Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs*

Zhendong Su[†] zhendong@cs.berkeley.edu
Manuel Fähndrich[‡] maf@microsoft.com
Alexander Aiken[†] aiken@cs.berkeley.edu

EECS Department
University of California, Berkeley

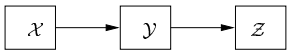
Abstract

Inclusion-based program analyses are implemented by adding new edges to directed graphs. In most analyses, there are many different ways to add a transitive edge between two nodes, namely through each different path connecting the nodes. This path redundancy limits the scalability of these analyses. We present *projection merging*, a technique to reduce path redundancy. Combined with cycle elimination [7], projection merging achieves orders of magnitude speedup of analysis time on programs over that of using cycle elimination alone.

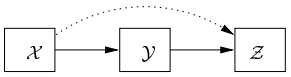
1 Introduction

Many constraint-based program analyses can be expressed in terms of set inclusion constraints. These analyses include shape analysis, closure analysis, soft typing systems, receiver-class analysis for object-oriented programs, and points-to analysis for pointer-based programs, among others [4, 5, 8, 11, 15, 16, 19, 20, 22].

Inclusion-based constraint systems are commonly represented as directed graphs. Nodes represent abstract program values and edges represent the set inclusion relation. For example, the constraints $\mathcal{X} \subseteq \mathcal{Y} \subseteq \mathcal{Z}$ are represented by the directed graph with nodes $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ and directed edges $(\mathcal{X}, \mathcal{Y})$ and $(\mathcal{Y}, \mathcal{Z})$ for the inclusion:



Solving constraints can be viewed as adding new edges to the graph to denote relationships implied by the initial system. By making implied constraints explicit, it becomes straightforward to answer queries about whether particular relationships hold. In the example above, adding the edge $(\mathcal{X}, \mathcal{Z})$ makes explicit the implied constraint $\mathcal{X} \subseteq \mathcal{Z}$:



It is well-known that solving many forms of inclusion constraints reduces to computing the dynamic transitive closure of the underlying graph representation, or equivalently to a context-free grammar reachability problem [17]. Thus, the worst case complexity of the best known algorithms for many inclusion-based analyses is cubic in the size of the analyzed program [13, 17]. Standard implementations of these analyses are efficient for small to medium size programs, but usually do not scale to large programs.

There are several reasons that straightforward implementations are impractical:

1. *Memory Requirement*
   The transitive closure of a directed graph with n nodes may have $\mathcal{O}(n^2)$ edges. Standard implementations compute an explicit representation of the transitive closure and therefore may consume a lot of memory.

2. *Cyclic Constraints*
   For inclusion constraints, if there is a cycle of inclusions (constraints of the form $\mathcal{X}_1 \subseteq \mathcal{X}_2 \subseteq \mathcal{X}_3 \ldots \subseteq \mathcal{X}_m \subseteq \mathcal{X}_1$ where the $\mathcal{X}_i$'s are set variables), up to m^2 edges may be added between the nodes on the cycle. This is undesirable because these variables are obviously equivalent in all solutions of the constraints.

3. *Redundant Paths*
   In constraint graphs, there are usually many different ways to add a transitive edge, namely through the many different paths that connect two nodes.

In [7], the authors present a simple but effective cycle elimination algorithm addressing the problem of cyclic constraints. Andersen's points-to analysis [5], a cubic time algorithm, is used as a case study.

In the same paper, the authors propose a representation of inclusion constraints, called *inductive form*, which helps overcome the problem of high memory usage. Instead of explicitly representing the least solution, an implicit representation is maintained. This implicit representation is sparser than standard implementations and is better suited for use with the suggested cycle elimination algorithm. A separate post-processing phase is needed to compute the least solution of the constraints. The separation into these two phases also helps reduce memory usage.

However, the problem of redundant paths is not addressed in [7]. The techniques in [7] are very effective for

*This research was supported in part by the National Science Foundation Young Investigator Award No. CCR-9457812 and NASA Contract No. NAG2-1210.

[†]EECS Department, University of California, Berkeley, 387 Soda Hall #1776, Berkeley, CA 94720-1776

[‡]One Microsoft Way, Redmond, WA 98052-6399

programs up to 50,000 lines of preprocessed code, but even for these programs, the problem with redundant paths is evident. For example, for the largest program considered in [7], redundant edge additions dominate (see [7], Table 3). On average, each transitive edge is added in four different ways.

This paper addresses the problem of redundant paths via a technique called *projection merging*. Combined with cycle elimination, projection merging yields orders of magnitude speedup for large programs.

The basic insight of the technique is the observation that inductive form treats different kinds of edges differently: *variable-variable* edges (edges between two variables) are less likely to be added than *source-sink* edges (edges between two constructed terms). Thus, translating the constraint graphs to expose more variable-variable paths can reduce redundant edge additions. Projection merging itself is quite simple to explain and implement, but the analysis of its behavior is subtle because of positive interactions with cycle elimination.

To validate the technique, we study the same points-to analysis for C [5, 21] used in [7] (See Appendix A for a brief description of the analysis). `gimp`, a program that before preprocessing has more than 440,000 non-comment lines of C, can be analyzed in less than half an hour with projection merging and cycle elimination. With cycle elimination alone, the analysis did not finish after more than 33 hours of processing.

The rest of the paper is structured as follows. In Section 2, we briefly describe the constraint language and inductive form. Section 3 presents projection merging. Section 4 presents experimental results to demonstrate the efficacy of the technique. Section 5 discusses related work, and Section 6 concludes.

2 Preliminaries

In this section, we introduce a constraint language and standard resolution rules. We also present *inductive form*, the particular constraint graph representation we use.

2.1 Set Constraints

This subsection covers basic material on set constraints. In particular, we work with a subset of the full language of set constraints [2,12]. Constraints are of the form $L \subseteq R$, where L and R are set expressions. Set expressions consist of set variables $\mathcal{X}, \mathcal{Y}, \ldots$ drawn from a countable set of variables *Vars*, terms constructed from n-ary constructors $c \in Con$, projection expressions, an empty set 0, and a universal set 1.

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

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.

Definition 2.1 (Positive and Negative Positions) In a constraint $se \subseteq se'$, we say the set expression se appears *positively* and the set expression se' appears *negatively*. The position of a subexpression is defined inductively.

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

Figure 1: Resolution rules

- If $c(se_1, \ldots, se_i, \ldots, se_n)$ appears positively in a constraint, then se_i appears positively if c is covariant in i; se_i appears negatively if c is contravariant in i. The case when $c(se_1, \ldots, se_i, \ldots, se_n)$ appears negatively is symmetric.
- If $proj(c, i, se)$ appears negatively in a constraint, then se appears negatively if c is covariant in i; se appears positively if c is contravariant in i. We require a projection expression $proj(c, i, se)$ to appear only in negative positions.

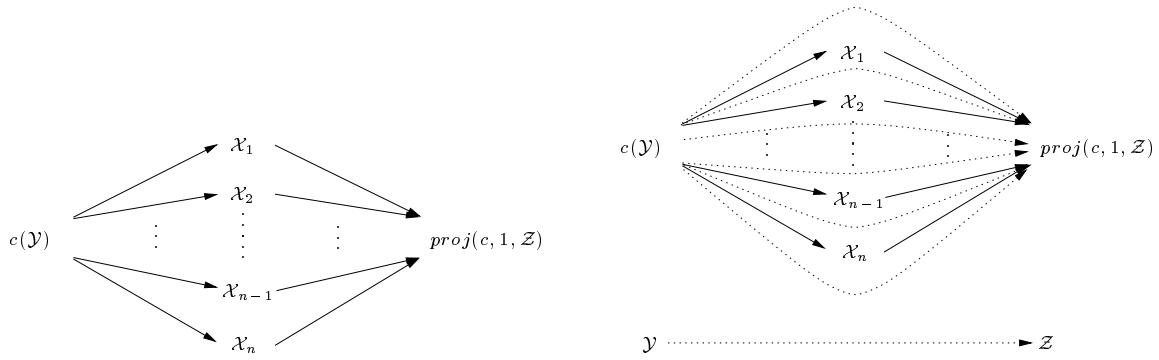
A projection expression $proj(c, i, se)$ has the effect of selecting the ith component se' of any expression with head constructor c on the left-hand side of the constraint, and then adding the new constraint $se' \subseteq se$. For example, a constraint

$$c(\mathcal{X}, \mathcal{Y}) \subseteq proj(c, 1, \mathcal{Z})$$

implies the constraint (see Figure 1)

$$\mathcal{X} \subseteq \mathcal{Z}$$

assuming c is covariant in its first argument. Note that $proj(c, i, se)$ has no effect if there is no expression with head constructor c; the constraint $d(\ldots) \subseteq proj(c, i, se)$ is trivially satisfied if $d \neq c$ (see Figure 1). The notation $proj(c, i, se)$ is closely related to the more standard set constraint notation $c^{-i}(se)$. We discuss why we need this new notation in Section 3.



(a) Before closure

(b) After closure

Figure 2: Example constraint graph A

2.2 Constraint Graphs

This subsection reviews the framework of [7].

The solved form of a constraint system is a directed graph $G = (V, E)$ closed under a set of rules for adding edges. 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. A constraint is atomic if it is one of the four forms

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

We use the resolution rules shown in Figure 1 to rewrite constraints into atomic form (note that non-atomic constraints are not represented in the constraint graph). Each rule states that the system of constraints on the left has the same solutions as the system on the right[1]. In a resolution engine these rules are used as left-to-right rewrite rules.

The key idea of inductive form is 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. A variable-variable edge $\mathcal{X} \longrightarrow \mathcal{Y}$ is represented as a successor edge on $\mathcal{X}$ if $o(\mathcal{X}) > o(\mathcal{Y})$; otherwise, it is represented as a predecessor edge on $\mathcal{Y}$.

The choice of the order function $o(\cdot)$ can affect the size of the closed constraint graph and the amount of work required for the closure. *Generation order*, the order in which variables are created as part of building the initial system of constraints, does very well in practice, and we have found experimentally that it is difficult to pick a better order function.

The other two kinds of edges are associated with the variables. A source-variable constraint $c(\dots) \subseteq \mathcal{X}$ is represented as a predecessor edge on $\mathcal{X}$, and a variable-sink constraint $\mathcal{X} \subseteq proj(\dots)$ or $\mathcal{X} \subseteq c(\dots)$ is represented as a successor edge on $\mathcal{X}$. The closure rule **Transitive** is given as:

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

[1]For a treatment of the semantics of set constraints, see [2, 3, 12].

Notice that L may be a source or a variable and R may be a sink or a variable. This closure rule combined with the resolution rules in Figure 1 produces a final graph in inductive form [3]. The least solution of the constraints is not explicit in inductive form, but it is easily computed by:

$$Sol_{least}(\mathcal{Y}) = \{c(\dots) \mid c(\dots) \in pred(\mathcal{Y})\} \cup \bigcup_{\mathcal{X} \in pred(\mathcal{Y})} Sol_{least}(\mathcal{X})$$

Note that this computation amounts to computing an *acyclic* transitive closure since $o(pred(\mathcal{Y})) < o(\mathcal{Y})$ for any set variable $\mathcal{Y}$. Furthermore, the least solution can be computed on demand. In practice, it is rarely necessary to compute the least solution for every variable in a constraint system because most applications require the solution of only a subset of all set variables.

A path in a constraint graph consists of a sequence of nodes and inclusions

$$L \subseteq \mathcal{X}_1 \subseteq \dots \subseteq \mathcal{X}_n \subseteq R$$

where L may be a source or a variable, and R a sink or a variable.

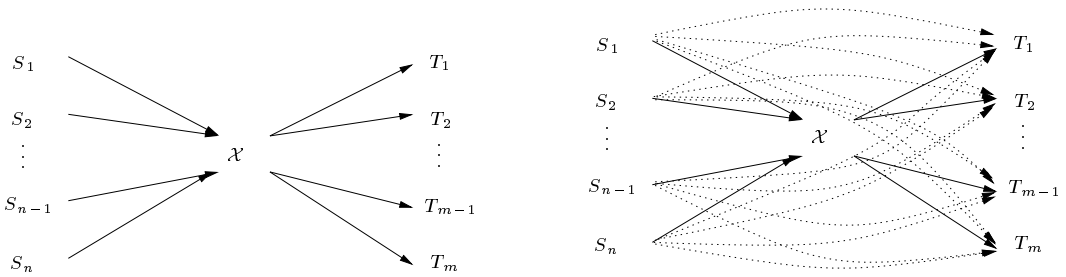
Definition 2.2 (Inductive Path) A path is *inductive* if an edge is added between the two end points on the path by applying the closure rule **Transitive** to the inclusions along the path. Equivalently, one can show that a path is inductive if and only if the two end points have the smallest indices (under the order function) on the path [7]. Sources and sinks are taken to have the smallest indices, *i.e.*, $-\infty$.

2.3 Redundant Paths

In inclusion constraint graphs, there are usually many paths connecting two nodes. Consider the example constraint graph in Figure 2a. In the graph, there are n paths connecting the two nodes $c(\mathcal{Y})$ and $proj(c, 1, \mathcal{Z})$, namely through the n variables $\mathcal{X}_1$ to $\mathcal{X}_n$. Thus, the implied constraint

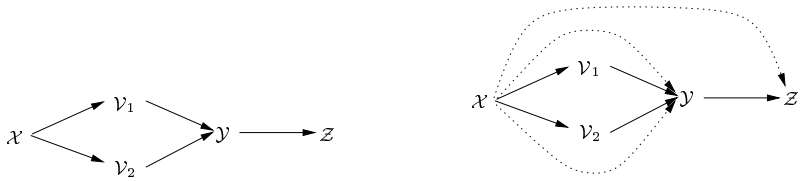
$$c(\mathcal{Y}) \subseteq proj(c, 1, \mathcal{Z})$$

is discovered n times, resulting in $n-1$ redundant additions of the edge $(\mathcal{Y}, \mathcal{Z})$ to the constraint graph. Figure 2b shows



(a) Before closure (b) After closure

Figure 3: Example constraint graph B



(a) Before closure (b) After closure

Figure 4: Example constraint graph C

the graph after closure. The n dotted lines between $c(\mathcal{Y})$ and $proj(c, 1, \mathcal{Z})$ do not appear in the final graph. We show them only to stress that the constraint

$$c(\mathcal{Y}) \subseteq proj(c, 1, \mathcal{Z})$$

is resolved n times.

For a program analysis problem, the aforementioned redundancy occurs because it is commonplace that some constructed value (a source) flows through many intermediate variables before a field is finally projected out. A related problem is depicted in Figure 3a, in which there are n sources flowing into the variable $\mathcal{X}$ and m sinks to project from $\mathcal{X}$. In this case, nm edges are matched up, one for each (S_i, T_j) for $1 \leq i \leq n$ and $1 \leq j \leq m$. The graph after closure is shown in Figure 3b.

Although every inductive path may cause an edge to be added between the two end points, the number of edge additions can be smaller. As an example, consider the constraint graph shown in Figure 4a. We assume the following ordering $o(\cdot)$ on the variables

$$o(\mathcal{X}) < o(\mathcal{Z}) < o(\mathcal{Y}) < o(\mathcal{V}_1) < o(\mathcal{V}_2)$$

Notice that there are two inductive paths between $\mathcal{X}$ and $\mathcal{Z}$, namely

$$\mathcal{X} \longrightarrow \mathcal{V}_1 \longrightarrow \mathcal{Y} \longrightarrow \mathcal{Z}$$

and

$$\mathcal{X} \longrightarrow \mathcal{V}_2 \longrightarrow \mathcal{Y} \longrightarrow \mathcal{Z}$$

However, the edge $(\mathcal{X}, \mathcal{Z})$ is only added once, through the path

$$\mathcal{X} \cdots\!\blacktriangleright \mathcal{Y} \longrightarrow \mathcal{Z}$$

because the redundant addition of $\mathcal{X} \cdots\!\blacktriangleright \mathcal{Y}$ does not cause the edge $\mathcal{X} \cdots\!\blacktriangleright \mathcal{Z}$ to be added a second time. The graph after closure is shown in Figure 4b.

Definition 2.3 (Join Point) Let p be an inductive path. Let $node(p)$ denote the set of nodes on the path. Further let $head(p)$ and $tail(p)$ denote the two end-points on the path. The *join point* of p, denoted by $join(p)$, is the variable with the minimum index among the variables $node(p) \setminus \{head(p), tail(p)\}$, *i.e.*, $o(join(p)) \leq o(\mathcal{X})$ for all $\mathcal{X} \in (node(p) \setminus \{head(p), tail(p)\})$

The following lemma (Lemma 2.4) gives a characterization of the number of edge additions between two nodes.

Lemma 2.4 Let n_1 and n_2 be two nodes in a constraint graph, and $P(n_1, n_2)$ be the set of inductive paths connecting the two nodes. The number of edge additions of (n_1, n_2), denoted by $\#(n_1, n_2)$, is given by

$$\#(n_1, n_2) = \mathbf{Card}(\{join(p) \mid p \in P(n_1, n_2)\})$$

where $\mathbf{Card}(\cdot)$ denotes the cardinality of a set.

Proof. [Sketch]
It suffices to demonstrate the following:

1. For any $p_1, p_2 \in P(n_1, n_2)$ with $join(p_1) = join(p_2)$, the edge (n_1, n_2) is added once through these two paths.
2. For any $p_1, p_2 \in P(n_1, n_2)$ with $join(p_1) \neq join(p_2)$, the edge (n_1, n_2) is added twice, once through each path.

Both can be shown by observing that the edge added between n_1 and n_2 along an inductive path p is added by a final application of rule **Transitive** to a 2-edge segment of the form

$$n_1 \cdots\cdots\blacktriangleright\ join(p)\ \cdots\cdots\blacktriangleright\ n_2$$

□

Because edges between variables are added only on inductive paths, inductive form gives a sparse representation for variable-variable edges. But every path in the constraint graph linking a source and a sink is inductive, and thus may cause an edge addition. It is shown in [7] that under a simple random graph model, the work to close a constraint graph is dominated by the addition of edges between sources and sinks, which is expected to be $O(n\sqrt{n})$, where n is the number of nodes in the graph. The cost of all other edges is expected to be $O(n \ln n)$. These results are supported by experimental data. For example, for the program `mume`, the number of edge additions between sources and sinks significantly dominate the rest (see Table 2, the columns labeled "s-s", edge additions between sources and sinks, and "Work Other", all other edge additions, under the column "Cycle Elimination Only": 13748716 versus 1106200). Thus, improving performance requires reducing the number of redundant edge additions between sources and sinks.

3 Projection Merging

This section presents *projection merging*. Before we present the technique, we first provide some intuition with two observations:

1. *In inductive form, variable-variable edges are sparse.*
   In inductive form, any path between two constructed terms may cause an edge to be added. For paths between variables, an edge is added only if the two variables have smaller indices than the other variables on the path. Thus variable-variable paths are preferred to source-sink paths.

2. *There are a small number of constructors with small arity.*
   For many constraint-based program analyses, only a small number of non-constant constructors are used (even though there may be many constant constructors serving, *e.g.*, as program point labels). Furthermore, these non-constant constructors usually have small arity. For our example points-to analysis, only two constructors are used, a *ref* constructor to denote locations (arity 3) and a *lam* constructor to denote functions (arity 3).

From the first observation we can conclude that by transforming the constraint system from paths between constructed terms to paths between variables, we may reduce the number of redundant edge additions.

Consider again the constraint graph in Figure 2a. The ultimate effect of this graph is to discover the constraint

$$c(\mathcal{Y}) \subseteq proj(c, 1, \mathcal{Z})$$

between the source $c(\mathcal{Y})$ and the sink $proj(c, 1, \mathcal{Z})$. This constraint is then resolved to the variable-variable constraint $\mathcal{Y} \subseteq \mathcal{Z}$.

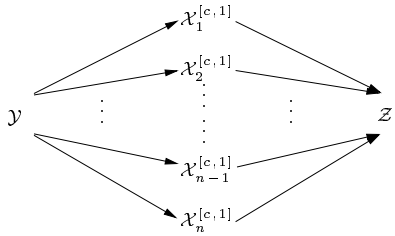


Figure 5: Transformed constraint graph A (Figure 2a)

If there were a way to bypass source $c(\mathcal{Y})$ and sink $proj(c, 1, \mathcal{Z})$ and instead deal directly with the flow of information from $\mathcal{Y}$ to $\mathcal{Z}$, then we could avoid the redundant edge additions between the source and sink and deal with only variable-variable paths. The key is to observe that when a constraint $\mathcal{X}_i \subseteq proj(c, 1, \mathcal{Z})$ in Figure 2a is first formed, we know that whatever lower bound $\mathcal{X}_i$ may have it will be projected by $\mathcal{X}_i$'s upper bound. Thus we may as well replace the constraint $\mathcal{X}_i \subseteq proj(c, 1, \mathcal{Z})$ by a constraint $\mathcal{X}_i^{[c,1]} \subseteq \mathcal{Z}$, where $\mathcal{X}_i^{[c,1]}$ is a variable standing for $c^{-1}(\mathcal{X}_i)$. Note that the new constraint involves only variables.

A naive method for systematically performing this transformation goes as follows. With each variable $\mathcal{X}$, constructor c, and argument position i in the original constraint system associate a projection of the form $proj(c, i, \mathcal{X}^{[c,i]})$, where $\mathcal{X}^{[c,i]}$ is a variable. The idea is that $\mathcal{X}^{[c,i]}$ represents $c^{-i}(\mathcal{X})$, the sum of all ith components of terms with head constructor c in the set $\mathcal{X}$. With this association of a variable with all its possible projections, we can transform the constraint system to expose the variable-variable paths not explicit before. By the second observation, not very many such projections are required.

As an example, we explain how the graph in Figure 2a may be transformed into the graph in Figure 5. We associate with each $\mathcal{X}_i$ a projection $proj(c, 1, \mathcal{X}_i^{[c,1]})$. Now any constraint

$$\mathcal{X}_i \subseteq proj(c, 1, \mathcal{Y})$$

is equivalent to

$$\mathcal{X}_i^{[c,1]} \subseteq \mathcal{Y}$$

Systematically applying this rule to the graph in Figure 2a gives the graph in Figure 5.

As indicated above, this approach has problems.

- *The approach is static.*
  By *static* we mean that the association of variables with their projections is made before performing the graph closure. For some variables, it is unnecessary to associate the variable with its possible projections. In addition, we may need to project only a particular field from a variable.

- *The approach is incomplete.*
  The association of a variable and its projections is made for variables appearing in the original system. What about the newly created variables $\mathcal{X}^{[c,i]}$?

Furthermore, these newly created variables must be assigned indices in the total order. If the indices are not chosen

(Proj-Creation)

$$S_0 \cup \{\mathcal{X} \subseteq proj(c,i,se)\} \quad \Rightarrow \quad \begin{cases} S_0 \cup & \begin{cases} \{\mathcal{X} \subseteq proj_s(c,i,\mathcal{X}^{[c,i]}), \mathcal{X}^{[c,i]} \subseteq se\} & c \text{ covariant in } i \\ \{\mathcal{X} \subseteq proj_s(c,i,\mathcal{X}^{[c,i]}), se \subseteq \mathcal{X}^{[c,i]}\} & c \text{ contravariant in } i \end{cases} \\ & \qquad \text{if } (\mathcal{X} \subseteq proj_s(c,i,\mathcal{X}^{[c,i]})) \notin S_0 \\ S_0 \cup & \begin{cases} \{\mathcal{X}^{[c,i]} \subseteq se\} & c \text{ covariant in } i \\ \{se \subseteq \mathcal{X}^{[c,i]}\} & c \text{ contravariant in } i \end{cases} \\ & \qquad \text{otherwise.} \end{cases}$$

(Proj-Transitive)

$$S_0 \cup \{\mathcal{X} \subseteq \mathcal{Y}, \mathcal{Y} \subseteq proj_s(c,i,\mathcal{Y}^{[c,i]})\} \quad \Rightarrow \quad S_0 \cup \{\mathcal{X} \subseteq \mathcal{Y}, \mathcal{Y} \subseteq proj_s(c,i,\mathcal{Y}^{[c,i]}), \mathcal{X} \subseteq proj(c,i,\mathcal{Y}^{[c,i]})\}$$
$$\text{— if } o(\mathcal{X}) < o(\mathcal{Y})$$

Figure 6: Rules specific for projection merging

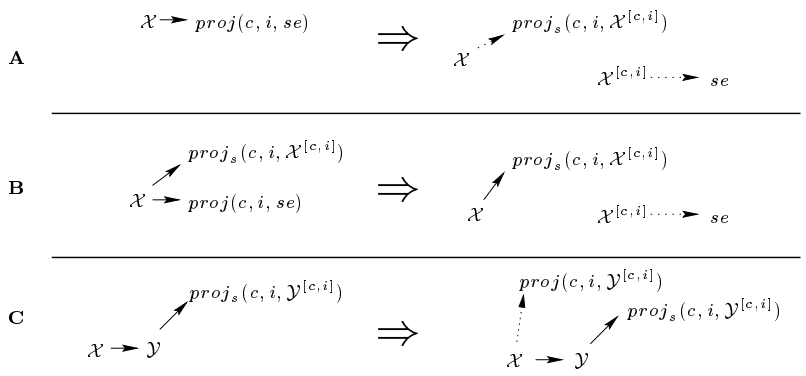


Figure 7: Graph representation of projection merging rules

with care, the benefits are lost. To stress this point, consider the example in Figure 2a. If the newly created variables $\mathcal{X}_1^{[c,1]}$ through $\mathcal{X}_n^{[c,1]}$ have large indices, we still add the edge $(\mathcal{Y}, \mathcal{Z})$ n times (due to inductive paths), which is no better than the original situation.

These problems can all be solved with an incremental approach. A variable $\mathcal{X}^{[c,i]}$ is needed only if the variable $\mathcal{X}$ has a projection upper bound, *i.e.*, we have $\mathcal{X} \subseteq proj(c,i,se)$ for some constructor c, index i, and expression se. We address the technical details of generating projection variables $\mathcal{X}^{[c,i]}$ "on the fly" in the next section, with an explanation of why it helps to reduce the number of redundant edge additions.

Note that in standard set constraint implementations such as SBA, constructed values (including projections) do not propagate backwards. Thus, projection upper bounds are not added to variables through constraint resolution. They only appear through the initial constraints. However, in inductive form, constructed values (including projections) can be propagated backwards along variable paths. Thus, a variable might accumulate many projections for the same constructor and index pair. Therefore, we need to apply the transformation dynamically for the projections propagated through constraint resolution.

We can now explain why we introduce the notation $proj(c,i,se)$ instead of using the more standard notation $c^{-i}(se)$. Consider a constraint

$$\mathcal{X} \subseteq proj(c,i,se)$$

which would be written

$$c^{-i}(\mathcal{X}) \subseteq se$$

in standard notation. The $c^{-i}(\ldots)$ form obscures the implied upper bound on $\mathcal{X}$, which is explicit using $proj(\ldots)$. Using $c^{-i}(\ldots)$ would thus complicate the definition of inductive form and the resolution rules and obscure the key point that sources propagate forward through the graph as lower bounds while sinks propagate backwards as upper bounds. As noted above, this is relevant only for inductive form implementations: using $proj(\ldots)$ and projection merging would have no benefit for standard implementations such as SBA.

3.1 The Algorithm

This subsection shows how to associate a variable with its *generic projections*. We extend the closure rules in Figure 1

and the **Transitive** rule. Then we show how to choose a variable ordering that ensures the efficient termination of the modified rules. Finally, we explain why this approach works well in reducing the number of redundant edge additions.

We first discuss a transformation on the initial constraints that removes all nested projections, *i.e.*, projections within projections or within constructed terms. We describe the case for projection expressions, and the case for constructors is similar. Replace each constraint with nested projections

$$se_1 \subseteq proj(c, i, se_2)$$

by

$$\begin{cases} \{se_1 \subseteq proj(c, i, \mathcal{X}),\ \mathcal{X} \subseteq se_2\} & \text{if } c \text{ is covariaint in } i \\ \{se_1 \subseteq proj(c, i, \mathcal{X}),\ se_2 \subseteq \mathcal{X}\} & \text{if } c \text{ is contravariant in } i \end{cases}$$

where $\mathcal{X}$ is a fresh set variable not appearing in the current constraint system. By repeated use of this rule, a system of initial constraints with nested projection operators is transformed into a system of constraints without nested projections in linear time, and the resulting constraints have size linear in the size of the original constraints. For later discussion we only consider constraints without nested projections. Note that the resolution rules in Figure 1 do not reintroduce nested projections.

Projection merging consists of two additional resolution rules **Proj-Creation** and **Proj-Transitive**. The two rules are shown in Figure 6. Figure 7 explains the rules in terms of constraint graphs.

The **Proj-Creation** rule is used to merge projection upper bounds. This rule uses a special marked sink, $proj_s(c, i, se)$. This *special projection* has the same meaning as the normal projection $proj(c, i, se)$. If for a set variable $\mathcal{X}$, there is not yet a constraint

$$\mathcal{X} \subseteq proj_s(c, i, \mathcal{X}^{[c,i]})$$

when a constraint

$$\mathcal{X} \subseteq proj(c, i, se)$$

is to be added to the constraint system, we replace the constraint by the two constraints

$$\mathcal{X} \subseteq proj_s(c, i, \mathcal{X}^{[c,i]})$$

and

$$\begin{cases} \mathcal{X}^{[c,i]} \subseteq se & \text{if } c \text{ is covariant in } i \\ se \subseteq \mathcal{X}^{[c,i]} & \text{if } c \text{ is contravariant in } i \end{cases}$$

where $\mathcal{X}^{[c,i]}$ is a fresh set variable. The variable $\mathcal{X}^{[c,i]}$ is called the *generic projection variable* of $\mathcal{X}$ for constructor c at index i. This case is depicted as part A in Figure 7. If, on the other hand, there is already a special projection on $\mathcal{X}$, adding the constraint

$$\mathcal{X} \subseteq proj(c, i, se)$$

causes

$$\mathcal{X}^{[c,i]} \subseteq se$$

or

$$se \subseteq \mathcal{X}^{[c,i]}$$

to be added depending on the variance of c in i. This case is depicted as part B in Figure 7.

Note that for each variable, there is at most one projection for a constructor and index pair (c, i). By observation 2 in Section 3, the total number of possible special projections is relatively small.

The new transitive rule **Proj-Transitive** deals with the case when a special projection is added transitively on a variable. The rule simply converts a special projection to a normal projection because to the variable $\mathcal{X}$, the projection $proj_s(c, i, \mathcal{Y}^{[c,i]})$ is only a normal sink. This rule is depicted as part C in Figure 7. Note that a constructed expression can also appear in the position of the variable $\mathcal{X}$, the rule for a constructed expression is the same.

In cases where the new rules are inapplicable, rules in Figure 1 and the **Transitive** rule are applied. To close a graph, we repeatedly apply all the rules until no rule is applicable.

The following theorem (Theorem 3.1) states that the resolution of the constraints under the new rules yields the same results as that under the standard rules in Figure 1 and the **Transitive** rule.

Theorem 3.1 (Correctness) The modified resolution rules preserve the least solution of the system. More precisely, for any variable $\mathcal{X}$ appearing in the original constraint system, the least solution for $\mathcal{X}$ is the same under both sets of closure rules.

Proof. [Sketch]

Simply notice that with the projection merging rules any value (a source) that is a lower bound on a non-generic projection variable $\mathcal{X}$ under the original rules still is a lower bound on $\mathcal{X}$ under the new rules, and *vice versa*.

□

We also need to establish that resolution under the new rules terminates. It suffices to show that only finitely many generic projection variables are generated. We assume that our order function $o(\cdot)$ is a one-to-one function that maps all set variables (an infinite set) to natural numbers $\mathbb{N}$.

Theorem 3.2 (Termination) For any constraint set S_0 and any one-to-one order function $o(\cdot)$ that maps set variables to natural numbers $\mathbb{N}$, finitely many generic projection variables are generated during the resolution of S_0 under the extended resolution rules.

Proof. As noted above, for each constructor and index pair (c, i), and each set variable, at most one generic projection variable is created. Thus it suffices to bound the number of variables that may have a projection upper bound. Let

$$Omax = \mathbf{max}\ \{\ o(\mathcal{X}) \mid \mathcal{X} \text{ is a variable appearing in } S_0\ \}$$

We claim that for each constructor and index pair (c, i), at most $Omax$ number of generic projection variables are created. The claim follows if there are at most $Omax$ variables that can have projection upper bounds with c as the constructor and i as the index, namely the set of variables having index no greater than $Omax$.

If a variable $\mathcal{X}$ has a projection upper bound, then there must exist an inductive path from $\mathcal{X}$ to a projection. We show by induction that all variables $\mathcal{X}$ with projection or special projection upper bounds have index $o(\mathcal{X}) \leq Omax$. Clearly this is true before any resolution rules are applied. There are two cases for the inductive step:

1. **Proj-Creation** creates special projection constraints $\mathcal{X} \subseteq proj_s(c, i, \mathcal{X}^{[c,i]})$ from existing constraints $\mathcal{X} \subseteq proj(c, i, \mathcal{Z})$. By the inductive hypothesis, $o(\mathcal{X}) \leq Omax$.

2. **Proj-Transitive** adds the constraint

$$\mathcal{X} \subseteq proj(c, i, \mathcal{Y}^{[c,i]})$$

from constraints

$$\mathcal{X} \subseteq \mathcal{Y} \subseteq proj_s(c, i, \mathcal{Y}^{[c,i]})$$

We know $o(\mathcal{X}) < o(\mathcal{Y})$ (by the conditions of **Proj-Transitive**) and $o(\mathcal{Y}) \leq Omax$ (by the inductive hypothesis). Therefore $o(\mathcal{X}) < Omax$.

Thus all such variables must have index no greater than $Omax$. Since there are finitely many constructor and index pairs, the number of new projection variables is bounded by $\mathcal{O}(MOmax)$, where M is the number of distinct constructor and index pairs.

□

Although resolution terminates for any order function, the ordering still affects the number of generic projection variables. The following corollary suggests a good ordering to use.

Corollary 3.3 If for each generic projection variable $\mathcal{X}^{[c,i]}$, we set $o(\mathcal{X}^{[c,i]}) > Omax$, then at most $\mathcal{O}(MN)$ new variables are generated, where M is the number of constructor and index pairs and N is the number of set variables in the original constraint system. In particular, if the number of constructor and index pairs is fixed, only a linear number of generic projection variables are generated.

Proof. There are only N set variables with index no greater than $Omax$. □

Note it follows that if each generic projection variable $\mathcal{X}^{[c,i]}$ has index larger than $Omax$, we only generate generic projection variables for the set variables in the original constraint system. We never generate a generic projection variable for another generic projection variable.

Recall that most constraint-based program analyses use a small number of non-constant constructors and these constructors usually have small arity. Thus the number of generated generic projection variables tends to be small, which is what makes projection merging viable.

To implement the ordering suggested by Corollary 3.3, one can use some large offset to separate the generic projection variables from other variables; for each generic projection variable we assign it an index that is its generation order plus this chosen large offset.

To test the importance of choosing a good ordering, we performed the following experiment. We tried both variable generation order and the above suggested ordering. There is a C source file in one of the benchmarks (`gimp`) which needs more than 23,000 seconds to analyze under generation order, whereas it took less than 2 seconds under the ordering with a large offset. Note that the generation order inter-mixes the regular set variables and generic projection variables and thus may add more generic projection variables. The generation order is the ordering used in [7] for the cycle elimination experiments. In [7], it is observed that the chosen order function does not make much difference for cycle elimination. However, for projection merging, the order function in use is very important for performance.

3.2 Discussion

Without creating many generic projection variables, projection merging turns many source-sink paths into variable-variable paths. We should expect that redundant paths between variable nodes do not produce as many redundant edge additions because of the properties of inductive form.

However, as hinted in the beginning of Section 3, there is a flaw in the argument. Reconsider the example in Figures 2a and 5. For the variables $\mathcal{X}_1$ through $\mathcal{X}_n$, the generic projection variables are $\mathcal{X}_1^{[c,1]}$ through $\mathcal{X}_n^{[c,1]}$ respectively. The generic projection variables have indices larger than the rest of the variables $\mathcal{Y}, \mathcal{Z}, \mathcal{X}_1, \ldots, \mathcal{X}_n$. In this case, the edge between $\mathcal{Y}$ and $\mathcal{Z}$ is added n times. Overall, there is no reduction in redundant work.

There is actually a trade-off in choosing the variable ordering. For generic projection variables, making the indices small reduces the number of redundant edges added (*e.g.*, Figures 2a and 5). However, small indices cause many more such variables to be created (Theorem 3.2).

On the other hand, if we let the generic projection variables have large indices, the number of generic projection variables generated is linearly bounded. However, as just shown, redundancy in constraint graphs is not reduced. It is worth mentioning that if we only use projection merging without cycle elimination, analysis time tends to be much longer than without projection merging. For the case where we use small indices for generic projection variables, this is because the number of generic projection variables generated is large. For the case where we add a large offset to generation order, the analysis time is about a constant factor slower because of the extra work to create the projection variables and to apply the extra rules.

The missing insight is a very subtle, but pronounced, positive interaction between projection merging and cycle elimination. We first explain a few details of cycle elimination. Cycle elimination focuses on dynamically finding and eliminating cycles of constraints between variables of the form $\mathcal{X}_1 \subseteq \mathcal{X}_2 \subseteq \ldots \subseteq \mathcal{X}_n \subseteq \mathcal{X}_1$. Since $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are equal in all solutions, the n variables can be replaced by a single variable, which becomes the *representative* of $\mathcal{X}_1, \ldots, \mathcal{X}_n$. For reasons discussed in [6], it turns out that the best choice of representative is the variable among $\mathcal{X}_1, \ldots, \mathcal{X}_n$ with the smallest index under the ordering $o(\cdot)$.

Since we collapse a cycle to the variable with the smallest index on the cycle, all the variables on the cycle now have a smaller index, which at first sight might appear to cause additional projection variables to be generated for the generic projection variables. This is not a problem because of the following:

- The variables on a cycle become one variable, and so only need at most one generic projection variable for each constructor and index pair (c, i).
- Eliminating a cycle does not increase the number of variables having indices no greater than $Omax$, and in fact may decrease the number of variables with indices no greater than $Omax$.

3.2.1 Reducing Redundant Additions

Now we are in the position to explain why projection merging combined with cycle elimination reduces redundant edge additions. We use a sequence of graph transformations to aid the explanation.

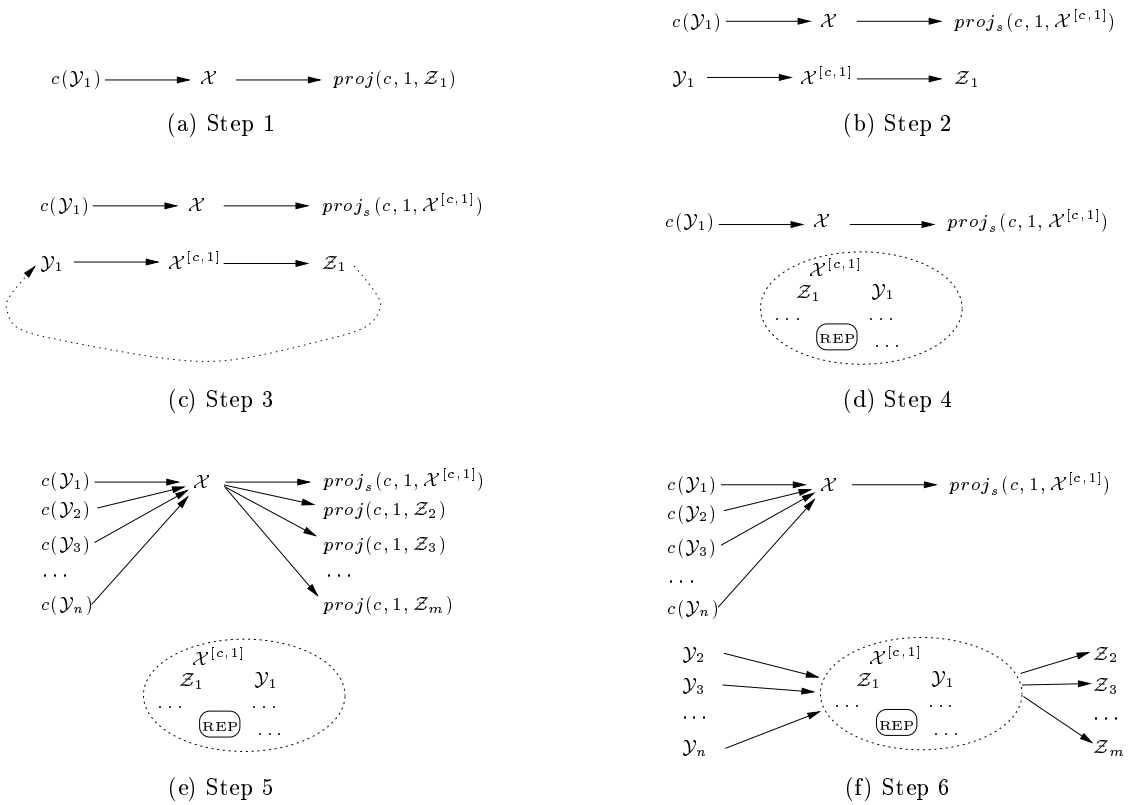


Figure 8: Reducing edge additions

First, we give a high level overview. A generic projection variable $\mathcal{X}^{[c,i]}$ of $\mathcal{X}$ starts with large index, since its index is given as generation order plus a large offset. In the process of resolving the constraints, $\mathcal{X}^{[c,i]}$ can potentially be on a cycle that is detected with online cycle detection and elimination. In that case, the cycle including $\mathcal{X}^{[c,i]}$ is collapsed to a single representative node $\mathcal{Y}$ with small index.

Suppose later we obtain another constraint $\mathcal{X} \subseteq proj(c,i,\mathcal{Z})$. This constraint transforms into the constraint $\mathcal{Y} \subseteq \mathcal{Z}$, because $\mathcal{X}^{[c,i]}$ is aliased to $\mathcal{Y}$. If $\mathcal{Z}$ is generated after $\mathcal{Y}$, we have $o(\mathcal{Y}) < o(\mathcal{Z})$, and any path ending with the edge $(\mathcal{Y},\mathcal{Z})$ is not inductive. Thus eliminating cycles with generic projection variables tends to reduce the redundancy in a constraint graph.

The following lemma states that generic projection variables, if found on a cycle, always have a representative that is not a generic projection variable.

Lemma 3.4 Let $\mathcal{X}_1 \subseteq \ldots \subseteq \mathcal{X}_n \subseteq \mathcal{X}_1$ be a cycle in a constraint graph. It holds that at least one of $\mathcal{X}_i$ for $1 \leq i \leq n$ is a not a generic projection variable.

Proof.

Let $\mathcal{X}^{[c,i]}$ and $\mathcal{Y}^{[d,j]}$ be two generic projection variables. Assume there is an edge $\mathcal{X}^{[c,i]} \subseteq \mathcal{Y}^{[d,j]}$ in the graph. We claim that $o(\mathcal{X}) < o(\mathcal{Y})$ (with $c = d$ and $i = j$) if c is covariant in i, and $o(\mathcal{X}) > o(\mathcal{Y})$ (with $c = d$ and $i = j$) if c is contravariant in i.

We consider the case when c is covariant in i. The case when c is contravariant in i is symmetric.

There are two ways that the edge $\mathcal{X}^{[c,i]} \subseteq \mathcal{Y}^{[d,j]}$ can be added:

1. through a path

   $$\mathcal{X} \longrightarrow \mathcal{Y} \longrightarrow proj_s(d,j,\mathcal{Y}^{[d,j]})$$

   where $c = d$ and $i = j$. By applying the **Proj-Transitive** rule, we get

   $$\mathcal{X} \longrightarrow proj(d,j,\mathcal{Y}^{[d,j]})$$

   Applying the **Proj-Creation** rule, we get the edge

   $$\mathcal{X}^{[c,i]} \longrightarrow \mathcal{Y}^{[d,j]}$$

   For the **Proj-Transitive** rule to be applicable, it must be the case that $o(\mathcal{X}) < o(\mathcal{Y})$.

2. through a path

   $$d(\ldots,\mathcal{X}^{[c,i]},\ldots) \longrightarrow \mathcal{Y} \longrightarrow proj_s(d,j,\mathcal{Y}^{[d,j]})$$

   where $\mathcal{X}^{[c,i]}$ is the jth argument of d. This case is impossible because $\mathcal{X}^{[c,i]}$ cannot appear inside a constructor.

Any cycle with $\mathcal{X}_1^{[c_1,i_1]} \subseteq \ldots \subseteq \mathcal{X}_n^{[c_n,i_n]} \subseteq \mathcal{X}_1^{[c_1,i_1]}$, requires that $c_1 = \ldots = c_n$ and $i_1 = \ldots = i_n$ and $o(\mathcal{X}_1) < \ldots < o(\mathcal{X}_n) < o(\mathcal{X}_1)$ or $o(\mathcal{X}_1) > \ldots > o(\mathcal{X}_n) > o(\mathcal{X}_1)$, which is impossible.

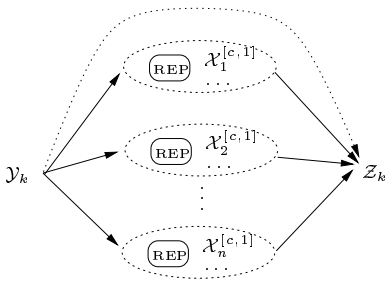


Figure 9: Reducing edge additions for constraint graph A (Figure 2a)

Thus every cycle must contain a variable that is not a generic projection variable.

□

We give now a more detailed explanation by graph transformations. Assume we start with the constraints shown in Figure 8a:

$$c(\mathcal{Y}_1) \subseteq \mathcal{X} \subseteq proj(c, 1, \mathcal{Z}_1)$$

Applying the **Proj-Creation** rule creates the constraints $\mathcal{X} \subseteq proj_s(c, 1, \mathcal{X}^{[c,1]})$ and $\mathcal{X}^{[c,1]} \subseteq \mathcal{Z}_1$. Applying the **Proj-Transitive** rule adds the constraints $c(\mathcal{Y}_1) \subseteq proj(c, 1, \mathcal{X}^{[c,1]})$, which is decomposed by the rule in Figure 1 into the constraint $\mathcal{Y}_1 \subseteq \mathcal{X}^{[c,1]}$. The resulting graph in shown in Figure 8b. After applying more rules to portions of the graph not shown, assume we discover that the variables $\mathcal{Y}_1$, $\mathcal{X}^{[c,1]}$, and $\mathcal{Z}_1$ are on a cycle. The situation is depicted in Figure 8c. This cycle of constraints is then collapsed to a single node with a representative for all the variables on the cycle. In Figure 8d, we show the collapsed cycle as a dotted oval, with (REP) as the representative. Note that the index of the representative is relatively small since it is the variable with the smallest index on the cycle, and the representative cannot be a generic projection variable by Lemma 3.4.

Now suppose the following constraints shown in Figure 8e are generated and added to the system[2]

$$\begin{array}{rcl} c(\mathcal{Y}_2) & \subseteq & \mathcal{X} \\ c(\mathcal{Y}_3) & \subseteq & \mathcal{X} \\ & \vdots & \\ c(\mathcal{Y}_n) & \subseteq & \mathcal{X} \end{array}$$

$$\begin{array}{rcl} \mathcal{X} & \subseteq & proj(c, 1, \mathcal{Z}_2) \\ \mathcal{X} & \subseteq & proj(c, 1, \mathcal{Z}_3) \\ & \vdots & \\ \mathcal{X} & \subseteq & proj(c, 1, \mathcal{Z}_m) \end{array}$$

Assume the variables $\mathcal{Y}_2, \ldots, \mathcal{Y}_n, \mathcal{Z}_2, \ldots, \mathcal{Z}_m$ have larger indices than the variables $\mathcal{Y}_1$ and $\mathcal{Z}_1$. Thus they have larger index than the representative (REP). After applying the resolution rules, we get the constraint graph shown in Figure 8f. Since (REP) has smaller index than the variables $\mathcal{Y}_2, \ldots, \mathcal{Y}_n, \mathcal{Z}_2, \ldots, \mathcal{Z}_m$, none of the paths

$$\mathcal{Y}_i \longrightarrow \text{(REP)} \longrightarrow \mathcal{Z}_j$$

for $2 \le i \le n$ and $2 \le j \le m$ is inductive. Thus none of the edges $(\mathcal{Y}_i, \mathcal{Z}_j)$ is added. In contrast, without projection merging, all the nm edges $(\mathcal{Y}_i, \mathcal{Z}_j)$ are added to the graph. The above scenario is the best case for projection merging. In practice, we can assume that a fraction f of the nodes $\mathcal{Y}_2, \ldots, \mathcal{Y}_n$ and $\mathcal{Z}_2, \ldots, \mathcal{Z}_m$ have lower index than the representative (REP). In that case, $f^2(nm)$ transitive edges are added to the graph instead of nm edges.

We now come back to the redundant paths problem shown in Figure 2 and show how it is handled by projection merging. Suppose the constraints

$$\begin{array}{rcl} c(\mathcal{Y}_k) & \subseteq & \mathcal{X}_1 \\ c(\mathcal{Y}_k) & \subseteq & \mathcal{X}_2 \\ & \vdots & \\ c(\mathcal{Y}_k) & \subseteq & \mathcal{X}_n \end{array}$$

$$\begin{array}{rcl} \mathcal{X}_1 & \subseteq & proj(c, 1, \mathcal{Z}_k) \\ \mathcal{X}_2 & \subseteq & proj(c, 1, \mathcal{Z}_k) \\ & \vdots & \\ \mathcal{X}_n & \subseteq & proj(c, 1, \mathcal{Z}_k) \end{array}$$

are added to the graph in Figure 2a. The generic projection variables for $\mathcal{X}_1$ through $\mathcal{X}_n$ are created; let these be $\mathcal{X}_1^{[c,1]}, \ldots, \mathcal{X}_n^{[c,1]}$. Assume that all of the generic projection variables are found to be on cycles and are identified with other variables as the representatives. Assume also that $\mathcal{Y}_k$ and $\mathcal{Z}_k$ have smaller indices than a fraction f of the representatives of the variables $\mathcal{X}_1^{[c,1]}, \ldots, \mathcal{X}_n^{[c,1]}$. After applying closure rules, we get the graph in Figure 9. The edge $(\mathcal{Y}_k, \mathcal{Z}_k)$ is added fn times because only fn of the paths

$$\mathcal{Y}_k \longrightarrow \text{(REP)} \longrightarrow \mathcal{Z}_k$$

are inductive.

The examples in this section illustrate how projection merging interacts positively with cycle elimination. There is another informal argument which is insightful. The random graph model in [7] shows that if indices are assigned randomly to variables, then inductive form adds asymptotically fewer variable-variable edges than source-sink edges. The problem with assigning all generic projection variables large indices is that the resulting graph is anything but random, and the examples show that there is no benefit. Cycle elimination, however, has the effect of arbitrarily perturbing the indices of generic projection variables. This apparently makes the graph sufficiently random that the behavior is more in line with the predictions in [7].

In Section 4, we will see that projection merging combined with cycle elimination significantly improves the analysis time. The effect on redundant edge additions is very dramatic. In one example, `gcc`, the average number of times an edge is redundantly added drops from 85 to 0.32.

4 Experiments

This section experimentally validates the idea of projection merging. We show that combined with cycle elimination,

[2] The addition of such constraints is common when the variable $\mathcal{X}$ is found to be the representative of a cycle.

Benchmark	AST Nodes	LOC	#Prog Vars
allroots	700	426	91
diff.diffh	935	293	122
anagram	1078	344	130
genetic	1412	323	154
ks	2284	574	210
ul	2395	441	141
ft	3027	1180	279
compress	3333	651	174
ratfor	5269	1532	388
compiler	5326	1888	320
assembler	6516	2980	796
ML-typecheck	6752	2410	557
eqntott	8117	2266	592
simulator	10946	4216	1125
less-177	15179	11988	1420
li	16828	5761	2313
flex-2.4.7	29960	9345	2871
pmake	31148	18138	2493
make-3.72.1	36892	15213	3061
inform-5.5	38874	12957	3186
tar-1.11.2	41035	18293	2471
sgmls-1.1	44533	30941	2890
screen-3.5.2	49292	23919	3235
cvs-1.3	51223	31130	4691
espresso	56938	21537	3981
gawk-3.0.3	71140	28326	3692
povray-2.2	87391	59689	4924
mume	312458	430947	28849
spice	452149	849258	24310
gs	504724	437211	35687
pgsql	718781	1344689	45299
gcc	1168907	411034	59991
gimp	2112848	7486733	178829

Table 1: Benchmark programs

projection merging significantly improves the execution time of Andersen's points-to analysis [5]. Note that the analysis time includes not only the time to close the constraint graph but also the time to compute the points-to sets for all the program variables. In our implementation of the analysis, we model structures as single atomic memory locations. Every field of a structure shares the same location.

4.1 Experimental Data

Our experiments were done with the BANE analysis toolkit [1] using a single processor on a SPARC Enterprise-5000 with 2048M of memory. We use the generation order of set variables, except for the generic projection variables, where we set the indices to be generation order plus a large offset (`0x0fffffff`). This ordering guarantees that the indices of generic projection variables are greater than the indices of other variables for the set of benchmarks we analyze.

We use the C benchmarks shown in Table 1. The benchmarks are those in [7] with six additional large programs: `mume`, `spice`, `gs`, `pgsql`, `gcc`, and `gimp`.

- `mume` is a multiuser dungeon program.
- `spice` (version 3f4) is a circuit simulation program.
- `gs` is ghostscript version 5.01 without the X library.
- `pgsql` is PostgreSQL, an Object-Relational DBMS derived from the Berkeley Postgres database management system.
- `gcc` is the GNU C compiler version 2.8.1.
- `gimp` is the GNU image manipulation program with the X library included.

For each benchmark, the table lists the number of abstract syntax tree (AST) nodes, the number of lines in the preprocessed source, and the number of program variables in the source. Notice that the six new benchmarks are much larger than those used in [7]. As an aside, `gimp` was the largest program we could obtain for these experiments ($440,000$ non-comment source lines before preprocessing). The Linux kernel without assembly files is a little larger ($550,000$ non-comment, non-assembler source lines before preprocessing) but uses a number of GNU extensions that our C parser does not support.

Two experiments were performed. In the first experiment, we analyzed all the benchmarks with cycle elimination only. In the second experiment, we combined cycle elimination and projection merging. For the experiment with cycle elimination only, all programs ran through the analysis except `gimp`. The analysis for `gimp` did not finish in 33 hours, after which the job was killed. For the second experiment, all programs ran through in less than half an hour.

Table 2 shows the results for the two experiments. For each experiment and each benchmark, we report the number of set variables, the number of edges in the final graph, the total number of source-sink edge additions including redundant ones, the total number of non-source-sink edge additions, the total number of edge additions, and the analysis time in seconds. We ran each experiment three times, and the one with the best execution time is presented in the table. The data for `gs` is abnormal. There are a large number of edges in the final graph under the experiment with cycle elimination only. We believe this is partly due to the lack of the X library mentioned above. Since the program is incomplete, there are fewer opportunities for collapsing cyclic constraints, which may make the final graph very large.

We show some plots to better demonstrate the effectiveness of projection merging with cycle elimination. In Figure 10, we plot the analysis time for both cycle elimination with projection merging and cycle elimination alone against the number of AST nodes of the parsed benchmarks. Note that all the plots are on a log-log scale. For small programs, the analysis times differ by very little. As the size of the program increases, redundant source-sink paths begin to dominate as predicted in the complexity analysis in [7] and the effect of projection merging becomes pronounced. The analysis time with projection merging is significantly smaller than without, especially for the last six large benchmarks. In Figure 11, we plot the speedup of the analysis time with projection merging over cycle elimination only, and the trend becomes more obvious. The speedups seem to be asymptotic: the speedup increases with the size of the programs. Figure 12 plots the total number of edge additions (Work) for both experiments. We see the same trend as in the case of analysis time. Figure 13 plots the ratio of the total edge additions, which has basically the same shape as Figure 11. Figure 14 plots the total number of redundant edge additions. We notice that the absolute number of redundant edge additions for cycle elimination only is significantly higher. In Figure 15, we plot, on average, how

Benchmark	Cycle Elimination Only						Projection Merging + Cycle Elimination					
			Work						Work			
	#Vars	Edges	s-s	Other	Total	Time(s)	#Vars	Edges	s-s	Other	Total	Time(s)
allroots	126	257	86	205	291	0.14	210	335	71	317	388	0.15
diff.diffh	184	363	127	282	409	0.16	269	412	109	350	459	0.17
anagram	208	346	81	290	371	0.18	318	411	106	324	430	0.21
genetic	228	391	115	300	415	0.21	436	610	129	507	636	0.24
ks	324	1222	641	1276	1917	0.40	511	1028	280	977	1257	0.45
ul	199	264	56	259	315	0.26	255	237	48	270	318	0.27
ft	393	1037	297	1029	1326	0.42	581	955	266	840	1106	0.53
compress	249	395	51	409	460	0.33	383	451	71	514	585	0.39
ratfor	599	1982	727	1853	2580	0.89	840	2493	577	2297	2874	1.05
compiler	439	1159	431	997	1428	0.69	656	1219	196	1281	1477	0.82
assembler	969	2235	874	1906	2780	1.45	1502	3336	688	3511	4199	1.61
ML-typecheck	793	4505	2814	5062	7876	1.77	1214	3833	980	4139	5119	1.87
eqntott	987	2950	922	3048	3970	1.11	1544	2997	799	2804	3603	1.47
simulator	1441	4141	4849	3943	8792	2.36	2066	3990	823	3647	4470	1.95
less-177	1852	7656	7289	8125	15414	3.36	2417	6478	1375	6504	7879	2.90
li	3202	14125	81948	21288	103236	15.49	4033	13234	3719	15467	19186	5.42
flex-2.4.7	3755	8224	2252	7787	10039	6.41	5207	8886	1732	8461	10193	6.96
pmake	3300	12537	41301	13588	54889	10.33	5172	11857	2232	12966	15198	6.25
make-3.72.1	4731	31820	159389	97474	256863	31.54	6861	55452	15389	168933	184322	31.38
inform-5.5	4364	18394	47315	20172	67487	13.48	8056	19587	3121	19069	22190	8.88
tar-1.11.2	4156	18038	32257	18888	51145	10.84	5476	18262	2092	19422	21514	7.84
sgmls-1.1	4255	34178	300563	41651	342214	47.10	6102	28082	5571	38275	43846	13.46
screen-3.5.2	6449	28098	154413	32185	186598	28.11	7900	17543	3686	17750	21436	9.72
cvs-1.3	6967	22120	27076	23557	50633	11.99	10444	22681	3433	25171	28604	11.80
espresso	6295	26697	120880	28730	149610	25.98	10078	24927	5150	26737	31887	12.76
gawk-3.0.3	6337	26052	108261	35195	143456	26.53	8519	22307	5217	24533	29750	13.02
povray-2.2	7673	65861	246225	70129	316354	51.12	11421	43898	10069	56052	66121	21.72
mume	52859	676765	13748716	1106200	14854916	1964.24	66800	228641	40938	276446	317384	137.27
spice	34748	220725	719903	146675	866578	211.19	50969	103198	21649	116610	138259	82.49
gs	96633	44406952	76168794	7788690	83957484	64675.54	121335	311999	123831	273115	396946	257.51
pgsql	89814	1279129	15649146	875655	16524801	3941.39	129726	353714	70691	631884	702575	569.52
gcc	135655	940361	80178428	1301387	81479815	11445.80	188853	427584	80942	485653	566595	503.41
gimp	∞	∞	∞	∞	∞	∞	378872	917401	209117	1011301	1220418	1494.12

Table 2: Benchmark data

many times an edge is added redundantly through different paths. Notice that for projection merging with cycle elimination, the number is consistently around 0.3 for almost all programs. For cycle elimination alone, the number is much larger for large programs, with the exception of **gs**, where cycle elimination alone adds each edge 0.9 times redundantly on average, and cycle elimination plus projection merging adds each edge an average of 0.27 times. These data show that projection merging with cycle elimination effectively solves the problem of redundant paths.

5 Related Work

Many researchers have studied the problem of points-to analysis. Andersen devised a natural inclusion-based points-to algorithm based on set constraints in his thesis [5]. Work by Shapiro and Horwitz [21] contrast Andersen's set based points-to analysis with the unification based points-to analysis of Steensgaard [23]. They conclude that while Andersen's analysis is substantially more precise than Steensgaard's, its running time is impractical. Work in [7], however, demonstrates that Andersen's points-to analysis can be made to scale much better with a special online cycle elimination technique and the inductive form representation. This paper presents techniques that allow Andersen's analysis to be applied to programs an order of magnitude larger than in [7].

Inclusion constraint resolution algorithms usually have at least $\mathcal{O}(n^3)$ time complexity. Work by Melski and Reps [17] gives some insight as to why this cubic bottleneck is difficult to break. They show the equivalence of some set-based analyses to context-free grammar reachability problems. Similar work by Heintze and McAllester [14] also gives some theoretical evidence of the difficulty for some subtyping and flow analyses. They show that certain data-flow and control-flow problems are 2NPDA-complete, a class of problems that has resisted a sub-cubic algorithm for over 30 years.

Heintze and McAllester also present a quadratic time algorithm for some restricted classes of the closure analysis problem for higher order functional programming languages [13]. The technique presented in [13] resembles projection merging in some ways, but there are important differences. [13] relies on a given, *non-recursive* type structure of the program that the algorithm exploits. While projection merging "discovers" whatever structure it needs and works with recursive structures, its worst-case complexity is still cubic.

6 Conclusions

Redundant paths in inclusion constraint graphs limit the practical application of constraint based program analyses to very large programs. In this paper, we present projection merging for speeding up constraint-based program analyses, and demonstrate that the technique together with cycle elimination can yield orders of magnitude improvements in analysis time. With projection merging, a cubic time points-to analysis for C can be applied to half million line programs in less than half an hour. We expect the technique to work well for other set-based analyses that use only a small number of non-constant constructors.

Acknowledgments

We thank Jeff Foster, David Gay, Ben Liblit, and the anonymous referees for their comments on an earlier draft of this paper.

References

[1] Aiken, A., Fähndrich, M., Foster, J., and Su, Z. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)* (March 1998), pp. 78–96.

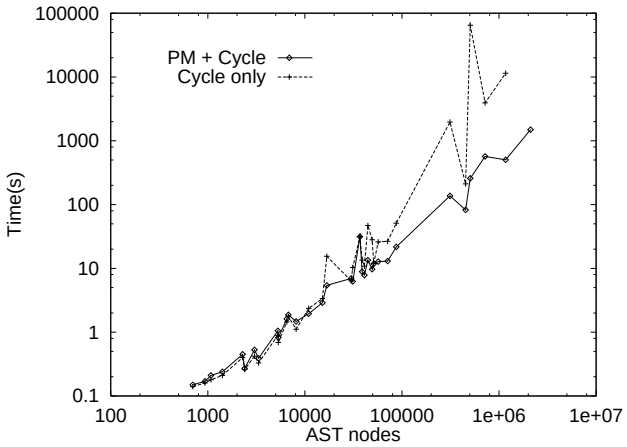


Figure 10: Total analysis times

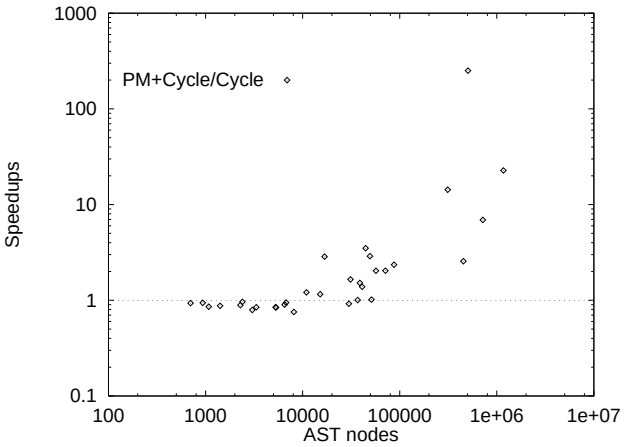


Figure 11: Speedups through projection merging

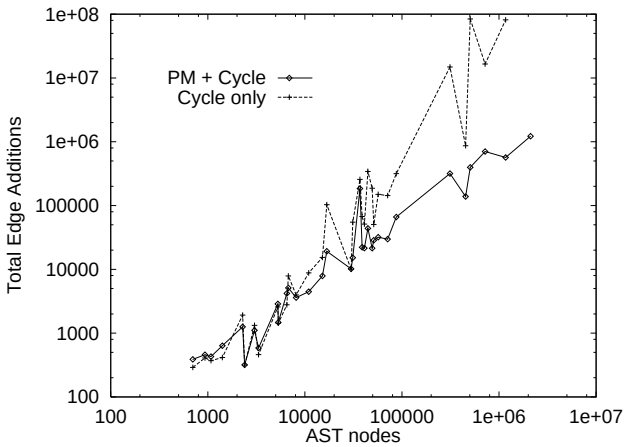


Figure 12: Total number of edge additions

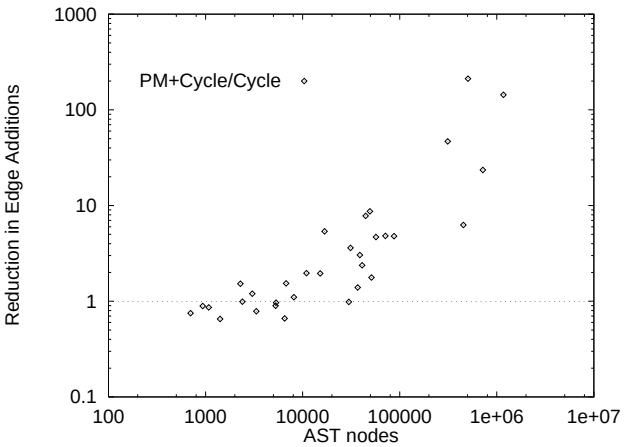


Figure 13: Reduction in edge additions

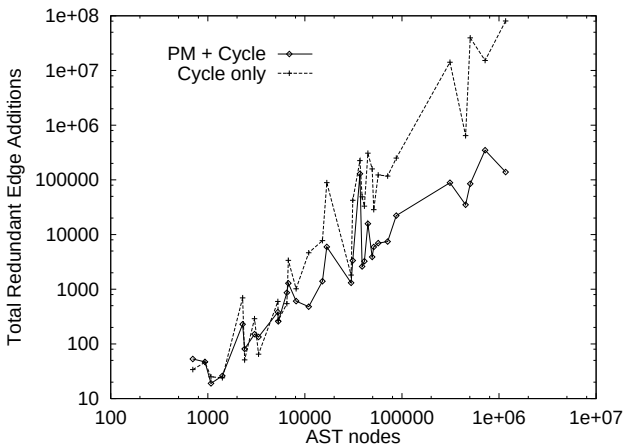


Figure 14: Total number of redundant edge additions

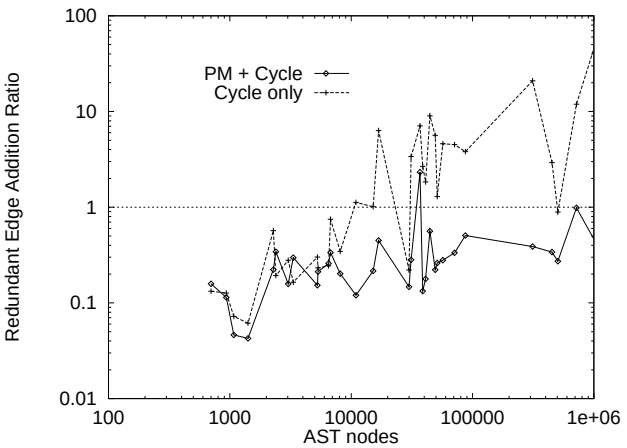


Figure 15: Ratio of redundant edge additions to total edges

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

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

[4] Aiken, A., Wimmers, E., and Lakshman, T. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages* (Jan. 1994), pp. 163–73.

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

[6] Fähndrich, M. *BANE: A Library for Scalable Constraint-Based Program Analysis*. PhD thesis, Computer Science Division, University of California, Berkeley, March 1999.

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

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

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

[10] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. Addison Wesley, 1996, ch. 10, pp. 199–200.

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

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

[13] Heintze, N., and McAllester, D. Linear-Time Subtransitive Control Flow Analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation* (1997), pp. 261–272.

[14] Heintze, N., and McAllester, D. On the Cubic Bottleneck in Subtyping and Flow Analysis. In *Proceedings of the 1997 IEEE 12th Anuual Symposium on Logic in Computer Science* (1997), pp. 342–351.

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

[16] Marlow, S., and Wadler, P. A Practical Subtyping System For Erlang. In *Proceedings of the International Conference on Functional Programming (ICFP '97)* (June 1997), pp. 136–149.

[17] Melski, D., and Reps, T. Interconvertibility of set constraints and context-free language reachability. In *Proceedings of the 1997 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97* (June 1997), pp. 74–89.

[18] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, 1990.

[19] Palsberg, J., and Schwartzbach, M. I. Object-Oriented Type Inference. In *Proceedings of the ACM Conference on Object-Oriented programming: Systems, Languages, and Applications* (Oct. 1991), pp. 146–161.

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

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

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

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

A Andersen's Points-to Analysis

For completeness we include a summary of Andersen's points-to analysis and its formulation using set constraints. The formulation here is basically the same as that in [7], except for the use of $proj(\ldots)$. In the current paper, we need to expose more low-level details of the system to explain projection merging. Thus, in the presentation of the analysis, we include the use of $proj(\ldots)$.

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* [5]. 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.

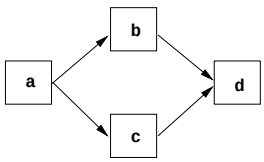


Figure 16: Example points-to graph

Figure 16 shows the points-to graph computed by Andersen's analysis for the following simple C program:

```
a = &b;
a = &c;
*a = &d;
```

Note that these points-to graphs are different from the constraint graphs discussed in this paper. Points-to graphs can be constructed from the least solution of the constraints.

A.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 16 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. In [7], a different formulation makes this association explicit and enables the use of a generic set constraint solver. Locations are modeled 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 [18].

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 [10]). A sound approach is to turn inclusions between references into equality for their contents: $ref(\mathcal{X}) \subseteq ref(\mathcal{Y}) \Leftrightarrow \mathcal{X} = \mathcal{Y}$.

This technique can be adapted to a purely inclusion-based system. 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 $\tau \subseteq proj(ref, 3, \mathcal{T})$. For example, if $ref(l_\mathtt{x}, \mathcal{X}_{l_\mathtt{x}}, \overline{\mathcal{X}_{l_\mathtt{x}}}) \subseteq \tau$, then the transitive constraint $ref(l_\mathtt{x}, \mathcal{X}_{l_\mathtt{x}}, \overline{\mathcal{X}_{l_\mathtt{x}}}) \subseteq proj(ref, 3, \mathcal{T})$ 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.

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

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

$$\frac{e : \tau \quad \tau \subseteq proj(ref, 2, \mathcal{T}) \quad \mathcal{T} \text{ fresh}}{*e : \mathcal{T}} \quad \text{(Deref)}$$

$$\frac{\begin{array}{c} e_1 : \tau_1 \qquad\qquad e_2 : \tau_2 \\ \tau_1 \subseteq proj(ref, 3, \mathcal{T}_1) \quad \tau_2 \subseteq proj(ref, 2, \mathcal{T}_2) \\ \mathcal{T}_2 \subseteq \mathcal{T}_1 \quad \mathcal{T}_1, \mathcal{T}_2 \text{ fresh} \end{array}}{e_1 = e_2 : \tau_2} \quad \text{(Asst)}$$

Figure 17: Constraint generation for Andersen's analysis

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 $\mathcal{X}_{l_\mathtt{x}} \subseteq proj(ref, 1, \mathcal{Y}_{l_\mathtt{x}})$ 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.

A.2 Constraint Generation

Figure 17 gives a subset of the constraint-generation rules for Andersen's analysis. For the full set of rules, see [9]. 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 x therefore denotes the location of x and not its contents.

We briefly describe the other rules in Figure 17. 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 $\mathcal{T}$ a superset of the points-to set of τ. This is achieved through the projection operator proj. 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 $\tau_1 \subseteq proj(ref, 3, \mathcal{T}_1)$ 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 16, a = &b, generates the constraints $\tau_1 = ref(l_\mathtt{a}, \mathcal{X}_{l_\mathtt{a}}, \overline{\mathcal{X}_{l_\mathtt{a}}}) \subseteq proj(ref, 3, \mathcal{T}_1)$, and so $\mathcal{T}_1 \subseteq \mathcal{X}_{l_\mathtt{a}}$, and $\tau_2 = ref(0, ref(l_\mathtt{b}, \mathcal{X}_{l_\mathtt{b}}, \overline{\mathcal{X}_{l_\mathtt{b}}}), \ldots) \subseteq proj(ref, 2, \mathcal{T}_2)$, 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 $ref(l_\mathtt{b}, \mathcal{X}_{l_\mathtt{b}}, \overline{\mathcal{X}_{l_\mathtt{b}}}) \subseteq \mathcal{X}_{l_\mathtt{a}}$.