Flow and Context Sensitive Points-to Analysis using Higher Order Reachability

Pritam M. Gharat  Uday P. Khedker
{pritamg,uday}@cse.iitb.ac.in
Indian Institute of Technology Bombay

Abstract
Computing precise (fully flow-sensitive and context-sensitive) and exhaustive (as against demand driven) points-to information is known to be computationally expensive. Therefore many practical tools approximate the points-to information trading precision for efficiency. This often has adverse impact on compute intensive analyses such as model checking. Past explorations in top-down approaches of fully flow and context sensitive interprocedural points-to analysis have not scaled. We explore the alternative of bottom-up interprocedural approach which constructs summary flow functions for procedures and uses them in the place of calls. This approach has been effectively used for many analyses. However, it is ineffective for flow and context sensitive points-to analysis which requires representing indirect accesses of pointees defined in the callers. This is conventionally handled by using placeholders which explicate the unknown locations resulting in either a large number of placeholders or multiple call-specific summary flow functions for a procedure.

We propose a bounded representation of summary flow functions for may points-to analysis called the higher order reachability graph (HRG). The conventional graph reachability based program analyses relate variables but not their pointees. HRGs relate the (transitively indirect) pointees of a variable with those of another variable in terms of indirection levels. A simple arithmetic on indirection levels discovers direct relationships without explicating the unknown pointees defined in the callers. This obviates the need of placeholders. Since the unknown pointees are left implicit, no information about aliasing patterns in the calling contexts is required and we construct a single summary flow function (HRG) per procedure. HRGs are bounded by the number of variables regardless of the number of statements. They are context independent, and hence suitable for context sensitive interprocedural analysis. Further, they are flow sensitive and enable strong updates within the calling contexts.

Our empirical measurements on SPEC benchmarks show that HRGs are compact and have a high reusability. We have been able to scale fully flow and context sensitive exhaustive points-to analysis to 158 kLoC using HRGs (compared to 35 kLoC reported earlier). Thus HRGs hold a promise of efficiency and scalability of points-to analysis without compromising on precision.

1 Introduction
Points-to analysis discovers information about indirect accesses in a program and its precision influences the precision and scalability of other program analyses significantly. Compute intensive analyses such as model checking are ineffective on programs containing pointers partly because of imprecision of pointer analyses.

In this paper, we focus on exhaustive (as against demand-driven [3, 7, 23]) points-to analysis with full flow and context sensitivity for precision. A flow sensitive analysis respects the control flow and computes separate data flow information at each program point. It provides more precise results but could be inefficient at the interprocedural level. A context sensitive analysis distinguishes between different calling contexts of procedures and restricts the analysis to interprocedurally valid control flow paths (i.e. control flow paths
int a, b, c, d;

01 g()
02 {
03     c = a*b;
04     f(); /* call 1 */
05     a = c*d;
06     f(); /* call 2 */
07 }

08 f()
09 {
10     a = b*c;
11 }

(a.1) Context independent representation of context sensitive summary flow function of procedure \( f \)
\[
\begin{align*}
    f(X) &= X \cdot 011 + 010
\end{align*}
\]

(a.2) Context dependent representation of context sensitive summary flow function of procedure \( f \)
\[
\begin{align*}
    f &= \{100 \mapsto 010, \ 001 \mapsto 011\}
\end{align*}
\]

(b) Context insensitive data flow information as a procedure summary of procedure \( f \)
\[
\begin{align*}
    f &= 010
\end{align*}
\]

Figure 1: Illustrating different kinds of procedure summaries for available expressions analysis. The set \( \{a*b, b*c, c*d\} \) is represented by the bit vector 111.

in which every return from a procedure is matched with a call to the procedure such that all call-return matchings are properly nested.

The top-down approach to context sensitive analysis propagates the information from callers to callees [29] effectively traversing the call graph top down. In the process, it analyzes a procedure each time a new data flow value reaches a procedure from some call. Several popular approaches fall in this category: call strings method [22], its value-based variants [10, 18] and the tabulation based functional method [19, 22]. By contrast, the bottom-up approaches [4, 6, 13, 20, 22, 24–29] avoid analyzing callees multiple times by constructing summary flow functions which are used in the calling contexts to incorporate the effect of procedure calls. Effectively, it traverses the call graph bottom up.

It is prudent to distinguish between three kinds of summaries of a procedure that can be created for minimizing the number of times it is re-analyzed:

(a.1) a bottom-up parameterized summary flow function which is context independent (context dependence is captured in the parameters),

(a.2) a top down enumeration of summary flow function in the form of input-output pairs for the input values reaching a procedure, and

(b) a bottom-up parameterless (and hence context insensitive) summary information.

Example 1. Figure 1 illustrates the three different kinds of summaries for available expressions analysis. Procedure \( f \) kills the availability of expression \( a*b \), generates the availability of \( b*c \), and is transparent to the availability of \( c*d \).

- Summary (a.1) is a parameterized flow function, summary (a.2) is an enumerated flow function, whereas summary (b) is a data flow value representing the effect of all calls of \( f \).

- Summaries (a.1) and (a.2) are context sensitive (because they compute distinct values for different calling contexts of \( f \)) whereas summary (b) is context insensitive (because it represents the same value regardless of the calling context of \( f \)).
Figure 2: Points-to analysis flow functions for basic pointer assignments.

- Summaries (a.1) and (b) are context independent (because they can be constructed without requiring any information from the calling contexts of \( f \)) whereas summary (a.2) is context dependent (because it requires information from the calling contexts of \( f \)).

\[ \sum_{\text{context independent}} \]

Note that context independence (in (a.1) above), achieves context sensitivity through parameterization and should not be confused with context insensitivity (in (b) above).

We focus on summaries of the first kind because we would like to avoid re-analysis and seek context sensitivity. We formulate our analysis on a language modelled on C. Section 2 describes the issues in constructing bottom up summary flow functions for points-to analysis and our contributions. Section 3 describes the concept of higher order reachability and introduces edge composition as the most important operation on HRG. Construction of HRG along with its semantics for points-to analysis is described in Sections 4 and 5 respectively. Points-to information computation using HRGs is presented in Section 7. Section 8 presents soundness proofs for the analysis. Section 9 describes the handling of advanced features of the language such as function pointers, structures, unions, heap, arrays and pointer arithmetic. Section 10 describes the related work. Section 11 presents the empirical measurements. Section 12 concludes the paper.

2 Motivation, Key Ideas, and Contributions

This section highlights the issues in constructing bottom up summary flow functions for points-to analysis and describes our contributions by showing how our representation of summary flow functions for points-to analysis overcomes the limitations of the past approaches.

2.1 Issues in Constructing Summary Flow Functions for Points-to Analysis

In this section, we highlight the difficulties faced to construct summary flow functions for points-to analysis followed by a brief overview of the past approaches along with their limitations.

2.1.1 Constructing Bottom-Up Summary Flow Functions

Construction of bottom-up parameterized summary flow functions requires
• reducing the compositions of statement level flow functions to summarize the effect of statements appearing in a control flow path, and

• merging reduced flow functions to combine the effect of multiple control flow paths reaching a join point in the control flow graph.

An important requirement of such a summary flow function is that it should be compact and that its size should be independent of the size of the procedure it represents. This seems hard because the flow functions need to handle indirect pointees of variables. When these pointees are defined in the calling procedures, their information is not available in a bottom-up construction; information reaching a procedure from its callees is available during bottom-up construction but not the information reaching from its callers. The presence of function pointers passed as parameters pose an additional challenge for bottom-up construction for a similar reason.

2.1.2 Modelling Access of Unknown Pointees

The main difficulty in reducing meets (i.e. merges) and compositions of points-to analysis flow functions is modelling the accesses of pointees when they are not known. For the statement sequence $x = \ast y; \ z = \ast x$ if the pointee information of $y$ is not available, it is difficult to describe the effect of these statements on points-to relations symbolically. A common solution for this is to use placeholders for indirect accesses. We motivate this need below and show its limitations.

Let $V$ and $P \subseteq V$ denote the sets of variables and pointers in a program. Then, the points-to information is subset of $PTG = P \times V$. For a given statement, a flow function for pointer analysis computes points-to information after the statement by incorporating its effect on the pointer information that holds before the statement. It has the form: $f : 2^{PTG} \rightarrow 2^{PTG}$. Figure 2 enumerates the space of flow functions for points-to analysis. The flow functions are named in terms of the variables appearing in the assignment statement and are parameterized on the input pointer information $X$ which may depend on the calling context. This is described in terms of placeholders in $X$ denoted by $\phi_1$ and $\phi_2$ in the Figure which are placeholders for the information in $X$. It is easy to see that the function space $F = \{ ad, cp, st, ld \}$ is not closed under composition.

Example 2. Let $f$ represent the composition of flow functions for the statement sequence $x = \ast y; \ z = \ast x$. Then

$$f(X) = ld_{zx}(ld_{xy}(X)) = (X - \{(x, l_1) \mid (x, l_1) \in L\} \cup \{(z, l_1) \mid (z, l_1) \in L\} )$$
$$\cup \{(x, \phi_2) \mid \{(y, \phi_1), (\phi_1, \phi_2) \} \subseteq X\}$$
$$\cup \{(z, \phi_3) \mid \{(y, \phi_1), (\phi_1, \phi_2), (\phi_2, \phi_3) \} \subseteq X\}$$

This has three placeholders and cannot be reduced to any of the four flow functions in the set. □

The use of placeholders prohibits compact representation of summary flow functions because a separate placeholder may be required for different occurrences of the same variable in different statements.

Example 3. Consider the following code snippet for constructing a summary flow function which would then be applied to the points-to relations before statement $s_1$.

```
s_1 : x = \ast y;
s_2 : z = q;
s_3 : p = \ast y;
```
Consider the possibility of constructing a flow sensitive summary flow function. Assume that we use $\phi_1$ as the placeholder to denote the pointee of $y$ and $\phi_2$ as the placeholder to denote the pointee of pointee of $y$. Clearly, we cannot guarantee that the pointee of $y$ or pointee of pointee of $y$ remains same in $s_1$ and $s_3$ because statement $s_2$ could have a side effect of changing either on them depending upon the aliases present in the calling context. Assuming that $\phi_3$ is the placeholder for $q$, if $*z$ is aliased to $y$ before statement $s_1$ then the placeholder for pointee of $y$ in $s_3$ will coincide with $\phi_3$ otherwise it will coincide with $\phi_1$. Similarly, if $z$ is aliased to $y$ before statement $s_1$ then the placeholder for pointee of pointee of $y$ in $s_3$ will coincide with $\phi_3$ otherwise it will coincide with $\phi_2$. Thus the decision to reuse the placeholder for a flow sensitive summary flow function is not easy.

This difficulty can be overcome by avoiding the kill due to $s_2$ and using $\phi_1$ for pointee of $y$ and $\phi_2$ for pointee of pointee of $y$ in both $s_1$ and $s_3$. If $z$ is aliased to $y$ or $*z$ is aliased to $y$ before statement $s_1$ then both $x$ and $p$ will point to both $\phi_2$ and $\phi_3$ which is imprecise. Effectively, the summary flow function becomes a flow insensitive.

Thus, introducing placeholders for the unknown pointees is not sufficient but the knowledge of aliases in the calling context is also equally important for introducing the placeholders. □

2.1.3 An Overview of Past Approaches

In this section, we explain two approaches that construct the summary flow functions for points-to analysis. Other related investigations have been reviewed in Section 10; the description in this section serves as a background to our contributions.

- **Using aliasing patterns to construct a collection of partial transfer functions (PTFs).**

  In this approach, a different summary flow function is constructed for every combination of aliases found in the calling contexts to decide the placeholders for representing the unknown pointees. This requires creation of multiple versions of a summary flow function which is represented by a collection of partial transfer functions (PTFs). A PTF is constructed for every possible combination of aliasing patterns that could occur for a given list of parameters and global variables accessed in a procedure [26].

  **Example 4.** For procedure $g$ of the program in Figure 3, three placeholders $\phi_1$, $\phi_2$, and $\phi_3$ have been used in the PTFs shown in Figures 4(a) and (b). The possibility that $x$ and $y$ may or may not be aliased
gives rise to two PTFs. □

The main limitation of this approach is that the number of PTFs could increase combinatorially with the number of dereferences of globals and parameters.

**Example 5.** For four dereferences, we need 15 PTFs. Consider four pointers \(a, b, c, d\). Either none of them is aliased (1 possibility); only two of them are aliased: \((a, b), (a, c), (a, d), (b, c), (b, d)\) or \((c, d)\) (6 possibilities); only three of them are aliased: \((a, b, c), (a, b, d), (a, c, d), (b, c, d)\) (total 4 possibilities); all four of them are aliased: \((a, b, c, d)\) (1 possibility); groups of aliases of two each: \(\{(a, b), (c, d)\}, \{(a, c), (b, d)\}, \{(a, d), (b, c)\}\) (3 possibilities). Thus the total number of PTFs is \(1 + 6 + 4 + 1 + 3 = 15\). □

PTFs that do not correspond to actual aliasing patterns occurring in a program are irrelevant. A hybrid approach \cite{29} excludes such PTFs by combining a top-down analysis for discovering aliasing patterns in a program with a bottom-up analysis to construct the corresponding PTFs. Yet, the number of PTFs could remain large.

Although this approach does not introduce any imprecision, our measurements show that the number of aliasing patterns occurring in practical programs is very large which limits the usefulness of this approach.

- **Single summary flow function without using aliasing patterns.**

This approach does not make any assumption about aliases in the calling context and constructs a single summary flow function for a procedure. For our example, this approach uses a new temporary \(\phi_3\) for pointee of \(y\) and also \(\phi_4\) for pointee of pointee of \(y\) in \(s_3\). In a degenerate case, the size of flow functions may be proportional to the number of statements represented by the summary flow function. This is undesirable because it may be better not to create summary flow functions and retain the original statements whose flow functions are applied one after the other.

Separate placeholders for different occurrences of a variable can be avoided if points-to information is not killed by the summary flow functions \cite{13,24,25}. Another alternative is to use flow insensitive summary flow functions \cite{4}. However, this introduces imprecision in both the cases.

A fundamental problem with placeholders is that they explicate unknown pointees by naming them, resulting in either a large number of placeholders or multiple summary flow functions for different aliasing patterns that exist in the calling contexts.
2.2 Our Key Ideas and Contributions

We overcome the difficulties outlined in Section 2.1 by representing the summary flow function of a procedure in the form of a graph called Higher order Reachability Graph (HRG) and use it for for flow and context sensitive points-to analysis.

Example 6. The HRG for our motivating example are illustrated in Figure 4(c). The edge labels (1,0) indicate that the source of the edge is assigned (indicated by indirection level 1) the address of the target of the edge (indicated by indirection level 0). Thus the assignment on line number 08 of the motivating example is represented by the edge \( a \xrightarrow{1.0} e \) in the HRG. The indirection level 2 in edge \( x \xrightarrow{2.1} z \) for line 10 indicates that the pointees of \( x \) are being defined; since \( z \) is read, its indirection level is 1. The indirection level on the edge eliminates the need of placeholders capturing a precise relationship between \( x \) and \( z \).

The assignment on line number 17 is represented by two edges in the HRG: \( y \xrightarrow{2.0} d \) and \( b \xrightarrow{1.0} d \). This is because \( y \) points to \( b \) along one path (line number 13) and hence flow function composition (line numbers 13 and 17) results in the edge \( b \xrightarrow{1.0} d \). However there is no information about the pointee of \( y \) along the other path and hence we also have the edge \( y \xrightarrow{2.0} d \). □

Nodes in an HRG represent variables (including pointer variables) and edges track indirection levels. Composition of edges using simple arithmetic resolves indirection levels yielding more direct relationships eventually leading to points-to edges with indirection level (1,0).

We perform edge composition without making any approximations, construct HRGs flow sensitively, and remember the order of edges in HRGs, thereby eliminating the imprecision caused by [13, 24, 25].

HRGs leave pointees whose information is not available during summary construction implicit. Hence:

(a) We do not need placeholders (unlike [13, 24, 26, 29]). This is possible because we encode indirection levels as edge labels by replacing a sequence of indirection operators “∗” by a number.

(b) We do not require any assumptions/information about aliasing patterns in the calling contexts (unlike [26, 29]) and construct a single summary flow function per procedure (unlike [26, 29]) without introducing the imprecision introduced by [13, 24, 25].

(c) The size of our summary flow function for a procedure does not depend on the number of statements in the procedure and is bounded by the number of global variables, formal parameters of the procedure, and its return value variable (unlike [13, 24, 25]).

(d) updates can be performed in the calling contexts (unlike [4, 13, 24, 25]).

This facilitates the scalability of fully flow and context sensitive exhaustive points-to analysis.

2.3 Our Language and Scope

We have described our formulations for a language modelled on C and have organized the paper based on the features included in the language. For simplicity of exposition, we describe our analysis at three levels designed to handle the different features of our language based on the treatment of indirect accesses of pointees defined in the callers of a procedure.

\[ This is somewhat similar to choosing a decimal representation for integers over Peano’s representation or replacing a unary language by a binary or n-ary language. ]
The effect of pointer assignment on concrete memory \( \mathcal{M} \) (i.e., memory created by a single control flow path) for reaching the shared node from \( x \) and \( y \):

| Pointer assignment | Memory Graph | Constraint | Edge traversals | Higher order path in \( \mathcal{M} \) |
|--------------------|--------------|------------|-----------------|-------------------------------|
| \( x = ky \)       | \( x \rightarrow \mathcal{M} \) | \( \mathcal{M}^1 \{x\} = \mathcal{M}^0 \{y\} \) | 1 edge, 0 edge | \( x \sim y \) |
| \( x = y \)        | \( x \rightarrow \mathcal{M} \) | \( \mathcal{M}^1 \{x\} = \mathcal{M}^1 \{y\} \) | 1 edge, 1 edge | \( x \sim y \) |
| \( x = \ast y \)   | \( x \rightarrow \mathcal{M} \) | \( \mathcal{M}^2 \{x\} = \mathcal{M}^2 \{y\} \) | 1 edge, 2 edges | \( x \sim y \) |
| \( \ast x = y \)   | \( x \rightarrow \mathcal{M} \) | \( \mathcal{M}^2 \{x\} = \mathcal{M}^1 \{y\} \) | 2 edges, 1 edge | \( x \sim y \) |

Figure 5: Higher order paths in concrete memory for basic pointer assignments. A double circle indicates a shared location while a thick arrow shows the newly created edge in the memory.

### Feature Level 1 | Level 2 | Level 3
---|---|---
Pointers to scalars | ✓ | |
Function Calls and Recursion | ✓ | |
Function Pointers | | ✓
Pointers to Structures, Unions, and Heap | | ✓
Pointer Arithmetic, Pointers to Arrays, Address Escaping Locals | | ✓

In the first three cases, the information flows from top to bottom of the call graph (caller to callee) and hence are naturally handled by the top-down approaches of interprocedural analysis. However, a special attention is required for representing this information in the bottom-up approaches. In case of recursion, the presence of cycle in the call graph requires a fixed point computation regardless of the approach used.

Levels 1 and 2 handle the core features of the language whereas level 3 handles the advanced features.

Our analysis for level 1 handles the access of pointers to scalars within a procedure and is described in Section 4. We first present our analysis for memory created along a single control flow path (called the concrete memory) and then extend it to incorporate the effect of all the control flow paths by modelling an abstract memory at a program point as an approximation of a collection of concrete memories along individual paths reaching the program point. Level 2 extends our analysis to the interprocedural level which is presented in Section 5. This section also shows how we handle recursion. We then extend our analysis to level 3 to handle function pointers, structure and heap in Section 9.

## 3 Higher Order Reachability Graph

This section defines higher order reachability graph (HRG) and edge composition for constructing it. HRG represents memory manipulations without needing placeholders for unknown pointees.

### 3.1 Basic Concepts and Notations

Initially we assume scalars and pointers in the stack and static memory; Section 9.2 presents extensions for structures and heap.

We assume a control flow graph (CFG) representation consisting of three address code statements. Program points \( u, v, t \) represent the points just before the execution of statements. The successors and predecessors of a program point \( u \) in the CFG are denoted by \( gsucc(u) \) and \( gpred(u) \) respectively. A control...
flow path $\pi$ is a sequence of program points $q_0, q_1, \ldots, q_m$, such that $q_0$ is the start of the program and $q_{i+1} \in \text{gsucc}(q_i)$. When we talk about a particular control flow path $\pi$, we use $\text{psucc}$ to denote successors along $\pi$. Thus, $q_{i+1} = \text{psucc}(\pi, q_i)$ and $q_j \in \text{psucc}^+(\pi, q_i), j > i$. In presence of cycles, program points could repeat; however, we do not explicate their distinct occurrences for notational convenience; the context is sufficient to make the distinction.

The concrete memory at a program point along a control flow path is an association between variables and their values and is represented by a function $M : V \rightarrow (V \cup C \cup \{?\})$ where “?” denotes an undefined value. For static analysis, when the effects of multiple control flow paths reaching a program point are incorporated in the memory, the resulting memory is a relation $M \subseteq V \times (V \cup C \cup \{?\})$. We call it an abstract memory because, in general, it is an over-approximation of the union of concrete memories along all paths reaching the program point. Given these two versions of $M$, it is convenient to view $M$ as a graph in which nodes from the set $V \cup C \cup \{?\}$. An edge $x \rightarrow y$ indicates that $x \in V$ contains the value $y \in C$ or $x \in P$ contains the address of $y \in V$. When concrete and abstract memories need to be distinguished, we denote them by $\mathcal{M}$ and $\mathcal{\overline{M}}$, respectively; otherwise we generically use $M[\pi]$; $\mathcal{M}_u, \pi$ denotes the memory associated with a particular occurrence of $u$ in a given $\pi$ whereas $\mathcal{\overline{M}}_u$ denotes the memory associated with all occurrences of $u$ in all possible $\pi$s.

Example 7. Assuming that the execution of the program in Figure 3 begins with procedure $f$, $\mathcal{M}$ after line 3 with its domain restricted to pointers, is $\{(a,?), (b,?), (x, a), (y,?), (z, w)\}; \mathcal{\overline{M}}$ is also the same. \(\square\)

The values or pointees of a set of variables $X \subseteq V$ in memory $M$ are computed by the application of $M$ to $X$:

$$M X = \{y \mid (x, y) \in M, x \in X\} \quad (1)$$

For the memory after line 3 in our example $M\{x, y, z\} = \{a,?, w\}$, $(M \circ M) \{x\}$ discovers the pointees of pointees of $x$. For composability of $M$, we extend its domain to $V \cup C \cup \{?\}$ by inclusion map because $V \subseteq V \cup C \cup \{?\}$. A composition of degree $i$, $M^i\{x\}$ discovers $i^{th}$ pointees of $x$ which involves $i$ transitive reads from $x$: first $i - 1$ addresses are read followed by the value in the last address. By definition, $M^0\{x\} = \{x\}$.

---

This increases the amount of notation but our explanations become simpler if we talk about the three separately: concrete memory, abstract memory, and generic memory when the distinction between concrete and abstract memory is not required.
For $\nu \in \text{psucc}(\pi, u)$, flow function $\delta(u, \nu)$ represents the effect of the statement between them. $\Delta(\pi, u, \nu)$ denotes the summary flow function representing the effect of a sequence of statements on $\pi$ between $u$ and $\nu$ (Definition $1$).

| Definition 1: Summary flow functions |
|--------------------------------------|
| **Concrete Memory and Summary Flow Function** |
| $\Xi(\pi, u, \nu) := \begin{cases} \delta(u, \nu) & v = \text{psucc}(\pi, u) \\ \delta(t, v) \circ \Xi(\pi, u, t) & t \in \text{psucc}^+(\pi, u), v = \text{psucc}(\pi, t) \end{cases}$ |
| $\Xi_{\nu, \pi} := (\Xi(\pi, u, \nu)) (\Xi_{u, \pi})$ |
| **Abstract Memory and Summary Flow Function** |
| $\Xi(u, \nu) := \begin{cases} \delta(u, \nu) & v \in \text{gsucc}(u) \\ \bigcup \delta(t, v) \circ \Xi(u, t) & t \in \text{gsucc}^+(u), v \in \text{gsucc}(t) \end{cases}$ |
| $\Xi_{\nu} := (\Xi(u, \nu)) (\Xi_{u})$ |

For concrete and abstract memories, $\Delta$ is denoted by $\Xi$ and $\Xi$; when this distinction is not required, we generically use $\Delta\Xi$.

Finally, $g \circ f(\cdot) = g(f(\cdot))$.

### 3.2 Higher Order Reachability for Points-to Analysis

Figure $5$ shows the effect of basic C-style pointer assignments in a concrete memory. It is easy to visualize their effect without knowing the pointees of $x$ and $y$ as explained below:

- For assignment $x = &y$, pointer $x$ is defined to point to $y$. After the assignment, traversing one edge from $x$ in the memory graph leads to the location reached by traversing zero edges from $y$ (i.e. $\mathcal{M}^1\{x\} = \mathcal{M}^0\{y\}$).

- For assignment $x = y$, pointer $x$ is defined to point to the pointee of $y$. Traversing one edge from both $x$ and $y$ in the memory graph leads to the same location (i.e. $\mathcal{M}^1\{x\} = \mathcal{M}^1\{y\}$).

- For assignment $x = \ast y$, pointer $x$ is defined to point to the pointee-of-pointee of $y$. Traversing one edge from $x$ and two from $y$ in the memory graph leads to the same location (i.e. $\mathcal{M}^1\{x\} = \mathcal{M}^2\{y\}$).

- For assignment $\ast x = y$, pointee of $x$ is defined to point to the pointee of $y$. Traversing two edges from $x$ and one edge from $y$ in the memory graph leads to the same location (i.e. $\mathcal{M}^2\{x\} = \mathcal{M}^1\{y\}$).

We generalize the observations made in the figure to define the concept of a higher order reachability graph. The generic view of higher order reachability graph is shown in Figure $6$.

| Definition 2: Higher order Reachability Graph (HRG) |
|------------------------------------------------------|
| Let the execution of a pointer assignment $s$ create a points-to edge $w \rightarrow z$ in a memory graph $M$. If $w \in M^i\{x\}$ and $z \in M^j\{y\}$, $s$ creates a path of order $i$ at $x$ from $y$, denoted $x \downarrow^i j \rightarrow y$, in $M$. It is represented by an edge $x \downarrow^i j \rightarrow y$ where $(i, j)$ is the indirection level (indlev) of the edge ($i$ is the indlev of $x$, and $j$ that of $y$). An HRG is an ordered set of such edges. |

The order $i$ of a higher order path $x \downarrow^i j \rightarrow y$ (or the indlev $(i, j)$) of edge $x \downarrow^i j \rightarrow y$ indicates that $\mathcal{M}^i\{x\} = \mathcal{M}^j\{y\}$ in a concrete memory, and $\mathcal{M}^i\{x\} \supseteq \mathcal{M}^j\{y\}$ in an abstract memory. Thus, traversing $j$ edges from $y$ leads to locations reachable from $x$ by traversing $i$ edges. This is because the $(i-1)^{th}$ pointees of $x$ are defined to

\*The reasons for using distinct notations are similar to those explained in footnote 4

10
The path order eliminates the need of placeholders and a simple arithmetic enables composition of paths summarizing the effect of multiple statements concisely. We call it a “higher order” path because it does not coincide with a directed path in $M$ unless $i = 1$ and $j = 0$. For $i = j = 1$, it represents the classical graph reachability used in program analyses [19] which relates variables but not their pointees.

For a pointer assignment between consecutive program points $u$ and $v$, the HRG for flow function $\delta(u, v)$ has a single edge and is same for both $M$ and $M$. These edges are $x \xrightarrow{1.0} y$, $x \xrightarrow{1.1} y$, $x \xrightarrow{1.2} y$, and $x \xrightarrow{2.1} y$ (see Figure (5) for $\overline{M}$). For $\overline{M}$, $x$ and $y$ may have multiple pointees and the memory graph may contain multiple shared locations.

We may create an HRG to (a) represent higher order paths in memory $M_u$, or (b) transform $M_u$ into $M_v$ to represent the effect of control flow paths from $u$ to $v$. This brings out an important insight: When used for (a), an HRG is an abstraction of memory; When used for (b), it is an abstraction of a memory transformer. This is analogous to a matrix which can be seen both as an absolute value, and also as a transformer (for a linear translation in space).

The rest of the paper builds on this theme using the concepts listed in Figure 7 where each layer is defined in terms of the layers below it. They are developed for a single execution path (memory $\overline{M}$, HRG $\overline{\Delta}$) and then lifted to multiple paths (memory $\overline{M}$, HRG $\overline{\Delta}$). They are grouped according to the two phases of our analysis: (a) the construction of HRGs, and (b) the use of HRGs for computing points-to information.

3.3 Defining Edge Composition

This section introduces the concept of edge composition. Operation for creating new edges for possible inclusion in HRGs. This incorporates the effect of a statement level flow function $\delta$ into a summary flow function $\Delta$.

Let $\delta$ be represented by an edge $\mathbf{n}$ (“new” edge) and consider edge $\mathbf{p} \in \Delta$ (“processed” edge). Edges $\mathbf{n}$ and $\mathbf{p}$ can be composed (denoted $\mathbf{n} \circ \mathbf{p}$) provided they have a common node called the pivot of composition. The goal is to reduce (i.e. simplify) $\mathbf{n}$ by using the information from $\mathbf{p}$. This is achieved by eliminating the pivot and joining the remaining two nodes resulting in a reduced edge $\mathbf{r}$. This requires the indlevs of the pivot in both the edges to be made same. For example, given edges $\mathbf{n} \equiv z \xrightarrow{i,j} x$ and $\mathbf{p} \equiv x \xrightarrow{k,l} y$ with a pivot $x$, if $j > k$, then the difference $j - k$ is added to $\mathbf{p}$ allowing it to be viewed as $x \xrightarrow{j,(l+j-k)} y$. This
balances the indlevs of $x$ in both the edges and creates a reduced edge $r \equiv z^1_{i,(l+j-k)} y$. Although this computes the transitive effect of edges, it cannot be modelled using matrix multiplication as explained in Section 3.5.1.

**Example 8.** In the first example in Figure 8, the indlevs of pivot $x$ in both $p$ and $n$ is same allowing us to join $z$ and $y$ through an edge $z^1_{0} y$. In the second example, the difference $(2 - 1)$ in the indlevs of $x$ is added to the nodes in $p$ viewing it as $x^2_{1} y$. This allows us to join $y$ and $z$ creating the edge $y^1_{1} z$. □

**Example 9.** Figure 9 exhaustively illustrates SS and TS edge compositions where $x$ is the pivot; these possibilities are explained in details in the rest of this section. In Ex. ts3, the indlev of pivot $x$ is already balanced. In Ex. ss2, balancing requires adding the difference in the indlev of $x$ $(2 - 1)$ to the indlev of $z$. In Ex. ss1, edge $n$ is already in a reduced form. We cannot reduce edge $p$ (resolve the dereference of the pivot $x$) by using information from edge $n$ (points-to information of $x$) owing to flow sensitivity. □

Observe that a simple arithmetic on the indirection levels obviates the need of placeholders.

$n$ and $p$ could be reduced edges representing the effect of multiple statements. For flow sensitivity, the statements represented by $n$ appear after the statements represented by $p$ in the control flow path for which $\Delta$ is constructed. Thus edge composition is not commutative although it is associative.

**Lemma 1.** Edge composition is associative.

\[(e_1 \circ e_2) \circ e_3 = e_1 \circ (e_2 \circ e_3)\]

**Proof.** Edge composition computes indlevs using arithmetic expressions involving binary plus (+) and binary minus (−). They can be made to associate by replacing binary minus (−) with binary plus (+) and unary minus (−), eg. $a + b + (-c)$ instead of $a + b - c$. □

Consider an edge composition $r = n \circ p$, $p \in \Delta$. Accumulating both $r$ and $n$ with $p$ in $\Delta$ is sound but may lead to imprecision; accumulating only $n$ with $p$ in $\Delta$ (and discarding $r$) is also sound but may lead to inefficiency. Since our goal is to include reduced edges and keep $\Delta$ small, an edge composition is desirable if and only if it is valid, useful, and conclusive. We explain these properties in Sections 3.4 and 3.5.

(a) A composition $n \circ p$ is valid only if it preserves flow sensitivity i.e., the statement(s) representing $n$ follow the statement(s) representing $p$ on some control flow path.

(b) A composition $n \circ p$ is useful only if it takes HRG closer to a points-to graph.

(c) A composition $n \circ p$ is conclusive only when the information supplied by $p$ used for reducing $n$ is not likely to be updated by intervening statements.
For compactness of HRG, we would like to include either \( r \) or \( n \) but not both. We include \( r \) only when the edge composition is desirable, otherwise we include \( n \). We ensure desirability by traversing a procedure along the control flow and examining different kinds of compositions based on the role of the pivot in edge composition.

### 3.4 Valid and Useful Edge Compositions

Let an edge \( n \) be represented by the triple \((S_n, (s_n^C, τ_n^C), T_n)\) where \( S_n \) and \( T_n \) are the source and target of the edge and the \textit{indlev} of the edge is \( s_n^C \cdot τ_n^C \) (\( C \) represents the count of indirection levels). Similarly, \( p \) is represented by the triple \((S_p, (s_p^C, τ_p^C), T_p)\) and the reduced edge \( r \) resulting from the composition \( n \circ p \) is represented by \((S_r, (s_r^C, τ_r^C), T_r)\). The \textit{indlev} \( s_r^C \cdot τ_r^C \) is obtained by balancing the \textit{indlev} of the pivot in edges \( p \) and \( n \).

Let the weight of an edge be defined as the sum of the \textit{indlev}s of the source and the target. The weight of a points-to edge \( x \xrightarrow{1.0} y \) is 1 which is the minimum weight that any edge can have. Every HRG \( Δ \) corresponds to a unique points-to graph which can be viewed as an HRG in which every edge has weight 1; this points-to graph is called the \textit{canonical} form of \( Δ \). An edge composition in \( Δ \) is useful only if it takes \( Δ \) closer to its canonical form. This requires the \textit{indlev} \((s_r^C, τ_r^C)\) of the reduced edge \( r \) must honour the following usefulness criterion

\[
s_r^C \leq s_n^C \land τ_r^C \leq τ_n^C
\]

Intuitively, this ensures that the \textit{indlev} of the new source and the new target does not exceed the corresponding \textit{indlev}s in the original edge \( n \). This reduces the average weight of the HRG and takes it closer to the points-to graph. Section 3.4.1 applies this criterion to each kind of composition described below and derives composition specific criteria.
### Possible ST Compositions

| Statement sequence | Memory graph | HRG edges |
|--------------------|--------------|-----------|
| $s^c_n < t^c_P$   | $p: y \xrightarrow{1.2} x$ | $n: x \xrightarrow{1.0} z$ |

- **Ex. st1**
  - $y = *x$
  - $x = &z$
  - $s^c_n < t^c_P$ (Additionally $s^c_p \leq t^c_P$)

| Statement sequence | Memory graph | HRG edges |
|--------------------|--------------|-----------|
| $t^c_n < t^c_P$   | $p: y \xrightarrow{1.2} x$ | $n: z \xrightarrow{1.1} x$ |

- **Ex. tt1**
  - $y = *x$
  - $z = x$
  - $t^c_n < t^c_P$ (Additionally $s^c_p \leq t^c_P$)

### Possible TT Compositions

| Statement sequence | Memory graph | HRG edges |
|--------------------|--------------|-----------|
| $s^c_n = t^c_P$   | $p: y \xrightarrow{1.2} x$ | $n: x \xrightarrow{2.0} z$ |

- **Ex. st2**
  - $y = x$
  - $x = &z$
  - $s^c_n = t^c_P$

| Statement sequence | Memory graph | HRG edges |
|--------------------|--------------|-----------|
| $t^c_n = t^c_P$   | $p: y \xrightarrow{1.1} x$ | $n: z \xrightarrow{1.0} y$ |

- **Ex. tt2**
  - $y = x$
  - $z = *x$
  - $t^c_n = t^c_P$ (Additionally $s^c_p \leq t^c_P$)

| Statement sequence | Memory graph | HRG edges |
|--------------------|--------------|-----------|
| $s^c_n > t^c_P$   | $p: y \xrightarrow{1.2} x$ | $n: x \xrightarrow{2.0} z$ |

- **Ex. st3**
  - $y = *x$
  - $x = &z$
  - $s^c_n > t^c_P$ (Additionally $s^c_p \leq t^c_P$)

| Statement sequence | Memory graph | HRG edges |
|--------------------|--------------|-----------|
| $t^c_n > t^c_P$   | $p: y \xrightarrow{1.1} x$ | $n: z \xrightarrow{1.0} y$ |

- **Ex. tt3**
  - $y = x$
  - $z = x$
  - $t^c_n > t^c_P$ (Additionally $s^c_p \leq t^c_P$)

---

**Figure 10:** Illustrating all possibilities of ST and TT compositions. See Figure 9 for illustrations of SS and TS compositions. In each case, the pivot of the composition is $x$.

---

We examine the role of the pivot for validity and usefulness. The pivot of a composition, denoted $P$, may be the source or the target variable of $n$ and $p$. This leads to four combinations (SS, TS, ST, TT) of $n \circ p$ as illustrated in Figures 9 and 10. Although we have implemented only TS and SS, we explain the four compositions using the following notation: Let $\ell^P$ denote the $(p^P)^{th}$ pointee of $P$ accessed by the edge $p$ and $\ell^n$ denote the $(p^n)^{th}$ pointee of $P$ accessed by the edge $n$. If there is a path from $\ell^P$ to $\ell^n$ (denoted $\ell^P \rightarrow \ell^n$) in the memory, the information from $p$ can be used to resolve the indirection levels in $n$.

- **SS composition.** In this case, $S_n = S_p$, i.e. the pivot is the source of both $n$ and $p$. It is eliminated and $T_p$ becomes the source and $T_n$ becomes the target of $r$.
  - For $s^c_n < s^c_p$ in Figure 9 (Ex. ss1), edge $p$ updates the pointee of $x$ and edge $n$ redefines $x$. As shown in the memory graph, there is no path between $\ell^P$ and $\ell^n$ and hence $y$ and $z$ are unrelated rendering this composition invalid. Similarly, edge composition is invalid for $s^c_n = s^c_p$ (Ex. ss2).
  - For $s^c_n > s^c_p$, there exists a path from $\ell^P$ to $\ell^n$; hence this composition is valid. For it to be useful, the indlev of the reduced edge $r$ should be smaller than the indlev of $n$. The usefulness criteria (Inequality 2) reduces to $t^c_P < s^c_p < s^c_n$ in this case (as derived in Section 3.4.1). Ex. ss2 in Figure 9 satisfies this constraint and creates a reduced edge $z \xrightarrow{1.0} y$.

- **TS composition.** In this case, $T_n = S_p$, i.e. the pivot is the target of $n$ and source of $p$. It is eliminated and $S_n$ becomes the source and $T_p$ becomes the target of $r$.
  - For $t^c_n < s^c_p$, $\ell_n \rightarrow \ell_p$ holds in the memory graph. However, this composition takes us away from the canonical form. In Ex. ts1, a composition would create an edge $z \xrightarrow{2.0} y$ whose indlev is
higher than that of \( n (z = \frac{1}{1}x) \). Hence, this composition is not useful. Thus, we require \( \ell_p \rightarrow \ell_n \)
to hold in the memory graph and not \( \ell_n \rightarrow \ell_p \) for usefulness of composition. However, this is
necessary but not sufficient for useful compositions. The usefulness criteria (Inequality 2) for \( TS \)
composition reduces to \( \tau_p^c \leq s_p^c \leq \tau_n^c \) as the necessary and sufficient condition (Section 3.4.1).

- For \( \tau_n^c \geq s_p^c \), this composition is valid because \( \ell_p \rightarrow \ell_n \) holds in the memory graph. Ex. \( ts2 \) and
\( ts3 \) in Figure 9 also satisfy the usefulness criteria and create reduced edges \( z \xrightarrow{1,1} y \) and \( z \xrightarrow{1,0} y \).

- \( TT \) composition. In this case, \( T_n = T_p \), i.e. the pivot is the target of both \( n \) and \( p \). It is eliminated and
\( S_n \) becomes the source and \( S_p \) becomes the target of \( r \).

  - For \( \tau_n^c < \tau_p^c \) (Ex. \( tt1 \)), the composition is not useful because \( \ell_p \rightarrow \ell_n \) does not hold in the
memory graph.

  - For \( \tau_n^c \geq \tau_p^c \), this composition is valid because \( \ell_p \rightarrow \ell_n \) holds in the memory graph. The usefulness
constraint (Inequality 2) for \( TT \) composition reduces to \( s_p^c \leq \tau_p^c \leq \tau_n^c \). Ex. \( tt2 \) and \( tt3 \) in Figure 10 satisfy the usefulness criteria to create \( z \xrightarrow{1,2} y \) and \( z \xrightarrow{1,1} y \) edges.

- \( ST \) composition. In this case, \( S_n = T_p \), i.e. the pivot is the source of \( n \) and target of \( p \). It is eliminated
and \( S_p \) becomes the source and \( T_n \) becomes the target of \( r \).

  - For \( s_n^c < \tau_p^c \) (Ex. \( st1 \)), there is no path between \( \ell_p \) and \( \ell_n \) because of the redefinition of \( x \); hence
this composition is invalid. Similarly, for \( s_n^c = \tau_p^c \) (Ex. \( st3 \)), the composition is invalid.

  - For \( s_n^c > \tau_p^c \), this composition is valid because \( \ell_p \rightarrow \ell_n \) holds in the memory graph. The usefulness
criteria (Inequality 2) reduces to \( s_p^c \leq \tau_p^c < s_n^c \) in this case. Ex. \( st2 \) in Figure 10 satisfies this
constraint and creates a reduced edge \( y \xrightarrow{2,0} z \).

Since Figures 9 and 10 covers all possible cases, we conclude that an edge composition is valid only if
there exists a path \( \ell_p \rightarrow \ell_n \) rather than \( \ell_n \rightarrow \ell_p \) between \( \ell_p \) and \( \ell_n \). Intuitively, such a path guarantees that the updates made by \( n \) do not disturb the higher order path represented by \( p \). Hence, the two higher order
paths can be composed by eliminating the pivot to create a new higher order path which is represented by
\( r \). In each case, imposing inequality (2) gives a necessary and sufficient condition for usefulness of edge
composition.

### 3.4.1 Deriving the Composition Specific Conditions for Usefulness of Edge Compositions

In this section, we derive usefulness criteria for performing edge compositions. The choice of a pivot and
these criteria together form a necessary and sufficient condition for performing a specific edge composition.

We show the derivation of the usefulness criterion for \( TS \) composition by examining the valid cases for it.
There are three cases to be considered: \( \tau_n^c > \tau_p^c \), \( \tau_n^c < \tau_p^c \) and \( \tau_n^c = \tau_p^c \). We have already seen that the
case \( \tau_n^c < s_p^c \) is invalid that results in an imprecision in points-to information and hence we ignore this case.
We derive a constraint for the case \( \tau_n^c > s_p^c \). The \( indlev \) \( s_p^c \tau_p^c \) of the reduced edge \( r \) for the case \( \tau_n^c > s_p^c \),
by balancing the \( indlev \) of the pivot \( T_n/S_p \) in edges \( n \) and \( p \), is given as

\[(s_p^c, \tau_p^c) = (s_n^c, \tau_p^c + \tau_n^c - s_p^c)\]

By imposing the usefulness constraint (Inequality 2) we get:
Figure 11: Excluding inconclusive compositions (reduced edges shown by dashes are excluded).

\[(T^c_n > S^c_p) \land (S^c_p \leq S^c_n) \land (\tau^c_p \leq \tau^c_n)\]

\[\Rightarrow (T^c_n > S^c_p) \land (S^c_p \leq S^c_n) \land (\tau^c_p + \tau^c_n - S^c_p \leq \tau^c_n)\]

\[\Rightarrow (T^c_n > S^c_p) \land (\tau^c_p \leq S^c_n)\]

\[\Rightarrow \tau^c_p \leq S^c_p < \tau^c_n\]

We can also derive a usefulness constraint for the case \(T^c_n = S^c_p\). The final condition for a useful TS composition combined for both the cases is:

\[\tau^c_p \leq S^c_p \leq \tau^c_n\]  \hspace{1cm} (TS composition)  \hspace{1cm} (3)

Similarly, we can derive the criterion for other compositions by examining the valid and useful cases for them which turn out to be:

\[\tau^c_p \leq S^c_p < S^c_n\]  \hspace{1cm} (SS composition)  \hspace{1cm} (4)

\[S^c_p \leq \tau^c_p \leq S^c_n\]  \hspace{1cm} (ST composition)  \hspace{1cm} (5)

\[S^c_p \leq \tau^c_p \leq \tau^c_n\]  \hspace{1cm} (TT composition)  \hspace{1cm} (6)

Example 10. A TS composition where \(n\) is \(z \xrightarrow{1} x\) and \(p\) is \(x \xrightarrow{2} y\) violating the constraint \(S^c_p < \tau^c_n\) (2 > 1) (Equation 3). Edge \(n\) needs pointees of \(x\) whereas edge \(p\) provides information about the pointees of pointees of \(x\). A TS composition in which \(n\) is \(z \xrightarrow{1} x\) and \(p\) is \(x \xrightarrow{1} y\), violates the constraint \(\tau^c_p \leq S^c_p\) (Equation 5). Edge \(n\) needs pointees of pointees of \(x\) whereas edge \(p\) provides information in terms of pointees of pointees of \(y\). □

In both these cases, edge composition \(n \circ p\) will take the HRG away from points-to graph and hence we do not perform such compositions. Similarly, we can reason about the usefulness constraint in Equation 4 for other types of compositions.

3.5 Conclusive Edge Compositions

Recall that \(r = n \circ p\) is valid and useful if we expect a path in the memory from \(\ell_p\) to \(\ell_n\), denoted \(\ell_p \rightarrow \ell_n\). This composition is conclusive when \(\ell_p\) remains accessible from the pivot \(P\) in \(p\) when \(n\) is composed with \(p\). It may become inaccessible from \(P\) because of a combined effect of the statements in a calling context and the statements in the procedure being processed. In such a case, the composition is undesirable and may lead to unsoundness if \(r\) replaces \(n\).

Since no information from calling context is available, we are forced to retain edge \(n\) in the HRG missing an opportunity of reducing it. Hence we propose the following conditions for conclusiveness:

(a) The statements of \(p\) and \(n\) should be consecutive on every control flow path.
(b) If the statements of \( p \) and \( n \) are not consecutive on some control flow path, we require that

(i) the intervening statements should not have an indirect assignment (e.g., \(*x = \ldots\) ), and

(ii) the pointee of pivot \( P \) in edge \( p \) has been found i.e. \( P^C = 1 \).

Example 11. Line 07 of procedure \( g \) Figure[11]indirectly defines \( a \) (because \( y \) points to \( a \) as defined on line 02 of procedure \( f \) whereas line 08 directly defines \( a \) overwriting the value assigned on line 06. Thus, \( x \) points to \( b \) and not \( c \) after line 09. However, during HRG construction of procedure \( g \), the relationship between \( y \) and \( a \) is not known. Thus, the composition of \( n \equiv x^{1,2}_{\rightarrow} y \) with \( p \equiv y^{2,0}_{\rightarrow} c \) results in \( r \equiv x^{1,0}_{\rightarrow} c \). In this case, \( \ell_p \) is \( c \), however it is not reachable from \( y \) anymore as the pointee of \( y \) (which is \( a \)) is re-defined by line 07 violating the condition \( P^C = 1 \). Thus this composition is not conclusive and we add \( n \equiv x^{1,2}_{\rightarrow} y \) instead of \( r \equiv x^{1,0}_{\rightarrow} c \).

Similarly, line 10 defines \( p \) directly whereas line 11 defines \( p \) indirectly (because \( q \) points to \( p \) as defined on line 03 of procedure \( f \)). The composition of \( n \equiv r^{1,1}_{\rightarrow} p \) with \( p \equiv p^{1,0}_{\rightarrow} t \) results in \( r \equiv r^{1,0}_{\rightarrow} t \). In this case, \( \ell_p \) is \( t \), however it is not reachable from \( p \) anymore as the pointee of \( p \) is re-defined indirectly by line 11 violating the condition that \( p \) and \( n \) should not have an intervening indirect assignment. Thus this composition is inconclusive and we add \( n \equiv r^{1,1}_{\rightarrow} p \) instead of \( r \equiv r^{1,0}_{\rightarrow} t \). \( \square \)

3.5.1 Can Edge Composition be Modelled as Matrix Multiplication?

Edge composition \( n \circ p \) computes transitive effects of edges \( n \) and \( p \). This is somewhat similar to the reachability computed in a graph: If there are edges \( x \rightarrow y \) and \( y \rightarrow z \) representing the facts that \( y \) is reachable from \( x \) and \( z \) is reachable from \( y \), then it follows that \( z \) is reachable from \( x \) and an edge \( x \rightarrow z \) can be created. If the graph is represented by an adjacency matrix \( A \) in which the element \((x, y)\) represents reachability of \( y \) from \( x \), matrix multiplication \( A \times A \) can be used to compute the transitive effect.

It is difficult to model edge composition in this manner because of the following reasons:

- Edge labels are pairs of numbers representing indirection levels. Hence we will need to device an appropriate operator and the usual multiplication would not work.

- Edge composition has some additional constraints over reachability because of desirability; undesirable compositions are not performed. These restrictions are difficult to model in matrix multiplication.

- Transitive reachability considers only the edges of the kind \( x \rightarrow y \) and \( y \rightarrow z \); i.e. the pivot should be the target of the first edge and the source of the second edge. Edge composition considers pivot as both source as well as target in both the edges and hence considers all four compositions \((SS, TT, TS, \text{and} \ ST)\). For example, we compose \( x^{1,0}_{\rightarrow} y \) and \( x^{2,0}_{\rightarrow} y \) in an SS composition to create a new edge \( z^{1,0}_{\rightarrow} y \). Transitive reachability computed using matrix multiplication can consider only \( TS \).

4 Constructing \( \Delta \) at the Intraprocedural Level

This section defines the computation of \( \Delta \) for concrete memory in Section[4.2]. It then further lifts the construction of HRGs for abstract memory in Section[4.3].

4.1 Edge Reduction in \( \Delta \)

Constructing \( \Delta \) requires us to reduce edge \( n \) by composing it with the edges present in \( \Delta \) to reduce the indirections in \( n \) using points-to information in \( \Delta \). This step is same for \( \overline{\sigma} \) and \( \overline{\sigma} \) and is formulated in Definition[5]for SS and \( TS \) compositions as given below:
Example 12. When $n$ represents a statement $x = *y$, we need multi level compositions: The first level composition identifies pointees of $y$ while the second level composition identifies the pointees of pointees of $y$. This is facilitated by function $mlc$. Consider the code snippet,

$$s_1 : y = &a;$$
$$s_2 : a = &b;$$
$$s_3 : x = *y;$$

$\Delta = \{ y \xrightarrow{1,0} a, a \xrightarrow{1,0} b \}$ when $n \equiv x \xrightarrow{1,2} y$ (statement $s_3$). $mlc(\{n\}, \Delta)$ returns an edge $x \xrightarrow{1,0} b$. This involves two consecutive $TS$ compositions. Firstly, there is a $TS$ composition $n \circ p$ with $y \xrightarrow{1,0} a$ as $p$. $TS_{\Delta}^{p} = \{ x \xrightarrow{1,1} a \}$ with $SS_{\Delta}^{n} = \emptyset$ as there is no $SS$ composition thereby satisfying the third case of $slc(n, \Delta)$. This forms the first level of composition and $slces$ is now called with $X = \{ x \xrightarrow{1,1} a \}$. The second $TS$ composition between the reduced edge $x \xrightarrow{1,1} a$ (as new $n$) and $a \xrightarrow{1,0} b$ (as $p$) results in a reduced edge $x \xrightarrow{1,0} b$. $slces$ is called again with $X = x \xrightarrow{1,0} b$ which returns $X$ thereby satisfying the base condition of $mlc$. $slc(x \xrightarrow{1,0} b, \Delta)$ returns $\{ x \xrightarrow{1,0} b \}$ because there are no further compositions as $n$ is already in its reduced form which is a points-to edge with order $(1,0)$. □

The following example contrasts combining the effect of single level compositions through $\times$ with multi level composition.

Example 13. Single level compositions need to be combined using $\times$ when $n$ represents $*x = y$; the source of the resulting edge is computed from $SS$ composition for $*x$ and the target is computed from $TS$ composition for $y$. Consider the code snippet.
4.2 Computing Points-to HRGs

We define points-to in terms of dynamic transitive closure. Apart from the reasons mentioned in Section 3.5.1, the following differences make it difficult to model edge reduction in terms of dynamic transitive closure:

- Edge reduction does not compute unrestricted transitive effects. Dynamic transitive closure computes unrestricted transitive effects.
- We do not perform closure. Either the final set computed by points-to is retained in \( \Delta \) or \( n \) is retained in \( \Delta \). Dynamic transitive closure implies retaining all edges including the edges computed in the intermediate steps.

### A Comparison with Dynamic Transitive Closure

It is tempting to compare edge reduction \( n \circ \Delta \) with dynamic transitive closure [12]: edge composition computes a new edge that captures the transitive effect and this is done repeatedly by points-to. However, the analogy stops at this abstract level. Apart from the reasons mentioned in Section 3.5.1, the following differences make it difficult to model edge reduction in terms of dynamic transitive closure:

- Edge reduction does not compute unrestricted transitive effects. Dynamic transitive closure computes unrestricted transitive effects.
- We do not perform closure. Either the final set computed by points-to is retained in \( \Delta \) or \( n \) is retained in \( \Delta \). Dynamic transitive closure implies retaining all edges including the edges computed in the intermediate steps.

#### 4.2 Computing Points-to HRGs \( \Xi(\pi, u, \nu) \) for a Single Control Flow Path

We define \( \Xi(\pi, u, \nu) \) recursively by extending \( \Xi(\pi, u, t) \) (denoted \( \Xi \)) to incorporate the effect of \( \delta(t, \nu) \) for computing the concrete memory \( M \) (Definition 4).

![Code snippet]

In the base case, \( t \) is same as \( u \) and hence \( \Xi = \emptyset \). \( \Xi \) is maintained as an ordered set of HRG edges where the order is governed by the order of inclusion of edges and ensures flow sensitivity. At the intraprocedural level, we assume that the subpath of \( \pi \) from \( u \) to \( \nu \) is free of calls. This is relaxed later in Section 5.

Extending \( \Xi(\pi, u, t) \) (denoted \( \Xi \)) to incorporate the effect of \( \delta(t, \nu) \) (denoted by the edge \( n \)) involves two steps:

- Reducing \( n \) by composing it with edges in \( \Xi \) denoted by \( n \circ \Xi \) (i.e. reduce indirections in \( n \) using points-to information in \( \Xi \)). This is explained in Definition 5.
- Updating \( \Xi \) with the reduced edges denoted by \( \Xi[n \circ \Xi] \).

The second step of updating \( \Delta \) with the reduced edges differs for \( \Xi \) and \( \Xi \). Definition 4 formulates it for \( \Xi \).

Given a reduced edge \( r \), this update, denoted \( \Xi[r] \), reorients the out edge of the source whose indlev matches that in \( r \); if no such edge exists in \( \Xi \), \( r \) is added to it. For this purpose, we view \( \Xi \) as a mapping \( V \times I \to V \times I \) where \( I \) is a set of integers and an edge \( x \xrightarrow{i,j} y \) as a pair \( ((x, i), (y, j)) \) in \( \Xi \). Then, the update of \( \Xi \) by an edge \( x \xrightarrow{i,j} y \) changes the mapping of \((x, i)\) in \( \Xi \) to \((y, j)\).
Example 14. Figure 12 shows the summary flow function along two paths in procedure \( g \) of our motivating example in Figure 3. The edges are numbered in the order of their inclusion. □

4.3 Constructing Points-to HRGs \( \overline{\pi}(u, v) \) for Multiple Control Flow Paths

In this section we define the construction of \( \overline{\pi}(u, v) \) over \( \overline{M} \).

4.3.1 Migrating from \( \overline{M} \) to \( \overline{M} \): An Overview

We compare the concepts to highlight the differences:

- **Memory update** \( M[e] \). A concrete memory \( \overline{M} \) is a function and the update \( \overline{M}[e] \) reorients the out edge of the source of \( e \). An abstract memory \( \overline{M} \) is a relation and the source of \( e \) may have multiple edges. This may require under-approximating deletion.

- **Edge composition** \( n \circ p \). This is same for both the memories.

- **Edge reduction** \( n \circ \Delta \). For a concrete \( \Delta \), the reduction \( n \circ \Delta \) creates a single edge whereas the reduction \( n \circ \overline{\Delta} \) involving an abstract \( \Delta \) could create multiple edges because \( \overline{\Delta}(u, v) \) needs to cover all paths from \( u \) to \( v \).

- **Summary flow function update** \( \Delta[e] \). Like memory update, \( \Delta \) update is exact whereas \( \overline{\Delta} \) update may have to be approximated.

\( \overline{\pi}(u, v) \) should be an over-approximation of \( \overline{\pi}(\pi, u, v) \) for every path \( \pi \) from \( u \) to \( v \). Hence, the inclusion of pointees of a pointer should be over-approximated while their removal should be under-approximated. In other words, the inclusion of edges in \( \overline{\Delta} \) may be over-approximated whereas the removal may have to be under-approximated by distinguishing between strong and weak updates.

4.3.2 Constructing \( \overline{\pi}(u, v) \)

We construct \( \overline{\pi}(u, v) \) by extending \( \overline{\pi}(u, t) \), \( \forall t \in g\text{succ}^+(u) \) (denoted \( \overline{\pi} \)) to include \( \delta(t, v) \) (denoted \( n \)) for \( v \in g\text{succ}(t) \). The rules of edge composition remain same except that \( n \circ \overline{\Delta} \) may compute multiple reduced
edges rather than a single edge as in \( n \circ \Delta \). All these edges must be included in \( \Delta \) but the edges to be removed from \( \Delta \) (represented by \( \text{conskill} \)) may be under-approximated if a strong update cannot be performed. When a strong update is performed, we delete all edges in \( \Delta \) whose source and \( \text{indlev} \) match that of the shared source of the reduced form of edge \( n \) (identified by \( \text{match}(e, X) \)) and the edges to be deleted from \( \Delta \) are defined by \( \text{conskill}(n \circ \Delta, \Delta) \). For weak update of \( \Delta \), \( \text{conskill}(n \circ \Delta, \Delta) = \emptyset \).

Since we perform \( \text{may} \) points-to analysis, the meet operation for HRGs is graph union as shown by the first equation in Definition 5. The \( \top \) value for the HRGs is not \( \emptyset \) but an artificial function \( \Delta \top \) for a more precise handling of function calls (Section 5).

**Definition 5: Construction of \( \Delta \)**

\[
\Delta(u, v) := \bigcup_{t \in \text{gpred}(v)} \left( (\Delta(u, t)) \left[ n \circ \Delta(u, t) \right] \right)
\]

where

\[
\Delta[X] := (\Delta - \text{conskill}(X, \Delta)) \cup (X)
\]

\[
\text{conskill}(X, \Delta) := \{ e_1 \mid e_1 \in \text{match}(e, \Delta), e \in X, \text{sources}(X) = 1 \}
\]

\[
\text{match}(e, X) := \{ e_1 \mid e_1 \in X, S_c = S_{e_1}, S_c = S_{e_1} \}
\]

\[
\text{sources}(X) := \{ (S_c, S_c) \mid e \in X \}
\]

**Identifying Strong and Weak Updates in \( \Delta \)**

When \( n \) represents \( x = y \), all \( x \xrightarrow{1, j} w \) edges in \( \Delta \) should be removed because \( x \) is being redefined. When \( n \) represents \( *x = y \), all \( z \xrightarrow{1, j} w \) edges in \( \Delta \) can be removed if \( x \) points-to \( z \) along every path leading to a strong update. In both these situations, all reduced edges of \( n \) have a single source—\( x \) in the former case and \( z \) in the latter. This is identified by \( \text{sources}(n \circ \Delta) \) in Definition 5.

When \( |\text{sources}(n \circ \Delta)| > 1 \), the reduced edges define multiple pointers and only a weak update is possible. However, when \( |\text{sources}(n \circ \Delta)| = 1 \) all reduced edges define the same pointer. This is necessary for a strong update but not sufficient because the pointer may not be defined along every path—there may be a definition free path which does not contribute to \( \text{sources}(n \circ \Delta) \). In order to identify whether the pointer is defined along every path or not, we introduce an upwards exposed version \( x' \) for every global variable \( x \) to represent its uses in \( \Delta \) which are not preceded by its definition in \( \Delta \). It is used as described below.
In order to eliminate a definition free path for a variable, say \( x \), from \( u \) to \( v \) we introduce a copy edge \( x \xrightarrow{1,1} x' \) at \( u \). This indicates that \( x \) is same as its upward exposed version \( x' \) at \( u \) and a reduced edge \( x \xrightarrow{1,i} y \) along any path from \( u \) to \( v \) removes the copy edge \( x \xrightarrow{1,1} x' \) indicating that \( x \) is redefined. This guarantees that \( |\text{sources}(n \circ N)| = 1 \) only when the source is defined along every path. However, if the source being defined is a (transitive) pointee of \( x \), then introduction of \( x' \) does not serve the purpose of discovering a definition free path for the pointee. We therefore view the transitive pointees of \( x \) (Figure 13(a)) as a collection of HRG edges (Figure 13(b)) which is represented by an aggregate edge (Figure 13(c)) where \( N \) is the set of natural numbers; \( S \) is a summary node representing all possible pointees. Hence, we insert an aggregate edge \( x' \xrightarrow{\mathbb{N},0} S \) at program point \( u \) for the upward exposed version of every variable \( x \). A reduced edge \( x \xrightarrow{i,j} y, \ i > 1 \) modifies the aggregate edge \( x' \xrightarrow{\mathbb{N},0} S \) to \( x' \xrightarrow{(\mathbb{N}−i),0} S \) indicating that \((i−1)^{th}\) dereference of \( x \) is redefined. The inclusion of aggregate and copy edges guarantee that \( |\text{sources}(n \circ N)| = 1 \) only when the source is defined along every path thereby eliminating the dashed path in Figure 13. This leads to a necessary and sufficient condition for strong updates.

Observe the upwards exposed versions of variables are required only for supporting updates at the intra and interprocedural level (Section 5). In other words, they are required for precision and not soundness.

**Example 15.** Consider the construction of \( \overline{\Delta}_g \) as illustrated in Figure 15. Edge \( g_1 \) created for line 8 of the program, kills edge \( a \xrightarrow{1,1} a' \). \( |\text{sources}\{\{g_1\}\}| = 1 \) and hence it kills all the edges whose source and indlev matches with that of the source \( a \) (i.e. \( a \xrightarrow{1,1} a' \)).

For line 10, since the pointees of \( x \) and \( z \) are not available in \( g \), edge \( g_2 \) is created from \( x' \) to \( z' \); this involves composition of \( x \xrightarrow{2,1} z \) with the edges \( x \xrightarrow{1,1} x' \) and \( z \xrightarrow{1,1} z' \). Edges \( g_3, g_4, g_5 \) and \( g_6 \) correspond to lines 11, 13, 14, and 16 respectively.

\( z \xrightarrow{1,1} z' \) edge is killed along both the paths (lines 11 and 14) and hence is struck off in \( \overline{\Delta}_g \) indicating \( z \) is must defined. On the other hand, \( y \xrightarrow{1,1} y' \) is killed only along one of the two paths and hence is retained by the control flow merge just before line 16. Similarly \( x' \xrightarrow{2,0} S \) in the aggregate edge \( x' \xrightarrow{\mathbb{N},0} S \) is retained indicating that pointee of \( x \) is not defined along all paths.

Edges \( g_3 \) and \( g_5 \) are may edges; however \( z \) is defined along all paths (indicated by the deletion of \( z \xrightarrow{1,1} z' \)); hence they can remove matching edges in the callers. Edge \( g_4 \) is not a must edge because \( y \) is defined along one of the two paths in \( g \) (indicated by the presence of \( y \xrightarrow{1,1} y' \)). Edge \( g_6 \) is a must edge and hence kills \( x \xrightarrow{1,1} x' \).

Line 17 creates edges \( g_7 \) and \( g_8 \); this is a weak update because \( y \) has multiple pointees (\( |\text{sources}\{\{g_7, g_8\}\}| \neq 1 \)). Hence \( b \xrightarrow{1,1} b' \) is not removed. Similarly, \( y' \xrightarrow{2,0} S \) in the aggregate edge \( y' \xrightarrow{\mathbb{N},0} S \) is not removed. □
5 Constructing $\mathfrak{m}$ at the Interprocedural Level

We have discussed intraprocedural points-to analysis using HRGs in Section 4. We now extend our analysis to the second level which includes handling function calls and recursion.

5.1 Handling Function Calls

Definition 6 shows how procedure calls are handled for constructing HRGs for summary flow functions.

| Definition 6: $\mathfrak{m}$ for a call $g()$ in procedure $f$ |
|---------------------------------------------------------------|
| /* let $\mathfrak{m}_f$ denote $\mathfrak{m}(\text{Start}_f, u)$ and $\mathfrak{m}_g$ denote $\mathfrak{m}(\text{Start}_g, \text{End}_g)$ */ |
| $\mathfrak{m}(\text{Start}_f, v) := \mathfrak{m}_g \circ \mathfrak{m}_f := \mathfrak{m}_f \bigl[ \mathfrak{m}_g \bigr] $ |
| where /* let $\mathfrak{m}_g$ be $\{ e_1, e_2, \ldots, e_k \} */ |
| $\mathfrak{m}_f \bigl[ \mathfrak{m}_g \bigr] := \mathfrak{m}_f \bigl[ e_1, \mathfrak{m}_g \bigr] \bigl[ e_2, \mathfrak{m}_g \bigr] \ldots \bigl[ e_k, \mathfrak{m}_g \bigr] $ |
| $\mathfrak{m}(e, \mathfrak{m}_g) := (\mathfrak{m}_f - \text{callkill}(e, \mathfrak{m}_f, \mathfrak{m}_g)) \cup (e \circ \mathfrak{m}_f) $ |
| callkill$(e, \mathfrak{m}_f, \mathfrak{m}_g) := \{ e_2 | e_2 \in \text{match}(e_1, \mathfrak{m}_f), e_1 \in e \circ \mathfrak{m}_f, \text{callsup}(e, \mathfrak{m}_f, \mathfrak{m}_g) \} $ |
| callsup$(e, \mathfrak{m}_f, \mathfrak{m}_g) := (| \text{sources}(e \circ \mathfrak{m}_f) | = 1) \land \text{mustedge}(e, \mathfrak{m}_g) $ |
| mustedge$(x \xrightarrow{i,j} y, \mathfrak{m}) \iff (x \xrightarrow{i,k} z \in \mathfrak{m} \Rightarrow k = j \land z = y) \land (i > 1 \land x \xrightarrow{i,0} s \notin \mathfrak{m}) \lor (i = 1 \land x \xrightarrow{1,1} x' \notin \mathfrak{m}) $ |

Consider two procedures $f$ and $g$ such that $f$ contains a call to $g$ between two consecutive program points $u$ and $v$, $v \in \text{gsucc}(u)$. Let $\text{Start}_f$ and $\text{End}_f$ denote the start and the end points of $f$, $\mathfrak{m}$ representing the control flow paths from $\text{Start}_f$ to $u$ (i.e., just before the call to $g$) is $\mathfrak{m}(\text{Start}_f, u)$; we denote it by $\mathfrak{m}_f$ for brevity. $\mathfrak{m}$ for the body of procedure $g$ is $\mathfrak{m}(\text{Start}_g, \text{End}_g)$; we denote it by $\mathfrak{m}_g$.

Then $\mathfrak{m}(\text{Start}_f, v)$ is computed as follows:

- Edges for actual-to-formal-parameter mapping are added to $\mathfrak{m}_f$.
- $\mathfrak{m}_f$ and $\mathfrak{m}_g$ are composed.
- An edge is created between the return variable of $g$ and the receiver variable in $f$ and is added to $\mathfrak{m}_f$.

Since HRGs are ordered sets of edges, their composition is simple: we select an edge $e$ from $\mathfrak{m}_g$ and perform an update $\mathfrak{m}_f \bigl[ e \circ \mathfrak{m}_f \bigr]$. We then update the resulting $\mathfrak{m}$ with the next edge from $\mathfrak{m}_g$. This is repeated until all edges of $\mathfrak{m}_g$ are exhausted. Intuitively, this amounts to inlining $\mathfrak{m}_g$ in $\mathfrak{m}_f$.

The update of $\mathfrak{m}_f$ with an edge $e$ from $\mathfrak{m}_g$ involves the following:

- Substituting the callee’s upwards exposed variable $x'$ occurring in $\mathfrak{m}_g$ by the caller’s original variable $x$ in $\mathfrak{m}_f$.
- Including reduced edges resulting from $e \circ \mathfrak{m}_f$.
- Performing a strong or weak update. An update of $\mathfrak{m}_f$ with $e$ is a strong update when $e$ defines a single pointer and is a must edge in $\mathfrak{m}_g$ (i.e., it is defined along all paths in $\mathfrak{m}_g$).

Note that if $e$ involves an upwards exposed variable $x'$, it should be composed with an original edge in $\mathfrak{m}_f$ rather than a reduced edge included in $\mathfrak{m}_f$ created by $e_1 \circ \mathfrak{m}_f$ for some $e_1 \in \mathfrak{m}_g$.

Strong update for summary flow function composition $\mathfrak{m}_f \circ \mathfrak{m}_g$ is identified by function callsup (Definition 6). Observe that $x \xrightarrow{1,1} x' \in \mathfrak{m}$ implies that $x \xrightarrow{1,0} y \in \mathfrak{m}$ is a may edge. $x' \xrightarrow{0,0} s \in \mathfrak{m}$ implies that
For indlev mn, regardless of the direction of the edge, m is for the source while n is for the target. Edges deleted by updates are struck off. The numbers in the subscripts of edge names (e.g., \( g_i, f_i \)) indicate the order of their inclusion.

**Example 16.** Consider the construction of \( \overline{\Delta} \) as illustrated in Figure 15. Edges \( f_1 \) and \( f_2 \) correspond to lines 2 and 3. The call on line 4 causes the composition of \( \overline{\Delta}_f = \{ f_1, f_2 \} \) with \( \overline{\Delta}_g \) selecting edges in the order \( g_1, g_2, \ldots, g_8 \). The edges from \( \overline{\Delta}_g \) with their corresponding names in \( \overline{\Delta}_f \) (denoted name-in-g/name-in-f) are: \( g_1/f_3, g_2/f_5, g_4/f_6, g_5/f_7, g_6/f_8, g_7/f_9, \) and \( g_8/f_{10} \). Edge \( f_4 \) is created by SS and TS compositions of \( g_2 \) with \( f_1 \) and \( f_2 \). Although \( x \) has a single pointee (along edge \( f_1 \)), the resulting update is a weak update because \( g_2 \) is a *may* edge indicated by the presence of \( x' \xrightarrow{2.0} s \) in the aggregate edge \( x' \xrightarrow{N.0} s \). If line 10 is moved just before line 16, then \( g_2 \) will be a *must* edge and during function composition, it will cause a strong update killing edge \( f_3 \).

Edges \( g_3 \) and \( g_5 \) together kill \( f_2 \). Note that the inclusion of \( f_7 \) does not kill \( f_3 \) because they both are from \( \overline{\Delta}_g \). Finally, the edge for line 5 (\( x \xrightarrow{2.1} z \)) undergoes an SS composition (with \( f_8 \)) and TS compositions
01 void f()
02 {
03     if (...) {
04         y = &a;
05     }
06     else {
07         y = &b;
08         f();
09     }
10 }

Figure 16: A recursive example demonstrating the need for $\Delta_T$.

(with $f_5$ and $f_7$). This creates edges $f_{11}$ and $f_{12}$. Since $x \xrightarrow{2.1} z$ is a must edge (indicated by the absence of $x' \xrightarrow{2.0} s$ from the aggregate edge $x' \xrightarrow{\{N-\{2\},0\}} s$) and $x$ has a single pointee (edge $f_8$), this is a strong update killing the edge $f_{10}$.

Observe that all edges in $\overline{\Delta}_f$ are canonical edges except $f_9$. In order to reduce $f_9$, we need the pointees of $y$ from its callers. □

### 5.2 Handling Recursion

The summary flow function $\overline{\Delta}$ of a procedure is complete only when it incorporates the effect of all its callees. Hence $\overline{\Delta}$ of callee procedures are constructed first to incorporate its effect in their callers resulting in a postorder traversal over the call graph. However, in case of recursion, $\overline{\Delta}$ of a callee procedure may not have been constructed yet because of the presence of cycle in the call graph. This requires us to begin with an approximate version of $\overline{\Delta}$ which is then refined to incorporate the effect of recursive calls. When the callee’s $\overline{\Delta}$ is computed, its call statements will have to be reprocessed needing a fixed point computation.

In the presence of cycle in the call graph because of recursion, we may have to over-approximate the initial $\overline{\Delta}$ for a callee. This is handled in the usual manner [9, 22] by over-approximating initial $\overline{\Delta}$ that computes $\top$ for may points-to analysis (which is $\emptyset$). Using any other function would be sound but imprecise. Such an HRG, denoted $\Delta_T$, kills all points-to relations and generates none. Clearly, $\Delta_T$ is not expressible as a HRG and is not a natural $\top$ element of the meet semi-lattice [9] of HRGs. It has the following properties related to the meet and composition:

- **Meet Operation.** Since we wish to retain the the meet operation $\cap$ as $\cup$, we extend it to define $\Delta \cup \Delta_T = \Delta$. Although $\Delta = \emptyset$ denoted as $\Delta_{id}$, also seems to satisfy this, it is an identify function and not a function computing $\top$ because it does not kill points-to information.

- **Composition.** Since $\Delta_T$ is a constant function returning $\top$ value of the lattice of may points-to analysis, it follows that $\forall \Delta, \Delta_T \circ \Delta = \Delta_T$ because $\Delta_T$ is a constant function returning the $\top$ value. Similarly, $\forall X, \forall \Delta, \Delta(T(X)) = \Delta(T) = \emptyset[\Delta]$ which implies that $\Delta \circ \Delta_T = \Delta$. This is because $\top$ for may points-to analysis is $\emptyset$ and empty memory updated with $\Delta$ returns $\Delta$. Note that $\Delta \circ \Delta_T = \Delta$ is an intermediate function because the fixed point computation induced by recursion will eventually replace $\Delta_T$ by appropriate summary flow function.

**Example 17.** In the example of Figure[16] if we use the initial $\overline{\Delta}$ for procedure $f$ at $n_4$ as $\Delta_{id}$ - an HRG with no edges, then the $\overline{\Delta}$ at the $Out$ of $n_4$ has an HRG with one edge $y \xrightarrow{1.0} b$. Thus, the summary flow function of procedure $f$ ($\overline{\Delta}_f$) computed at $n_5$ after the meet is shown in Figure [16](b). After reprocessing the call at
We first define the semantics of \( \text{Out} \) from \( u \). The resulting memory after applying \( M \) requires evaluating \( e \) in \( \overline{M} \) and then reorienting the existing edges. Suppose the evaluation \( \overline{M} \) of Figure 12(a) is \( M \) of Figure 12(b) representing the other control flow path to the same \( n \). When we apply \( \overline{M} \) of Figure 12(b) representing the other control flow path to the same \( M \) before the call to \( g \), the resulting \( \overline{M} \) is \( \{(a, d), (b, d), (x, b), (y, b), (z, v)\} \).

Example 18. For our motivating example, let \( \overline{M} \) before the call to \( g \) be \( \{(a, ?), (b, ?), (x, a), (y, ?), (z, w)\} \). The resulting memory after applying \( \overline{M} \) of Figure 12(a) is \( \{(a, w), (b, ?), (x, b), (y, ?), (z, u)\} \). When we apply \( \overline{M} \) of Figure 12(b) representing the other control flow path to the same \( M \) before the call to \( g \), the resulting \( M \) is \( \{(a, e), (b, d), (x, b), (y, b), (z, v)\} \).

6 Semantics of the Application of \( \Delta \) to \( M \)

We first define the semantics of \( \overline{M} \) and then extend it to \( M \).

6.1 Semantics of the Application of \( \overline{M} \) to \( M \)

The initial state of a control flow path \( \pi \) is \( \sigma_0 = (\pi, q_0, \overline{M}_0) \) with \( \overline{M}_0 = \{(x, ?) \mid x \in V\} \). Since \( \overline{M}_0 \) is a total function, \( \overline{M} \) is defined for all variables at all program points. Let \( \overline{M}\{a\} = \{b\} \) implying that \( a \) points-to \( b \) in \( \overline{M} \). Suppose that, as a consequence of execution of a statement, \( a \) ceases to point to \( b \) and instead points to \( c \). The memory resulting from this change is denoted by \( \overline{M}[a \mapsto c] \).

Definition 7[6] provides the semantics of the application of \( \overline{M}(\pi, u, v) \) to \( M \) in terms of state transitions from \( u \) to \( v \) along \( \pi \).

\[
\begin{align*}
\overline{M}_{\pi, \pi} := & \overline{M}_u \pi \{ \overline{M}(\pi, u, v) \} \\
\text{where} & \quad \text{/* let } \overline{M} \text{ be } \{e_1, e_2, \ldots, e_k\} */ \\
\begin{array}{l}
\overline{M}[\overline{M}] := (\ldots ((\overline{M}[e_1]) [e_2]) \ldots [e_k]) := \overline{M}[e_1] [e_2] \ldots [e_k] \\
\overline{M}[x \overset{i,j}{\rightarrow} y] := \overline{M}[w \mapsto z] \text{ where } w = \overline{M}^{-1}[x], \ z = \overline{M}^j[y] \\
\overline{M}[x \overset{i,j}{\rightarrow} y] := w^1_{z} z \text{ where } w = \overline{M}^{-1}[x], \ z = \overline{M}^j[y]
\end{array}
\end{align*}
\]

The evaluation of an edge \( x \overset{i,j}{\rightarrow} y \) in \( \overline{M} \), denoted \( [x \overset{i,j}{\rightarrow} y]_{\overline{M}} \), creates a points-to edge by discovering the locations reached indirectly from \( x \) and \( y \). The memory update due to an HRG edge \( e \equiv x \overset{i,j}{\rightarrow} y \), denoted \( \overline{M}[x \overset{i,j}{\rightarrow} y] \), requires evaluating \( e \) in \( \overline{M} \) and then reorienting the existing edges. Suppose the evaluation \( [x \overset{i,j}{\rightarrow} y]_{\overline{M}} \) computes \( w^1_{z} z \), then, the result of \( \overline{M}[x \overset{i,j}{\rightarrow} y] \) is \( \overline{M}[w \mapsto z] \); although the two notations look similar, the arrow in the first indicates that it is an HRG edge whereas the arrow in the second indicates that a mapping is being changed. Effectively we change \( \overline{M} \) such that \( \overline{M}^j[x] = \overline{M}^j[y] \).

Example 18. For our motivating example, let \( \overline{M} \) before the call to \( g \) be \( \{(a, ?), (b, ?), (x, a), (y, ?), (z, w)\} \). The resulting memory after applying \( \overline{M} \) of Figure 12(a) is \( \{(a, w), (b, ?), (x, b), (y, ?), (z, u)\} \). When we apply \( \overline{M} \) of Figure 12(b) representing the other control flow path to the same \( M \) before the call to \( g \), the resulting \( M \) is \( \{(a, e), (b, d), (x, b), (y, b), (z, v)\} \).

6.2 Semantics of the Application of \( \overline{M} \) to \( M \)

Definition[8] provides the semantics of \( \overline{M}(u, v) \) by showing how \( \overline{M} \) is computed from \( M \).

26.
Effectively we change the points-to information at every program point within that procedure. For the main function the creates a set of points-to edges by discovering the locations reached indirectly from Points-to analysis using HRGs is performed in two phases; first phase involves constructing bottom-up con-

8 Computing Points-to Information using HRGs

Points-to analysis using HRGs is performed in two phases; first phase involves constructing bottom-up context independent summary flow functions (HRGs), whereas the second phase involves computing points-to information at every program point within a procedure using the summary flow functions. This requires computing the boundary information (BI) for every procedure which involves capturing the points-to information reaching that procedure from all of its callers. The BI so computed is then used to compute the points-to information at every program point within that procedure. For the main function the BI is computed from static initializations. In the presence of recursion, BI may require a fixed point computation.

The computation of points-to information within a procedure from its BI can be achieved in two ways:
Figure 17: An example demonstrating the bypassing performed.

(a) For a procedure $r$, since all $\Delta(\text{Start}_r, u)$ (i.e. the summary flow function representing the effect of all paths from the start of $r$ to $u$) have been constructed, points-to information at $u$ can be computed simply by applying $\Delta(\text{Start}_r, u)$ to $BI$ (i.e. the points-to information reaching $\text{Start}_r$).

This approach is oblivious to intraprocedural control flow and does not involve fixed point computation for loops although it requires fixed point computation for finalizing $BI$ of $r$ in the presence of recursive calls involving $r$.

(b) Points-to information in $r$ is computed using

- $\delta(u, v)$ (i.e. the statement level flow functions) for all non-call statements, and
- $\Delta(\text{Start}_q, \text{End}_q)$ for a statement calling procedure $q$.

This approach requires a fixed point computation to handle loops within procedure $r$ (apart from the fixed point computation required for finalizing $BI$ of $r$ in the presence of recursive calls involving $r$).

Example 19. The $BI$ of procedure $g$ ($BI_g$) in the example of Figure[17] is the points-to information reaching $g$ from its callers $f$ and $h$. Thus, the $BI_g$ is a union of HRG at the $Out$ of line numbers 05 and 17. Let $\Delta_{10}$ represent the HRG at line number 10. Then the points-to information at line number 10 is $(\Delta_{10} \circ BI_g)$ as discussed in Section [5]. Similarly, the points-to information at line number 11 can be computed by $(\Delta_{11} \circ BI_g)$. □

Our measurements show that when we compute the points-to information using $\Delta$ (i.e. the first approach), it takes more time whereas the second approach requires much less time. In fact the first approach takes much more time compared to the time taken to construct $\Delta$. This may appear surprising because the second approach requires an additional fixed point computation for handling loops. The reason the first approach requires more time is that the HRG at $u$ represents a cumulative effect of the statement level flow functions from $\text{Start}_r$ to $u$. The HRGs tend to become larger with the length of a control flow path because they contain cumulative effect of all statements appearing in the path. Thus computing points-to information using HRGs for consecutive statements involves redundant computations.

28


Example 20. In our example above, $\Delta_{10}$ has only one edge \( y \xrightarrow{1,1} z' \) (ignoring the aggregate and copy edges) whereas $\Delta_{11}$ consists of two edges \( y \xrightarrow{1,1} z' \) and \( x' \xrightarrow{1,2} z' \) incorporating the effect of all the control flow paths from start of procedure \( g \) to line number 11 which also includes the effect of line number 10.

As an alternative, we can compute points-to information using statement level flow functions using the points-to information computed for the \( \ln \) of the statement (instead of \( B1 \)) thereby avoiding redundant computations. Thus at line number 10, we have \( y \xrightarrow{1,1} z \) and at line number 11 we have only \( x \xrightarrow{2,1} z \).

For a call statement, we can use the HRG representing the summary flow function of the callee instead of propagating the values through the body of the callee. This reduces the computation of points-to information to an intraprocedural analysis. \( \square \)

Our measurement show that computing \( B1 \) of a procedure from all its call points is expensive because many points-to pairs reaching a call may not be accessed by the callee procedure. Thus the efficiency of analysis can be enhanced significantly by filtering out the points-to information which is irrelevant to a procedure but merely passes through it unchanged. This concept of \textit{bypassing} has been successfully used for data flow values of scalars \([16,17]\). HRGs support this naturally for pointers because $\overline{M}$ contains edges involving upwards exposed versions of variables which allow us to separate relevant information from the irrelevant information which can be bypassed. More specifically, if a variable has upwards exposed version in an HRG, then it means that there is a use of the variable in the procedure which requires pointee information from the callers. Hence the points-to information of such a variable is relevant. If there is no upwards exposed version of a variable in an HRG, its points-to information is irrelevant and can be discarded from the \( B1 \) of the procedure effectively bypassing the call.

Example 21. In our example of Figure[17] the HRG at the \textit{Out} of line number 11 (which represents the summary flow function of procedure \( g \)) contains upwards exposed versions of variables \( x \) and \( z \) indicating that some pointees of \( x \) and \( z \) from the calling context are accessed in the procedure \( g \). Since the \textit{indlev} of \( x' \) is 2 which is the source of one of the HRG edge, its pointee is being defined by \( g \). Thus, pointee of \( x \) needs to be propagated to the procedure \( g \). Similarly, the \textit{indlev} of \( z' \) is 1 which is the target of an HRG edge specifying that pointee of \( z \) is being assigned to some pointer in procedure \( g \). Thus, pointees of \( x \) and \( z \) are accessed in procedure \( g \) but are defined in the calling context and hence should be part of the \( B1 \) of procedure \( g \). Note that points-to information of \( p \) or \( q \) is neither accessed nor defined by procedure \( g \) and hence can be bypassed. Thus, \( B1_g \) is not the union of HRGs at the \textit{Out} of line numbers 05 and 17. It excludes edges such as \( p \xrightarrow{1,0} c \) and \( q \xrightarrow{1,0} w \) as they are irrelevant to procedure \( g \) and hence are bypassed. \( \square \)

8 Soundness of Summary flow Functions

Definition[9] lists the soundness claims which are followed by their proofs. The soundness of $\overline{M}$ (i.e., HRG for abstract memory) is shown by arguing that it is an over-approximation of concrete memories. We begin by defining the notion of state transition which is used for showing the soundness of $\overline{M}$ (i.e., HRG for concrete memory).

The state $\sigma \in S$ at a particular occurrence of a program point \( u \) along a control flow path $\pi$ is $\langle \pi, u, \overline{M}_u, \pi \rangle$. The execution of the program along the control flow path $\pi$ is modelled by a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_m$ where $\sigma_i = (\pi, q_i, \overline{M}_i)$ such that

- $\sigma_0 = (\pi, q_0, \overline{M}_0)$ is the initial state with $\overline{M}_0 = \{(x, ?) \mid x \in V\}$,
- $\sigma_{i+1} := \tau(\sigma_i)$ where $\tau : S \rightarrow S$ is a state transition function.

Let \( u \) and \( v \) denote $q_i$ and $q_{i+1}$ and let $\delta(u, v)$ denote the flow function of the statement appearing between
them. Then,

\[ \tau(\pi, u, M_{u, \pi}) := (\pi, v, M_{v, \pi}), \quad \text{where} \]

\[ M_{v, \pi} := (\delta(u, v)) (M_{u, \pi}) \]  

**Definition 9: Soundness of \( \Delta \) and \( \overline{\Delta} \)**

**Soundness of Concrete Summary Flow Function \( \overline{\Delta} \)**

\[ [n] (M_{u, \pi}[p]) := [n \circ p] M_{u, \pi} \]

\[ \tau^k(\pi, u, M_{u, \pi}) = (\pi, v, M_{v, \pi}) \]

\[ \Rightarrow M_{v, \pi} = M_{u, \pi}[\Delta(\pi, u, v)] \]

**Soundness of Abstract Summary Flow Function \( \overline{\Delta} \)**

\[ \text{kill}(\pi, n, M_{u, \pi}) := \{ e_1 \mid e_1 \in \text{match}(e, M_{u, \pi}), e \in [n] M_{u, \pi} \} \]

\[ \text{memkill}(n, M_{u, \overline{\Delta}(u, v)}) \subseteq \bigcap_{\pi \in \text{Paths}(u, v)} \text{kill}(\pi, n, M_{u, \pi}) \]

\[ n \circ \overline{\Delta}(u, v) \supseteq \bigcup_{\pi \in \text{Paths}(u, v)} n \circ \Delta(\pi, u, v) \]

\[ (\overline{\Delta}(u, v)) (M_{u, \overline{\Delta}}) \subseteq \bigcup_{\pi \in \text{Paths}(u, v)} (\Delta(\pi, u, v)) (M_{u, \pi}) \]

### 8.1 Soundness of Concrete Summary Flow Function

\( \Delta(\pi, u, v) \) is sound because the effect of the reduced edge is identical to the effect of the original edge on \( M_{u, \pi} \); hence the evaluation of an edge \( n \) in memory \( M_{u, \pi} \) updated with edge \( p \), is same as the evaluation of the reduced edge \( n \circ p \) in \( M_{u, \pi} \).

**Lemma 2.** The evaluation of an edge \( n \) in memory \( M_{u, \pi} \) updated with edge \( p \), is same as the evaluation of the composed edge \( n \circ p \) in \( M_{u, \pi} \).

\[ [n] (M_{u, \pi}[p]) = [n \circ p] M_{u, \pi} \]  

**Proof.** The lemma trivially follows when \( n \) and \( p \) do not compose because they have independent effects on \( M_{u, \pi} \) provided the order of execution is followed.

Consider TS composition for \( n \circ p \). Let edge \( n \equiv x \xrightarrow{i,j} y \) and edge \( p \equiv y \xrightarrow{k,l} z \). From Section 3.3 \( n \circ p = x \xrightarrow{i,(l+j-k)} z \) for a useful composition.

- For the RHS of (8.2.a), the evaluation of \( n \circ p \) in \( M_{u, \pi} \) results in \([n \circ p] M_{u, \pi} = s_1 \xrightarrow{1,0} t_1 \) where \( s_1 = M_{u, \pi}^{i-1}\{x\} \) and \( t_1 = M_{u, \pi}^{i+j-k}\{z\} \). Thus edge \( s_1 \xrightarrow{1,0} t_1 \) imposes the constraint

\[ M_{u, \pi}^i\{x\} = M_{u, \pi}^{i+j-k}\{z\} \]  

(8.2.b)

- For the LHS of (8.2.a), edge \( p \) updates \( M_{u, \pi} \) as follows \([p] M_{u, \pi} = s_2 \xrightarrow{i} t_2 \) where the pointer \( s_2 = M_{u, \pi}^{k-1}\{y\} \) and the pointee \( t_2 = M_{u, \pi}^{i}\{z\} \). \([p] M_{u, \pi} \) is defined in terms of \( M \) by the following constraint resulting from the inclusion of the edge \( s_2 \xrightarrow{1,0} t_2 \).

\[ M_{u, \pi}^{k}\{y\} = M_{u, \pi}^{i}\{z\} \]

(8.2.c)
The evaluation of $n$ in the updated memory $\overline{M}_{u, \pi} [p]$ results in
\[
\llbracket n \rrbracket (\overline{M}_{u, \pi} [p]) = s^3 \xrightarrow{1.0} t_3
\]
where $s_3 = (\overline{M}_{u, \pi} [p])^{i-1} \{ x \}$ and $t_3 = (\overline{M}_{u, \pi} [p])^j \{ y \}$. Edge $s_3 \xrightarrow{1.0} t_3$ imposes the following constraint on $\overline{M}_{u, \pi} [p]$.
\[
(\overline{M}_{u, \pi} [p])^i \{ x \} = (\overline{M}_{u, \pi} [p])^j \{ y \}
\]
In order to map this constraint to $\overline{M}_{u, \pi}$, we need to combine it with constraint (8.2.d), replace $(\overline{M}_{u, \pi} [p])$ by $\overline{M}_{u, \pi}$ and solve them together.
\[
\begin{align*}
\overline{M}_{u, \pi}^i \{ x \} &= \overline{M}_{u, \pi}^j \{ y \} \land \overline{M}_{u, \pi}^k \{ z \} = \overline{M}_{u, \pi}^l \{ z \} \\
\Rightarrow \overline{M}_{u, \pi}^i \{ x \} &= \overline{M}_{u, \pi}^j \{ y \} \land \overline{M}_{u, \pi}^{k+(j-k)} \{ y \} = \overline{M}_{u, \pi}^{l+(j-k)} \{ z \} \\
\Rightarrow \overline{M}_{u, \pi}^i \{ x \} &= \overline{M}_{u, \pi}^{i+j-k} \{ z \}
\end{align*}
\]
Constraint (8.2.d) is identical to constraint (8.2.b). Since the effect on the memory is identical, the two evaluations are identical.

The equivalence of evaluations for $SS$ composition between $n$ and $p$ can be proved in a similar manner. \hfill □

**Lemma 3.** The evaluation of an edge $n$ in memory $\overline{M}_{u, \pi}$ updated with $\overline{\pi}$ is same as the evaluation of the reduced edge $n \circ \overline{\pi}$ in $\overline{M}_{u, \pi}$.

\[
\llbracket n \rrbracket (\overline{M}_{u, \pi} [\overline{\pi}]) = \llbracket n \circ \overline{\pi} \rrbracket \overline{M}_{u, \pi}
\]

**Proof.** Let $\overline{\pi}_m$ denote $\overline{\pi}(\pi, u, \nu)$, where the subpath of $\pi$ from $u$ to $\nu$ contains $m$ pointer assignment statements. We prove the lemma by induction on $m$. From Definition 4
\[
\overline{\pi}_m = \overline{\pi}_{m-1} [e_m \circ \overline{\pi}_{m-1}]
\]
(8.3.a)
\[
= \overline{\pi}_{m-1} [e]
\]
where $e = e_m \circ \overline{\pi}_{m-1}$ (8.3.b)

For basis $m = 1$, $\overline{\pi}_1$ contains a single edge and $\overline{\pi}_0 = \emptyset$. Hence the basis holds from Lemma 2. For the inductive hypothesis, assume
\[
\llbracket n \rrbracket (\overline{M}_{u, \pi} [\overline{\pi}_m]) = \llbracket n \circ \overline{\pi}_m \rrbracket \overline{M}_{u, \pi}
\]
(8.3.c)
To prove,
\[
\llbracket n \rrbracket (\overline{M}_{u, \pi} [\overline{\pi}_{m+1}]) = \llbracket n \circ \overline{\pi}_{m+1} \rrbracket \overline{M}_{u, \pi}
\]
For $m + 1$, the RHS of (8.3.c) becomes
\[
\begin{align*}
\llbracket n \circ \overline{\pi}_{m+1} \rrbracket \overline{M}_{u, \pi} \\
\Rightarrow \llbracket n \circ (\overline{\pi}_m [e_{m+1} \circ \overline{\pi}_m]) \rrbracket \overline{M}_{u, \pi} & \text{ (using (8.3.a) for } \overline{\pi}_{m+1}) \\
\Rightarrow \llbracket n \circ (\overline{\pi}_m [e]) \rrbracket \overline{M}_{u, \pi} & \text{ (let } e_{m+1} \circ \overline{\pi}_m = e) \quad (8.3.d) \\
\Rightarrow \llbracket n \rrbracket (\overline{M}_{u, \pi} [\overline{\pi}_m [e]]) & \text{ (from (8.3.d) and (8.3.c))} \quad (8.3.e) \\
\Rightarrow \llbracket n \rrbracket (\overline{M}_{u, \pi} [\overline{\pi}_m [e]]) & \text{ (from (8.3.c) and (8.3.b))} \\
\end{align*}
\]
Hence the lemma. \hfill □

**Theorem 1.** Let sub-path of $\pi$ from $u$ to $\nu$ contain $k$ statements. Then, $\tau^k(\pi, u, \overline{M}_{u, \pi}) = (\pi, \nu, \overline{M}_u, \pi) \Rightarrow \overline{M}_{u, \pi} = \overline{M}_{u, \pi} [\overline{\pi}(\pi, u, \nu)]$

**Proof.** From Lemma 3, the effect of the reduced form $e \circ \overline{\pi}$ of an edge $e$ on memory $\overline{M}_{u, \pi}$ is identical to the effect of $e$ on $\overline{M}_{u, \pi}$ updated with $\overline{\pi}$. This holds for every edge in $\overline{\pi}$ and the theorem follows from induction on the number of statements covered by $\overline{\pi}$. \hfill □
8.2 Soundness of Abstract Summary Flow Function

\( \overline{X}(u, v) \) is sound because it under-approximates the removal of HRG edges and over-approximates the inclusion of HRG edges compared to \( \overline{X}(\pi, u, v) \) for any \( \pi \) from \( u \) to \( v \).

The update of concrete memory \( \overline{M}_{\pi, u} \) (Definition 7) reorients the edges without explicitly defining the edges being removed. We can rewrite the equation as:

\[
\overline{M}_{\pi, u} \left[ n \right] = (\overline{M}_{\pi, u} - \text{kill}(\pi, n, \overline{M}_{\pi, u})) \cup \{ n \} \overline{M}_{\pi, u}
\]

(8)

\[
\text{kill}(\pi, n, \overline{M}_{\pi, u}) = \{ e_1 \mid e_1 \in \text{match}(e, \overline{M}_{\pi, u}), e \in \left[ n \right] \overline{M}_{\pi, u} \}
\]

(9)

Let \( \text{Paths}(u, v) \) denote the set of all control flow paths from \( u \) to \( v \).

**Lemma 4.** Abstract summary flow function under-approximates the removal of information.

\[
\text{memkill}(n, \overline{M}_{\pi, u}, \overline{X}(u, v)) \subseteq \bigcap_{\pi \in \text{Paths}(u, v)} \text{kill}(\pi, n, \overline{M}_{\pi, u})
\]

(8.4.a)

**Proof.** Observe that \( \text{memkill} \) (Definition 8) is more conservative than \( \text{kill} \) (Equation 9) because it additionally requires that \( n \) should cause a strong update. From Definition 8 for causing a strong update, \( n \) must be defined along every path and the removable edges must define the same source along every path. Hence 8.4.a follows. □

**Lemma 5.** Abstract summary flow function over-approximates the inclusion of information.

**Proof.** Since the rules of composition are same for both \( \overline{X} \) and \( \overline{X} \), it follows from Definition 3 that,

\[
n \circ \overline{X}(u, v) \supseteq \bigcup_{\pi \in \text{Paths}(u, v)} n \circ \overline{X}(\pi, u, v)
\]

□

**Theorem 2.** Abstract summary flow function \( \overline{X}(u, v) \) is a sound approximation of all concrete summary flow functions \( X(\pi, u, v) \).

\[
(\overline{X}(u, v)) (\overline{M}_u) \supseteq \bigcup_{\pi \in \text{Paths}(u, v)} (X(\pi, u, v)) (\overline{M}_{\pi, u})
\]

**Proof.** It follows because killing is under-approximated (Lemma 4) and generation is over-approximated (Lemma 5). □

9 Handling Advanced Features for Points-to Analysis using HRGs

This section describes handling of function pointers, structures, unions and heap. This section also describes how our analysis handles arrays and pointer arithmetic.
void f()
{
    fp = p;
    x = &a;
    g(fp);
    fp = q;
    z = &b;
    g(fp);
    z = &c;
    g(fp);
}

void g(fp)
{
    fp();
}

void p()
{
    y = x;
}

void q()
{
    y = z;
}

Figure 18: An example demonstrating the top-down traversal of call graph for handling function pointers.

9.1 Handling Function Pointers

In the presence of indirect calls (e.g., a call through a function pointer in C), the callee procedure is not known at compile time. In our case, construction of the HRG of a procedure requires incorporating the effect of the HRGs of all its callees and in the presence of indirect calls, we would not know the callees whose HRGs should be used at an indirect call site.

If the function pointers are defined locally, their effect can be handled easily because the pointees of function pointers would be available during summary construction. Consider the function pointers that are passed as parameters or global function pointers that are defined in the callers. A top-down interprocedural pointer analysis would be able to handle such function pointers naturally because the information flows from callers to callees and hence the pointees of function pointers would be known at the call sites. However, a bottom-up interprocedural analysis such as ours, works in two phases and the information flows from

- the callees to callers when summary flow functions are constructed, and from
- the callers to callees when summary flow functions are used for computing the points-to information.

We can expect the function pointer values to be available in the second phase but they are actually required in the first phase.

It is important to observe that the basic requirement of a bottom-up approach is that the callee procedures should have been processed before caller procedures are processed. More specifically,

The order in which the construction of summary flow functions of procedures begins is not as important as the order in which the construction of summary flow functions completes. If a procedure $r$ calls procedure $q$, all we need is that the construction summary flow function of $q$ should have been completed before we expect to complete the construction of the summary flow function of $r$. This requirement can be satisfied by beginning to construct the summary flow function of $r$ before that of $q$; when a call to $q$ is encountered, analysis of $r$ can be suspended and $q$ can be processed completely before resuming the analysis or $f$.

Thus, we can traverse the call graph top-down and yet construct bottom-up context independent summary flow functions. We start the analysis with $main$ function and suspend the construction of its summary
flow function $\overline{\mathcal{M}}_{\text{main}}$ when a call is encountered and then analyze the callee first. After the completion of construction of summary flow function of the callee, then the construction of $\overline{\mathcal{M}}_{\text{main}}$ is resumed. Thus, the construction of summary flow function of callees is completed before the construction of summary flow function of their caller. Only the function pointer value from the calling context is used to build a summary flow function.

Observe that a summary flow function so constructed, is context independent for the rest of the pointers but is customized for a specific value of a function pointer that is passed as a parameter or is defined globally. In other words, a procedure with an indirect call should have a different summary flow function for distinct values of function pointer for context sensitivity. This is important because the call chains starting at the call through function pointer in that procedure could be different.

Example 22. In the example of Figure 18 we first analyze the procedure $f$ as we are traversing the call graph top-down and suspend the construction of its summary flow function at the call site at line number 05 to analyze its callee which is procedure $g$. We construct a customized summary flow function for procedure $g$ with $fp = p$. The pointee information of $x$ is not used for summary construction of $g$. In procedure $g$, there is a call through function pointer whose value is $p$ as extracted from the calling context, we now suspend the summary construction of $g$ and summary flow function of $p$ is constructed first and its effect is incorporated in $g$ with $\overline{\mathcal{M}} = \{y \xrightarrow{1,1} x\}$. We then resume with the summary flow function construction of procedure $f$ by incorporating the effect of procedure $g$ at line number 05 which results in a reduced edge $y \xrightarrow{1,0} a$ by performing the required edge compositions.

At the call site at line number 07, procedure $g$ is analyzed again with a different value of $fp$ and this time procedure $q$ is the callee which is analyzed and whose effect is incorporated to construct summary flow function for procedure $g$ with $\overline{\mathcal{M}} = \{y \xrightarrow{1,1} z\}$ for $fp = q$. Note that procedure $g$ has two summary flow functions constructed for different values of function pointer $fp$ so far encountered. However, procedure $p$ and $q$ has only one summary flow function as they do not have any calls through function pointers. At line number 07, $y$ now points to $b$ as $z$ points to $b$ (because $\overline{\mathcal{M}}_g = \{y \xrightarrow{1,1} z\}$ for $fp = q$).

The third call to $g$ at line number 10 does not require re-analysis of procedure $g$ as summary flow function is already constructed because value of $fp$ is not changed. So the summary flow function of procedure $g$ $\overline{\mathcal{M}}_g = \{y \xrightarrow{1,1} z\}$ for $fp = q$ is reused at line number 10. The pointee of $y$ however is now $c$ as the pointee of $z$ has changed. □

### 9.2 Handling Structures, Unions, and Heap Data

We have seen the construction of HRGs for pointers to scalars. In this section, we describe the construction of HRGs for pointers to structures, unions, and heap allocated data. We use allocation site based abstraction for heap in which all locations allocated at a particular allocation site are over-approximated and are treated alike. This approximation allows us to handle the unbounded nature of heap to as if it was bounded. However, since the allocation site might not be available during summary construction phase (because they are occur in the callers), the heap accesses within loop remain unbounded and we need additional summarization techniques to bound them. This section first introduces the concept of indirection lists (indlist) for handling structures and heap accesses which is then followed by an explanation of the summarization technique we have used.

The indlev values $i,j$ of an edge $x \xrightarrow{i,j} y$ represents $i$ dereferences of $x$ and $j$ dereferences of $y$. We can also view the indlev $i,j$ as lists (also referred to as indirection list indlist) containing the dereference operator ($\ast$) of length $i$ and $j$. This representation naturally allows handling structures and heap field sensitively by using indirection lists containing field dereferences. With this view, We can represent the two statements at line numbers 08 and 09 in the example of Figure 19 by HRG edges in the following two ways:
struct node *x, *y;
struct node z;

01 struct node{
02 {
03 struct node *m, *n;
04 };
05 
06 void f()
07 {
08 x = malloc(...);
09 y = x;
10 w = y->n;
11 g();
12 }

13 void g()
14 {
15 while(...) {
16 y = x->m;
17 x = y->n;
18 z.m = x;
19 }
20 }

Figure 19: An example for modelling structures and heap.

- **Field Sensitively.** $y \rightarrow [\ast, \ast] \rightarrow x$ and $w \rightarrow [\ast, \ast, \ast] \rightarrow y$; field sensitivity is achieved by enumerating the field dereferences.
- **Field Insensitively.** $y \rightarrow [\ast] \rightarrow x$ and $w \rightarrow [\ast, n] \rightarrow y$; no distinction made between any field dereference.

The dereference of $y \rightarrow n$ on line 09 is represented by an indlist $[\ast, n]$. The access $z.m$ on line 18 can be treated as a separate variable which is represented by a node $z.m$ with an indlist $[\ast]$ in the HRG. We can also represent $z.m$ with a node $z$ and an indlist $[m]$. For structures and heap, we ensure field sensitivity by maintaining indlist in terms of field names. Unions are handled similarly to structures.

Recall that an edge composition $n \circ p$ involves balancing the indlev of the pivot in $n$ and $p$. With indlist replacing indlev, the operations remain similar in spirit although now they become operations on lists rather than operations on numbers.

**Example 23.** Consider the example in Figure 19. Edge composition $n \circ p$ requires balancing indlevs of the pivot (Section 3.3) which involves computing the difference between the indlev of the pivot in $n$ and $p$. This difference is then added to the indlev of the non-pivot node in $n$ or $p$. Recall that an edge composition is useful (Section 3.4) only when the indlev of the pivot in $n$ is greater than or equal to the indlev of the pivot in $p$. Thus, in our example with $p \equiv y \rightarrow [1, 1] \rightarrow x$ and $n \equiv w \rightarrow [1, 2] \rightarrow y$ with $y$ as pivot, an edge composition is useful because indlev of $y$ in $n$ (which is 2) is greater than indlev of $y$ in $p$ (which is 1). The difference (2-1) is added to the indlev of $x$ (which is 1) resulting in an reduced edge $r \equiv w \rightarrow [1, (2-1+1)] \rightarrow x$. □

Analogously we can define similar operations for indlist. An edge composition is useful if the indlist of the pivot in edge $p$ is the prefix of the indlist of the pivot in edge $n$. In our example, the indlist of $y$ in $p$ (which is $[\ast]$) is prefix of the indlist of $y$ in $n$ (which is $[\ast, n]$) and hence the edge composition is useful.

The operation of computing the difference in the indlev of the pivot is replaced by the remainder operation remainder : indlist $\times$ indlist $\rightarrow$ indlist which takes two indlists as its arguments where one is prefix of the other and returns the suffix of the indlist which does not match with the other. Finally, the addition of the difference in the indlevs of the pivot to the indlev of one of the other two nodes is replaced by an append

---

This does not matter for the first edge but matters for the second edge.
operation which is denoted by \#. The definitions of the operations on \textit{indlist} are given below:

\begin{align*}
\text{prefix}(il_1, il_2) & := il_2 = il_1 \# il_3 & (10) \\
\text{remainder}(il_1, il_2) & := \begin{cases} il_3 & \text{prefix}(il_1, il_2) \land il_2 = il_1 \# il_3 \\
\epsilon & \text{Otherwise} \end{cases} & (11)
\end{align*}

**Example 24.** In our example, since \textit{prefix}([*], [*, n]) returns \textit{True}, \textit{remainder}([*], [*, n]) returns [n] and this \textit{indlist} is appended to the \textit{indlist} of node \textit{x} (which is [*]) resulting in a new \textit{indlist} [*, # n] = [*, n] and a reduced edge \( w \xrightarrow{[*], [*, n]} x \). □

We use allocation site based abstraction for heap. Thus, line number 07 of procedure \textit{f} can be viewed as an HRG edge \( x \xrightarrow{1,0} \text{heap}_{07} \) where heap_{07} is the heap location create at this allocation site. We expect the heap to be bounded by this abstraction but the allocation site may not be available during the summary construction as is the case in our example where heap is accessed through pointers \textit{x} and \textit{y} in a loop in procedure \textit{g} whereas allocation site is available in procedure \textit{f} at line number 07.

**Example 25.** The fixed point computation for the loop in procedure \textit{g} will never terminate as the length of the indirection list keeps on increasing. In the first iteration of the loop, at its exit, the edge composition results into a reduced edge \( x \xrightarrow{[*, [*, m, n]]} y \). In the next iteration, the reduced edge is now \( x \xrightarrow{[*, [*, m, m, n]]} y \) indicating the access pattern of heap. This continues as the length of the indirection list keeps on increasing leading to a non-terminating sequence of computations. Heap access where the allocation site is locally available does not face this problem of non-termination. □

This indicates the need of additional summarization techniques. We bound the indirection lists by \( k\)-m-limiting technique which limits the length of indirection lists to \( k \) and maintains field sensitivity up to \( m \leq k \) dereferences. All dereferences beyond \( k \) are treated field insensitively and all dereferences beyond \( k \) are treated as an unbounded number of field insensitive dereferences.

### 9.3 Using SSA Form for Compact HRGs

Although the Static Single Assignment (SSA) form is not a language feature, it is ubiquitous in any real IR of practical programs. In this section we show how we have used the SSA productively to make our analysis more efficient and construct compact HRGs.

SSA form makes use-def chains explicit in the IR because every use has exactly one definition reaching it and every definition dominates all its uses. Thus for every local non-address taken variable, we traverse the SSA chains transitively until we reach a statement whose right hand side has an address taken variable, a global variable, or a formal parameter. In the process, all definitions involving SSA variables on the left hand side are ignored.

\[ s_1: x_\_1 = \&a; \]
\[ s_2: y = x_\_1; \]

**Example 26.** Consider, the above code snippet, the HRG edge \( x_\_1 \xrightarrow{1,0} a \) corresponding to statement \( s_1 \) is not added to the HRG. Statement \( s_2 \) defines a global pointer \textit{y} which is assigned the pointee of \textit{x}_{-1} (use of \textit{x}_{-1}). The explicit use of use-def chain helps to identify the pointee of \textit{x}_{-1} even though there is no corresponding edge in the HRG. SSA resolution leads to an edge \( y \xrightarrow{1,0} a \) which the desired result, also indicating the fact that SSA resolution is similar to edge composition. □

The use of SSA has the following two advantages:

- The HRG size is small because local variables are eliminated.
• No special filtering required for eliminating local variables from the summary flow function of a procedure. These local variables are not in the scope of the callers and hence should be eliminated before a summary flow function is used at its call sites.

9.4 Handling Arrays, Pointer Arithmetic, and Address Escaping Locals

An array is treated as a single variable in the following sense: Accessing a particular element is seen as accessing every possible element and updates are treated as weak updates. This applies to both the situations: when arrays of pointers are manipulated, as well as when arrays are accessed through pointers. Arrays are maintained flow insensitively by our analysis.

For pointer arithmetic, we approximate the pointer being defined to point to every possible location. All address taken local variables are treated as globals because they can escape the scope of the procedure.

10 Related Work

Section 2.1 introduced two broad categories of constructing summary flow functions for pointer analysis. Some methods using placeholders do not make any assumptions about the calling contexts and hence, construct context independent summaries [12, 13, 21, 24, 25]. Introducing placeholders leads to larger summary flow functions causing inefficiency in fixed point computation at the intraprocedural level thereby prohibiting flow sensitivity for scalability. Also, these methods cannot distinguish between may and must information and do not perform strong updates thereby losing precision.

Among the general frameworks for constructing procedure summaries, the formalism proposed by Sharir and Pnueli [22] is limited to finite lattices of data flow values. It was implemented using graph reachability in [14, 19, 20]. A general technique for constructing procedure summaries [5] has been applied to unary uninterpreted functions and linear arithmetic. However, the presented program model does not include pointers.

Symbolic procedure summaries [26, 28] involve computing preconditions and corresponding postconditions (in terms of aliases). A calling context is matched against a precondition and the corresponding postcondition gives the result. However, the number of calling contexts in a program could be unbounded hence constructing summaries for all calling contexts could lose scalability. This method requires statement level transformers to be closed under composition; a requirement which is not satisfied by pointer analysis. For example, composition of two loads involving context dependent values would result in a transformer in three unknowns which does not belong to the basic transformers of pointer analysis. We overcome this problem using higher order paths. Saturn [6] also creates such summaries which are sound but may not be precise across applications because they depend on context information.

Some approaches use customized summaries and combine the top-down and bottom-up analyses to construct summaries for only those calling contexts that occur in a given program [29]. This choice is controlled by the number of times a procedure is called. If this number exceeds a fixed threshold, a summary is constructed using the information of the calling contexts that have been recorded for that procedure. A new calling context may lead to generating a new precondition and hence a new summary.

Indirect calls such as calls through function pointers or virtual calls for bottom-up approaches are handled in the following two ways. Special care is taken when the pointees of the function pointer or the type of receiver object is not available locally but depends on the calling context.

• All the procedures whose type matches with the type of function pointer through which call is made are assumed to be potential callees of that call. The summary flow function of all such procedures are applied at the call site and then merged resulting in imprecise results. Such an approach is context
insensitive in presence of function pointers. In case of virtual calls, class hierarchy analysis (CHA) is
used for identifying the callees.

- In the second approach, when sufficient information about the indirect call is not available, then the
construction of summary flow function is suspended and it is resumed only when the information is
discovered by gathering the information from the calling context which is achieved by moving up the
call graph.

## 11 Implementation and Measurements

We have implemented HRG based points-to analysis in GCC 4.7.2 using the GCC’s LTO framework and
have carried out measurements on SPEC CPU2006 benchmarks on a machine with 16 GB RAM with 8
64-bit Intel i7-4770 CPUs running at 3.40GHz.

Our method eliminates local variables using the SSA form and $\overline{\Delta}$ are computed for global variables
alone. However, points-to information is computed for local variables also. Arrays are approximated by
treating them as a single variable and hence an access to one element of an array is considered as an access
of the entire array. This forces us to perform weak updates in case of arrays and hence we maintain their
information flow insensitively. SSA variables are maintained flow insensitively. We approximate heap by
using $k$-limiting technique and our implementation maintains the indirection lists of field dereferences of
length $k = 2$.

Unlike the conventional approaches [26,28], our analysis computes $\overline{\Delta}$ that does not depend on the aliases
in the calling context. The actually observed number of aliasing patterns (Table 2) suggests that it is unde-
sirable to construct multiple PTFs for a procedure indiscriminately.

| Program   | kLoC | # Ptr stmts ($\times 10^3$) | Time in seconds for | HRG Constr. | HRG + NB | HRG + B | Stmt ff + NB | Stmt ff + B |
|-----------|------|----------------------------|---------------------|------------|----------|---------|--------------|-------------|
| lbm       | 0.9  | 0.37                       | 0.10                | 0.22       | 0.21     | 0.26    | 0.28         |
| mcf       | 1.6  | 0.48                       | 75.29               | 33.73      | 30.05    | 1.25    | 0.91         |
| libquantum| 2.6  | 0.34                       | 6.47                | 10.23      | 1.95     | 8.21    | 1.85         |
| bzip2     | 5.7  | 1.65                       | 3.17                | 11.11      | 8.71     | 4.73    | 3.30         |
| milc      | 9.5  | 2.54                       | 7.36                | 6.08       | 5.89     | 4.29    | 5.61         |
| sjeng     | 10.5 | 0.70                       | 9.36                | 39.66      | 25.75    | 14.75   | 7.56         |
| hmmer     | 20.6 | 6.79                       | 38.23               | 51.73      | 14.86    | 31.32   | 13.50        |
| h264ref   | 36.1 | 17.77                      | 208.47              | 1262.07    | 199.34   | 457.26  | 74.62        |
| gobmk     | 158.0| 212.83                     | 652.78              | 3652.99    | 1624.46  | 1582.62 | 1373.88      |

Table 1: Time Measurements. Data flow values can be computed using: Stmt ff (Statement level flow
function), B (Bypassing), NB (No Bypassing).
Program | # of call sites | # of procs. | Proc. count for different buckets of # of calls (reuse of HRGs) | # of procs. requiring different no. of PTFs based on the no. of aliasing patterns | # of procs. for different sizes of HRG in terms of the number of edges | # of procs. for different % of context ind. info. (for non-empty HRGs)
---|---|---|---|---|---|---
| | | | | | | |
| lbm | 30 | 19 | 5 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 13 | 0 | 13 | 4 | 2 | 0 | 0 | 0 | 3 | 0 | 0 | 3 |
| mcf | 29 | 23 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 10 | 5 | 2 | 3 | 2 | 1 | 5 | 1 | 1 | 6 |
| libquantum | 277 | 80 | 24 | 1 | 11 | 4 | 3 | 7 | 3 | 1 | 0 | 14 | 4 | 42 | 10 | 7 | 12 | 9 | 0 | 20 | 12 | 1 | 5 |
| bzip2 | 288 | 89 | 35 | 7 | 2 | 1 | 22 | 0 | 0 | 0 | 28 | 2 | 62 | 13 | 4 | 5 | 5 | 0 | 26 | 0 | 0 | 1 |
| mlibc | 782 | 190 | 60 | 15 | 9 | 1 | 37 | 8 | 0 | 1 | 35 | 25 | 157 | 11 | 19 | 2 | 7 | 0 | 6 | 10 | 9 | 14 |
| sjeng | 726 | 133 | 46 | 20 | 5 | 6 | 14 | 3 | 1 | 3 | 10 | 14 | 99 | 20 | 6 | 3 | 5 | 0 | 3 | 4 | 10 | 17 |
| hmer | 1328 | 275 | 93 | 33 | 22 | 11 | 62 | 5 | 3 | 4 | 88 | 32 | 167 | 56 | 20 | 15 | 15 | 2 | 54 | 20 | 11 | 23 |
| h264ref | 2393 | 566 | 171 | 60 | 22 | 16 | 85 | 17 | 5 | 3 | 102 | 46 | 419 | 76 | 23 | 15 | 30 | 3 | 54 | 13 | 27 | 53 |
| gobmk | 9379 | 2697 | 317 | 110 | 99 | 134 | 206 | 30 | 9 | 10 | 210 | 121 | 1374 | 93 | 8 | 1083 | 97 | 42 | 41 | 1192 | 39 | 51 |

Table 2: Sizes and Effectiveness of \( \Delta \) and the Statistics of the Benchmarks used.

| Program | # of points-to pairs (G+NB) | # of points-to pairs (L+NB) | # of points-to pairs (G+B) | # of points-to pairs (L+B) |
|---|---|---|---|---|
| | 0 | 1-2 | 3-4 | 5-8 | 9-50 | 50+ | 0 | 1-2 | 3-4 | 5-8 | 9-50 | 50+ | 0 | 1-2 | 3-4 | 5-8 | 9-50 | 50+ |
| lbm | 0 | 10 | 8 | 1 | 0 | 0 | 3 | 9 | 6 | 0 | 1 | 0 | 6 | 11 | 2 | 0 | 0 | 0 | 3 | 10 | 5 | 0 | 1 | 0 |
| mcf | 0 | 0 | 0 | 0 | 22 | 1 | 3 | 3 | 2 | 13 | 0 | 9 | 8 | 1 | 1 | 4 | 0 | 8 | 5 | 1 | 2 | 7 | 0 |
| libquantum | 4 | 4 | 1 | 0 | 27 | 44 | 60 | 14 | 5 | 0 | 0 | 1 | 37 | 13 | 0 | 14 | 16 | 0 | 61 | 15 | 4 | 0 | 0 | 0 |
| bzip2 | 13 | 0 | 0 | 0 | 76 | 0 | 59 | 11 | 12 | 4 | 3 | 0 | 67 | 8 | 4 | 4 | 6 | 0 | 59 | 11 | 12 | 4 | 3 | 0 |
| milc | 8 | 1 | 6 | 5 | 170 | 0 | 61 | 59 | 26 | 21 | 20 | 3 | 136 | 26 | 20 | 6 | 8 | 0 | 83 | 56 | 25 | 21 | 9 | 2 |
| sjeng | 12 | 3 | 5 | 1 | 29 | 83 | 80 | 21 | 10 | 10 | 12 | 0 | 101 | 17 | 6 | 2 | 7 | 0 | 89 | 16 | 12 | 8 | 8 | 0 |
| hmer | 51 | 1 | 1 | 8 | 63 | 151 | 72 | 97 | 31 | 35 | 35 | 5 | 152 | 55 | 22 | 22 | 23 | 1 | 72 | 97 | 30 | 40 | 32 | 4 |
| h264ref | 38 | 0 | 5 | 4 | 30 | 489 | 113 | 127 | 86 | 59 | 147 | 34 | 412 | 62 | 35 | 28 | 25 | 4 | 231 | 120 | 56 | 71 | 75 | 13 |
| gobmk | 1884 | 0 | 0 | 0 | 20 | 793 | 2280 | 264 | 65 | 59 | 28 | 1 | 2370 | 89 | 22 | 82 | 117 | 17 | 2341 | 224 | 50 | 56 | 24 | 2 |

Table 3: Number of points-to pairs with and without Bypassing (B and NB). G stands for Global variables and L stands for Local variables.
### Table 4: Comparing average number of points-to pairs and number of pointees for dereferences. G (Global variables), L (Local variables), Arr (Arrays), NB (No Bypassing), B (Bypassing).

| Program  | Avg. # of pointees per pointer | # of inconclusive compositions |
|----------|-------------------------------|--------------------------------|
|          | avg. # of pointees per dereference | GCC | LFCPA | HRG | GCC | LFCPA | HRG | GCC | LFCPA |
|          | HRG | LFCPA | LFCPA | HRG | GCC | LFCPA | HRG | GCC | LFCPA |
| lbm      | 1.31 | 1.42  | 2.21  | 17.74 | 0.05 | 1.09  | 2.25 | 1.50 | 0.00   |
| mcf      | 18.73 | 6.10  | 10.48 | 34.74 | 1.22 | 4.25  | 2.57 | 0.62 | 0.94   |
| libquantum | 139.50 | 22.50 | 1.11  | 4.49  | 3.34 | 1.50  | 2.93 | 0.83 | 0.00   |
| gzip2    | 43.39 | 8.38  | 1.89  | 31.46 | 0.94 | 1.72  | 2.94 | 0.33 | 1.30   |
| mlc      | 21.15 | 16.32 | 4.52  | 14.06 | 31.73 | 1.18  | 2.58 | 1.61 | 0.00   |
| sjeng    | 445.22 | 64.81  | 3.07  | 2.68  | 0.98  | 2.71  | 0.00  | 0.00  | 4.00   |
| hminer   | 43.49 | 5.85  | 6.00  | 59.35 | 1.56 | 1.04  | 3.62 | 0.91 | 4.00   |
| h264ref  | 279.71 | 9.24  | 16.29 | 98.84 | 0.98 | 3.97  | 0.00  | 0.00  | 8.00   |
| gobmk    | 11.98 | 1.73  | 6.34  | 4.08  | -     | 0.65  | 3.71 | -    | 0.00   |

The effectiveness of a summary flow function based approach depends on three important factors:

- compactness of HRGs
- percentage of points-to information that does not depend on the calling contexts, and
- reusability.

Table 2 gives the statistics of the effectiveness of $\Delta$ discovered in the benchmark programs. The $\Delta$ for a significant number of procedures is the identity flow function indicated by empty HRGs. Besides, in six out of nine benchmarks, most procedures have a significantly high amount of context independent information. Further, our measurements indicate that many procedures have many calls indicating a high reuse of $\Delta$. Thus, our data confirms the usefulness of constructing summary flow function $\Delta$.

Interestingly, computing points-to information using HRGs seems to take much more time than constructing HRGs. As discussed in Section 7, computing points-to information at every program point within a procedure using the $Bl$ of the procedure and the summary flow function ($\Delta$) is expensive because of the cumulative effect of the $\Delta$. Table 1 gives the time measurements which confirms the observation that application of statement level flow functions is much more efficient than the application of $\Delta$ for computing points-to information at every program point.

Redundant computations can also be avoided by filtering out the points-to pairs reaching a call but not accessed in the callee procedure. This concept of bypassing has been used effectively for data flow values of scalars but requires externally supplied points-to information for handling pointers [16, 17]. HRGs supports bypassing inherently as explained in Section 7. The efficiency achieved by bypassing is evident from the time measurements in Table 1 and also by the number of points-to pairs computed at every program point from the Table 3. The number of points-to pairs per statement given in Table 4 indicates that bypassing reduces the number of points-to pairs at a program point significantly. We have applied the technique of bypassing only to the flow sensitive points-to information.

We have also compared our analysis with GCC and LFCPA [11] by computing the number of point-to pairs per assignment for flow sensitive data and the number of point-to pairs per function for flow insensitive data. The number of points-to pairs per function for GCC is large because it is flow and context insensitive. The number of points-to pairs per statements is much smaller for LFCPA as it is liveness based. However, LFCPA does not seem to scale beyond 35 kLoC (to the best of our knowledge, LFCPA represents the state of the art in fully flow and context sensitive exhaustive points-to analysis). We have also computed the average of number of pointees of dereferenced variables which is maximum for GCC as shown in Table 4.
As discussed in Section 3.5, there are instances when the $n$ cannot be replaced by its reduced edge $r$. We observed that practically there are very few instances when such a situation arises and we have recorded these numbers in Table 4. All of them are in single digits.

12 Conclusions and Future Work

Constructing bounded summary flow functions for flow and context sensitive points-to analysis seems hard because of indirectly accessed pointees whose information is not available during summary construction. Conventionally, they have been handled by using placeholders. However, a fundamental problem with this is that the placeholders explicate the locations by naming them. This results in either (a) a large number of placeholders, or (b) multiple summary flow functions for different aliasing patterns in the calling contexts.

We propose the concept of higher order reachability which allows us to relate (transitively indirect) pointees of a variable with those of other variables. A simple arithmetic on indirection levels allows unknown locations to be left implicit and facilitates composition of higher order paths to find direct relationships without the need of placeholders. Since the locations are left implicit, no information about aliasing patterns in the calling contexts is required and we construct a single summary flow function (HRG) per procedure. HRGs are bounded by the number of variables, are flow sensitive, and are able to perform strong updates within calling contexts. Further, HRGs inherently lift the bypassing of irrelevant scalar values across procedure calls to bypassing of points-to information thereby aiding scalability significantly. Our measurements on SPEC benchmarks show that HRGs are small enough to scale fully flow and context sensitive exhaustive points-to analysis to programs as large as 158 kLoC (as compared to 35 kLoC of LFCPA [11]). In future, we expect to scale the method to still larger programs by using memoisation, and constructing and applying HRGs incrementally thereby eliminating redundancies across fixed point computations.

The concept of higher order reachability lifts the classical graph reachability to an abstraction of memory involving pointers. The way matrices represent values as well as transformations, HRGs represent memory as well as memory manipulations performed by a program in terms of loading, storing, and copying memory addresses. Any analysis that is influenced by these operations may be able to use HRGs by combining them with the original abstractions of the analysis. We plan to explore this direction in future.

References

[1] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah and Marina Blanton, editors, Algorithms and Theory of Computation Handbook, pages 9.1–9.14. Chapman & Hall/CRC, 2010.

[2] Camil Demetrescu and Giuseppe F. Italiano. Maintaining dynamic matrices for fully dynamic transitive closure. Algorithmica, 51(4):387–427, May 2008.

[3] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’08, pages 270–280, New York, NY, USA, 2008. ACM.

[4] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up context-sensitive pointer analysis for java. In Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings, pages 465–484, 2015.

[5] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In R. De Nicola, editor, European Symp. on Programming, ESOP 2007, volume 4421 of LNCS, pages 253–267, 2007.
[6] B. Hackett and A. Aiken. How is aliasing used in systems software? In Proc. 14th FSE, 2006.

[7] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI ’01, pages 24–34, New York, NY, USA, 2001. ACM.

[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. SIGACT News, 32(1):60–65, March 2001.

[9] U. P. Khedker, A. Sanyal, and B. Sathe. Data Flow Analysis: Theory and Practice. Taylor & Francis (CRC Press, Inc.), Boca Raton, FL, USA, 2009.

[10] Uday P. Khedker and Bageshri Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: resurrecting the classical call strings method. In Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction, CC’08/ETAPS’08, pages 213–228, 2008.

[11] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Liveness-based pointer analysis. In Proceedings of the 19th International Conference on Static Analysis, SAS’12, pages 265–282, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Lian Li, Cristina Cifuentes, and Nathan Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In Proceedings of the 2013 International Symposium on Memory Management, ISMM ’13, pages 85–96, New York, NY, USA, 2013. ACM.

[13] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. Modular heap analysis for higher-order programs. In Proceedings of the 19th International Conference on Static Analysis, SAS’12, pages 370–387, Berlin, Heidelberg, 2012. Springer-Verlag.

[14] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the ifds algorithm. In Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction, CC’10/ETAPS’10, pages 124–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings, pages 165–180, 2004.

[16] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for c-like languages. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12, Beijing, China - June 11 - 16, 2012, pages 229–238, 2012.

[17] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014, page 49, 2014.

[18] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in soot using value contexts. In Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis, SOAP ’13, pages 31–36, New York, NY, USA, 2013. ACM.
[19] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’95, pages 49–61, New York, NY, USA, 1995. ACM.

[20] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci., 167(1&2):131–170, 1996.

[21] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO ’12, pages 264–274, New York, NY, USA, 2012. ACM.

[22] A. Sharir M., Pnueli. Two approaches to interprocedural data flow analysis. S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, (ch. 7), pages 189–234, 1981.

[23] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA ’05, pages 59–76, New York, NY, USA, 2005. ACM.

[24] Ru D. Sălcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In In VMCAI. Springer-Verlag, 2005.

[25] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA ’99, pages 187–206, New York, NY, USA, 1999. ACM.

[26] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 1–12, 1995.

[27] Dacong Yan, Guoqing Xu, and Atanas Rountev. Rethinking soot for summary-based whole-program analysis. In Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP ’12, pages 9–14, New York, NY, USA, 2012. ACM.

[28] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 221–234, New York, NY, USA, 2008. ACM.

[29] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, pages 249–258, New York, NY, USA, 2014. ACM.