Precise Points-To Analysis with Witnesses

Extended Version

Bor-Yuh Evan Chang
University of Colorado, Boulder
bec@cs.colorado.edu

Sriram Sankaranarayanan
University of Colorado, Boulder
srirams@cs.colorado.edu

Manu Sridharan
IBM T.J. Watson Research Center
msridhar@us.ibm.com

Abstract

We present a framework for augmenting points-to analyses with *witness generation* and *refutation*. A witness for a points-to relationship $p \mapsto q$ is a sequence of program assignments that causes p to point to q when executed. Witnesses for points-to results could significantly improve the usability of points-to analysis, both by helping explain the results to users and by enabling a clean architecture for refining precision on demand.

Our framework is designed around performing targeted *witness search* while incorporating the program's control and data flow via *witness refuters* to improve precision. The witness search algorithm leverages a pre-computed points-to analysis to narrow its search space while refining the memory abstraction used in the pre-computed results. Witness refuters can prune branches of the witness search based on more precise modeling of program semantics, such as flow, context, or even path sensitivity. Furthermore, refuters can remedy imprecisions caused by dynamic memory allocation.

A symbolic encoding of the witness search and refutation enables the use of powerful SAT solvers to perform this search. Using a prototype implementation of our ideas, we present experimental results that suggest that symbolic witness search can be applied to find witnesses on-demand.

1. Introduction

> *... explaining errors is often more difficult than finding them. A misunderstood explanation means the error is ignored or, worse, transmuted into a false positive. The heuristic we follow: Whenever a checker calls a complicated analysis subroutine, we have to explain what that routine did to the user, and the user will then have to (correctly) manually replicate that tricky thing in his/her head.*
>
> — Engler et al. [3]

Programming languages such as C and Java make extensive use of pointers. As a result, many program analysis questions over these languages require pointer analysis as a primitive to find the set of all memory locations that a given pointer may address. This problem is of fundamental importance and many different techniques have been proposed at different levels of precision [10].

Still, it is widely held that pointer analysis can be difficult to employ in practice, leading many real-world tools to eschew pointer analysis at the cost of soundness. The primary challenge lies in finding the right balance between scalability and precision. Scalable pointer analyses such as Steensgaard's analysis [23] and Andersen's analysis [1]) can lack sufficient precision to prove properties of interest, whereas more precise heap analyses such as shape analysis [20]) often do not scale to large programs. A large volume of work has sought to address this problem over the past two decades. Yet, an equally important challenge for bug finding tools is the difficulty of explaining analysis results that rely on pointer aliasing to the user; many bug-finding systems simply give up on reporting any aliasing-related bugs since developers have great difficulty understanding the reports [3].

In this paper, we present a framework for tackling this issue by providing witnesses to points-to edges. Intuitively, a witness for points-to relationship $p \mapsto q$ is a sequence of statements W from the input program such that executing W causes p to point to q. A witness can be seen as a type of certificate indicating the reasoning used by an analysis to derive a particular points-to relation. Witnesses could be useful in a number of contexts, and in particular they could be instrumental in overcoming the pointer analysis challenges described earlier.

Witnesses provide an elegant basis for interfacing between points-to analyses of varying precision, thereby enabling a general framework for *refinement-based points-to analysis*. Previous work has shown that client-driven, refinement-based points-to analysis is a promising technique for finding a "sweet spot" in the trade-off of scalability versus precision [9, 21]. However, previous systems have typically integrated core analyses and refinements into a monolithic system, making it harder to add new refinement techniques.

In contrast, witnesses provide a clean way to integrate analyses with arbitrarily varying precision requirements. Given a witness W for some points-to analysis result, refinement with various sensitivities can be directly formulated in terms of the program semantics:

- **Control-Flow Compatibility:** Does W lie on some control-flow path in the program?
- **Flow Sensitivity:** Does W lie on some control-flow path p such that its reaching definitions are preserved (i.e., no other statement on p "kills" a memory location between a def and use from W)?
- **Context Sensitivity:** Does W lie on a valid interprocedural control-flow path?
- **Path Sensitivity:** Does W lie on some *feasible* control-flow path?

The simplicity with which these refinements can be described reflects how various refinement techniques (both previously known and novel) could be cleanly integrated via witness generation.

Witnesses could also help greatly in explaining analysis results involving pointer aliasing to developers, as they directly indicate *why* the analysis determines that such aliasing is possible. For example, in a typestate analysis [24], if the typestate of an object is changed via multiple aliases, a tool could optionally display the points-to analysis witnesses explaining the aliasing to the user. Mishandled aliasing remains the cause of many difficult-to-diagnose programming errors, and effective witness-generating points-to analyses could have a significant impact on helping programmers avoid and fix such bugs.

Enhancing points-to analyses with witness generation presents some challenges. First, given a standard flow-insensitive analysis like Andersen's [1], it is not immediately obvious how to generate witnesses for its results, even ignoring the control flow of the original program. The key difficulty lies in "bridging the gap" between the flow-*insensitive* view of the program taken by the points-to analysis and the flow-*sensitive* nature of the witness—it is a *sequence* of statements. We also desire a technique that enables filtering of witnesses via refinement. Performing this refinement on full witnesses may lead to much redundant work (e.g., if many witnesses share the same infeasible sub-sequence).

We present a framework for witness-generating points-to analysis that addresses these issues. In our framework, a core *witness search* algorithm searches for possible witnesses for a points-to result based on some initial conservative points-to analysis. This search is carefully designed to both exploit the results of the initial points-to analysis (to avoid "obviously invalid" witnesses) and to enable arbitrary refinement of the heap model, independent of the heap abstraction used by the initial analysis. Refinement is performed via *witness refuters* that are given partial witnesses during the search. Refuters are able to prune branches of the witness search based on their more precise modeling of program semantics, and the set of refuters employed can be varied to achieve fine-grained control of overall precision in a targeted manner.

As an initial instantiation of our framework, we focus on the problem of computing a precise witness-generating flow-*insensitive* points-to analysis. As shown in previous work [5, 11], common flow-insensitive points-to analyses like Andersen's analysis [1] may not compute precise results, even ignoring control flow and dynamic memory (Cf. Section 2). Hence, the witness generation process must be able to filter imprecise results even for flow-insensitive witnesses.

In summary, we make the following contributions:

- We describe a framework for witness-generating points-to analysis (Sections 5 and 6) for a natural, operational definition of a points-to witness (Section 4). A novel aspect of our framework is the interaction between witness generators and witness refuters that permit on-demand precision refinement (Section 7).
- As a result of our instantiating framework with specific refuters, we get a precise witness-generating algorithm for programs without dynamic memory allocation (Section 5.3) and a precise semi-algorithm for programs with dynamic memory (Section 6.3). In both cases, a simple "conflict detection" witness refuter is able to guarantee (flow-insensitive) precision.
- We present an encoding of the witness search as a Boolean formula such that any solution to the formula corresponds to a witness (Section 5.4). As a result, the search for witnesses can be carried out using efficient SAT solvers.
- We provide initial evidence of the feasibility of symbolic witness search based on a prototype implementation (Section 8).

Supplementary material in the form of an extended version with proofs, our prototype implementation, the benchmarks used in our evaluations and the witnesses found will be made available to the interested reader on-line at the URL: `http://pl.cs.colorado.edu/witnesses`.

2. Overview

In this section, we give an overview of of our framework for witness-generating points-to analyses. We first give some brief points-to analysis background and show an example of a points-to analysis witness. We then present our algorithm for witness search at a high level, and finally we discuss how witness refuters are used to refine precision.

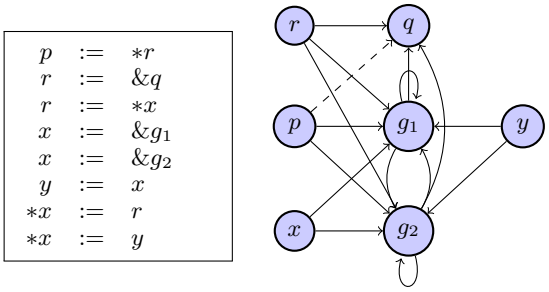


Figure 1. The result of Andersen's analysis on the problem shown left. We shall show a witness for the dashed edge $p \mapsto q$.

Briefly, a points-to analysis computes an over approximation of the memory locations that pointers may point to during program execution. Andersen's analysis [1] is a well-known *flow-insensitive* points-to analysis, that is, an analysis that assumes program statements can execute in any order and any number of times. To handle dynamic memory allocation, Andersen's analysis employs a heap abstraction in which the memory allocated by all executions of a `malloc` statement are modeled with a single abstract location.

Figure 1 shows an example program and the points-to graph computed by Andersen's analysis for the program (with edges from variables to locations the variable may point to). Since the analysis is flow insensitive, the program is simply shown as a set of statements, with control flow elided. One problem with the analysis output is that it may not be immediately evident how some of the edges in the points-to graph arose (e.g., why the analysis concludes that p can point to q).

Our algorithms augment a standard points-to analysis by generating a *witness* for any points-to relationship in the result. For a given points-to edge $p \mapsto q$, a witness is a sequence of statements W from the original program such that executing W causes p to point to q. Witnesses may vary in how much they abstract the semantics of the original program. Here, we focus on flow-insensitive witnesses, which like Andersen's analysis assume that statements may execute in any order and any number of times. We discuss flow-sensitive witnesses in Section 7.

The following statement sequence is a witness for $p \mapsto q$ in the program from Figure 1 (a comment shows the effect of each statement on the heap):

$$
\begin{array}{llll}
1: r := \&q; & // \; r \mapsto q & 5: x := \&g_2; & // \; x \mapsto g_2 \\
2: x := \&g_1; & // \; x \mapsto g_1 & 6: *x := y; & // \; g_2 \mapsto g_1 \\
3: y := x; & // \; y \mapsto g_1 & 7: r := *x; & // \; r \mapsto g_1 \\
4: *x := r; & // \; g_1 \mapsto q & 8: p := *r; & // \; p \mapsto q
\end{array}
$$

Presenting the witness above along with the fact $p \mapsto q$ significantly increases understandability compared to presenting the fact alone. Furthermore, generating the above witness immediately enables many types of refinement, for example, ensuring that the statements in the witness lie on a valid (possibly interprocedural) control-flow path or that they respect the semantics of dynamic memory allocation (see further discussion in Section 7).

Our witness search algorithm discovers witnesses in a goal-directed manner, prepending statements based on precomputed *edge dependence rules*. For a points-to edge e, an edge dependence rule captures a program statement and other points-to edges that enable e to arise. Here is an example edge dependence for the program and points-to graph from Figure 1:

$$p \mapsto q \xleftarrow{p:=*r} \{r \mapsto g_1, g_1 \mapsto q\}$$

$$
\begin{array}{lll}
\{\underline{p \mapsto q}\} & \xleftarrow{p:=*r} & \{\underline{r \mapsto g_1}, g_1 \mapsto q\} \\
& \xleftarrow{r:=*x} & \{x \mapsto g_2, \underline{g_2 \mapsto g_1}, g_1 \mapsto q\} \\
& \xleftarrow{*x:=y} & \{\underline{x \mapsto g_2}, y \mapsto g_1, g_1 \mapsto q\} \\
& \xleftarrow{x:=\&g_2} & \{y \mapsto g_1, \underline{g_1 \mapsto q}\} \\
& \xleftarrow{*x:=r} & \{\underline{y \mapsto g_1}, x \mapsto g_1, r \mapsto q\} \\
& \xleftarrow{y:=x} & \{\underline{x \mapsto g_1}, r \mapsto q\} \\
& \xleftarrow{x:=\&g_1} & \{\underline{r \mapsto q}\} \\
& \xleftarrow{r:=\&q} & \emptyset
\end{array}
$$

Figure 2. Algorithm state for a successful witness search for $p \mapsto q$ for program from Figure 1. The witness can be read off the edge labels from bottom to top.

This rule states that $p \mapsto q$ can arise through an execution of the statement $p := *r$ in an environment where $r \mapsto g_1$ and $g_1 \mapsto q$ hold *simultaneously*. The rule set is constructed using some precomputed conservative points-to analysis like Andersen's, and hence it captures all possible ways that a points-to edge can arise. Guiding the witness search with edge dependence rules yields a far more targeted search than doing naïve enumeration of statement sequences.

Figure 2 shows the state maintained by our algorithm in discovering the aforementioned witness for $p \mapsto q$ in Figure 1. At each point in the search, the algorithm maintains a set of points-to relations that must hold simultaneously in order to complete the witness. In a successful witness search, the initial set is a singleton containing the desired points-to relation (in this case $p \mapsto q$) and the final set is $\emptyset$. At each step of the search, a points-to edge e is chosen from the current set, and the edge dependence rules are used to choose a statement that could cause e and update the simultaneous points-to set appropriately. In Figure 2, the edge e chosen at each step (i.e., the edge satisfied by the *next* chosen statement) is underlined.

As the backward search is performed, a *refuter* may use the current partial witness and knowledge of program semantics to terminate that branch of the search. For flow-insensitive witness generation, the core refuter performs a kind of conflict detection, terminating the search if the current simultaneous points-to set requires a single location to point to two distinct locations at once. Here is an example of a conflict arising in the witness search for $p \mapsto q$:

$$
\begin{array}{lll}
\{\underline{p \mapsto q}\} & \xleftarrow{p:=*r} & \{\underline{r \mapsto g_1}, g_1 \mapsto q\} \\
& \xleftarrow{r:=*x} & \{x \mapsto g_2, g_2 \mapsto g_1, \underline{g_1 \mapsto q}\} \\
& \xleftarrow{*x:=r} & \{\mathbf{x \mapsto g_2}, g_2 \mapsto g_1, \mathbf{x \mapsto g_1}, r \mapsto q\}
\end{array}
$$

The conflict on x is shown in bold. The refuter indicates that this partial witness is invalid, forcing the witness search to backtrack and try a different sequence. As shown in previous work [5, 11], even without dynamic memory allocation, Andersen's analysis may compute some points-to results for which there is no flow-insensitive witness. Our witness search with conflict detection can detect such cases (since it will find no witness), yielding a precise flow-insensitive points-to analysis.

Sections 3, 4, and 5 detail the basic algorithm for witness generation overviewed in this section. This algorithm is extended to handle dynamic memory allocation in Section 6. We describe extensions to flow-sensitive witness generation in Section 7.

3. Points-To Analysis and Memory Abstractions

To set a baseline for describing our witness-generating analysis framework, we discuss in this section various kinds of points-to analysis problems and corresponding memory abstractions, which form the core of any points-to analysis algorithm. Most of the definitions in this section are reformulations of known concepts.

3.1 Finite Memory

We begin by defining the flow-insensitive points-to analysis problem on finite memory, that is, on a fixed, static set of locations.

Def. 3.1 (Finite-Memory Points-To Analysis Problem)**.** *A (flow-insensitive) finite-memory points-to analysis problem* Π *consists of a finite set of variables* X *along with a set of assignments* A*.*

Each assignment in A *conforms to one of the following forms:*

$$*^d p := \&q \qquad *^{d_1} p := *^{d_2} q, \qquad \text{where } p, q \in X\,.$$

The expression $*^d p$ *denotes the application of* $d \geq 0$ *dereferences to pointer* p*, while* $\&q$ *takes the address of* q*. Note that* $*^0 p \equiv p$*.*

Intuitively, the goal of a points-to analysis is to answer queries of the form $p \mapsto q$: is there a sequence of assignments from A that causes p to point to q? We will define the points-to analysis result more precisely in Section 4.

The standard formulation above restricts the problem to a single procedure with no dynamic memory allocation. Each location points to at most one memory location at a time. In practice, *summary variables* are often used to conservatively model constructs like aggregates (e.g., arrays or structures), dynamically allocated memory, and local variables in a recursive context as a bundle of memory locations. The handling of summary variables is discussed in Section 5.5. The problem is flow-insensitive, as program control flow is ignored to produce a set of assignments as input.

Def. 3.2 (Points-To Graph)**.** *A points-to graph* $G\colon (V, E)$ *(corresponding to a problem* Π*) consists of a set of vertices* V *and directed edges* E *such that* $V \supseteq X$ *and* $E \subseteq V \times V$*. An edge* $v_1 \mapsto v_2$ *says that* v_1 *may point to* v_2 *under some execution of assignments drawn from* A*. For convenience, we use the notation* $V(G)$ *or* $E(G)$ *to indicate the vertex set or edge set component of* G*, respectively.*

3.2 Dynamic Memory

We now define the full points-to analysis problem, which includes the possibility of dynamic memory allocation:

Def. 3.3 (Points-To Analysis Problem)**.** *A (flow-insensitive) points-to analysis problem* Π *consists of a finite set of variables* X *along with a set of pointer assignments* A*. Each assignment in* A *conforms to one of the following forms:*

$$*^d p := \&q \qquad *^{d_1} p := *^{d_2} q \qquad *^d p := \texttt{malloc}(), \qquad p, q \in X\,.$$

A problem with dynamic memory can be transformed conservatively into a finite memory problem by introducing a fresh summary variable g for each statement with `malloc()`. However, such a transformation may introduce imprecision (Cf. Section 6).

4. Exact Graphs and Realizability

In this section, we build on the points-to analysis definitions of Section 3 to define a witness-generating points-to analysis. In particular, we define concrete semantics for pointer assignments in terms of *exact points-to graphs* and define witnesses over these graphs.

Def. 4.1 (Exact Points-To Graph)**.** *A points-to graph* $G^\natural\colon (V, E)$*, corresponding to a problem* Π *with variables* X*, wherein* $V \supseteq X$ *and* $E \subseteq V \times V$*, is* exact *iff each vertex* $v \in V$ *has at most one outgoing points-to edge in* $G^\natural$*.*

An exact points-to graph corresponds to a single concrete memory state, whereas a may points-to graph obtained as a result of Andersen's analysis may be viewed as the *join* of many exact points-to

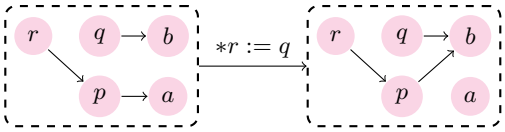


Figure 3. Transformation of an exact graph through assignment. For clarity in presentation, we draw exact graphs with red nodes.

graphs. Furthermore, if a vertex p does not have an outgoing edge in the exact points-to graph $G^\natural$, it is assumed that p holds the address of some *junk* memory location that does not correspond to the addresses of any of the vertices in V. For presentation, we continue to write G for points-to graphs in general and only write $G^\natural$ specifically to clarify that the graph must be exact.

We now define the operational semantics of pointer assignments from a points-to analysis problem (see Section 3.2) over exact graphs $G^\natural$. Let $G_0^\natural, G_1^\natural$ be two exact graphs and $a \in A$ be an assignment. We write $G_0^\natural \xrightarrow{a} G_1^\natural$ to denote that $G_1^\natural$ is reachable in one step from $G_0^\natural$:

- The assignment is of the form $a\colon *^d p := \&q$. There exists a (simple) path of length d in $G_0^\natural$ from node p to some intermediate node p' (denoted $p \rightsquigarrow_d p'$). Let us suppose that $p' \mapsto p'' \in E(G_0^\natural)$. The graph $G_1^\natural$ is defined as follows:

  $$V(G_1^\natural) \stackrel{\text{def}}{=} V(G_0^\natural)\,.$$
  $$E(G_1^\natural) \stackrel{\text{def}}{=} (E(G_0^\natural) - \{p' \mapsto p''\}) \cup \{p' \mapsto q\}\,.$$

  In other words, the existing outgoing edge for p' in $G_0^\natural$ is replaced in $G_1^\natural$ by an edge from p' to q. If p' does not have an outgoing edge in $G_0^\natural$, then $E(G_1^\natural) \stackrel{\text{def}}{=} E(G_0^\natural) \cup \{p' \mapsto q\}$.

- The assignment is of the form $a\colon *^d p := \texttt{malloc}()$. Suppose that $p \rightsquigarrow_d p'$ and $p' \mapsto p'' \in E(G_0^\natural)$. The graph $G_1^\natural$ is defined as follows:

  $V(G_1^\natural) \stackrel{\text{def}}{=} V(G_0^\natural) \cup m$ where $m \notin V(G_0^\natural)$ is fresh location .

  $E(G_1^\natural) \stackrel{\text{def}}{=} (E(G_0^\natural) - \{p' \mapsto p''\}) \cup \{p' \mapsto m\}\,.$

  This case is similar to the first one, except that we create a fresh node m to represent the dynamically allocated memory. Like in the previous case, if an old edge from p' does not exist in $G_0^\natural$, the new one is just added and similarly for the remaining case.

- The assignment is of the form $a\colon *^{d_1} p := *^{d_2} q$, where $d_1, d_2 \geq 0$. There exist paths $p \rightsquigarrow_{d_1} p'$ and $q \rightsquigarrow_{d_2+1} q'$ of length d_1 and (d_2+1), respectively, for $p', q' \in V$:

  $$V(G_1^\natural) \stackrel{\text{def}}{=} V(G_0^\natural)\,.$$
  $$E(G_1^\natural) \stackrel{\text{def}}{=} (E(G_0^\natural) - \{p' \mapsto p''\}) \cup \{p' \mapsto q'\}\,.$$

Note that the semantics implicitly allow the execution of an assignment involving $*p$ for an exact graph $G^\natural$ only when there is an outgoing edge from p. Similar consideration applies for $*^d p$ wherein we require a path from p to some p' of length d.

Figure 3 illustrates the transformation of an exact points-to graph through an assignment. We can now define *realizable* points-to edges.

Def. 4.2 (Realizable Graph, Edges, and Subgraphs)**.** *A graph G is* realizable *iff there exists a sequence of assignments $a_1, \ldots, a_N$ such that $G_0 \xrightarrow{a_1} G_1 \xrightarrow{a_N} G_N \equiv G$, where $G_0\colon (X, \emptyset)$ is the* initial graph *of the points-to-analysis problem (X, A).*

An edge $v_1 \mapsto v_2 \in V \times V$ is realizable *iff there exists a realizable graph G such that $v_1 \mapsto v_2 \in E(G)$. Generally, a subset of edges $E \subseteq V \times V$ is (simultaneously) realizable if there exists a realizable graph G such that $E \subseteq E(G)$.*

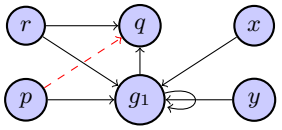


Figure 4. An example demonstrating imprecision in Andersen's analysis. This graph is the result of Andersen's analysis on the problem in Ex. 4.1 (a slight modification of that in Figure 1). The edge $p \mapsto q$ (shown as a dashed red line) is no longer realizable.

We now formally state the problem of precise flow-insensitive points-to analysis.

Def. 4.3 (Precise Flow-Insensitive Points-To Analysis)**.** *Given a points-to analysis problem $\Pi\colon (X, A)$, a precise flow-insensitive points-to analysis computes all the edges $E_r \subseteq X \times X$ that are realizable.*

Note that we only require the analysis to compute points-to relations over program variables, not dynamic memory locations, as is standard in precise points-to analysis formulations [5]. The interface could easily be extended to handle dynamic memory, given some naming scheme for dynamic locations.

Associated with each realizable edge, we can now define a witness sequence.

Def. 4.4 (Witness Sequence)**.** *Let e be a realizable edge. A witness sequence for e is a sequence of assignments $a_1, \ldots, a_N$ starting from the initial graph G_0: $G_0 \xrightarrow{a_1} G_1 \to \cdots \xrightarrow{a_N} G_N \equiv G$, such that $e \in G$.*

A witness-generating precise flow-insensitive points-to analysis produces realizable points-to edges with witness sequences. Note that Andersen's analysis [1], well studied in the literature, is *not* a precise flow-insensitive points-to analysis, as pointed out in previous work [5, 11]. We illustrate this situation using a slightly modified version of our example from Section 2:

Example 4.1. *Consider the following set of pointer assignments:*

$$\{p := *r,\ r := \&q,\ r := *x,\ x := \&g_1,\ y := x,\ *x := r,\ *x := y\}\,.$$

This example is identical to that of Figure 1 except $x := \&g_2$ has been removed. Figure 4 shows the Andersen's analysis result for this example. Note that the edge $p \mapsto q$ is no longer realizable, as realizing the edge requires that either: (1) pointer r points-to $g1$ and q simultaneously; or (2) pointer $g1$ points-to $g1$ and q simultaneously (also shown in Figure 6). Neither situation is possible since a pointer can only hold one value at a time.

This example indicates the need for refutation of points-to edges during witness generation, even for generating precise flow-insensitive witnesses.

5. Witness Generation for Finite Memory

We now present our framework for witness-generating points-to analysis. At the core of our framework is a goal-directed, backwards witness search over possible exact points-to graphs. The immediate result of this section is a witness-generating, precise flow-insensitive points-to analysis for the finite-memory points-to analysis problem (Def. 3.1). We present handling of dynamic memory allocation in Section 6.

At a high-level, our technique proceeds in three stages:

Compute Conservative Points-To Result. First, we run a conservative flow-insensitive points-to analysis, such as Andersen's [1].

Generate Edge Dependency Rules. Second, based on the conservative points-to result, we create *edge dependency rules* that capture ways a points-to edge may arise. This rule generation can be done offline to take advantage of an optimized, off-the-shelf points-to analysis, but it can also be performed online during the execution of Andersen's analysis.

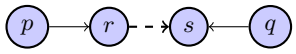
Search for Witnesses. Third, we search for witness sequences for a given edge, on demand, using the edge dependency rules. Our technique may also produce a *refutation* that no witness sequence exists.

This search can be viewed as a cooperation between a witness enumerator and a *refuter* that detects when an inconsistent state has been reached. A novel aspect of our approach is that one can get various levels of precision simply by using different kinds of refuters (see Section 7).

5.1 Generating Edge Dependency Rules

The first step is to derive a set of *edge dependency rules* based on a points-to graph produced by a conservative points-to analysis. We first illustrate edge dependency rule construction through an example.

Example 5.1. *Let G be a points-to graph derived as the result of a conservative points-to analysis. Consider the assignment $a\colon *p := q$, wherein edges $p \mapsto r$, $q \mapsto s$, and $r \mapsto s$ exist in G, as illustrated below:*



In terms of realizability, the following claim can be made in this situation:

> *Edge $r \mapsto s$ is realizable (using assignment a) if the edge set $\{p \mapsto r,\ q \mapsto s\}$ is simultaneously realizable.*

Note that the converse of this statement need not be true—the edge $r \mapsto s$ may be realizable using another set of edges and/or a different pointer assignment.

In our framework, this assertion is represented by a dependency rule*:*

$$r \mapsto s \xleftarrow{a\colon *p:=q} \{p \mapsto r, q \mapsto s\}$$

This dependency rule indicates that the edge $r \mapsto s$ can be produced as a result of the assignment a whenever the edges $p \mapsto r$ and $q \mapsto s$ can be realized simultaneously.

Def. 5.1 (Edge Dependency Rules)**.** *In general, an* edge dependency rule r *has the following form:*

$$e_l \xleftarrow{a} \{e_{r1}, \ldots, e_{rm}\}\ ,$$

indicating that the edge e_l can be realized through the assignment a whenever the edges $e_{r1}, \ldots, e_{rm}$ can be simultaneously realized.

As noted earlier, edge dependency rules can be constructed for a given problem in two ways: either offline given the points-to graph resulting from any conservative points-to analysis or online during the course of executing Andersen's analysis whenever an assignment is processed.

Offline Construction of Rules. The offline construction of rules can be performed over a points-to graph G that results from a conservative analysis. Let us first consider assignments of the form $*^m p := *^n q$. For each such assignment, we generate a set of rules as follows:

$$r \mapsto q \xleftarrow{r:=\&q} \emptyset \qquad x \mapsto g_1 \xleftarrow{x:=\&g_1} \emptyset \qquad y \mapsto g_1 \xleftarrow{y:=x} x \mapsto g_1$$

$$g_1 \mapsto q \xleftarrow{*x:=r} x \mapsto g_1, r \mapsto q \qquad g_1 \mapsto g_1 \xleftarrow{*x:=r} x \mapsto g_1, r \mapsto g_1$$

$$g_1 \mapsto g_1 \xleftarrow{*x:=y} x \mapsto g_1, y \mapsto g_1$$

$$r \mapsto g_1 \xleftarrow{r:=*x} x \mapsto g_1, g_1 \mapsto g_1 \qquad r \mapsto q \xleftarrow{r:=*x} x \mapsto g_1, g_1 \mapsto q$$

$$p \mapsto g_1 \xleftarrow{p:=*r} r \mapsto g_1, g_1 \mapsto g_1 \qquad p \mapsto q \xleftarrow{p:=*r} r \mapsto g_1, g_1 \mapsto q$$

Figure 5. The edge dependency rules for the problem in Ex. 4.1.

- Let $\text{paths}(p, m)$ denote the set of all paths of length m starting from p in G, and let $\text{paths}(q, n+1)$ be the set of all paths of length $n+1$ starting from q in G.
- Consider each pair of paths

  $$\pi_1\colon p \rightsquigarrow_m p' \in \text{paths}(p, m) \qquad \text{and}$$
  $$\pi_2\colon q \rightsquigarrow_{n+1} q' \in \text{paths}(q, n+1)\ .$$

  For each such pair, we generate the following edge dependency rule:

  $$p' \mapsto q' \xleftarrow{*^m p:=*^n q} E(\pi_1) \cup E(\pi_2)\ ,$$

  wherein $E(\pi_i)$ denotes the edges in the path π_i for $i \in \{1, 2\}$.

The case for assignments of the form $*^m p := \&q$ is essentially the same, so we elide it here. (We discuss $*^m p := \texttt{malloc}()$ in Section 6). Overall, we obtain the set of rules for a finite-memory problem $\Pi\colon (X, A)$ by taking all such rules generated from all assignments $a \in A$.

Theorem 5.1 (Complexity of Offline Rule Construction)**.** *Given a conservative points-to graph $G\colon (V, E)$ corresponding to a problem $\Pi\colon (X, A)$, the number of rules derived and the time taken to generate these rules is $O\left(|A| \cdot |E|^{d+1}\right)$ where d is the maximum number of total dereferences in any assignment $a \in A$.*

Proof. A rule involving an assignment of the form $*^m p := *^n q$ consists of a pair of paths $p \rightsquigarrow_m x$ and $q \rightsquigarrow_{n+1} y$ in the points to graph. The number of such paths is upper bounded by $|E|^{m+n+1} = |E|^{d+1}$. A similar argument holds for assignments of the form $*^m p := \&q$. Overall, the number of rules is upper bounded by $O\left(|A| \cdot |E|^{d+1}\right)$. Since each step of the algorithm generates a new rule, by considering a new assignment a and a new pair of paths, the running time corresponds to the number of rules generated. □

Note that the time taken is polynomial in the size of the problem and the number of edges in the points-to graph (which is in turn at most quadratic in the number of variables). The time taken is exponential in the number of dereferences in the pointer assignments, but usually this number is very small. For statements processed for input to Andersen's analysis, $d \leq 1$, in which case the overall complexity is polynomial.

Online Construction of Rules. Consider a points-to edge e discovered in the course of Andersen's analysis while processing an assignment a. The edges traversed at this step to produce e are exactly the dependence edges needed to create an edge dependency rule (as in the offline construction case).

Example 5.2. *Figure 5 shows the edge dependency rules derived from the result of Andersen's Analysis for the problem in Example 4.1. The set of pointer assignments is repeated here:*

$$\{p := *r,\ r := \&q,\ r := *x,\ x := \&g_1,\ y := x,\ *x := r,\ *x := y\}\ .$$

Theorem 5.2. *The realizability of an edge e is dependent on the set of edge dependency rules as follows:*

(1) For each dependency rule $r_i : e \xleftarrow{a} E_i$, if the subset of edges E_i is (simultaneously) realizable then e is realizable.

(2) Given the set of all dependency rules $\{r_1, \ldots, r_m\}$ with e on the left side: $r_i : e \xleftarrow{a} E_i$, the edge e is realizable only if the subset E_j corresponding to some rule r_j is (simultaneously) realizable.

Proof. (1) Let $r_i : e \xleftarrow{a} E_i$ be a rule. It can be verified by inspecting the rule construction process that the assignment a can be applied construct the edge e whenever the edges in the set E_i are all simultaneously present in an exact graph.

(2) Let $R(e) = \{r \mid r : e \xleftarrow{a} E\}$ be the set of all rules that have the edge e on the LHS. Let us assume that for all rules $r_j : e \xleftarrow{a_j} E_j \in R(e)$, the set E_j is not simultaneously realizable. We wish to show that e is not realizable.
Let $G_0 \xrightarrow{a_0} G_1 \xrightarrow{a_1} \cdots G_n \xrightarrow{a_n} G$ be a sequence of exact graphs such that the final graph G has the edge e and furthermore, $e \notin E(G_i)$ for $i \in [0, n]$. Such a sequence always exists if e is realizable.
We now consider the assignment a_n and the graph G_n. Let $E_n \subseteq E(G_{n-1})$ be the set of edges that are "involved" in the assignment. This set can be defined based on the nature of the assignment a_n as follows:
- For an assignment of the form $a_n : *^m p := *^n q$, there must be a path of size m from p to the head of e and a path of size $n+1$ from q to the tail of e. The edges on this path constitute the set E_n.
- For an assignment of the form $a_n : *^m p := \&q$, there must be a path of size m from p. The edges on this path constitute the set E_n.

Clearly, a conservative pointer analysis should have concluded that the edges in E_n are possible may-points-to relations, and further, the rule generation process will generate the rule: $r_n : e \xleftarrow{a} E_n$. This leads to a contradiction with our original assumption that the edges in E_n are not simultaneously realizable.

□

This characterization leads to a search algorithm based on the set of edge dependency rules for a problem.

5.2 Witness Enumeration

Once edge dependency rules are generated, witness search is performed via witness *enumeration*, which constructs possible partial witnesses, and *refutation*, which cuts off exploration of invalid partial witnesses. We describe enumeration here, assuming no refutation is performed, and then discuss a flow-insensitive refuter in Section 5.3.

Consider a rule $r : e \xleftarrow{a} E$. Rule r states that we can realize edge e if we can realize the set of edges E simultaneously (i.e., in a state satisfying E, executing a creates the desired points-to e). Intuitively, we can realize the set E if we can find a chain of rules that realize each edge in E. Thus, enumeration proceeds by repeatedly rewriting edge sets based on dependency rules until reaching the empty set; the statements associated with the rules employed become the candidate witness.

We define the rewriting of edge sets with rules as follows.

Def. 5.2 (Rewriting Edge Sets)**.** *Let E_1 be a non-empty set. We write $E_1 \xrightarrow{r} E_2$ for a rule $r : e \xleftarrow{a} E'$ such that $e \in E_1$ and*

$$E_2 = (E_1 - \{e\}) \cup E' .$$

Example 5.3. *Consider the set of rules derived for Example 4.1 as shown in Figure 5. Starting from the set $E : \{r \mapsto g_1, g_1 \mapsto g_1\}$ and using the rule $r : g_1 \mapsto g_1 \xleftarrow{*x:=y} x \mapsto g_1, y \mapsto g_1$, we can rewrite set E to a set E' as follows:*

$$E : \{r \mapsto g_1, g_1 \mapsto g_1\} \quad \xrightarrow{r} \quad E' : \{x \mapsto g_1, y \mapsto g_1, r \mapsto g_1\} .$$

Often, we will write such transitions using the same format as the rule itself:

$$E : \{r \mapsto g_1, g_1 \mapsto g_1\} \quad \xleftarrow{*x:=y} \quad E' : \{x \mapsto g_1, y \mapsto g_1, r \mapsto g_1\} .$$

5.3 Flow-Insensitive Refutation

Here we describe our first *refuter* R_{fin} for precise, witness-generating flow-insensitive points-to analysis (for finite memory). A refuter is invoked at each rewriting step of the witness enumeration process. The refuter is able to terminate that branch of the search if the partial witness is invalid (to guarantee precision) or redundant (to guarantee termination).

In general, a refuter guarantees precision by ensuring the witness search follows some feasible execution (in a particular concrete semantics). Refuter R_{fin} guarantees precision by detecting edge sets that require a singleton location to simultaneously point to more than one location (violating the exact points-to graph constraint in Def. 4.1). Recall the definition of realizability (Def. 4.2). It states that a set of edges E is realizable if it is a subset of edges in a realizable graph. A realizable graph must be an exact points-to graph. Therefore, R_{fin} simply detects when the exactness constraint is violated, which we call a *conflict set*:

Def. 5.3 (Conflict Set)**.** *A set of edges E is a* conflict set *iff there exist two or more outgoing edges $v \mapsto v_1, v \mapsto v_2 \in E$ for some vertex v.*

We note that a conflicting set of edges is not simultaneously realizable. Hence, whenever a conflict set is detected, R_{fin} terminates that branch of the search.

In addition to conflict detection, R_{fin} guarantees termination by stopping cyclic rewriting of edge sets. Intuitively, if we have $E_1 \xrightarrow{r_1} E_2 \xrightarrow{r_2} \ldots \xrightarrow{r_n} E_n$, wherein $E_n \supseteq E_1$, the corresponding statements have simply restored the points-to edges in E_1. Hence no progress has been made toward a complete witness. Clearly, since all cyclic rewriting is terminated, and we have a finite number of possible edge sets (since memory is finite), R_{fin} guarantees termination.

We now prove the correctness of R_{fin} by showing that if a witness exists for a points-to edge, witness enumeration with R_{fin} will find a witness. We also show that when R_{fin} is employed, any witness output by enumeration is valid.

Lemma 5.1 (Head Expansion of Realizability)**.** *Let E, E' be two edge sets such that $E \xrightarrow{r} E'$ for some edge dependency rule r such that E is non-conflicting. If E' is simultaneously realizable, then so is the set E.*

Proof. Let us assume that the rule r is of the form $r : e \xleftarrow{a} F$. Also let $e : s \to t$. Since $E \xrightarrow{r} E'$, it follows that (1) $e \in E$, (2) $F \subseteq E'$, and $E - \{e\} \subseteq E'$.

Let $G_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} G_n$ be a sequence of exact graphs starting from the initial graph G_0 such that $E' \subseteq E(G_n)$. Clearly, such a sequence exists since we assumed that E' is simultaneously realizable. Since $F \subseteq E'$, we have that $F \subseteq E(G_n)$. Applying the assignment a to the graph G_n, we may we obtain a graph G with the edge e. We now argue that edges in $E - \{e\}$ will also remain unaffected by the assignment a. First, $E - \{e\} \subseteq E' \subseteq E(G_n)$. Secondly, an edge in $e_1 \in E - \{e\}$ can be removed by the assignment a only if e_1 shared the same source node as e. However,

this means that the set E has a conflict. Therefore, the sequence:

$$G_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} G_n \xrightarrow{a} G\,.$$

yields an exact graph G such that $E \subseteq E(G)$, proving the simultaneous realizability of E. □

We now prove a key lemma about witness enumeration:

Lemma 5.2 (Complete Enumeration)**.** *For each sequence of exact graphs showing realizability of an edge e:*

$$s : G_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} G_n\,,$$

wherein $a_1, \ldots, a_n$ are assignments, G_0 is the initial graph and $e \in G_n$, there is a corresponding sequence of edge sets:

$$w : E_n : \{e\} \rightarrow E_{n-1} \ldots \rightarrow E_0 : \emptyset\,,$$

wherein, for each $i \in [1, n]$ $E_i = E_{i-1}$ or $E_i \xrightarrow{r_i} E_{i-1}$. Furthermore, (A) $E_i \subseteq E(G_i)$, (B) E_i does not contain a conflict and (C) the assignment sequence $b_1, \ldots, b_m$ obtained from w (via the rules r_i employed when $E_i \neq E_{i-1}$) is a subsequence of $a_1, \ldots, a_n$.

Proof. The sequence of edge sets $E_n \ldots E_0$ is constructed starting so that the properties stated in the lemma are maintained invariant as follows.

1. Let $E_n = \{e\}$.
2. We construct E_i, given E_{i+1} as follows. Consider the assignment $G_i \xrightarrow{a_i} G_{i+1}$. There are two cases to consider:
   (case-1) The assignment a_i applied to G_i produces an edge $e \in E_{i+1}$ such that $e \notin E(G_i)$. Let $r : e \xleftarrow{a_i} E'$ be the corresponding rule application. We then set $E_i = (E_{i+1} - \{e\}) \cup E'$.
   (case-2) The assignment a_i applied to G_i produces an edge e' such that $e' \notin E_{i+1}$. In this case, we conclude that the edge e' produced is irrelevant for the overall realizability of the edge e. We just set $E_i = E_{i+1}$.

   We now prove the invariants are maintained through our construction above.
   (A) To prove that $E_i \subseteq E(G_i)$, we observe that it is true for $i = n$. Next, we prove that if $E_{i+1} \subseteq E(G_{i+1})$ the construction above ensures that $E_i \subseteq E(G_i)$. This is achieved using a case-by-case reasoning on the nature of the assignment a_i used and which of the two cases identified in the construction above applies.
   (B) E_i cannot contain a conflict, since $E_i \subseteq E(G_i)$ and $E(G_i)$ cannot contain a conflict for any exact graph G_i.
   (C) This is again immediate from the construction above. The assignment sequence is a subsequence since case-2 of the construction above skips assignments that are *irrelevant* to the production of the edge e whose realizability we care about.

□

Lemma 5.2 effectively states that for every realizable edge, there is a witness sequence. Going further, it also states that the witness enumeration can potentially generate *every* flow insensitive witness that realizes a given edge e. This will be useful while searching for witnesses that respect constraints arising from the control flow in the original program (Cf. Section 7) and in the handling of dynamic memory (Cf. Section 6.

Using the two lemmas above, we can prove the main result.

Theorem 5.3 (Realizability)**.** *(A) An edge e is realizable iff there exists a sequence of rewrites*

$$w :\ E_0 \colon \{e\} \xrightarrow{r_1} E_1 \xrightarrow{r_2} \cdots \xrightarrow{r_N} E_N \colon \emptyset\,,$$

such that none of the sets $E_0, \ldots, E_N$ are conflicting.

(B) Furthermore, it is also possible to find w such that $E_i \not\supseteq E_j$ for all $i < j$.

Proof. (A, $\Rightarrow$) Let us assume that an edge e is realizable. Furthermore let $a_1, \ldots, a_N$ be some sequence of assignments that results in a subgraph G with the edge e starting from the empty initial graph G_0 as

$$G_0 \xrightarrow{a_1} G_1 \cdots \xrightarrow{a_N} G_N\,.$$

Using Lemma 5.2, we conclude the existence of a witness sequence:

$$E_0 : \{e\} \xrightarrow{r_1} \ldots \xrightarrow{r_N} E_N : \emptyset$$

We know further that each set E_i in this sequence does not have a conflict, yielding the required witness sequences over sets of edges.

(A, $\Leftarrow$) Let us assume, on the other hand, that a witness sequence $E_m : \{e\} \xrightarrow{r_m} \cdots \xrightarrow{r_1} \emptyset$. We know from Lemma 5.1 that if $E_i \xrightarrow{r} E_{i+1}$ and E_{i+1} is simultaneously realizable then so is E_i. The rest of the proof is done by induction starting from the set $E_N : \emptyset$ which is trivially simultaneously realizable. We then conclude that $E_0 : \{e\}$ is realizable.

(B) Let

$$E_0 : \{e\} \xrightarrow{r_1} \ldots \xrightarrow{r_N} E_N : \emptyset$$

be the shortest sequence that shows the realizability of $\{e\}$. Assume, however, that $E_i \supset E_j$ for some $i > j$ and furthermore, amongst all such pairs (i, j) in the sequence, consider the pair (i, j) wherein j has the largest value. We can produce a smaller sequence by skipping the subsequence $E_j \rightsquigarrow E_i$. The suffix $E_i \rightsquigarrow \emptyset$ is now used to construct a new sequence $E_j \rightharpoonup F_{j+1} \rightsquigarrow \emptyset$. Consider the rewrite: $E_{i+\alpha} \xrightarrow{r} E_{i+1+\alpha}$. Let $r : e_\alpha \xleftarrow{a} E'$ be the rule applied. By induction, we construct $F_{j+\alpha} \subset E_{i+\alpha}$ based on two cases:

C-1 If $e_\alpha \in F_{j+\alpha}$, we apply the rule r_α to $F_{j+\alpha}$ to obtain $F_{j+\alpha+1}$.
C-2 If $e_\alpha \notin F_{j+\alpha}$, then we do not apply a rule and simply let $F_{j+1+\alpha} = F_{j+\alpha}$.

Overall, the new sequence obtained

$$E_0 \rightsquigarrow E_j (\equiv F_j) \rightharpoonup F_{j+1} \cdots \rightharpoonup \emptyset$$

wherein $F_{j+\alpha} \subseteq E_{i+\alpha}$ can also be shown to serve as a witness sequence for $\{e\}$ and is of length at least one less than the shortest sequence. This leads to a contradiction. □

Example 5.4. *Considering the problem from Ex. 4.1, the following sequence of rule applications demonstrates the realizability of the edge $r \mapsto g_1$:*

$$\{r \mapsto g_1\} \xleftarrow[y:=x]{r:=*x} \{x \mapsto g_1, g_1 \mapsto g_1\} \xleftarrow{*x:=y} \{x \mapsto g_1, y \mapsto g_1\} \xleftarrow{x:=\&g_1} \{x \mapsto g_1\} \xleftarrow{} \emptyset\,.$$

The sequence of assignments corresponding to the set of rule applications provides the witness sequence:

$$x := \&g_1; \quad y := x; \quad *x := y; \quad r := *x;\,.$$

The converse of Theorem 5.3 can be applied to show that a given edge is not realizable. To do so, we search over the sequence of applicable rules, stopping our search when a conflicting set or a superset of a previously encountered set of edges is encountered.

Example 5.5. *A refutation tree for the non-realizability of edge $p \mapsto q$ is shown in Fig. 6.*

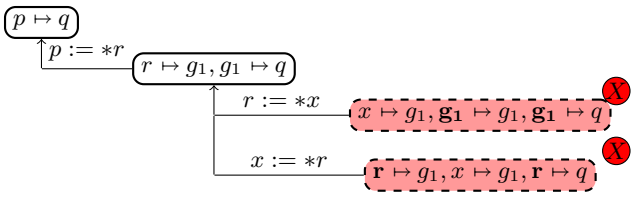


Figure 6. Refutation tree for non-realizable edge $p \mapsto q$.

Using the fact that our search remains over non-conflicting edges, the size of the sets $E_i \subseteq E$ cannot exceed the number of nodes in the graph. Therefore, we can claim the following upper bound on the witness length.

Lemma 5.3. *For a flow-insensitive analysis problem with n variables, if an edge e is realizable then it has a witness sequence of size $O(2^{n \lg n})$.*

Proof. During a witness search the number of possible *non-conflicting* edge sets encountered is bounded by $O(n^n) = O(2^{n \lg n})$. □

We have been unable to improve the bound or find a family of examples proving that it is tight. Doing so can help resolve a long standing question regarding the complexity of flow-insensitive pointer analyses [11].

As a result of these observations, we conclude that the overall complexity of the algorithm for finding witnesses remains in polynomial space.

Theorem 5.4. *The search for a witness through rule application always terminates. Furthermore, the problem of checking realizability of an edge for a flow insensitive points-to analysis problem can be solved in polynomial space.*

Proof. Each edge set can be stored in space $O(n \lg n)$ since the size of set can be at most n and each edge requires $O(\lg n)$ bits to store. Therefore, the search can be done in polynomial space. Appealing to Savitch's theorem to determinize the search concludes the proof. □

This bound matches known bounds established by Muth and Debray [14]. Interestingly, our technique can lead to a better upper-bound on the complexity if a bound on the size of the largest witness can be established.

Claim 5.1. *The problem of checking edge realizability is in NP,* provided it can be shown that each realizable edge has a polynomial size witness.

Proof. If each realizable edge had a polynomial length witness, this witness serves as a polynomial length certificate to show the realizability of the edge. The problem therefore belongs to NP. □

Thus far, we have been unable to prove such a polynomial bound or exhibit a family of problems wherein some realizable edge does not have a polynomial length witness. However, it seems likely that a bound that is polynomial in the problem size will exist on the size of the shortest length witness for a realizable edge.

5.4 Symbolic Witness Search

In this section, we discuss a symbolic encoding for witness search and proving unrealizability. Consider a set of edges E and a set of rules R over E. To prove the realizability of a given edge e, we wish to check for the existence of a sequence of rule applications of a given length $k > 0$: $\sigma_k : r_1, \ldots, r_k$ so that starting from the set $E_0 : \{e\}$ we obtain $E_0 \rightsquigarrow_\sigma \emptyset$.

An explicit search over configurations of edges with backtracking may be expensive since the number of sequences to be considered is exponential in k. Therefore, we wish to leverage the numerous search heuristics found in modern SAT solvers to perform this search symbolically.

For each k, we will describe a Boolean formula $\varphi(k)$ that will encode the presence of a witness of length k. We will also show that this formula can be adapted to infer the absence of witnesses for *all depths*.

Propositions: For each edge $e \in E$ and depth $i \in [1, k+1]$, the Boolean variable $\mathsf{Edg}(i, e)$ denotes the presence of edge e in the set obtained at depth i.

Similarly, for depths $i \in [i, k]$, the Boolean variable $\mathsf{Rl}(i, e)$ will be used to denote the application of the rule r at depth i (to obtain the set at depth $i + 1$).

Note that there is no rule application at the last step. A unique feature of the symbolic search is that we allow multiple *non-conflicting* rule applications at each step (see Def. 5.4) .

Boolean Encoding: The singleton $E_s : \{e\}$ describes the starting edge set. This is encoded by the following clause:

$$\mathsf{init}(e) : \mathsf{Edg}(e, 1) \wedge \bigwedge_{e' \in E, e' \neq e} \neg \mathsf{Edg}(e', 1) .$$

Recall that a pair of edges conflict if they have the same source location (which is not a summary variable, cf. Section 5.5). For each pair of conflicting edges e_A, e_B and for depth i, we add the clause:

$$\mathsf{edgeConflict}(e_A, e_B, i) : \neg \mathsf{Edg}(e_A, i) \vee \neg \mathsf{Edg}(e_B, i) .$$

We now define a notion of a conflict on the rules

Def. 5.4 (Conflicting Rules)**.** *Rules $r_1 : e_1 \xleftarrow{a_1} E_1$ and $r_2 : e_2 \xleftarrow{a_2} E_2$ are* conflicting *iff one of the following conditions holds: (a) $e_1 = e_2$, or (b) e_1 conflicts with some edge in E_2, or (c) e_2 conflicts with some edge in E_1.*

Lemma 5.4. *If two rules r_1, r_2 are not conflicting, then they may be applied in any order on a given set of edges to yield the same result. This extends to any set of pairwise non-conflicting rules.*

The clause $\mathsf{ruleConflict}(r_1, r_2, i) : \neg \mathsf{Rl}(r_1, i) \vee \neg \mathsf{Rl}(r_2, i)$ avoids conflicting rule applications at the same depth.

The clause $\mathsf{someRule}(i) : \bigvee_{r \in R} \mathsf{Rl}(r, i)$ asserts that some rule must be applied at depth i.

We add a clause to enforce that a rule $r : e \leftarrow E$ is applicable at depth i only if the corresponding edge e is present at that depth:

$$\mathsf{ruleApplicability}(r, i) : \mathsf{Rl}(r, i) \Rightarrow \mathsf{Edg}(e, i), \text{ where } r : e \leftarrow E .$$

Using the clauses $\mathsf{ruleToEdgeRelation}(e, i)$ defined below, we relate the set of edges at depth $i + 1$ to the rule applied to the set at depth i:

$$\mathsf{Edg}(e, i+1) \Leftrightarrow \left(\begin{array}{c} \mathsf{Edg}(e, i) \wedge \left(\bigwedge_{r:\, e \leftarrow E} \neg \mathsf{Rl}(r, i)\right) \\ \text{(* } e \text{ already existed and not destroyed *)} \\ \bigvee \\ \bigvee_{r' : e' \leftarrow E,\ e \in E} \mathsf{Rl}(r', i) \\ \text{(* } e \text{ produced by some rule application *)} \end{array} \right)$$

During the witness search, if we encounter an edge set E_i at depth i, such that $E_i \supseteq E_j$ for a smaller depth $j < i$ then the search can be stopped and a different set of rule applications explored.

This is encoded as follows:

$$\mathsf{notSubsumes}(i,j) : \neg\left(\bigwedge_{e\in E} \mathsf{Edg}(e,i) \Rightarrow \mathsf{Edg}(e,j)\right).$$

Overall Encoding: The overall encoding for an edge e is the conjunction of all the clauses described earlier.

$$\varphi(e_s,k) : \bigwedge_{i\in[1,k]} \begin{bmatrix} & \mathsf{init}(e_s) \\ \bigwedge_{e_1,e_2,\text{conflicting}} & \mathsf{edgConflict}(e_1,e_2,i) \\ \bigwedge_{r_1,r_2,\text{conflicting}} & \mathsf{ruleConflict}(r_1,r_2,i) \\ \wedge & \mathsf{someRule}(i) \\ \wedge \bigwedge_{r\in R} & \mathsf{ruleApplicability}(r,i) \\ \wedge \bigwedge_{e\in E} & \mathsf{ruleToEdge}(e,i) \\ \wedge \bigwedge_{j\in[1,i-1]} & \mathsf{notSubsumes}(i,j) \end{bmatrix}.$$

The overall witness search for edge e, consists of increasing the depth bound k incrementally until either

(A) $\varphi(e,k)$ is unsatisfiable indicating a proof of unrealizability of the edge e, or

(B) $\varphi(e,k) \wedge \mathsf{emptySet}(k+1)$ is satisfiable yielding a witness, wherein, the clause $\mathsf{emptySet}(i) : \bigwedge_{e\in E} \neg\mathsf{Edg}(e,i)$ encodes an empty set of edges.

Lemma 5.5. *If $\varphi(e,k)$ is unsatisfiable then there cannot exist a witness for e for any depth $l \geq k$.*

If $\varphi(e,k) \wedge \mathsf{emptySet}(k+1)$ is satisfiable then there is a witness for the realizability of the edge e.

5.5 Summary Variables

In practice, problems arising from programming languages such as C will contain complications such as union types, structure types handled field insensitively, local variables in a recursive function, thread local variables and dynamic memory allocations. Such constructs are often handled conservatively through *summary variables*. These summary variables model a collection of an unbounded number of concrete memory locations.

The semantics for summary variables allow them to point simultaneously to multiple locations. Furthermore, the assignments to these variables are modeled using *weak updates*, wherein some of the values stored prior to the assignment may be retained.

The presence of such summary variables does not affect flow-insensitive may points-to analysis. Likewise, edge dependency rules can also be generated in the presence of summary variables using the may points-to graph. However, witness search and refutation require minor modifications to find witnesses in the presence of summary variables:

Modified Rule Application: Consider a rule $r : e \xleftarrow{a} E_r$, wherein the source vertex of edge e corresponds to a summary variable. The application of rule to a set of edges E, $E \xrightarrow{r} E'$ is modified to allow for two outcomes: (a) $E' = (E - \{e\}) \cup E_r$ (as before), or (b) $E' = (E \cup E_r)$. The second outcome allows for the edge e to be retained in the set of edges to model a weak update.

Modified Refutation: The definition of a conflict is modified to ensure that two edges $p \mapsto q$ and $p \mapsto r$ will not conflict when p is a summary vertex.

The symbolic witness search described in Section 5.4 can also be suitably modified to carry out a witness search and refutation in the presence of summary nodes.

6. Witness Generation for Dynamic Memory

In this section, we refine the witness search algorithm from Section 5 to generate precise witnesses in the presence dynamic memory allocation. Section 5.5 describes a witness generation scheme in the presence of summary variables, based on a semantics with weak updates. While the witness generation scheme presented there is a simple way to soundly model summary variables, it restricts the refuter in its ability to detect conflicts. Intuitively, the central issue is that a weak update-based scheme does not search over the concrete execution semantics as in Section 5.2, which allowed us to obtain precise answers with R_{fin} (cf. Section 5.3).

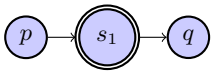
In this section, we present a precise flow-insensitive witness generation technique in the presence of dynamic memory allocation. Our approach expands each summary variable, on demand, into multiple "concrete" *instances* so that conflict detection along the lines of R_{fin} can be applied. Therefore, we first extend edge dependency rules with *instance variables*. The resulting witness search with instances exhibits the desired enumeration properties. Thus, we extend the flow-insensitive refuter from Section 5.3 to produce a precise semi-algorithm for flow-insensitive points-to analysis for programs with dynamic memory (see Def. 3.3).

6.1 Extending Dependency Rule Generation

Recall that the witness search algorithm described in Section 5.2 works as follows: at each step, it selects an edge e to realize and picks a rule that can create e with an assignment a and then constructs an exact points-to graph that must hold before the application of a to create the edge e. Since at each step, the points-to graph (represented by the set of edges) is exact, the resulting trace yields a witness sequence.

The edge dependency rules capture how to transition between exact points-to graphs. We now describe modifications to these rules in the presence of summaries to retain precision. In particular, the realizability of an edge depends not only on the simultaneous existence of edges along a dereference path, but also that the points-to facts over summary vertices along a path hold for the same concrete instances. The following example clarifies this point:

Example 6.1. *Consider the assignment $x := *p$ with the following points-to graph:*



where s_1 is a summary vertex (shown with a double circle). Recall that s_1 represents an unspecified number of concrete instances that we shall denote $s_1[\alpha_1], \ldots, s_1[\alpha_k]$.

This assignment causes the edge $x \mapsto q$ only if in some "concretization" of this points-to graph where p points to a concrete instance $s_1[\alpha]$ and the very same *instance $s_1[\alpha]$ points to q.*

In essence, we need to generate edge dependency rules that specify constraints over instances of summary vertices in a points-to graph. Therefore, we introduce the notion of an *instance variable* ι that allows us to quantify over instances of summary vertices. We write $s[\iota]$ to denote some instance ι of the summary vertex s.

Def. 6.1 (Edge Dependency Rules with Instances)**.** *A points-to fact $f \in F$ involves non-summary vertices and instances of summary vertices. Specifically, we disallow the direct use of summary variables in our points-to facts.*

points-to facts	f	$::=$	$u_1 \mapsto u_2$
pointers	u	$::=$	$t \mid s[\iota],\ t$ *is non-summary vertex.*
instance variables	ι		

An edge dependency rule r over a set of instance variables $\vec{\iota}$ has the following form:

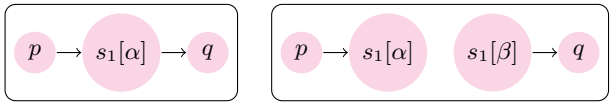
$$f_l \xleftarrow{a} \{f_{r1}, \ldots, f_{rm}\}$$

indicating that the points-to fact f_l can be realized through the assignment a whenever the facts $f_{r1}, \ldots, f_{rm}$ can be simultaneously realized, for all instantiations of the instance variables $\vec{\iota}$.

Continuing Example 6.1, the following dependency rule is obtained:

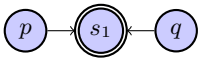
$$x \mapsto q \xleftarrow{x:=*p} \{p \mapsto s_1[\iota], s_1[\iota] \mapsto q\}$$

Observe that the instance variable ι captures the constraint that for x to point to q, the location that p points to must be the same as the location that points to q. To better see what this rule captures, let us define an instance points-to graph where all edges are between non-summary vertices and instances, representing the points-to facts. For example, avoiding the points-to graph with summaries as in Example 6.1, we instead operate over instance points-to graphs such as the following:



For clarity, we write α and β with subscripts to denote specific ground instances, implicitly stating that $\alpha \neq \beta$. We reserve ι with subscripts for instance variables in dependency rules. The first instance graph says there is one instance of s_1 that is pointed to by p and points to q, while the second graph says there are two instances of s_1 where p points to one instance and the other instance points to q. The above dependency rule essentially says that for $x \mapsto q$ to hold after the assignment $x := *p$, we must have the first instance graph and not the second.

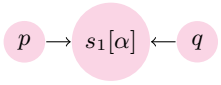
The same summary vertex may also appear with different instances in a dependency rule if the particular instance does not matter for establishing a given points to fact. For example, consider the assignment $*p := q$ with the following points-to graph:



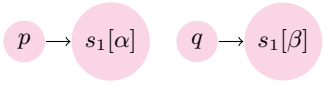
This assignment makes an instance of s_1 point to some other (or possibly the same) instance of s_1. The effect of this assignment is captured succinctly as follows:

$$s_1[\iota_1] \mapsto s_1[\iota_2] \xleftarrow{*p:=q} \{p \mapsto s_1[\iota_1], q \mapsto s_1[\iota_2]\}$$

This single rule captures both the case where p and q point to same instance or different instances of s_1 depending on whether ι_1 and ι_2 are instantiated with the same or different ground instances. For example, in the instance graph



the effect of $*p := q$ is to create the self-loop (i.e., $s_1[\alpha] \mapsto s_1[\alpha]$). Whereas, in



the effect is to create a points-to relation between two objects (i.e., $s_1[\alpha] \mapsto s_1[\beta]$).

Rule generation is only a slight extension of the algorithm described in Section 5.1: for a dereference path π, we introduce instance variables for each summary vertex along π. More precisely, we write $\pi[\vec{\iota}]$ as an instance path through a points-to graph where each occurrence of any summary node has been annotated with a unique instance variable from $\vec{\iota}$. For example, for a path $\pi\colon p \mapsto s_1 \mapsto s_2 \mapsto s_1$ where the s_i's are summary vertices, we have an instance path

$$\pi[\iota, \iota', \iota'']\colon p \mapsto s_1[\iota] \mapsto s_2[\iota'] \mapsto s_1[\iota''] \,.$$

Thus, rules for an assignment of the form $*^m p := *^n q$ are generated as follows: for each pair of paths

$$\pi_1\colon p \rightsquigarrow_m v_1 \in \text{paths}(p, m) \qquad \text{and}$$
$$\pi_2\colon q \rightsquigarrow_{n+1} v_2 \in \text{paths}(q, n+1)\,,$$

we create the following edge dependency rule:

$$\text{end}(\pi_1[\vec{\iota}_1]) \mapsto \text{end}(\pi_2[\vec{\iota}_2]) \xleftarrow{*^m p:=*^n q} E(\pi_1[\vec{\iota}_1]) \cup E(\pi_2[\vec{\iota}_2])$$

where $\text{end}(\pi[\vec{\iota}])$ yields the last node in the given instance path and such that $\vec{\iota}_1$ and $\vec{\iota}_2$ are distinct (i.e., $\vec{\iota}_1 \cap \vec{\iota}_2 = \emptyset$).

Let us consider an initial conservative points-to analysis that uses the typical abstraction for dynamic memory allocation where all concrete locations created by a particular statement is represented by a single summary vertex. Any initial points-to result thus uses a summary vertex for each `malloc()` site. Therefore, for each allocation statement $p := \texttt{malloc}()$, we have the following rule:

$$p \mapsto s[\iota] \xleftarrow{p:=\texttt{malloc}()} \{\}$$

where s is the summary vertex for this allocation statement. Intuitively, we can always witness the creation of an instance of s using this allocation statement. For simplicity, we show a left-hand side with no dereferences; to generate rules for $*^m p := \texttt{malloc}()$ in general, it follows from our above description of the $*^m p := *^n q$ case.

6.2 Extending Witness Enumeration

The witness search from Section 5.2 need only be extended slightly to instantiate during the rewriting of edge sets. We write $f(\vec{\iota})$ as a points-to edge with instance variables from $\vec{\iota}$ (and analogously for edge sets), and we write $[\vec{\alpha}/\vec{\iota}]F$ for the substitution of $\vec{\alpha}$ for $\vec{\iota}$ in F.

Def. 6.2 (Rewriting Edge Sets with Instantiation)**.** *Let F_1 be a non-empty set. We write*

$$F_1 \overset{r}{\rightharpoonup} F_2$$

for a rule $r\colon f(\vec{\iota}) \xleftarrow{a} F'(\vec{\iota}, \vec{\iota}')$ such that $f(\vec{\alpha}) \in F_1$ and

$$F_2 = (F_1 - \{f\}) \cup [\vec{\alpha}, \vec{\beta}/\vec{\iota}, \vec{\iota}']F' \,.$$

The new instance variables $\vec{\beta}$ introduced by r may be chosen to existing or fresh instances.

Algorithmically, we introduce new instances only as needed to avoid conflict. Conceptually, we search for witnesses with successively more dynamically allocated objects.

6.3 Extending Flow-Insensitive Refutation

Recall that the job of a refuter is to limit the witness search to feasible executions. Complete precision is obtained when it can guarantee that the search always follows the concrete semantics. In the case of dynamic memory, the precise refuter R_{dyn} must also detect conflict sets (Def. 5.3) like R_{fin} and additionally capture the behavior of memory allocation.

The defining property of `malloc()` that R_{dyn} must capture is that each execution of `malloc()` creates a new object. Thus, R_{dyn} should detect a conflict during witness search if an instance α is supposed to exist before its creation.

Def. 6.3 (Malloc-Existence Conflict)**.** *An instance α is said to be* post-allocated *in a rewrite step of the form*

$$F \cup \{v \mapsto s[\alpha]\} \xleftarrow{*^m p:=\texttt{malloc}()} F$$

for some edge set F, vertex v, summary s, variable p, and integer m. Let $F_0 \rightharpoonup \cdots \rightharpoonup F$ be a partial witness search sequence where $\vec{\alpha}$ are instances post-allocated up to F. Edge set F is said to be in malloc-existence conflict *if any of the instances in $\vec{\alpha}$ appear in F.*

For example, the following rewrite step results in a malloc-existence conflict:

$$\{p \mapsto s_1[\alpha], s_1[\alpha] \mapsto q\} \xleftarrow{p:=\texttt{malloc()}} \{s_1[\alpha] \mapsto q\}\,.$$

Note that the above definition also rules out instantiations of post-allocated instances.

We now illustrate precise witness-generation with dynamic memory using a variant of the example in Figure 1 of Section 2.

Example 6.2. *Consider the following set of assignments:*

$$\{p := *r,\ r := \&q,\ r := *x,\ x := \texttt{malloc}(),\\ y := x,\ *x := r,\ *x := y\}\,.$$

The points-to graph in Figure 4 is the Andersen's result on this set where g_1 is the summary node representing the `malloc()` *site. The following sequence of rule applications yields a witness that respects R_{dyn} for $p \mapsto q$:*

$$\begin{array}{lll}
\{p \mapsto q\} & \xleftarrow{p:=*r} & \{r \mapsto g_1[\alpha], g_1[\alpha] \mapsto q\}\\
& \xleftarrow{r:=*x} & \{x \mapsto g_1[\beta], g_1[\beta] \mapsto g_1[\alpha], g_1[\alpha] \mapsto q\}\\
& \xleftarrow{*x:=y} & \{x \mapsto g_1[\beta], y \mapsto g_1[\alpha], g_1[\alpha] \mapsto q\}\\
& \xleftarrow{x:=\texttt{malloc()}} & \{y \mapsto g_1[\alpha], g_1[\alpha] \mapsto q\}\\
& \xleftarrow{*x:=r} & \{y \mapsto g_1[\alpha], x \mapsto g_1[\alpha], r \mapsto q\}\\
& \xleftarrow{y:=x} & \{x \mapsto g_1[\alpha], r \mapsto q\}\\
& \xleftarrow{x:=\texttt{malloc()}} & \{r \mapsto q\} \quad \xleftarrow{r:=\&q} \quad \emptyset\,.
\end{array}$$

In particular, the following is the assignment sequence witness:

$$r := \&q; x := \texttt{malloc}(); y := x; *x := r;\\ x := \texttt{malloc}(); *x := y; r := *x; p := *r;\ .$$

The witness produced here is essence the witness shown in Figure 2, except the $x := \&g_1$ and $x := \&g_2$ assignments have been replaced with calls to `malloc()`. In Example 5.5, we demonstrated that not having the second statement $x := \&g_2$ yields no witnesses, and thus, all witnesses for $p \mapsto q$ here must have at least two calls to `malloc()`. Intuitively, we can see this issue in the above witness trace. The second rule application uses a second instance of g_1, namely β; choosing to reuse would result in a conflict set on $g_1[\alpha]$.

In the above trace, we can also see the role of malloc-existence conflict checking. Without this conflict check, at the third rule application, the witness search could have instead chosen to discharge $x \mapsto g_1[\beta]$ with $x := \texttt{malloc}()$ instead of $g_1[\beta] \mapsto g_1[\alpha]$ with $*x := y$. However, such a trace would generate a bogus witness with respect to the semantics of `malloc()`, as this state says that $g_1[\beta]$ exists before its creation.

The precision and soundness argument for R_{dyn} follow the structure of the one for R_{fin}, as instantiation restricts our witness search to exact graphs (as in the finite memory case from Section 5). In particular, we need analogous lemmas to Lemma 5.1 (Head Expansion of Realizability) and Lemma 5.2 (Complete Enumeration), though the notion of conflict now includes malloc-existence conflict (Def. 6.3). With these lemmas, we have the corresponding statement about realizability:

Theorem 6.1 (Realizability)**.** *An edge f is realizable iff there exists a sequence of rewrites*

$$w:\ F_0 \colon \{f\} \xrightarrow{r_1} F_1 \xrightarrow{r_2} \cdots \xrightarrow{r_N} F_N \colon \emptyset\,,$$

such that none of the sets $F_0, \ldots, F_N$ are conflicting.

Note that we can no longer make a statement about termination. An exhaustive witness enumeration with instantiation is, in general, non-terminating, as avoiding conflicts may cause infinite instantiation. A naïve cycle detection where we consider edge sets equivalent modulo instances clearly enforces termination, but this algorithm turns out to be unsound. That is, there are examples, at least with non-Andersen style statements, where witnesses exists but this cycle detection refutation will falsely rule them out. Due to space constraints, we do not discuss the examples here. The question of decidability of precise, flow-insensitive points-to analysis with dynamic memory remains open.

6.4 Pigeon Hole Refuters *and all that*

Thus far, the idea of instantiating summary variables for dynamic memory was presented. However, as noted earlier, summaries have many other uses, such as modeling arrays and structures. In this section, we present ideas for refuting summary variables in the case of arrays using *pigeon hole refuters*.

In particular, we may use summary variables for statically allocated arrays $v[]$ that may point to a multiple locations under the weak-update semantics. However, the number of such locations is bounded by the statically allocated size of the array. This directly yields a refuter that we call the pigeon hole refuter.

Let E be a set of points-to facts involving summary as well as non-summary nodes wherein summary nodes $s_1, \ldots, s_k$ are assumed to denote statically allocated arrays of size $z_1, \ldots, z_k$.

Def. 6.4 (Pigeon Hole Refuter)**.** *Set E is conflicting if some summary node s_i must simultaneously point to more than z_i nodes.*

Pigeon hole refuters can also partially handle structures or unions, incorporating a degree of field sensitivity in an otherwise field-insensitive analysis based on the number of fields present in a structure.

Other kinds of refuters could use the types of the variables to forbid certain points-to relationships. For instance, *type masking* analysis can be performed to ascertain the set of all type-casts (both explicit and implicit) that are possible in the program. As a result, it may be possible to improve the pigeon hole refuter.

Example 6.3. *Consider a structure type s containing one integer pointer field and two string pointer fields with the type mask information that no integer pointer has been cast to a string pointer in the program.*

The pigeon hole refuter simply checks that any summary variable of type s may not point to more than 3 other locations in a given set of points-to fact.

This refuter may be enhanced with type information to ensure that no more than one integer typed location and two string typed locations may be pointed to.

7. Flow-Sensitive Refutation

In this section, we describe the construction of refuters for generating witnesses that respect the control flow of the program. One could similarly construct context- or path-sensitive refuters, showing the versatility of our framework.

We first define a points-to analysis problem with flow constraints and a witness for such a problem.

Def. 7.1 (Flow Sensitive Problem)**.** *A flow sensitive finite memory points-to analysis problem consists of (1) a* control flow graph *$C : \langle L, E_C, \ell_{\text{init}}\rangle$ with control flow locations L, edges between these locations $E_C \subseteq L \times L$ and a marked initial location ℓ_{init}, (2) a set of pointer variables X and (3) labeled assignments $\langle a, s\rangle$, wherein each assignment corresponds to an edge $s \in E_C$.*

It is possible to define two notions of flow sensitive witnesses for an edge e at location ℓ:

Basic Flow Compatibility Sequence of assignments that realize e correspond to some valid control flow path in the graph C from ℓ_{init} to ℓ.

Full Flow Sensitivity Sequence of assignments along a valid control flow path such that none of the assignments along intermediate edges in the path *changes* the value of a variable between its definition and its use in the witness.

Refuters. Refuters for flow sensitivity and flow compatibility can be built on top of the basic *flow insensitive* witness search and refuter described in Section 5.

In order to find a witness for edge e for location ℓ, we start from the initial configuration $\{e\}$ and use the flow insensitive witness enumerator to explore all possible sequence of rule applications. For each search path, we use the following refuters to ensure flow compatibility or flow sensitivity:

Flow Compatibility Refutation The set of rule applications along current enumerated path does not correspond to any valid control flow path leading to location ℓ.

Flow Sensitive Refutation The current path is *flow incompatible* or there is an assignment that interferes with a use-def chain in the witness sequence.

The overall refutation strategy described above does not automatically guarantee termination of the technique or the possibility of a symbolic witness search. Therefore, we reformulate the witness enumeration to keep track of a location for the symbolic encoding.

Efficient Witness Search for Flow Compatibility The refuter for flow compatibility maintains a location ℓ in the CFG with the set of points-to edges considered by the witness enumeration. The witness search is refuted whenever the enumerator applies an assignment that is *flow incompatible* with the current location.

Def. 7.2 (Flow Compatible Assignment)**.** *An assignment (a_i, s_i) is flow compatible with location ℓ iff there is a path from the target of edge s_i to ℓ in the CFG.*

A flow sensitive witness search for the realizability of an edge e at location ℓ starts with the initial configuration $(\ell, \{e\})$ and searches for a *flow compatible* sequence of rule applications to yield the configuration $(\ell_{\mathsf{init}}, \emptyset)$:

Def. 7.3 (Flow Compatible Witness Sequence)**.** *A flow compatible witness sequence is a set of rule applications of the form:*

$$(\ell_1 : \ell,\ E_1 : \{e\}) \xrightarrow{r_1} (\ell_1,\ E_1) \xrightarrow{r_2} \ldots \xrightarrow{r_n} (\ell_n : \ell_{\mathsf{init}},\ E_n : \emptyset)\,,$$

such that $E_i \xrightarrow{r_i} E_{i+1}$ is a valid rule application. Furthermore, we require that the assignment (a_i, s_i) associated with rule r_i, be flow compatible with the location ℓ_i.

Theorem 7.1 (Flow Compatible Witness Search)**.** *The search for a flow compatible witness can be carried out in polynomial space. Further, for each edge e and location ℓ there exists a SAT formula $\varphi_F(k, e, \ell)$ that is satisfiable if there is a flow compatible witness for (ℓ, e) of size k, and a formula $\varphi'_F(k, e, \ell)$ that is unsatisfiable if there is no witness of size k or longer.*

8. Implementation and Evaluation

In this section, we now describe a symbolic implementation of the witness generation and refutation technique for finite memory flow-insensitive analysis. We also evaluate our implementation on some benchmark C programs.

Our implementation uses a C language front end CIL [15] to generate a set of pointer analysis constraints for a given program. The constraint generator is currently field insensitive. Unions, structures and dynamic memory allocation are handled with *summary variables*. Our constraint generator uses a built-in Steensgaard analysis to resolve function pointers. Currently, locals of (potentially) recursive methods are not treated as summaries.

The constraints are analyzed using an implementation of Andersens analysis. Our implementation uses a *semi-naive iteration* strategy to handle changes in the pointer graphs incrementally. Other optimizations such as cycle detection have not been implemented.

Witness Finder: Our witness finder implementation uses a symbolic witness search algorithm based on an encoding to SAT, as described in Section 5.4. The SAT encoding allows parallel application of non-conflicting edge dependency rules to find longer witnesses using a search of smaller depth. Currently, our implementation uses the solver Yices [7].

Results: Table 1 shows the performance of our prototype on some benchmark examples. The symbolic witness search technique described in Section 5.4 was run for all the edges in the program for a fixed depth limit of 20. We note that in all these instances our technique was able to find witnesses within the depth limit for all the pointers in a small amount of time. This indicates that Andersen's analysis is precise for all these examples. However, it must be noted that a large number of summary variables in these examples reduces opportunities for refuters. Specifically, dynamic memory allocation was involved in a large fraction of the witnesses that were examined. Finally, a comparison of the depth of the witnesses and the depth of the SAT encoding suggests that allowing non-conflicting rule applications in parallel pays off in discovering longer witnesses at a shallower depth.

Our attempts to solve larger examples (20-200K) have at present been unsuccessful. These examples generate points to graphs ranging from tens of thousand to hundreds of thousand edges, making our SAT encoding intractable by the solver being used. We are currently investigating implementing ideas along the lines of *bootstrapping*, wherein the witness search may focus on a smaller subset of edges in the points-to graph [12].

9. Related Work

Our work was partially inspired by previous work on the complexity of precise points-to analyses, which only produce points-to results for which a witness exists. Horwitz [11] discussed the precision gap between Andersen's analysis and precise flow-insensitive analysis and proved the NP-hardness of the precise problem. Chakaravarthy [5] gave a polynomial-time algorithm for precise flow-insensitive analysis for programs with well-defined types.

Muth and Debray [14] provide an algorithm for a variant of precise flow-sensitive points-to analysis that can be viewed as producing witnesses by enumerating all possible assignment sequences and storing the exact points-to graph. However, the algorithm is meant to prove a PSPACE-completeness result, and it is not practical. Others have studied the complexity and decidability of precise flow-sensitive and partially-flow-sensitive points-to analysis [13, 16, 18].

Concerning programmer understandability, some recent work has looked at producing reasons for failure in the context of dataflow analysis [25]. The approach is based on a domain-specific language for expressing certain kinds of dataflow facts, which then yield reason-producing analysis. Witnesses target user understanding, but in contrast, they also enable refinement through refutation.

The edge reduction rules derived in our approach are similar, in spirit, to the reduction from pointer analysis problems to graph reachability as proposed by Reps [17]. However, a derivation in

Table 1. Results of witness search over some benchmark C programs. Legend: **#Vars**: # ptr variables, **Summ**: # summary variables, **#Cons**: # of pointer assignments, T_{PA}: Andersen analysis time, **#Edg**: edges in pts to graph, T_R: Time to generate rules, T_{WS}: Total time to generate witnesses, **% wit**: % of edges for which witness found.

Prog.	SLOC	# Vars (Summ.)	# Cons.	T_{PA}	# Edg	T_R	# Rules	T_{WS}	% wit.	SAT depth		Wit. length	
										avg.	max.	avg.	max.
vacation	1200	292 (53)	114	0	59	0	66	.1	100 %	2.2	6	2.5	9
smdb	2450	129 (21)	39	0	11	0	11	0	100 %	1.4	2	1.4	2
at-ftpd	7594	1132 (276)	746	0	235	0	267	0.9	100%	2.8	8	3.8	14
bc-1.06	7800	813 (233)	720	0	453	0	655	107	100 %	7	18	10.1	43
ifconfig	8900	2185 (402)	1024	0	247	0	274	.6	100 %	2.4	10	2.8	14
arp	9980	2379 (431)	1116	0	287	0	317	0.8	100 %	2.7	13	3.1	31
netstat	10750	2210 (403)	1029	0.01	250	0	277	0.8	100%	2.8	13	3.2	19

this CFL for a points-to edge need not always yield a witness. In analogy with Andersen's analysis, the derivation may ignore conflicts in the intermediate configurations. Finding a derivation in a CFL without conflicting intermediate configurations reduces to temporal model checking of push-down systems. This observation, however, does not seem to yield a better complexity bound [4].

Our work employs SAT solvers to perform symbolic search for witnesses to points-to edges. Symbolic pointer analysis using BDDs have been shown to outperform explicit techniques in some cases by promoting better sharing of information [2, 26].

Client-driven, refinement-based analysis has been successfully employed for a number of clients, including typestate checking [6, 8], downcast verification [21], and finding security bugs [9]. However, as discussed in Section 1, existing analyses are often architected with a specific set of possible refinements in mind, making it non-obvious if it is possible to extend the system with new refinements that may be required for different clients. In contrast, we have shown that quite a broad set of refinements are possible with our witness-based approach. Also, previous systems did not attempt to address the issue of explaining aliasing results to a user.

A witness can be viewed as a type of backward program slice-limited to those executions in which a particular points-to relationship holds while ignoring control dependence. As with slices, we foresee witnesses as playing a useful rule in program understanding tools, perhaps in combination with techniques for showing slice subsets like thin slicing [22]. In this regard, our techniques are comparable to abstract notions of slicing such as *semantic slicing* that can expose a set of program paths on which a particular abstract dataflow fact may depend [19]. The witness search algorithm with instantiation draws inspiration from materialization [20], a partial concretization operation, that underlie most shape analyzers.

10. Conclusion

We have demonstrated a framework for precise witness-generating pointer analysis using the combination of a witness search and various forms of refutations. In the future, our goal is to use this framework to prove complexity bounds on the pointer analysis problem. We are also working on optimizations that will enable our approach to be applied to explain the output of bug finding tools.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[2] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *PLDI '03*, 2003.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamski, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Comm. of the ACM*, 53(2), 2010.

[4] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, 1997.

[5] V. T. Chakaravarthy. New results on the computability and complexity of points - to analysis. In *POPL*, 2003.

[6] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, 2004.

[7] B. Dutertre and L. de Moura. The YICES SMT solver. Cf. `http://yices.csl.sri.com/tool-paper.pdf`, last viewed Jan. 2009.

[8] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[9] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1-2), 2005.

[10] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, 2001.

[11] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1), 1997.

[12] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08*, 2008.

[13] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4), 1992. ISSN 1057-4514.

[14] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *POPL*, 2000.

[15] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, 2002.

[16] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5), 1994.

[17] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40, 1998.

[18] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. On the complexity of partially-flow-sensitive alias analysis. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[19] X. Rival. Understanding the origin of alarms in astrée. In *SAS*, volume 3672 of *LNCS*, 2005.

[20] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3), 2002.

[21] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, 2006.

[22] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *PLDI*, 2007.

[23] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[24] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986. ISSN 0098-5589.

[25] D. von Dincklage and A. Diwan. Explaining failures of program analyses. In *PLDI*, 2008.

[26] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04*, 2004.