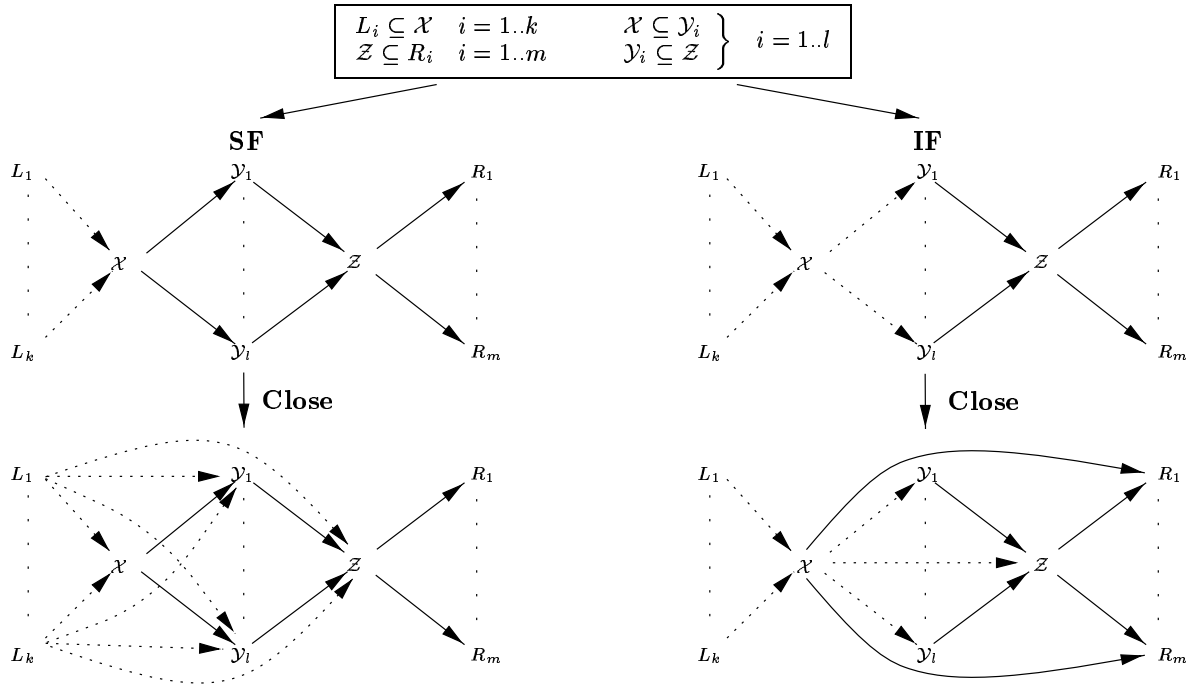

Figure 2: Example constraints in SF and IF

may apply. Note that in this case, L is always of the form $c(\dots)$. 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 $\mathcal{X}$, the closure rule must be applied exactly to all combinations of predecessor and the successor edges of $\mathcal{X}$.

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 $L_1 \dots L_k$ are sources and $R_1 \dots R_m$ are sinks. The closure of the standard form adds transitive edges from each source L_i to all variables reachable from $\mathcal{X}$ *i.e.*, $\mathcal{Y}_1 \dots \mathcal{Y}_l, \mathcal{Z}$. Note that the edges from $L_1 \dots L_k$ to $\mathcal{Z}$ are added l times each, namely along all l edges $\mathcal{Y}_i \longrightarrow \mathcal{Z}$. The total work of closing the graph is $2kl$ 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 $\mathcal{Z} \longrightarrow \mathcal{X}$ in Figure 2, forming a strongly connected component $\mathcal{X}, \mathcal{Y}_1, \dots, \mathcal{Y}_l, \mathcal{Z}$. If we collapse this component before adding the transitive edges $L_i \cdots\!\blacktriangleright \mathcal{Y}_j$, none of the $2kl$ transitive edge additions $L_i \cdots\!\blacktriangleright \mathcal{Y}_j$ 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 $\mathcal{X} \subseteq \mathcal{Y}$ can be represented either as a successor edge ($\mathcal{Y} \in succ(\mathcal{X})$) or as a predecessor edge ($\mathcal{X} \in pred(\mathcal{Y})$). The representation for a particular edge is chosen as a function of a fixed total order $o : Vars \to \mathbb{N}$ on the variables. Edges in the constraint graph are represented as follows:

$$\mathcal{X} \subseteq \mathcal{Y} \quad \begin{cases} \mathcal{X} \longrightarrow \mathcal{Y} & \text{if } o(\mathcal{X}) > o(\mathcal{Y}) \\ & \text{a successor edge} \\ \mathcal{X} \cdots\!\blacktriangleright \mathcal{Y} & \text{if } o(\mathcal{X}) < o(\mathcal{Y}) \\ & \text{a predecessor edge} \end{cases}$$

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:

$$L \cdots\!\blacktriangleright \mathcal{X} \longrightarrow R \quad \Leftrightarrow \quad L \subseteq R$$

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:

$$LS(\mathcal{Y}) = \{c(\dots) \mid c(\dots) \cdots\!\blacktriangleright \mathcal{Y}\} \cup \bigcup_{\mathcal{X} \cdots\!\blacktriangleright \mathcal{Y}} LS(\mathcal{X}) \tag{1}$$

```
insert_succ_edge(vertex from, vertex to)
{ // variable vertices: o(from) > o(to)
  if (pred_chain(from, to)) { // Cycle found
    collapse_cycle (...);
  }
  else
    insert_into_successor_list(from, to);
}
```

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

Figure 3: Algorithms for cycle detection

By the ordering $o(\cdot)$, we have $o(\mathcal{X}) < o(\mathcal{Y})$ for all $\mathcal{X} \cdots\!\blacktriangleright \mathcal{Y}$. Thus there exists a variable $\mathcal{Z}_1$ with minimum index $o(\mathcal{Z}_1)$ that has no predecessor edges to any other variables and $LS(\mathcal{Z}_1) = \{c(\ldots) \mid c(\ldots) \cdots\!\blacktriangleright \mathcal{Z}_1\}$. Then $LS(\mathcal{Z}_i)$ is computed using $LS(\mathcal{Z}_j)$ 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 $o(\mathcal{X}) < o(\mathcal{Z}) < o(\mathcal{Y}_i)$. Note the extra variable-variable edge $\mathcal{X} \cdots\!\blacktriangleright \mathcal{Z}$ added by the closure rule for IF. As a result of this edge, the closure of IF adds edges from $\mathcal{X}$ to all R_i. Each of the variables $\mathcal{Y}_1, \ldots, \mathcal{Y}_l, \mathcal{Z}$ has a single predecessor edge to $\mathcal{X}$, and thus their least solution is equal to $LS(\mathcal{X}) = \{L_1, \ldots, L_k\}$. The total work of closing the graph is $l + m$ edge additions, of which $l - 1$ additions are redundant, namely the addition of edge $\mathcal{X} \cdots\!\blacktriangleright \mathcal{Z}$ through all $\mathcal{Y}_i$, 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 l.

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 $u = v_0$, $v = v_k$, $v_1..v_{k-1}$ are variable nodes, and $v_{i-1} \longrightarrow\!\blacktriangleright v_i \in E$ or $v_{i-1} \cdots\!\blacktriangleright v_i \in E$ for $i = 1..k$. 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 $(\mathcal{X}_0, \ldots, \mathcal{X}_k)$ consisting entirely of successor edges $\mathcal{X}_{i-1} \longrightarrow\!\blacktriangleright \mathcal{X}_i$ for $i = 1..k$ (a *successor chain*), or consisting entirely of predecessor edges $\mathcal{X}_{i-1} \cdots\!\blacktriangleright \mathcal{X}_i$ for $i = 1..k$ (a *predecessor chain*).

A path $(\mathcal{X}_0, \ldots, \mathcal{X}_k)$ forms a cycle if $\mathcal{X}_0 = \mathcal{X}_k$ 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.

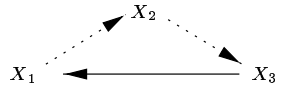

Figure 4: A cyclic graph in IF

Our algorithm (Figure 3) for online cycle elimination is a straight-forward implementation of the following idea. When adding a successor edge $\mathcal{X} \longrightarrow\!\blacktriangleright \mathcal{Y}$, we search (using pred_chain) along all predecessor edges starting from $\mathcal{X}$ for a predecessor chain $\mathcal{Y} \cdots\!\blacktriangleright^{+} \mathcal{X}$. Similarly, if we add a predecessor edge $\mathcal{X} \cdots\!\blacktriangleright \mathcal{Y}$, we search (using succ_chain, not shown) along all successor edges starting from $\mathcal{Y}$ for a successor chain $\mathcal{Y} \longrightarrow\!\blacktriangleright^{+} \mathcal{X}$. 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 $\mathcal{X} \longrightarrow\!\blacktriangleright \mathcal{Y}$, we search (using succ_chain) along all successor edges starting from $\mathcal{Y}$ for a successor chain $\mathcal{Y} \longrightarrow\!\blacktriangleright^{+} \mathcal{X}$. 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 $\mathcal{X}_3 \longrightarrow\!\blacktriangleright \mathcal{X}_1$ is added last, since in this case, the predecessor chain $\mathcal{X}_1 \cdots\!\blacktriangleright \mathcal{X}_2 \cdots\!\blacktriangleright \mathcal{X}_3$ 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 $\mathcal{X}_2 \longrightarrow\!\blacktriangleright \mathcal{X}_1$ and the sub-cycle $(\mathcal{X}_1, \mathcal{X}_2)$ 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).[3] 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.

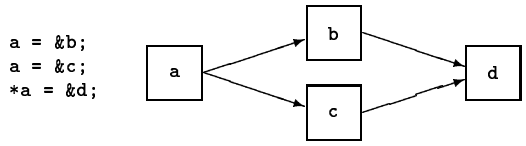

Figure 5: Example points-to graph

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 $\mathcal{R}$. 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 $\mathcal{X}_{l_1}, \ldots, \mathcal{X}_{l_n}$ denoting the set of locations pointed to by $l_1, \ldots, l_n$. The example in Figure 5 has the set formulation

$$\begin{aligned} \mathcal{X}_{l_\mathtt{a}} &= \{l_\mathtt{b}, l_\mathtt{c}\} \\ \mathcal{X}_{l_\mathtt{b}} &= \{l_\mathtt{d}\} \\ \mathcal{X}_{l_\mathtt{c}} &= \{l_\mathtt{d}\} \end{aligned}$$

The association between a location l_i and its points-to set $\mathcal{X}_{l_i}$ 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 $ref(\{l_i\}, \mathcal{X}_{l_i})$ akin to reference types in languages like ML [MTH90].

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: $ref(\mathcal{X}) \subseteq ref(\mathcal{Y}) \Leftrightarrow \mathcal{X} = \mathcal{Y}$.

$$\frac{}{\mathtt{x} : ref(l_\mathtt{x}, \mathcal{X}_{l_\mathtt{x}}, \overline{\mathcal{X}_{l_\mathtt{x}}})} \quad \text{(Var)}$$

$$\frac{e : \tau}{\&e : ref(0, \tau, \overline{\tau})} \quad \text{(Addr)}$$

$$\frac{e : \tau \quad \tau \subseteq ref(1, \mathcal{T}, \overline{0}) \quad \mathcal{T} \text{ fresh}}{*e : \mathcal{T}} \quad \text{(Deref)}$$

$$\frac{\begin{array}{c} e_1 : \tau_1 \qquad e_2 : \tau_2 \\ \tau_1 \subseteq ref(1, 1, \overline{\mathcal{T}_1}) \quad \tau_2 \subseteq ref(1, \mathcal{T}_2, \overline{0}) \\ \mathcal{T}_2 \subseteq \mathcal{T}_1 \quad \mathcal{T}_1, \mathcal{T}_2 \text{ fresh} \end{array}}{e_1 = e_2 : \tau_2} \quad \text{(Asst)}$$

Figure 6: Constraint generation for Andersen's analysis

We adapt this technique to a purely inclusion-based system using a novel approach. We intuitively treat a reference $l_\mathtt{x}$ as an object with a location name and two methods $\mathsf{get} : \mathsf{void} \to \mathcal{X}_{l_\mathtt{x}}$ and $\mathsf{set} : \mathcal{X}_{l_\mathtt{x}} \to \mathsf{void}$, 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 *ref* constructor that corresponds to the domain of the set function, and is thus contravariant. A location $l_\mathtt{x}$ is then represented by $ref(l_\mathtt{x}, \mathcal{X}_{l_\mathtt{x}}, \overline{\mathcal{X}_{l_\mathtt{x}}})$ (to improve readability we overline contravariant arguments). To update an unknown location τ with a set $\mathcal{T}$, it suffices to add a constraint $\tau \subseteq ref(1, 1, \overline{\mathcal{T}})$. For example, if $ref(l_\mathtt{x}, \mathcal{X}_{l_\mathtt{x}}, \overline{\mathcal{X}_{l_\mathtt{x}}}) \subseteq \tau$, then the transitive constraint $ref(l_\mathtt{x}, \mathcal{X}_{l_\mathtt{x}}, \overline{\mathcal{X}_{l_\mathtt{x}}}) \subseteq ref(1, 1, \overline{\mathcal{T}})$ is equivalent to $\mathcal{T} \subseteq \mathcal{X}_{l_\mathtt{x}}$ (due to contravariance), which is the desired effect. Dereferencing is analogous, but involves the covariant points-to set of the *ref* constructor.

To formally express Andersen's points-to graph, we must associate with each location $l_\mathtt{x}$ a set variable $\mathcal{Y}_{l_\mathtt{x}}$ for the set of abstract location names and a constraint $\mathcal{X}_{l_\mathtt{x}} \subseteq ref(\mathcal{Y}_{l_\mathtt{x}}, 1, 0)$ that constrains $\mathcal{Y}_{l_\mathtt{x}}$ to be a superset of all names of locations in the points-to set $\mathcal{X}_{l_\mathtt{x}}$. The points-to graph is then defined by the least solution for $\mathcal{Y}_{l_i}$. In our implementation we avoid using the location names l_i and the variables $\mathcal{Y}_{l_i}$, 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 [FFA97]. 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 τ 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 $ref(l_\mathtt{x}, \mathcal{X}_{l_\mathtt{x}}, \overline{\mathcal{X}_{l_\mathtt{x}}})$ associated with $\mathtt{x}$ therefore denotes the location of $\mathtt{x}$ and not its contents.

We briefly describe the other rules in Figure 6. The

Benchmark	AST Nodes	LOC	Total #Vars	Nodes	Initial Edges	Strongly Connected Components (SCC)					
						Initial graph			Final graph		
						#Vars	#SCC	max	#Vars	#SCC	max
allroots	700	428	171	264	141	5	2	3	19	5	10
diff.diffh	935	293	319	537	297	6	3	2	13	6	3
anagram	1078	344	219	360	216	10	5	2	23	9	5
genetic	1412	324	264	415	249	4	2	2	14	5	4
ks	2284	574	335	515	329	6	3	2	37	3	33
ul	2395	445	207	325	176	4	2	2	7	3	3
ft	3027	1179	510	859	487	0	0	—	70	5	57
compress	3333	652	241	364	241	17	7	4	27	9	6
ratfor	5269	1540	1024	1804	1020	20	7	6	104	8	80
compiler	5326	1895	1378	2028	1292	12	6	2	39	9	20
assembler	6516	2987	1811	2885	1497	9	4	3	96	10	27
ML-typecheck	6752	4903	1855	3096	1908	29	9	9	296	15	185
eqntott	8117	2316	1371	2209	1442	50	15	12	232	21	137
simulator	10946	4230	2764	4578	2317	6	3	2	290	5	267
less-177	15179	12046	3527	6171	3389	30	12	4	357	14	288
li	16828	5761	7111	12213	6283	24	10	4	1736	5	1727
flex-2.4.7	29960	9358	5617	9694	5591	57	19	11	355	19	281
pmake	31210	25129	7009	11530	6507	63	24	8	775	23	690
make-3.72.1	36892	15214	7839	12514	6994	156	63	7	966	57	727
inform-5.5	38874	12845	9565	15058	8687	34	15	6	505	19	410
tar-1.11.2	41497	18312	6095	9735	6459	347	89	29	831	79	515
screen-3.5.2	49292	23943	12806	18706	8631	108	46	6	894	31	822
cvs-1.3	51223	31195	13848	20735	10497	340	87	31	755	92	408
sgmls-1.1	53874	35155	9539	15372	9740	91	37	9	1201	41	1071
espresso	56938	21583	11490	19067	12271	333	149	10	1773	170	1355
gawk-3.0.3	71091	27381	11590	18083	10658	400	58	86	1796	40	1573
povray-2.2	87391	59689	13401	21837	11937	198	87	9	1578	80	1299

Table 1: Benchmark data common to all experiments

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 $\mathcal{T}$ a superset of the points-to set of τ. The second constraint in the assignment rule (Asst) transforms the right-hand side τ_2 from an L-value to an R-value $\mathcal{T}_2$, as in (Deref) (recall these rules infer sets representing L-values). The first constraint $\tau_1 \subseteq ref(1, 1, \overline{\mathcal{T}_1})$ makes $\mathcal{T}_1$ a subset of the points-to set of τ_1. The final constraint $\mathcal{T}_2 \subseteq \mathcal{T}_1$ expresses exactly the intuitive meaning of assignment: the points-to set $\mathcal{T}_1$ of the left-hand side contains at least the points-to set $\mathcal{T}_2$ of the right-hand side. For example, the first statement of Figure 5, `a = &b`, generates the constraints $\tau_1 = ref(l_{\mathtt{a}}, \mathcal{X}_{l_{\mathtt{a}}}, \overline{\mathcal{X}_{l_{\mathtt{a}}}}) \subseteq ref(1, 1, \overline{\mathcal{T}_1})$, and so $\mathcal{T}_1 \subseteq \mathcal{X}_{l_{\mathtt{a}}}$, and $\tau_2 = ref(0, ref(l_{\mathtt{b}}, \mathcal{X}_{l_{\mathtt{b}}}, \overline{\mathcal{X}_{l_{\mathtt{b}}}}), \dots) \subseteq ref(1, \mathcal{T}_2, \overline{0})$, and so $ref(l_{\mathtt{b}}, \mathcal{X}_{l_{\mathtt{b}}}, \overline{\mathcal{X}_{l_{\mathtt{b}}}}) \subseteq \mathcal{T}_2$. The final constraint $\mathcal{T}_2 \subseteq \mathcal{T}_1$ implies the desired effect, namely $ref(l_{\mathtt{b}}, \mathcal{X}_{l_{\mathtt{b}}}, \overline{\mathcal{X}_{l_{\mathtt{b}}}}) \subseteq \mathcal{X}_{l_{\mathtt{a}}}$.

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 constraints (before closing the graph). Furthermore, the table contains the combined size of all non-trivial strongly connected components (SCC), the number of components, and the size of the largest component, both for the initial graph (before closure) and for the final graph (in any experiment). The difference in the combined size of SCCs between the initial and the final graph shows the need for online cycle elimination. If all cycles were present in the initial graph, online cycle elimination would be unnecessary.

Experiment	Description
SF-Plain	Standard form, no cycle elimination
IF-Plain	Inductive form, no cycle elimination
SF-Oracle	Standard form, with full (oracle) cycle elimination
IF-Oracle	Inductive form, with full (oracle) cycle elimination
SF-Online	Standard from, using IF online cycle elimination
IF-Online	Inductive form, with online cycle elimination

Table 4: Experiments

We use a single well-engineered constraint resolution library to compare SF and IF. To validate that our results are not a product of our particular implementation, we compare our implementation of standard form to an independent implementation of points-to analysis written in C by Shapiro and Horwitz [SH97]. Their implementation corresponds to SF without cycle elimination, and we empirically verify that our implementation of SF produces the same trend on our benchmark suite. The scatter plot in Figure 12 shows that our implementation of SF without cycle elimination is usually between 2 times faster and 2 times slower than SH (horizontal lines) on a subset of the benchmarks[4] with a few exceptions where our implementation is significantly faster (`flex`, `li`, `cvs`, `inform`), and one program where our implementation is substantially slower (`tar`).

We performed the six experiments shown in Table 4. The first two are plain runs of the points-to analysis using SF and IF without cycle elimination. SF-Plain corresponds to classic implementations of set-based analyses. The experiments SF-Oracle and IF-Oracle precompute the strongly connected components of the final graph and use that information as

[4] Not all benchmarks ran through SH.

Benchmark	IF-Plain			SF-Plain			IF-Oracle			SF-Oracle		
	Edges	Work	Time(s)	Edges	Work	Time(s)	Edges	Work	Time(s)	Edges	Work	Time(s)
allroots	384	441	0.06	290	309	0.05	322	367	0.06	261	284	0.07
diff.diffh	711	782	0.13	606	651	0.11	686	762	0.10	588	637	0.10
anagram	510	557	0.09	412	450	0.10	450	483	0.07	369	393	0.09
genetic	515	572	0.13	446	496	0.10	488	542	0.10	432	481	0.14
ks	3385	15562	0.46	1278	2332	0.17	663	978	0.15	850	1181	0.15
ul	315	428	0.11	280	381	0.10	301	414	0.13	272	373	0.11
ft	3766	19821	0.61	1204	1733	0.19	1008	1251	0.33	832	1024	0.21
compress	492	742	0.15	356	529	0.14	376	570	0.13	317	493	0.13
ratfor	5777	24987	1.30	3158	5070	0.59	2302	3309	0.62	2407	3470	0.63
compiler	2733	3548	0.61	3027	3992	0.59	2571	3378	0.72	2655	3476	0.74
assembler	5219	9844	1.21	4016	4916	0.85	3552	4326	1.44	3640	4367	0.83
ML-typecheck	17908	159253	5.38	21850	149501	3.45	3826	6005	1.24	6606	9592	1.15
eqntott	15667	132250	3.63	4291	7985	0.85	2927	4030	1.06	2692	3808	0.81
simulator	29935	571836	14.48	35280	282857	6.07	4838	6578	1.98	12967	16740	2.05
less-177	76789	3047615	64.43	72045	678170	12.51	8412	13796	3.31	34029	49028	3.79
li	1130427	177872021	4349.04	1740142	103639138	1629.02	13360	76391	10.62	470945	786496	29.73
flex-2.4.7	98301	2231558	52.20	15736	27498	4.70	12954	20795	6.61	13100	21147	4.99
pmake	364732	28462390	669.97	306775	8582058	134.45	15485	47806	6.90	133477	234533	12.71
make-3.72.1	747878	96349223	2287.20	693860	43941197	624.01	19967	119389	12.24	268863	484661	21.24
inform-5.5	236017	15236795	359.31	222182	5505608	90.64	31991	95132	14.53	115437	178675	13.44
tar-1.11.2	278820	19175919	433.98	250474	5842839	89.30	18711	37250	6.81	94316	136277	8.15
screen-3.5.2	963242	130598316	2988.00	640161	38797130	610.87	—	—	—	—	—	—
cvs-1.3	214229	6702126	161.33	116513	891083	22.96	26364	43685	9.98	67253	106422	11.28
sgmls-1.1	1706442	362826558	8686.45	1472647	140857874	2077.24	25050	179301	19.15	532076	905743	44.35
espresso	859093	78018098	1903.74	741876	23276456	373.30	32318	123335	15.46	343268	591392	28.83
gawk-3.0.3	1653812	267208993	6605.69	922422	45499857	686.92	—	—	—	—	—	—
povray-2.2	2710342	490070572	12159.00	1984147	179010002	2966.48	—	—	—	—	—	—

Table 2: Benchmark data for IF-Plain, SF-Plain, IF-Oracle, and SF-Oracle

Benchmark	AST	SCC	IF-Online				SF-Online			
		#Vars	Elim.	Edges	Work	Time(s)	Elim.	Edges	Work	Time(s)
allroots	700	19	10	360	408	0.09	7	286	303	0.06
diff.diffh	935	13	7	697	767	0.11	6	607	656	0.12
anagram	1078	23	13	493	536	0.11	6	407	441	0.09
genetic	1412	14	6	502	556	0.13	2	444	494	0.11
ks	2284	37	31	1136	1742	0.23	13	1182	2006	0.20
ul	2395	7	4	306	419	0.11	1	279	380	0.11
ft	3027	70	44	1390	1809	0.29	22	1241	1828	0.24
compress	3333	27	15	448	669	0.16	10	356	526	0.14
ratfor	5269	104	64	2893	4168	0.97	15	3079	4849	0.71
compiler	5326	39	17	2703	3524	0.72	6	3013	3961	0.64
assembler	6516	96	70	4061	5016	1.13	25	4004	4831	1.00
ML-typecheck	6752	296	238	6519	11168	1.67	40	21158	148172	4.00
eqntott	8117	232	163	4074	6264	1.10	56	3931	6408	1.02
simulator	10946	290	206	7344	14365	2.89	91	32521	130041	4.40
less-177	15179	357	259	10943	18121	3.65	141	58195	126138	5.83
li	16828	1736	1284	28386	166951	30.25	678	1260930	3285073	96.86
flex-2.4.7	29960	355	279	14678	23842	6.50	125	15246	25828	4.92
pmake	31210	775	550	21413	83686	14.94	291	213492	484218	21.85
make-3.72.1	36892	966	785	40498	283025	40.50	479	471810	1740032	54.40
inform-5.5	38874	505	422	35374	110442	18.64	260	168979	378157	20.13
tar-1.11.2	41497	831	674	24216	50122	9.29	413	153572	347858	14.92
screen-3.5.2	49292	894	781	39728	255411	40.57	553	384980	1044965	46.29
cvs-1.3	51223	755	581	30677	52408	12.91	263	89744	171794	13.20
sgmls-1.1	53874	1201	1075	46568	314633	53.55	830	901331	4086287	113.63
espresso	56938	1773	1231	41390	155881	27.89	515	545501	1126267	55.61
gawk-3.0.3	71091	1796	1438	36193	176097	31.16	615	590639	1575731	74.42
povray-2.2	87391	1578	1292	87139	336573	58.63	782	1382071	8037796	224.89

Table 3: Benchmark data for IF-Online and SF-Online

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, IF-Online and SF-Online, which use the online cycle detection and elimination algorithm described in Section 2.5. Furthermore, the oracle experiments directly compare the graph representations of IF 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 IF-Plain. 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 IF include the time to compute the least solution. Figure 7 plots the analysis time for both SF-Plain and IF-Plain 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 IF because cycles add many redundant variable-variable edges in IF that lead to redundant work.

The low numbers for the oracle runs IF-Oracle and SF-Oracle in Table 2 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 IF 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

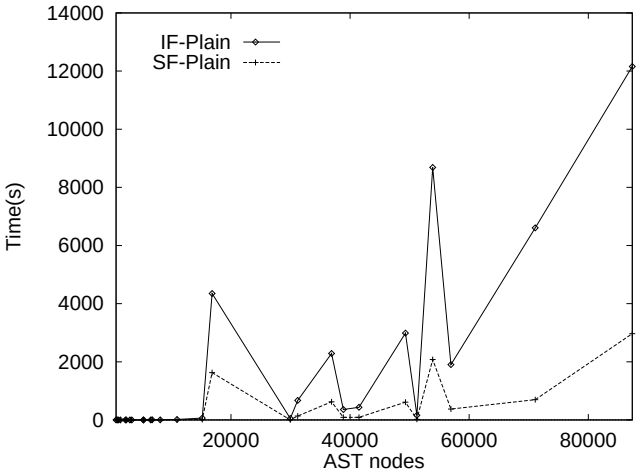

Figure 7: SF and IF without cycle elimination

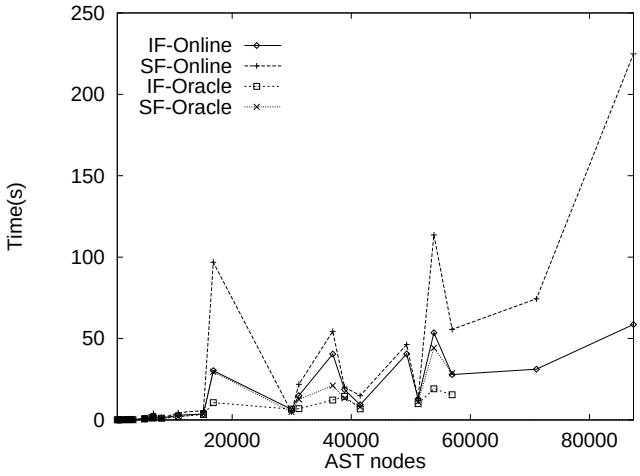

Figure 8: Analysis times with cycle detection and oracle

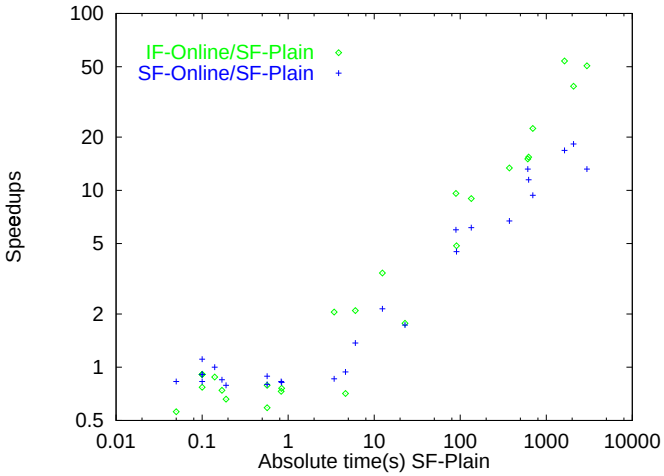

Figure 9: Speedups through online cycle detection

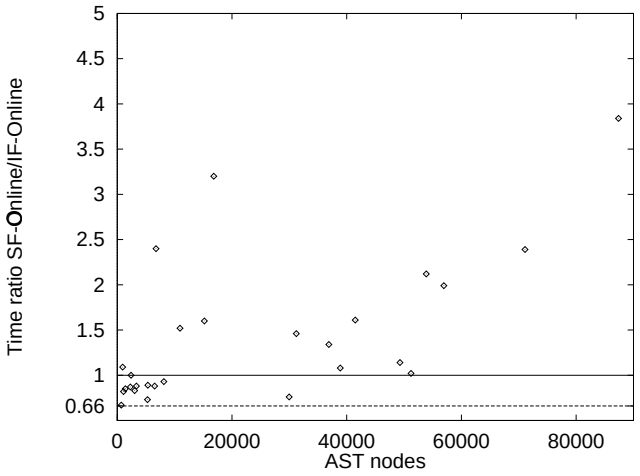

Figure 10: Speedups through inductive form

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.[5] 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 50% 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

[5] The outlier is the program `flex`; although `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.

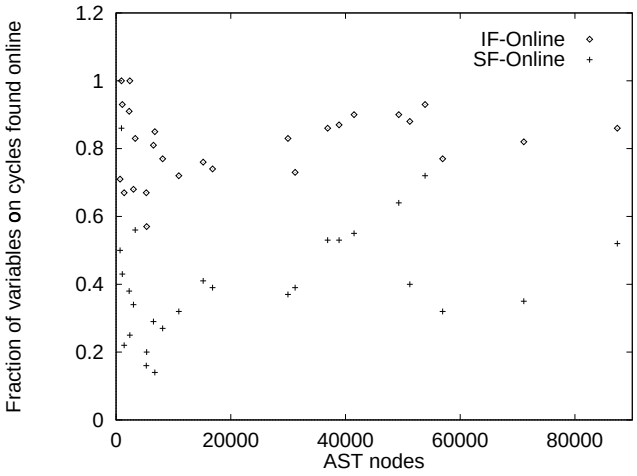

Figure 11: Fraction of variables on cycles found online

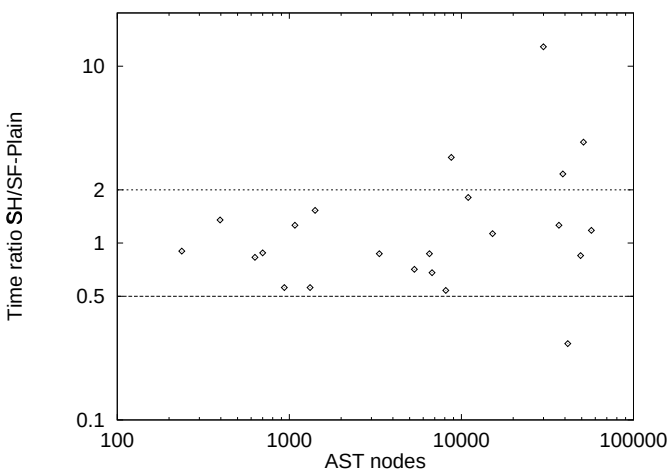

Figure 12: Relative execution times of Shapiro and Horwitz's SF implementation of C points-to analysis (SH) over SF-Plain

2 is small. Thus cycles of larger size are detected with small probability. IF counteracts this trend by adding transitive variable-variable edges, thereby shortening cycle lengths.

5 An Analytical Model

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

(Q1) Why is IF a better representation than SF?

(Q2) Why is partial online cycle elimination fast?

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

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 $\mathcal{R}$. 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.*, IF-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 IF.

Theorem 5.2 For random graphs with $p = \frac{2}{n}$, the expected number of variable nodes reachable through predecessor or successor chains in IF 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 the approximate densities of the initial and final IF graphs, respectively.

Theorem 5.1 answers the first question (Q1). It explains why SF-Oracle does on average 4.1 times more work than IF-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 IF 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 IF 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_i 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* ρ if (u, v) would be added by the graph closure rule considering only the nodes and edges of ρ.

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 $X^{\mathsf{SF}}_{(u,v)}$ 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 $\mathbf{E}(X^{\mathsf{SF}}_{(u,v)})$ of a given edge (u,v) and sum over all possible edges.

For the standard form we consider two kinds of edges, $(c,\mathcal{X})$ and (c,c'). We now calculate $\mathbf{E}(X^{\mathsf{SF}}_{(c,\mathcal{X})})$ and $\mathbf{E}(X^{\mathsf{SF}}_{(c,c')})$. Notice that the edge $(c,\mathcal{X})$ must be added through a simple path from c to $\mathcal{X}$. 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 $\mathcal{X}$ 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 $\mathcal{X}$ or c and c') is a path is p^{i+1}. We obtain the following:

$$\begin{aligned}\mathbf{E}(X^{\mathsf{SF}}_{(c,\mathcal{X})}) &= \sum_{i=1}^{n-1}\binom{n-1}{i}i!p^{i+1}\\ \mathbf{E}(X^{\mathsf{SF}}_{(c,c')}) &= \sum_{i=1}^{n}\binom{n}{i}i!p^{i+1}\end{aligned}$$

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

$$\mathbf{E}(X^{\mathsf{SF}}) = mn\mathbf{E}(X^{\mathsf{SF}}_{(c,\mathcal{X})}) + m(m-1)\mathbf{E}(X^{\mathsf{SF}}_{(c,c')})$$

5.2 Edge Additions in Inductive Form

Define the random variables $X^{\mathsf{IF}}_{(u,v)}$ 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: $(\mathcal{X}_1,\mathcal{X}_2)$, $(\mathcal{X},c)$, $(c,\mathcal{X})$, and (c_1,c_2). Notice that the probability that a given edge (u,v) is added through a simple path ρ of $l\geq 3$ nodes from u to v depends only on l. Thus we let $P_i(u,v)$ denote the probability that the edge (u,v) is added through a simple path from u to v with i nodes. We have the following equations:

$$\begin{aligned}\mathbf{E}(X^{\mathsf{IF}}_{(\mathcal{X}_1,\mathcal{X}_2)}) &= \sum_{i=1}^{n-2}\binom{n-2}{i}i!p^{i+1}P_{i+2}(\mathcal{X}_1,\mathcal{X}_2)\\ \mathbf{E}(X^{\mathsf{IF}}_{(\mathcal{X},c)}) &= \mathbf{E}(X^{\mathsf{IF}}_{(c,\mathcal{X})})\\ &= \sum_{i=1}^{n-1}\binom{n-1}{i}i!p^{i+1}P_{i+2}(\mathcal{X},c)\\ \mathbf{E}(X^{\mathsf{IF}}_{(c,c')}) &= \sum_{i=1}^{n}\binom{n}{i}i!p^{i+1}P_{i+2}(c,c')\end{aligned}$$

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 ρ from u to v with l nodes, the following holds:

1. $P_l(u,v)=\frac{2}{l(l-1)}$ if u and v are variable nodes;
2. $P_l(u,v)=\frac{1}{l-1}$ if one of u and v is a variable node and the other is a constructed node;
3. $P_l(u,v)=1$ 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 $P_l(u,v)\leq\frac{2}{l(l-1)}$. Recall that $o(\mathcal{X})$ is the index of variable $\mathcal{X}$. Assume the edge (u,v) is added through a path $(u,\mathcal{X}_1,\ldots,\mathcal{X}_{l-2},v)$, we claim that $o(u)$ and $o(v)$ are the smallest indices on the path, *i.e.*, $o(u)<o(\mathcal{X}_i)$ and $o(v)<o(\mathcal{X}_i)$ 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 $u\cdots\!\!\blacktriangleright\mathcal{X}_1$ and $\mathcal{X}_1\longrightarrow v$ are in the graph and these edges imply that $o(u)$ and $o(v)$ are less than $o(\mathcal{X}_1)$. Suppose the claim is true for paths with at most $k\geq 3$ nodes. Consider a path $(u,\mathcal{X}_1,\ldots,\mathcal{X}_{k-1},v)$ with $k+1$ nodes such that the edge (u,v) is added through the path. Notice there must exist a $\mathcal{X}_i$ with $1\leq i\leq k-1$ such that the edges $(u,\mathcal{X}_i)$ and $(\mathcal{X}_i,v)$ are added and $o(u)<o(\mathcal{X}_i)$ and $o(v)<o(\mathcal{X}_i)$. By induction, the claim holds for the shorter paths $(u,\ldots,\mathcal{X}_i)$ and $(\mathcal{X}_i,\ldots,v)$. Thus, $o(u)$ and $o(v)$ must be the smallest indices on the path. There are $n!$ possible orderings on the n variables and we claim that there are $\binom{n}{l}(2(l-2)!)(n-l)!$ of them satisfying the above condition. There are $\binom{n}{l}$ 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

$$\begin{aligned}P_l(u,v) &\leq \frac{\binom{n}{l}(2(l-2)!)(n-l)!}{n!}\\ &= \frac{2}{l(l-1)}.\end{aligned}$$

We now show $P_l(u,v)\geq\frac{2}{l(l-1)}$. Let $o(\cdot)$ be an ordering such that $o(u)$ and $o(v)$ are the smallest indices on the path $(u,\mathcal{X}_1,\ldots,\mathcal{X}_l,v)$. 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 $(u,\mathcal{X}_1,\ldots,\mathcal{X}_{k-1},v)$ with $k+1$ nodes such that $o(u)$ and $o(v)$ have the smallest indices. Let $\mathcal{X}_i$ be the node such that $o(\mathcal{X}_i)<o(\mathcal{X}_j)$ for all $1\leq j\leq k-1$ with $i\neq j$. By induction, the claim holds for the two subpaths $(u,\ldots,\mathcal{X}_i)$ and $(\mathcal{X}_i,\ldots,v)$, *i.e.*, the edges $(u,\mathcal{X}_i)$ and $(\mathcal{X}_i,v)$ are added through the respective subpaths. Thus, the edge (u,v) is added through the given path. Therefore, $P_l(u,v)\geq\frac{2}{l(l-1)}$. □

Since there are $m(m-1)$ edges of the form (c,c'), $2mn$ edges of the form $(\mathcal{X},c)$ or $(c,\mathcal{X})$, and $n(n-1)$ edges of the form $(\mathcal{X}_1,\mathcal{X}_2)$, the expected number of edge additions for the inductive form is given by

$$\begin{aligned}\mathbf{E}(X^{\mathsf{IF}}) = {}& m(m-1)\mathbf{E}(X^{\mathsf{IF}}_{(c,c')}) +\\ & 2mn\mathbf{E}(X^{\mathsf{IF}}_{(\mathcal{X},c)}) +\\ & n(n-1)\mathbf{E}(X^{\mathsf{IF}}_{(\mathcal{X}_1,\mathcal{X}_2)})\end{aligned}$$

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 $\frac{(n+m)^2}{n}$ edges. In practice, initial constraint graphs are sparse; all our benchmark programs produce initial graphs of approximately this density.

We have the following approximation [Knu73]

$$\sum_{i=1}^{n} \binom{n}{i} i! \left(\frac{1}{n}\right)^i \approx \sqrt{\frac{\pi n}{2}} \quad (2)$$

Using equation (2) we simplify $\mathbf{E}(X^{\mathsf{SF}})$ and $\mathbf{E}(X^{\mathsf{IF}})$ as follows

$$\begin{aligned} \mathbf{E}(X^{\mathsf{SF}}) &\approx m\left(\sqrt{\frac{\pi n}{2}} - 1\right) + \frac{m(m-1)}{n}\sqrt{\frac{\pi n}{2}} \\ &= \left(m + \frac{m(m-1)}{n}\right)\sqrt{\frac{\pi n}{2}} - m \\ \mathbf{E}(X^{\mathsf{IF}}) &\approx \frac{m(m-1)}{n}\sqrt{\frac{\pi n}{2}} + 2m\ln n + n \end{aligned}$$

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 $\frac{m}{n}$ is about $\frac{2}{3}$ (See Table 1). Thus, asymptotically, $\mathbf{E}(X^{\mathsf{SF}})/\mathbf{E}(X^{\mathsf{IF}})$ 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 $\mathcal{X}$ be any variable node and let $R_{\mathcal{X}}$ be the random variable denoting the number of nodes reachable from $\mathcal{X}$ through a predecessor chain. Using the same method for calculating the expected number of edge additions, we consider all simple paths starting with $\mathcal{X}$ involving only variable nodes. We thus have

$$\mathbf{E}(R_{\mathcal{X}}) \leq \sum_{i=1}^{n-1} \binom{n-1}{i} i! p^i \frac{1}{(i+1)!}$$

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

$$\begin{aligned} \mathbf{E}(R_{\mathcal{X}}) &\leq \sum_{i=1}^{n-1} \binom{n-1}{i} i! (\frac{k}{n})^i \frac{1}{(i+1)!} \\ &< \frac{1}{k}(e^k - 1 - k) \end{aligned}$$

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

$$\begin{aligned} \mathbf{E}(R_{\mathcal{X}}) &< \frac{1}{2}(e^2 - 1 - 2) \\ &\approx 2.2 \end{aligned}$$

completing the proof of Theorem 5.2. Note that for graphs denser than $p = \frac{2}{n}$ the value $\mathbf{E}(R_{\mathcal{X}})$ 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 $\mathcal{O}(n^3)$ 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

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[AW92] A. Aiken and E. Wimmers. Solving Systems of Set Constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[AW93] A. Aiken and E. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, January 1994.

[FA96] M. Fähndrich and A. Aiken. Making Set-Constraint Based Program Analyses Scale. In *First Workshop on Set Constraints at CP'96*, Cambridge, MA, August 1996. Available as Technical Report CSD-TR-96-917, University of California at Berkeley.

[FF97] C. Flanagan and M. Felleisen. Componential Set-Based Analysis. In PLDI'97 [PLD97].

[FFA97] J. Foster, M. Fähndrich, and A. Aiken. Flow-Insensitive Points-to Analysis with Term and Set Constraints. Technical Report UCB//CSD-97-964, U. of California, Berkeley, August 1997.

[FFK+96] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, chapter 10, pages 199–200. Addison Wesley, 1996.

[Hei92] N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.

[Hei94] N. Heintze. Set Based Analysis of ML Programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–17, June 1994.

[HJ90] N. Heintze and J. Jaffar. A decision procedure for a class of Herbrand set constraints. In *Symposium on Logic in Computer Science*, pages 42–51, June 1990.

[HM97] N. Heintze and D. McAllester. Linear-Time Subtransitive Control Flow Analysis. In PLDI'97 [PLD97].

[JM79] N. D. Jones and S. S. Muchnick. Flow Analysis and Optimization of LISP-like Structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[Knu73] D. Knuth. *The Art of Computer Programming, Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Mass., 2 edition, 1973.

[Koz93] D. Kozen. Logical Aspects of Set Constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proc. 1993 Conf. Computer Science Logic (CSL '93)*, volume 832 of *Lecture Notes in Computer Science*, pages 175–188. Springer-Verlag, 1993.

[Mos96] C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MW97] S. Marlow and P. Wadler. A Practical Subtyping System For Erlang. In *Proceedings of the International Conference on Functional Programming (ICFP '97)*, June 1997.

[PLD97] *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[Pot96] F. Pottier. Simplifying Subtyping Constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, January 1996.

[PS91] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the ACM Conference on Object-Oriented programming: Systems, Languages, and Applications*, October 1991.

[Rey69] J. C. Reynolds. *Automatic Computation of Data Set Definitions*, pages 456–461. Information Processing 68. North-Holland, 1969.

[SH97] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Shi88] O. Shivers. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[Shm83] O. Shmueli. Dynamic Cycle Detection. *Information Processing Letters*, 17(4):185–188, 8 November 1983.

[Ste96] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[TS96] V. Trifonov and S. Smith. Subtyping Constrained Types. In *Proceedings of the 3rd International Static Analysis Symposium*, pages 349–365, September 1996.