Flow- and Context-Sensitive Points-To Analysis Using Generalized Points-To Graphs

Pritam M. Gharat[1], Uday P. Khedker[1(✉)], and Alan Mycroft[2]

[1] Indian Institute of Technology Bombay, Mumbai, India
{pritamg,uday}@cse.iitb.ac.in
[2] University of Cambridge, Cambridge, UK
am@cl.cam.ac.uk

Abstract. Bottom-up interprocedural methods of program analysis construct summary flow functions for procedures to capture the effect of their calls and have been used effectively for many analyses. However, these methods seem computationally expensive for flow- and context-sensitive points-to analysis (FCPA) which requires modelling unknown locations accessed indirectly through pointers. Such accesses are commonly handled by using placeholders to explicate unknown locations or by using multiple call-specific summary flow functions. We generalize the concept of points-to relations by using the counts of indirection levels leaving the unknown locations implicit. This allows us to create summary flow functions in the form of *generalized points-to graphs* (GPGs) without the need of placeholders. By design, GPGs represent both memory (in terms of classical points-to facts) and memory transformers (in terms of generalized points-to facts). We perform FCPA by progressively reducing generalized points-to facts to classical points-to facts. GPGs distinguish between *may* and *must* pointer updates thereby facilitating strong updates within calling contexts.

The size of GPGs is linearly bounded by the number of variables and is independent of the number of statements. Empirical measurements on SPEC benchmarks show that GPGs are indeed compact in spite of large procedure sizes. This allows us to scale FCPA to 158 kLoC using GPGs (compared to 35 kLoC reported by liveness-based FCPA). Thus GPGs hold a promise of efficiency and scalability for FCPA without compromising 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. Computationally intensive analyses such as model checking are ineffective on programs containing pointers partly because of imprecision of pointer analyses [1,8].

We focus on exhaustive (as against demand driven [2,7,22]) points-to analysis with full flow- and context-sensitivity for precision. A top-down context sensitive

P.M. Gharat—Partially supported by a TCS Fellowship.

X. Rival (Ed.): SAS 2016, LNCS 9837, pp. 212–236, 2016.
DOI: 10.1007/978-3-662-53413-7_11

analysis propagates the information from callers to callees [28] thereby analyzing a procedure each time a new data flow value reaches its call(s). Some approaches in this category are: call strings method [21], its value-based variants [10,17] and the tabulation based functional method [18,21]. By contrast, bottom-up approaches avoid analyzing callees multiple times by constructing *summary flow functions* which are used at call sites to incorporate the effect of procedure calls [3,6,13,19,21,23–28].

It is prudent to distinguish between three kinds of summaries (see [4] for examples) that can be created for a procedure: *(a)* a bottom-up parameterized summary flow function which is context independent, *(b)* a top-down enumeration of summary flow function in the form of input-output pairs for the input values reaching a procedure, and *(c)* a bottom-up parameterless (and hence context-insensitive) summary information. Context independence (in *(a)* above), achieves context-sensitivity through parameterization and should not be confused with context-insensitivity (in *(c)* above).

We focus on summaries of the first kind. Their construction requires *composing* statement-level flow functions to represent a sequence of statements, and *merging* the composed functions to represent multiple control flow paths reaching a program point. These summaries should be compact and their size should be independent of the number of statements. This seems hard because of the presence of indirect pointees. The composition of the flow functions for a sequence of statements $x = *y$; $z = *x$ cannot be reduced to a flow function of the basic pointer assignments for 3-address code ($x = \&y$, $x = y$, $x = *y$, and $*x = y$).

Our Key Idea and Approach. We generalize the concept of points-to relations by using the counts of indirection levels leaving the unknown locations implicit. This allows us to create summary flow functions in the form of *generalized points-to graphs* (GPGs) whose size is linearly bounded by the number of variables (Sect. 2). By design, GPGs can represent both memory (in terms of classical points-to facts) and memory transformers (in terms of generalized points-to facts).

Example 1. Consider procedure g of Fig. 1 whose GPG is shown in Fig. 2(c). The edges in GPGs track indirection levels: indirection level 1 in the label (1,0) indicates that the source is assigned the address (indicated by indirection level 0) of the target. Edge $a \xrightarrow{1,0} e$ is created for line 8. 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 combined effect of lines 13 (edge $y \xrightarrow{1,0} b$) and 17 (edge $y \xrightarrow{2,0} d$) results in the edge $b \xrightarrow{1,0} d$. However edge $y \xrightarrow{2,0} d$ is also retained because there is no information about the pointee of y along the other path reaching line 17. □

The generalized points-to facts are composed to create new generalized points-to facts with smaller indirection levels (Sect. 3) whenever possible thereby converting them progressively to classical points-to facts. This is performed in

```
01 void f()
02 {  x = &a;
03    z = &w;
04    g();
05    *x = z;
06 }
07 void g()
08 {  a = &e;
09    if (...) {
10       *x = z;
11        z = &u;
12    } else
13    {  y = &b;
14       z = &v;
15    }
16    x = &b;
17    *y = &d;
18 }
```

Procs. *g* and *f* are used for illustrating intraprocedural and interprocedural GPG construction respectively.

Fig. 1. A program fragment used as a running example through the paper. All variables are global.

two phases: construction of GPGs, and use of GPGs to compute points-to information. GPGs are constructed flow-sensitively by processing pointer assignments along the control flow of a procedure and collecting generalized points-to facts (Sect. 4).

Function calls are handled context-sensitively by incorporating the effect of the GPG of a callee into the GPG of the caller (Sect. 5). Loops and recursion are handled using a fixed point computation. GPGs also distinguish between *may* and *must* pointer updates thereby facilitating strong updates.

Section 6 shows how GPGs are used for computing classical points-to facts. Section 7 presents the empirical measurements. Section 8 describes the related work. Section 9 concludes the paper. A detailed technical report [4] describes how we handle advanced issues (e.g. structures, heap memory, function pointers, arrays, pointer arithmetic) and also provides soundness proofs.

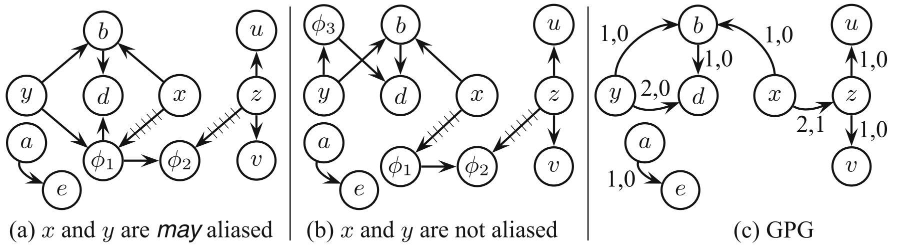
The Advantages of GPGs Over Conventional Summaries. Indirect accesses of unknown locations have been commonly modelled using *placeholders* (called extended parameters in [25] and external variables in [13]).

The *partial transfer function* (PTF) based method [25] uses placeholders to construct a collection of PTFs for a procedure for different aliasing patterns involving formal parameters and global variables accessed in the procedure.

Example 2. For procedure g of the program in Fig. 1, three placeholders ϕ_1, ϕ_2, and ϕ_3 have been used in the PTFs shown in Figs. 2(a) and (b). The possibility that x and y may or may not be aliased gives rise to two PTFs. □

The number of PTFs could be combinatorial in the number of dereferences of globals and parameters. PTFs that do not correspond to actual aliasing patterns can be excluded by combining a top-down analysis for discovering aliasing patterns with the bottom-up construction of PTFs [25,28]. Yet, the number of PTFs could remain large.

An alternative approach makes no assumption about aliases in the calling context and constructs a single summary flow function for a procedure. In a degenerate case, it may require a separate placeholder for the same variable in different statements and the size of the summary flow functions may be proportional to the number of statements.



(a) x and y are *may* aliased (b) x and y are not aliased (c) GPG

Fig. 2. PTFs/GPG for proc. g of Fig. 1. Edges deleted due to flow-sensitivity are struck off.

Example 3. For the code snippet on the right, we need two different placeholders for y in statements s_1 and s_3 because statement s_2 could change the pointee of y depending upon whether $*z$ is aliased to y. □

s_1: $x = *y;$
s_2: $*z = q;$
s_3: $p = *y;$

Separate placeholders for different occurrences of a variable can be avoided if points-to information is not killed by the summary flow functions [13,23,24]. Another alternative is to use flow-insensitive summary flow functions [3]. However, both these cases introduces imprecision.

A fundamental problem with placeholders is that they explicate unknown locations by naming them, resulting in either a large number of placeholders (e.g., a GPG edge $\cdot \xrightarrow{i,j} \cdot$ would require $i + j - 1$ placeholders) or multiple summary flow functions for different aliasing patterns that exist in the calling contexts.

Since we use edges to track indirection levels leaving unknown locations implicit: *(a)* placeholders are not needed (unlike [13,23–25,28]), *(b)* aliasing patterns from calling contexts are not needed and a single summary per procedure is created (unlike [25,28]), *(c)* the size of summary is linearly bounded by the number of variables regardless of the number of statements (unlike [13,23,24]), and *(d)* updates can be performed in the calling contexts (unlike [3,13,23,24]). This facilitates the scalability of fully flow- and context-sensitive exhaustive points-to analysis.

2 Generalized Points-To Graphs (GPGs)

We define the basic concepts assuming scalars and pointers in the stack and static memory; see [4] for extensions to handle structures, heap, function pointers, etc.

2.1 Memory and Memory Transformer

We assume a control flow graph representation containing 3-address code statements. Program points ***t***, ***u***, ***v*** represent the points just before the execution of statements. The successors and predecessors of a program point are denoted by *succ* and *pred*; succ^* pred^* denote their reflexive transitive closures. A *control* *flow path* is a finite sequence of (possibly repeating) program points $q_0, q_1, \ldots, q_m$ such that $q_{i+1} \in \mathsf{succ}(q_i)$.

Let L and $\mathsf{P} \subseteq \mathsf{L}$ denote the sets of locations and pointers respectively. Every location has a content and an address. The *memory* at a program point is a relation $M \subseteq \mathsf{P} \times (\mathsf{L} \cup \{?\})$ where "?" denotes an undefined location. We view M as a graph with $\mathsf{L} \cup \{?\}$ as the set of nodes. An edge $x \rightarrow y$ in M indicates that $x \in \mathsf{P}$ contains the address of location $y \in \mathsf{L}$. The memory associated with a program point $\boldsymbol{u}$ is denoted by $M_{\boldsymbol{u}}$; since $\boldsymbol{u}$ could appear in multiple control flow paths and could also repeat in a control flow path, $M_{\boldsymbol{u}}$ denotes the memory associated with all occurrences of $\boldsymbol{u}$.

The pointees of a set of pointers $X \subseteq \mathsf{P}$ in M are computed by the application $M\ X = \{y \mid (x, y) \in M, x \in X\}$. A composition of degree i, $M^i\{x\}$ discovers the i^{th} pointees of x which involves i transitive reads from x: first $i - 1$ addresses are read followed by the content of the last address. For composability of M, we extend its domain to $\mathsf{L} \cup \{?\}$ by inclusion map. By definition, $M^0\{x\} = \{x\}$.

For adjacent program points $\boldsymbol{u}$ and $\boldsymbol{v}$, $M_{\boldsymbol{v}}$ is computed from $M_{\boldsymbol{u}}$ by incorporating the effect of the statement between $\boldsymbol{u}$ and $\boldsymbol{v}$, $M_{\boldsymbol{v}} = (\delta(\boldsymbol{u}, \boldsymbol{v}))\,(M_{\boldsymbol{u}})$ where $\delta(\boldsymbol{u}, \boldsymbol{v})$ is a *statement-level flow function* representing a *memory transformer*. For $\boldsymbol{v} \in \mathsf{succ}^*(\boldsymbol{u})$, the effect of the statements appearing in all control flow paths from $\boldsymbol{u}$ to $\boldsymbol{v}$ is computed by $M_{\boldsymbol{v}} = (\Delta(\boldsymbol{u}, \boldsymbol{v}))\,(M_{\boldsymbol{u}})$ where the memory transformer $\Delta(\boldsymbol{u},\boldsymbol{v})$ is a *summary flow function* mapping the memory at $\boldsymbol{u}$ to the memory at $\boldsymbol{v}$. Definition 1 provides an equation to compute Δ without specifying a representation for it. Since control flow paths may contain cycles, Δ is the maximum fixed point of the equation where *(a)* the composition of Δs is denoted by $\circ$ such that $(g \circ f)(\cdot) = g(f(\cdot))$, *(b)* Δs are merged using $\sqcap$, *(c)* B captures the base case, and *(d)* Δ_{id} is the identity flow function. Henceforth, we use the term memory transformer for a summary flow function Δ. The rest of the paper proposes GPG as a compact representation for Δ. Section 2.2 defines GPG and Sect. 2.3 defines its lattice.

Definition 1: Memory Transformer Δ

$$\Delta(\boldsymbol{u}, \boldsymbol{v}) := \mathsf{B}(\boldsymbol{u}, \boldsymbol{v}) \ \sqcap \prod_{\substack{\boldsymbol{t} \,\in\, \mathsf{succ}^*(\boldsymbol{u}) \\ \boldsymbol{v} \,\in\, \mathsf{succ}(\boldsymbol{t})}} \delta(\boldsymbol{t}, \boldsymbol{v}) \circ \Delta(\boldsymbol{u}, \boldsymbol{t})$$

$$\mathsf{B}(\boldsymbol{u}, \boldsymbol{v}) := \begin{cases} \Delta_{id} & \boldsymbol{v} = \boldsymbol{u} \\ \delta(\boldsymbol{u}, \boldsymbol{v}) & \boldsymbol{v} \in \mathsf{succ}(\boldsymbol{u}) \\ \emptyset & \text{otherwise} \end{cases}$$

2.2 Generalized Points-To Graphs for Representing Memory Transformers

The classical memory transformers explicate the unknown locations using placeholders. Effectively, they use a low level abstraction which is close to the memory defined in terms of classical points-to facts: Given locations $x, y \in \mathsf{L}$, a classical points-to fact $x \rightarrow y$ in memory M asserts that x holds the address of y. We propose a higher level abstraction of the memory without explicating the unknown locations.

Pointer assignment	Pointers defined	Pointees	GPG edge	Pointers over-written	Effect on M after the assignment
$x = \&y$	$M^0\{x\}$	$M^0\{y\}$	$x \xrightarrow{1,0} y$	$M^0\{x\}$	$M^1\{x\} = M^0\{y\}$
$x = y$	$M^0\{x\}$	$M^1\{y\}$	$x \xrightarrow{1,1} y$	$M^0\{x\}$	$M^1\{x\} = M^1\{y\}$
$x = *y$	$M^0\{x\}$	$M^2\{y\}$	$x \xrightarrow{1,2} y$	$M^0\{x\}$	$M^1\{x\} = M^2\{y\}$
$*x = y$	$M^1\{x\}$	$M^1\{y\}$	$x \xrightarrow{2,1} y$	$M^1\{x\}$ or none	$M^2\{x\} \supseteq M^1\{y\}$

Fig. 3. GPG edges for basic pointer assignments in C.

Definition 2: *Generalized Points-to Graph (GPG).* Given locations $x, y \in L$, a *generalized points-to fact* $x \xrightarrow{i,j} y$ in a given memory M asserts that every location reached by $i - 1$ dereferences from x can hold the address of every location reached by j dereferences from y. Thus, $M^i\{x\} \supseteq M^j\{y\}$. A *generalized points-to graph* (GPG) is a set of edges representing generalized points-to facts. For a GPG edge $x \xrightarrow{i,j} y$, the pair (i, j) represents indirection levels and is called the *indlev* of the edge (i is the *indlev* of x, and j is the *indlev* of y).

Figure 3 illustrates the generalized points-to facts corresponding to the basic pointer assignments in C. Observe that a classical points-to fact $x \rightarrow y$ is a special case of the generalized points-to fact $x \xrightarrow{i,j} y$ with $i = 1$ and $j = 0$; the case $i = 0$ does not arise.

The generalized points-to facts are more expressive than the classical points-to facts because they can be composed to create new facts as shown by the example below. Section 3 explains the process of composing the generalized points-to facts through *edge composition* along with the conditions when the facts can and ought to be composed.

s_1: $x = \&y$;
s_2: $z = x$;

Example 4. Statements s_1 and s_2 to the right are represented by GPG edges $x \xrightarrow{1,0} y$ and $z \xrightarrow{1,1} x$ respectively. We can compose the two edges by creating a new edge $z \xrightarrow{1,0} y$ indicating that z points-to y. Effectively, this converts the generalized points-to fact for s_2 into a classical points-to fact. □

Imposing an ordering on the set of GPG edges allows us to view it as a sequence to represent a flow-sensitive memory transformer. A reverse post order traversal over the control flow graph of a procedure dictates this sequence. It is required only at the interprocedural level when the effect of a callee is incorporated in its caller. Since a sequence is totally ordered but control flow is partially ordered, the GPG operations (Sect. 5) internally relax the total order to ensure that the edges appearing on different control flow paths do not affect each other.

While the visual presentation of GPGs as graphs is intuitively appealing, it loses the edge-ordering; we annotate edges with their ordering explicitly when it matters.

A GPG is a uniform representation for a memory transformer as well as (an abstraction of) memory. This is analogous to a matrix which can be seen both as a transformer (for a linear translation) and also as an absolute value. A points-to analysis using GPGs begins with generalized points-to facts $\cdot \xrightarrow{i,j} \cdot$ representing memory transformers which are composed to create new generalized points-to facts with smaller *indlev* s thereby progressively reducing them to classical points-to facts $\cdot \xrightarrow{1,0} \cdot$ representing memory.

2.3 The Lattice of GPGs

Definition 3 describes the meet semi-lattice of GPGs. For reasons described later in Sect. 5, we need to introduce an artificial $\top$ element denoted $\Delta_\top$ in the lattice. It is used as the initial value in the data flow equations for computing GPGs (Definition 5 which instantiates Definition 1 for GPGs).

Definition 3: Lattice of GPGs

$$\Delta \in \{\Delta_\top\} \cup \{(\mathcal{N}, \mathcal{E}) \mid \mathcal{N} \subseteq N,\ \mathcal{E} \subseteq E\}$$

where

$$N := L \cup \{?\}$$

$$E := \{x \xrightarrow{i,j} y \mid x \in P,\ y \in N,\ 0 < i \leq |N|,\ 0 \leq j \leq |N|\ \}$$

$$\Delta_1 \sqsubseteq \Delta_2 \Leftrightarrow (\Delta_2 = \Delta_\top) \vee (\mathcal{N}_1 \supseteq \mathcal{N}_2 \wedge \mathcal{E}_1 \supseteq \mathcal{E}_2)$$

$$\Delta_1 \sqcap \Delta_2 := \begin{cases} \Delta_1 & \Delta_2 = \Delta_\top \\ \Delta_2 & \Delta_1 = \Delta_\top \\ (\mathcal{N}_1 \cup \mathcal{N}_2,\ \mathcal{E}_1 \cup \mathcal{E}_2) & \text{otherwise} \end{cases}$$

The sequencing of edges is maintained externally and is explicated where required. This allows us to treat a GPG (other than $\Delta_\top$) as a pair of a set of nodes and a set of edges. The partial order is a point-wise superset relation applied to the pairs. Similarly, the meet operation is a point-wise union of the pairs. It is easy to see that the lattice is finite because the number of locations L is finite (being restricted to static and stack slots). When we extend GPGs to handle heap memory [4], explicit summarization is required to ensure finiteness. The finiteness of the lattice and the monotonicity of GPG operations guarantee the convergence of GPG computations on a fixed point; starting from $\Delta_\top$, we compute the maximum fixed point.

For convenience, we treat a GPG as a set of edges leaving the set of nodes implicit; the GPG nodes can always be inferred from the GPG edges.

2.4 A Hierarchy of GPG Operations

Figure 4 lists the GPG operations based on the concept of the generalized points-to facts. They are presented in two separate columns according to the two phases of our analysis and each layer is defined in terms of the layers below it. The operations are defined in the sections listed against them in Fig. 4.

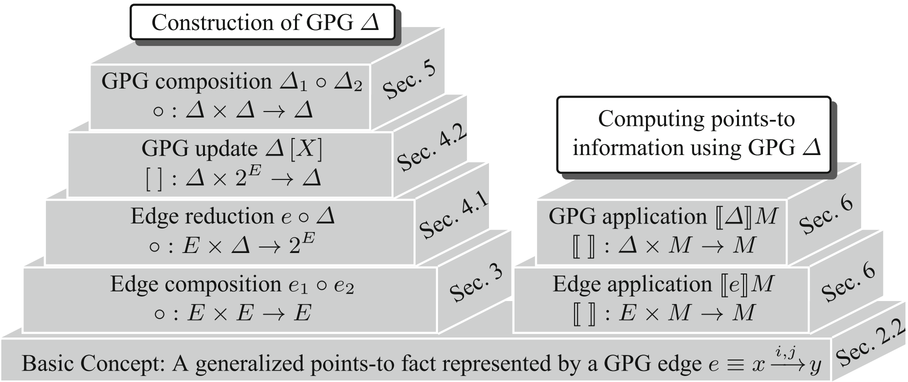

Construction of GPG Δ
GPG composition $\Delta_1 \circ \Delta_2$
$\circ : \Delta \times \Delta \to \Delta$
Sec. 5
GPG update $\Delta\,[X]$
$[\,] : \Delta \times 2^E \to \Delta$
Sec. 4.2
Edge reduction $e \circ \Delta$
$\circ : E \times \Delta \to 2^E$
Sec. 4.1
Edge composition $e_1 \circ e_2$
$\circ : E \times E \to E$
Sec. 3
Computing points-to information using GPG Δ
GPG application $[\![\Delta]\!]M$
$[\![\ ]\!] : \Delta \times M \to M$
Sec. 6
Edge application $[\![e]\!]M$
$[\![\ ]\!] : E \times M \to M$
Sec. 6
Basic Concept: A generalized points-to fact represented by a GPG edge $e \equiv x \xrightarrow{i,j} y$
Sec. 2.2

Fig. 4. A hierarchy of operations for points-to analysis using GPGs. Each operation is defined in terms of the layers below it. E denotes the set of GPG edges. By abuse of notation, we use M and Δ also as types to indicate the signatures of the operations. The operators "$\circ$" and "$[\![\ ]\!]$" are overloaded and can be disambiguated using the types of the operands.

Constructing GPGs. An *edge composition* $e_1 \circ e_2$ computes a new edge e_3 equivalent to e_1 using the points-to information in e_2 such that the *indlev* of e_3 is smaller than that of e_1. An *edge reduction* $e_1 \circ \Delta$ computes a set of edges X by composing e_1 with the edges in Δ. A *GPG update* $\Delta_1\,[X]$ incorporates the effect of the set of edges X in Δ_1 to compute a new GPG Δ_2. A *GPG composition* $\Delta_1 \circ \Delta_2$ composes a callee's GPG Δ_2 with GPG Δ_1 at a call point to compute a new GPG Δ_3.

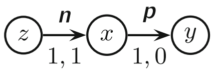
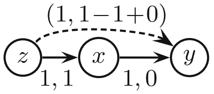
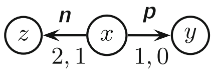
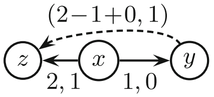
Using GPGs for computing points-to information. An *edge application* $[\![e]\!]M$ computes a new memory M' by incorporating the effect of the GPG edge e in memory M. A *GPG application* $[\![\Delta]\!]M$ applies the GPG Δ to M and computes a new memory M' using edge application iteratively.

These operations allow us to build the theme of a GPG being a uniform representation for both memory and memory transformers.

3 Edge Composition

This section defines edge composition as a fundamental operation which is used in Sect. 4 for constructing GPGs. Some considerations in edge composition (explained in this section) are governed by the goal of including the resulting edges in a GPG Δ.

Let a statement-level flow function δ be represented by an edge ***n*** ("new" edge) and consider an existing edge ***p*** $\in \Delta$ ("processed" edge). Edges ***n*** and ***p*** can be composed (denoted ***n*** $\circ$ ***p***) provided they have a common node called the *pivot* of composition (since a pivot can be the source or target of either of the edges, there are four possibilities as explained later). The goal is to *reduce* (i.e., simplify) ***n*** by using the points-to information from ***p***. This is achieved by using

Statement sequence	GPG	
	Before composition	After composition
$x = \&y$ $z = x$		
$x = \&y$ $*x = z$		

Regardless of the direction of an edge, i in *indlev* (i, j) represents its source while j represents its target. Balancing the *indlev*s of x (the pivot of composition) in $\boldsymbol{p}$ and $\boldsymbol{n}$ allows us to join y and z to create a reduced edge $\boldsymbol{r} = \boldsymbol{n} \circ \boldsymbol{p}$ shown by dashed arrows.

Fig. 5. Examples of edge compositions for points-to analysis.

the pivot as a bridge to join the remaining two nodes resulting in a reduced edge $\boldsymbol{r}$. This requires the *indlev* s of the pivot in both edges to be made the same. For example, given edges $\boldsymbol{n} \equiv z \xrightarrow{i,j} x$ and $\boldsymbol{p} \equiv x \xrightarrow{k,l} y$ with a pivot x, if $j > k$, then the difference $j - k$ can be added to the *indlev* s of nodes in $\boldsymbol{p}$, to view $\boldsymbol{p}$ as $x \xrightarrow{j,(l+j-k)} y$. This balances the *indlev* s of x in the two edges allowing us to create a reduced edge $\boldsymbol{r} \equiv z \xrightarrow{i,(l+j-k)} y$. Although this computes the transitive effect of edges, in general, it cannot be modelled using multiplication of matrices representing graphs as explained in our technical report [4].

Example 5. In the first example in Fig. 5, the *indlev* s of pivot x in both $\boldsymbol{p}$ and $\boldsymbol{n}$ is the same allowing us to join z and y through an edge $z \xrightarrow{1,0} y$. In the second example, the difference $(2-1)$ in the *indlev* s of x can be added to the *indlev* s of nodes in $\boldsymbol{p}$ viewing it as $x \xrightarrow{2,1} y$. This allows us to join y and z creating the edge $y \xrightarrow{1,1} z$. □

Let an edge $\boldsymbol{n}$ be represented by the triple $(\mathsf{S}_{\boldsymbol{n}},(\mathsf{s}^c_{\boldsymbol{n}},\mathsf{t}^c_{\boldsymbol{n}}),\mathsf{T}_{\boldsymbol{n}})$ where $\mathsf{S}_{\boldsymbol{n}}$ and $\mathsf{T}_{\boldsymbol{n}}$ are the source and the target of $\boldsymbol{n}$ and $(\mathsf{s}^c_{\boldsymbol{n}},\mathsf{t}^c_{\boldsymbol{n}})$ is the *indlev*. Similarly, $\boldsymbol{p}$ is represented by $\left(\mathsf{S}_{\boldsymbol{p}},(\mathsf{s}^c_{\boldsymbol{p}},\mathsf{t}^c_{\boldsymbol{p}}),\mathsf{T}_{\boldsymbol{p}}\right)$ and the reduced edge $\boldsymbol{r} = \boldsymbol{n} \circ \boldsymbol{p}$ by $(\mathsf{S}_{\boldsymbol{r}},(\mathsf{s}^c_{\boldsymbol{r}},\mathsf{t}^c_{\boldsymbol{r}}),\mathsf{T}_{\boldsymbol{r}})$; $(\mathsf{s}^c_{\boldsymbol{r}},\mathsf{t}^c_{\boldsymbol{r}})$ is obtained by balancing the *indlev* of the pivot in $\boldsymbol{p}$ and $\boldsymbol{n}$. The pivot of a composition, denoted $\mathbb{P}$, may be the source or the target of $\boldsymbol{n}$ and $\boldsymbol{p}$. This leads to four combinations of $\boldsymbol{n} \circ \boldsymbol{p}$: *SS*, *TS*, *ST*, *TT*. Our implementation currently uses *TS* and *SS* compositions illustrated in Fig. 6; *ST* and *TT* compositions are described in the technical report [4].

- *TS* composition. In this case, $\mathsf{T}_{\boldsymbol{n}} = \mathsf{S}_{\boldsymbol{p}}$ i.e., the pivot is the target of $\boldsymbol{n}$ and the source of $\boldsymbol{p}$. Node $\mathsf{S}_{\boldsymbol{n}}$ becomes the source and $\mathsf{T}_{\boldsymbol{p}}$ becomes the target of the reduced edge $\boldsymbol{r}$.
- *SS* composition. In this case, $\mathsf{S}_{\boldsymbol{n}} = \mathsf{S}_{\boldsymbol{p}}$ i.e., the pivot is the source of both $\boldsymbol{n}$ and $\boldsymbol{p}$. Node $\mathsf{T}_{\boldsymbol{p}}$ becomes the source and $\mathsf{T}_{\boldsymbol{n}}$ becomes the target of the reduced edge $\boldsymbol{r}$.

Consider an edge composition $\boldsymbol{r} = \boldsymbol{n} \circ \boldsymbol{p}$, $\boldsymbol{p} \in \Delta$. For constructing a new Δ, we wish to include $\boldsymbol{r}$ rather than $\boldsymbol{n}$: Including both of them is sound but may lead to imprecision; including only $\boldsymbol{n}$ is also sound but may lead to inefficiency because

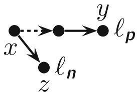
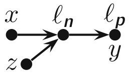
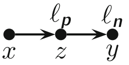
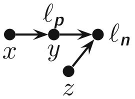
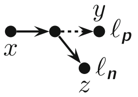
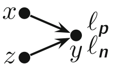
Possible *SS* Compositions			Possible *TS* Compositions		
Statement sequence	Memory graph	GPG edges	Statement sequence	Memory graph	GPG edges
$S_n^c < S_p^c$			$T_n^c < S_p^c$		
$\boxed{\text{Ex. } ss1}$ $*x = \&y$ $x = \&z$		**p**: $x \xrightarrow{2,0} y$ **n**: $x \xrightarrow{1,0} z$ (*irrelevant*)	$\boxed{\text{Ex. } ts1}$ $*x = \&y$ $z = x$		**p**: $x \xrightarrow{2,0} y$ **n**: $z \xrightarrow{1,1} x$ (not *useful*)
$S_n^c > S_p^c$ (Additionally $T_p^c \leq S_p^c$)			$T_n^c > S_p^c$ (Additionally $T_p^c \leq S_p^c$)		
$\boxed{\text{Ex. } ss2}$ $x = \&z$ $*x = \&y$		**p**: $x \xrightarrow{1,0} z$ **n**: $x \xrightarrow{2,0} y$ **r**: $z \xrightarrow{1,0} y$	$\boxed{\text{Ex. } ts2}$ $x = \&y$ $z = *x$		**p**: $x \xrightarrow{1,0} y$ **n**: $z \xrightarrow{1,2} x$ **r**: $z \xrightarrow{1,1} y$
$S_n^c = S_p^c$			$T_n^c = S_p^c$ (Additionally $T_p^c \leq S_p^c$)		
$\boxed{\text{Ex. } ss3}$ $*x = \&y$ $*x = \&z$		**p**: $x \xrightarrow{2,0} y$ **n**: $x \xrightarrow{2,0} z$ (*irrelevant*)	$\boxed{\text{Ex. } ts3}$ $x = \&y$ $z = x$		**p**: $x \xrightarrow{1,0} y$ **n**: $z \xrightarrow{1,1} x$ **r**: $z \xrightarrow{1,0} y$

Fig. 6. Illustrating all exhaustive possibilities of *SS* and *TS* compositions (the pivot is x). Dashed edges are killed. Unmarked compositions are *relevant* and *useful* (Sect. 3); since the statements are consecutive, they are also *conclusive* (Sect. 3) and hence *desirable*.

it forsakes summarization. An edge composition is *desirable* if and only if it is *relevant*, *useful*, and *conclusive*. We define these properties below and explain them in the rest of the section.

(a) A composition $\boldsymbol{n} \circ \boldsymbol{p}$ is *relevant* only if it preserves flow-sensitivity.
(b) A composition $\boldsymbol{n} \circ \boldsymbol{p}$ is *useful* only if the *indlev* of the resulting edge does not exceed the *indlev* of $\boldsymbol{n}$.
(c) A composition $\boldsymbol{n} \circ \boldsymbol{p}$ is *conclusive* only when the information supplied by $\boldsymbol{p}$ used for reducing $\boldsymbol{n}$ is not likely to be invalidated by the intervening statements.

When the edge composition is *desirable*, we include $\boldsymbol{r}$ in the Δ being constructed, otherwise we include $\boldsymbol{n}$. In order to explain the *desirable* compositions, we use the following notation: Let $\ell_{\boldsymbol{p}}$ denote a $(\mathbb{P}_{\boldsymbol{p}}^c)^{th}$ pointee of pivot $\mathbb{P}$ accessed by $\boldsymbol{p}$ and $\ell_{\boldsymbol{n}}$ denote a $(\mathbb{P}_{\boldsymbol{n}}^c)^{th}$ pointee of $\mathbb{P}$ accessed by $\boldsymbol{n}$.

Relevant Edge Composition. An edge composition is *relevant* if it preserves flow-sensitivity. This requires the indirection levels in $\boldsymbol{n}$ to be reduced by using the points-to information in $\boldsymbol{p}$ (where $\boldsymbol{p}$ appears before $\boldsymbol{n}$ along a control flow path) but not vice-versa. The presence of a points-to path in memory (which is the transitive closure of the points-to edges) between $\ell_{\boldsymbol{p}}$ and $\ell_{\boldsymbol{n}}$ (denoted by $\ell_{\boldsymbol{p}} \twoheadrightarrow \ell_{\boldsymbol{n}}$ or $\ell_{\boldsymbol{n}} \twoheadrightarrow \ell_{\boldsymbol{p}}$) indicates that $\boldsymbol{p}$ can be used to resolve the indirection levels in $\boldsymbol{n}$.

Example 6. For $\mathsf{S}^c_{\boldsymbol{n}} < \mathsf{S}^c_{\boldsymbol{p}}$ in Fig. 6 (Ex. *ss1*), edge $\boldsymbol{p}$ updates the pointee of x and edge $\boldsymbol{n}$ redefines x. As shown in the memory graph, there is no path between $\ell_{\boldsymbol{p}}$ and $\ell_{\boldsymbol{n}}$ and hence y and z are unrelated rendering this composition *irrelevant*. Similarly, edge composition is *irrelevant* for $\mathsf{S}^c_{\boldsymbol{n}} = \mathsf{S}^c_{\boldsymbol{p}}$ (Ex. *ss3*).

For $\mathsf{S}^c_{\boldsymbol{n}} > \mathsf{S}^c_{\boldsymbol{p}}$ (Ex. *ss2*), $\ell_{\boldsymbol{p}} \twoheadrightarrow \ell_{\boldsymbol{n}}$ holds in the memory graph and hence this composition is *relevant*. For Ex. *ts1*, $\ell_{\boldsymbol{n}} \twoheadrightarrow \ell_{\boldsymbol{p}}$ holds; for *ts2*, $\ell_{\boldsymbol{p}} \twoheadrightarrow \ell_{\boldsymbol{n}}$ holds; for *ts3* both paths hold. Hence, all three compositions are *relevant*. □

Useful Edge Composition. The *usefulness* of edge composition characterizes progress in conversion of the generalized points-to facts to the classical points-to facts. This requires the *indlev* ($\mathsf{S}^c_{\boldsymbol{r}}$, $\mathsf{T}^c_{\boldsymbol{r}}$) of the reduced edge $\boldsymbol{r}$ to satisfy:

$$\mathsf{S}^c_{\boldsymbol{r}} \le \mathsf{S}^c_{\boldsymbol{n}} \ \wedge\ \mathsf{T}^c_{\boldsymbol{r}} \le \mathsf{T}^c_{\boldsymbol{n}} \qquad (1)$$

Intuitively, this ensures that the *indlev* of the new source and the new target does not exceed the corresponding *indlev* in the original edge $\boldsymbol{n}$.

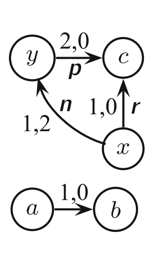
Example 7. Consider Ex. *ts1* of Fig. 6, for $\mathsf{T}^c_{\boldsymbol{n}} < \mathsf{S}^c_{\boldsymbol{p}}$, $\ell_{\boldsymbol{n}} \twoheadrightarrow \ell_{\boldsymbol{p}}$ holds in the memory graph. Although this composition is *relevant*, it is not *useful* because the *indlev* of $\boldsymbol{r}$ exceeds the *indlev* of $\boldsymbol{n}$. For this example, a *TS* composition will create an edge $z \xrightarrow{2,0} y$ whose *indlev* is higher than that of $\boldsymbol{n}$ ($z \xrightarrow{1,1} x$). □

Thus, we need $\ell_{\boldsymbol{p}} \twoheadrightarrow \ell_{\boldsymbol{n}}$, and not $\ell_{\boldsymbol{n}} \twoheadrightarrow \ell_{\boldsymbol{p}}$, to hold in the memory graph for a *useful* edge composition. We can relate this with the *usefulness* criteria (Inequality 1). The presence of path $\ell_{\boldsymbol{p}} \twoheadrightarrow \ell_{\boldsymbol{n}}$ ensures that the *indlev* of edge $\boldsymbol{r}$ does not exceed that of $\boldsymbol{n}$. The *usefulness* criteria (Inequality 1) reduces to $\mathsf{T}^c_{\boldsymbol{p}} \le \mathsf{S}^c_{\boldsymbol{p}} < \mathsf{S}^c_{\boldsymbol{n}}$ for *SS* composition and $\mathsf{T}^c_{\boldsymbol{p}} \le \mathsf{S}^c_{\boldsymbol{p}} \le \mathsf{T}^c_{\boldsymbol{n}}$ for *TS* composition.

From Fig. 6, we conclude that an edge composition is *relevant* and *useful* only if there exists a path $\ell_{\boldsymbol{p}} \twoheadrightarrow \ell_{\boldsymbol{n}}$ rather than $\ell_{\boldsymbol{n}} \twoheadrightarrow \ell_{\boldsymbol{p}}$. *Intuitively, such a path guarantees that the updates made by* $\boldsymbol{n}$ *do not invalidate the generalized points-to fact represented by* $\boldsymbol{p}$. Hence, the two generalized points-to facts can be composed by using the pivot as a bridge to create a new generalized points-to fact represented by $\boldsymbol{r}$.

Conclusive Edge Composition. Recall that $\boldsymbol{r} = \boldsymbol{n} \circ \boldsymbol{p}$ is *relevant* and *useful* if we expect a path $\ell_{\boldsymbol{p}} \twoheadrightarrow \ell_{\boldsymbol{n}}$ in the memory. This composition is *conclusive* when location $\ell_{\boldsymbol{p}}$ remains accessible from the pivot $\mathbb{P}$ in $\boldsymbol{p}$ when $\boldsymbol{n}$ is composed with $\boldsymbol{p}$. Location $\ell_{\boldsymbol{p}}$ may become inaccessible from $\mathbb{P}$ because of a combined effect of the statements in a calling context and the statements in the procedure being processed. Hence, the composition is *undesirable* and may lead to unsoundness if $\boldsymbol{r}$ is included in Δ instead of $\boldsymbol{n}$.

```
01 void p()
02 {  y = &a;
03    q();
04 }
05 void q()
06 {  *y = &c;
07    a = &b;
08    x = *y;
09 }
```



Example 8. Line 6 in the code on the right indirectly defines a (because of the assignment on line 2) whereas line 7 directly defines a overwriting the value. Thus, x points to b and not c after line 8. When the GPG for procedure q is constructed, the relationship between y and a is not known. Thus, the composition of $\boldsymbol{n} \equiv x \xrightarrow{1,2} y$ with $\boldsymbol{p} \equiv y \xrightarrow{2,0} c$ results in $\boldsymbol{r} \equiv x \xrightarrow{1,0} c$. Here $\ell_{\boldsymbol{p}}$ is c, however it is not reachable from y anymore as the pointee of y is redefined by line 7. □

Since the calling context is not available during GPG construction, we are forced to retain edge $\boldsymbol{n}$ in the GPG, thereby missing an opportunity of reducing the *indlev* of $\boldsymbol{n}$. Hence we propose the following condition for *conclusiveness*: The statements corresponding to $\boldsymbol{p}$ and $\boldsymbol{n}$ should be consecutive on every control flow *(a)* the intervening statements should not have an indirect assignment (e.g., $*x = \ldots$), and *(b)* the pointee of pivot $\mathbb{P}$ in edge $\boldsymbol{p}$ should have been found i.e., $\mathbb{P}^{c}_{\boldsymbol{p}} = 1$.

In the example above, condition *(b)* is violated and hence we add $\boldsymbol{n} \equiv x \xrightarrow{1,2} y$ to the GPG of procedure q instead of $\boldsymbol{r} \equiv x \xrightarrow{1,0} c$. This avoids a greedy reduction of $\boldsymbol{n}$ when the available information is *inconclusive*.

4 Constructing GPGs at the Intraprocedural Level

In this section we define edge reduction, and GPG update; GPG composition is described in Sect. 5 which shows how procedure calls are handled.

4.1 Edge Reduction $\boldsymbol{n} \circ \Delta$

Edge reduction $\boldsymbol{n} \circ \Delta$ uses the edges in Δ to compute a set of edges whose *indlev* s do not exceed that of $\boldsymbol{n}$ (Definition 4). The results of *SS* and *TS* compositions are denoted by SS^{n}_{Δ} and TS^{n}_{Δ} which compute *relevant* and *useful* edge compositions; the *inconclusive* edge compositions are filtered out independently. The edge ordering is not required at

Definition 4: Edge reduction in Δ

$$\boldsymbol{n} \circ \Delta \;:=\; \mathit{mlc}(\{\boldsymbol{n}\}, \Delta)$$

where

$$\mathit{mlc}(X, \Delta) := \begin{cases} X & \mathit{slces}(X, \Delta) = X \\ \mathit{mlc}(\mathit{slces}(X, \Delta), \Delta) & \text{Otherwise} \end{cases}$$

$$\mathit{slces}(X, \Delta) := \bigcup_{e \in X} \mathit{slc}(e, \Delta)$$

$$\mathit{slc}(\boldsymbol{n}, \Delta) := \begin{cases} \mathit{SS}^{n}_{\Delta} \bowtie \mathit{TS}^{n}_{\Delta} & \mathit{SS}^{n}_{\Delta} \neq \emptyset, \mathit{TS}^{n}_{\Delta} \neq \emptyset \\ \{\boldsymbol{n}\} & \mathit{SS}^{n}_{\Delta} = \mathit{TS}^{n}_{\Delta} = \emptyset \\ \mathit{SS}^{n}_{\Delta} \cup \mathit{TS}^{n}_{\Delta} & \text{Otherwise} \end{cases}$$

$$\mathit{SS}^{n}_{\Delta} := \{\boldsymbol{n} \circ \boldsymbol{p} \mid \boldsymbol{p} \in \Delta, S_{\boldsymbol{n}} = S_{\boldsymbol{p}}, T^{c}_{\boldsymbol{p}} \leq s^{c}_{\boldsymbol{p}} < s^{c}_{\boldsymbol{n}}\}$$

$$\mathit{TS}^{n}_{\Delta} := \{\boldsymbol{n} \circ \boldsymbol{p} \mid \boldsymbol{p} \in \Delta, T_{\boldsymbol{n}} = S_{\boldsymbol{p}}, T^{c}_{\boldsymbol{p}} \leq s^{c}_{\boldsymbol{p}} \leq T^{c}_{\boldsymbol{n}}\}$$

$$X \bowtie Y := \{\, (S_{\boldsymbol{n}}, (s^{c}_{\boldsymbol{n}}, T^{c}_{\boldsymbol{p}}), T_{\boldsymbol{p}}) \mid \boldsymbol{n} \in X, \boldsymbol{p} \in Y\}$$

the intraprocedural level; a reverse post order traversal over the control flow graph suffices.

A single-level composition (slc) combines SS^n_Δ with TS^n_Δ. When both TS and SS compositions are possible (first case in slc), the join operator $\bowtie$ combines their effects by creating new edges by joining the sources from SS^n_Δ with the targets from TS^n_Δ. If neither of TS and SS compositions is possible (second case in slc), edge $\boldsymbol{n}$ is considered as the reduced edge. If only one of them is possible, its result becomes the result of slc (third case). Since the reduced edges computed by slc may compose with other edges in Δ, we extend slc to multi-level composition (mlc) which recursively composes edges in X with edges in Δ through function slces which extends slc to a set of edges.

Example 9. When $\boldsymbol{n}$ 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 on the right. $\Delta = \{y \xrightarrow{1,0} a, a \xrightarrow{1,0} b\}$ for $\boldsymbol{n} \equiv x \xrightarrow{1,2} y$ (statement s_3).

This involves two consecutive TS compositions. The first composition involves $y \xrightarrow{1,0} a$ as $\boldsymbol{p}$ resulting in $\mathit{TS}^n_\Delta = \{x \xrightarrow{1,1} a\}$ and $\mathit{SS}^n_\Delta = \emptyset$. This satisfies the third case of slc. Then, slces is called with $X = \{x \xrightarrow{1,1} a\}$. The second TS composition between $x \xrightarrow{1,1} a$ (as a new $\boldsymbol{n}$) and $a \xrightarrow{1,0} b$ (as $\boldsymbol{p}$) results in a reduced edge $x \xrightarrow{1,0} b$. slces is called again with $X = \{x \xrightarrow{1,0} b\}$ which returns X, satisfying the base condition of mlc. □

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

Example 10. Single-level compositions are combined using $\bowtie$ when $\boldsymbol{n}$ represents $*x = y$.

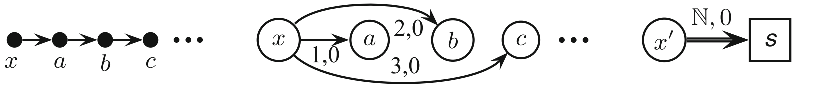
For the code snippet on the right, SS^n_Δ returns $\{a \xrightarrow{1,1} y\}$ and TS^n_Δ returns $\{x \xrightarrow{2,0} b\}$ when $\boldsymbol{n}$ is $x \xrightarrow{2,1} y$ (for statement s_3). The join operator $\bowtie$ combines the effect of TS and SS compositions by combining the sources from SS^n_Δ and the targets from TS^n_Δ resulting in a reduced edge $\boldsymbol{r} \equiv a \xrightarrow{1,0} b$. □

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

4.2 Constructing GPGs $\Delta(u, v)$

For simplicity, we consider Δ only as a collection of edges, leaving the nodes implicit. Further, the edge ordering does not matter at the intraprocedural level and hence we treat Δ as a set of edges. The construction of Δ assigns sequence numbers in the order of inclusion of edges; these sequence numbers are maintained externally.

By default, the GPGs record the *may* information but a simple extension in the form of *boundary definitions* (described in the later part of this section) allows them to record the *must* information. This supports distinguishing between strong and weak updates and yet allows a simple set union to combine the information.



(a) Memory view (b) GPG view (c) Aggregate edge

Fig. 7. Aggregate edge for handling strong and weak updates. For this example, $\mathsf{s} = \{a, b, c, \ldots\}$.

Definition 5: Construction of Δ

Assumption : $\boldsymbol{n}$ is $\delta(\boldsymbol{t},\boldsymbol{v})$ and Δ is a set of edges

$$\Delta(\boldsymbol{u},\boldsymbol{v}) := B(\boldsymbol{u},\boldsymbol{v}) \;\cup \bigcup_{\substack{\boldsymbol{t}\,\in\, succ^{+}(\boldsymbol{u})\\ \boldsymbol{v}\,\in\, succ(\boldsymbol{t})}} (\Delta(\boldsymbol{u},\boldsymbol{t}))\,[\boldsymbol{n} \circ \Delta(\boldsymbol{u},\boldsymbol{t})]$$

$$B(\boldsymbol{u},\boldsymbol{v}) := \begin{cases} \boldsymbol{n} & \boldsymbol{v} \in succ(\boldsymbol{u}) \\ \emptyset & \text{otherwise} \end{cases}$$

where

$$\Delta\,[X] := (\Delta - conskill(X,\Delta)) \;\cup\; (X)$$

$$conskill(X,\Delta) := \big\{e_1 \mid e_1 \in match(e,\Delta), e \in X, |def(X)| = 1\big\}$$

$$match(e,\Delta) := \{e_1 \mid e_1 \in \Delta,\; S_e = S_{e_1},\; s^c_e = s^c_{e_1}\}$$

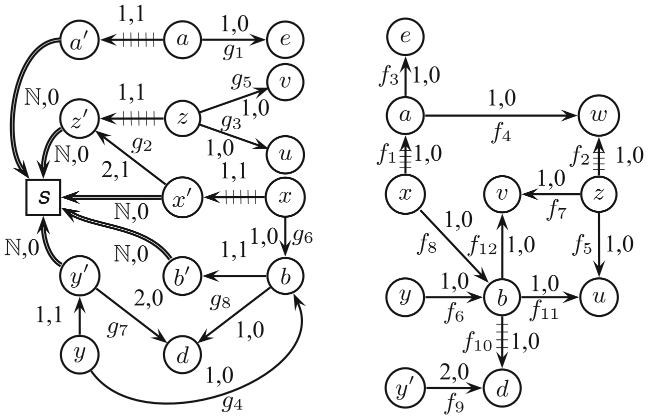
$$def(X) := \big\{(S_e, s^c_e) \mid e \in X\big\}$$

Definition 5 is an adaptation of Definition 1 for GPGs. Since Δ is viewed as a set of edges, the identity function Δ_{id} is $\emptyset$, meet operation is $\cup$, and $\Delta(\boldsymbol{u},\boldsymbol{v})$ is the least fixed point of the equation in Definition 5. The composition of a statement-level flow function ($\boldsymbol{n}$) with a summary flow function ($\Delta(\boldsymbol{u},\boldsymbol{t})$) is performed by GPG update which includes all edges computed by edge reduction $\boldsymbol{n}\circ\Delta(\boldsymbol{u},\boldsymbol{t})$; the edges to be removed are under-approximated when a strong update cannot be performed (described in the rest of the section). When a strong update is performed, we exclude those edges of Δ whose source and *indlev* match that of the shared source of the reduced edges (identified by $match(e,\Delta)$). For a weak update, $conskill(X,\Delta) = \emptyset$ and X contains reduced edges. For an *inconclusive* edge composition, $conskill(X,\Delta) = \emptyset$ and $X = \{\boldsymbol{n}\}$.

Extending Δ to Support Strong Updates. Conventionally, points-to information is killed based on the following criteria: An assignment $x = \ldots$ removes all points-to facts $x \longrightarrow \cdot$ whereas an assignment $*x = \ldots$ removes all points-to facts $y \longrightarrow \cdot$ where x *must*-points-to y; the latter represents a *strong update.* When x *may*-points-to y, no points-to facts can be removed representing a *weak update.*

Observe that the use of points-to information for strong updates is inherently captured by edge reduction. In particular, the use of edge reduction allows us to model both of the above criteria for edge removal uniformly as follows: the reduced edges should define the same pointer (or the same pointee of a given pointer) along every control flow path reaching the statement represented by $\boldsymbol{n}$. This is captured by the requirement $|def(X)| = 1$.

When $|def(X)| > 1$, the reduced edges define multiple pointers (or different pointees of the same pointer) leading to a weak update resulting in no removal of edges from Δ. When $|def(X)| = 1$, all reduced edges define the same pointer (or the same pointee of a given pointer). However, this is necessary but not sufficient



(a) $\varDelta$ for g

(b) $\varDelta$ for f

Regardless of the direction of the arrow, i in *indlev* (i,j) represents its source while j represents its target. Edges deleted by updates are struck off. Subscript k in edge names g_k, f_k indicates the order of edge inclusion.

Copy and aggregate edges have not been shown for $\varDelta$ for f.

Fig. 8. $\varDelta$ for procedures f and g of Figure 1.

for a strong update because the pointer may not be defined along all the paths—there may be a path which does not contribute to *def*(X). We refer to such paths as definition-free paths for that particular pointer (or some pointee of a pointer). The possibility of such a path makes it difficult to distinguish between strong and weak updates.

Since a pointer x or its transitive pointees may be defined along some control flow path from ***u*** to ***v***, we eliminate the possibility of definition-free paths from ***u*** to ***v*** by introducing *boundary definitions* of the following two kinds at ***u***: *(a)* a pointer assignment $x = x'$ where x' is a symbolic representation of the initial value of x at ***u*** (called the *upwards exposed* version of x), and *(b)* a set of assignments representing the relation between x' and its transitive pointees. They are represented by special GPG edges—the first, by a *copy edge* $x \xrightarrow{1,1} x'$ and the others, by an *aggregate* edge $x' \xrightarrow{\mathbb{N},0} \mathsf{s}$ where $\mathbb{N}$ is the set of all possible *indlev* s and s is the summary node representing all possible pointees. As illustrated in Fig. 7, $x' \xrightarrow{\mathbb{N},0} \mathsf{s}$ is a collection of GPG edges (Fig. 7(b)) representing the relation between x with it transitive pointees at ***u*** (Fig. 7(a)).

A reduced edge $x \xrightarrow{1,j} y$ along any path from ***u*** to ***v*** removes the copy edge $x \xrightarrow{1,1} x'$ indicating that x is redefined. A reduced edge $x \xrightarrow{i,j} y$, $i > 1$ modifies the aggregate edge $x' \xrightarrow{\mathbb{N},0} \mathsf{s}$ to $x' \xrightarrow{(\mathbb{N}-\{i\}),0} \mathsf{s}$ indicating that $(i-1)^{th}$ pointees of x are redefined.

The inclusion of aggregate and copy edges guarantees that $|$*def*$(X)| = 1$ only when the source is defined along every path. This leads to a necessary and sufficient condition for strong updates. Note that the copy and aggregate edges improve the precision of analysis and are not required for its soundness.

Example 11. Consider the construction of Δ_g as illustrated in Fig. 8(c). Edge g_1 created for line 8 of the program, kills edge $a \xrightarrow{1,1} a'$ because $|\mathit{def}(\{g_1\})| = 1$. 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.

The $z \xrightarrow{1,1} z'$ edge is killed along both paths (lines 11 and 14) and hence is struck off in Δ_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 $x' \xrightarrow{2,0} \mathsf{s}$ in the aggregate edge is retained indicating that pointee of x is not defined along all paths. Edge g_6 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 ($|\mathit{def}(\{g_7, g_8\})| \neq 1$). Hence $b \xrightarrow{1,1} b'$ is not removed. Similarly, $y' \xrightarrow{2,0} \mathsf{s}$ in the aggregate edge $y' \xrightarrow{\mathbb{N},0} \mathsf{s}$ is not removed. □

5 Constructing GPGs at the Interprocedural Level

Definition 6 shows the construction of GPGs at the interprocedural level by handling procedure calls. Consider a procedure f containing a call to g between two consecutive program points $\boldsymbol{u}$ and $\boldsymbol{v}$. Let Start_g and End_g denote the start and the end points of g. Δ representing the control flow paths from Start_f to $\boldsymbol{u}$ (i.e., just before the call to g) is $\Delta(\mathit{Start}_f, \boldsymbol{u})$; we denote it by Δ_f for brevity. Δ for the body of procedure g is $\Delta(\mathit{Start}_g, \mathit{End}_g)$; we denote it by Δ_g.

Since GPGs are sequences of edges, $\Delta_g \circ \Delta_f$ involves selecting an edge e in order from Δ_g and performing an update $\Delta_f\ [e \circ \Delta_f]$. We then update the resulting Δ with the next edge from Δ_g. This is repeated until all edges of Δ_g are exhausted. The update of Δ_f with an edge e from Δ_g involves the following: *(a)* substituting the callee's upwards exposed variable x' occurring in Δ_g by the caller's original variable x in Δ_f, *(b)* including the reduced edges $e \circ \Delta_f$, and *(c)* performing a strong or weak update.

A copy edge $x \xrightarrow{1,1} x' \in \Delta$ implies that x has not been defined along some path. Similarly, an aggregate edge $x' \xrightarrow{\mathbb{N},0} \mathsf{s} \in \Delta$ implies that some $(i-1)^{th}$ pointees of x, $i > 1$ have not been defined along some path. We use these to define $\mathit{mustdef}(x \xrightarrow{i,j} y, \Delta)$ which asserts that the $(i-1)^{th}$ pointees of x, $i > 1$ are defined along every control flow path. We combine it with $\mathit{def}(x \xrightarrow{i,j} y \circ \Delta)$ to define *callsup* for identifying strong updates. Note that we need *mustdef* only at the interprocedural level and not at the intraprocedural level. This is because, when we use Δ_g to compute Δ_f, performing a strong update requires knowing whether the source of an edge in Δ_g has been defined along every control flow path in g. However, we do not have the control flow information of g when we analyze f. When a strong update is performed, we delete all edges in Δ_f that match $e \circ \Delta_f$. These edges are discovered by taking a union of $\mathit{match}(e_1, \Delta_f)$, $\forall e_1 \in (e \circ \Delta_f)$.

Definition 6: *Δ for a call $g()$ in procedure f*

/* let Δ_f denote $\Delta(\mathit{Start}_f, \boldsymbol{u})$ and Δ_g denote $\Delta(\mathit{Start}_g, \mathit{End}_g)$ */

$\Delta(\mathit{Start}_f, \boldsymbol{v}) \;:=\; \Delta_g \circ \Delta_f$

$\Delta_g \circ \Delta_f \;:=\; \Delta_f\,[\Delta_g]$

where /* let Δ_g be $\{e_1, e_2, \ldots e_k\}$ */

$$\Delta_f\,[\Delta_g\,] := \Delta_f\,[\,e_1, \Delta_g\,]\,[\,e_2, \Delta_g\,] \ldots [\,e_k, \Delta_g\,]$$

$$\Delta_f\,[e, \Delta_g] := (\Delta_f - \mathit{callkill}(e, \Delta_f, \Delta_g)) \;\cup\; (e \circ \Delta_f)$$

$$\mathit{callkill}(e, \Delta_f, \Delta_g) := \big\{e_2 \mid e_2 \in \mathit{match}(e_1, \Delta_f),\; e_1 \in e \circ \Delta_f,\; \mathit{callsup}(e, \Delta_f, \Delta_g)\big\}$$

$$\mathit{callsup}(e, \Delta_f, \Delta_g) := (|\mathit{def}(e \circ \Delta_f)| \;=\; 1) \wedge \mathit{mustdef}\,(e, \Delta_g)$$

$$\mathit{mustdef}\,(x \xrightarrow{i,j} y, \Delta) \Leftrightarrow \big(x \xrightarrow{i,k} z \in \Delta \Rightarrow k = j \wedge z = y\big) \;\bigwedge\; \Big(\big(i > 1 \;\wedge\; x' \xrightarrow{i,0} \mathsf{s} \notin \Delta\big) \;\vee\; \big(i = 1 \;\wedge\; x \xrightarrow{1,1} x' \notin \Delta\big)\Big)$$

The total order imposed by the sequence of GPG edges is interpreted as a partial order as follows: Since the edges from Δ_g are added one by one, if the edge to be added involves an upwards exposed variable x', it should be composed with an original edge in Δ_f rather than a reduced edge included in Δ_f created by $e_1 \circ \Delta_f$ for some $e_1 \in \Delta_g$. Further, it is possible that an edge e_2 may kill an already added edge e_1 that coexisted with it in Δ_g. However, this should be prohibited because their coexistence in Δ_g indicates that they are *may* edges. This is ensured by checking the presence of multiple edges with the same source in Δ_g. For example, edge f_7 of Fig. 8(d) does not kill f_5 as they coexist in Δ_g.

Example 12. Consider the construction of Δ_f as illustrated in Fig. 8(d). Edges f_1 and f_2 correspond to lines 2 and 3. The call on line 4 causes the composition of $\Delta_f = \{f_1, f_2\}$ with Δ_g selecting edges in the order $g_1, g_2, \ldots, g_8$. The edges from Δ_g with their corresponding names in Δ_f (denoted name-in-g/name-in-f) are: g_1/f_3, g_3/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 the source of g_2 is *may*-defined indicated by the presence of $x' \xrightarrow{2,0} \mathsf{s}$ in the aggregate edge $x' \xrightarrow{\mathbb{N},0} \mathsf{s}$.

Edges g_3/f_5 and g_5/f_7 together kill f_2. Note that the inclusion of f_7 does not kill f_5 because they both are from Δ_g. Finally, the edge for line 5 ($x \xrightarrow{2,1} z$) undergoes an *SS* composition (with f_8) and *TS* compositions (with f_5 and f_7). This creates edges f_{11} and f_{12}. Since $x \xrightarrow{2,1} z$ is accompanied by the aggregate edge $x' \xrightarrow{\mathbb{N}-\{2\},0} \mathsf{s}$ indicating that the pointee of x is *must*-defined, and x has a single pointee (edge f_8), this is a strong update killing edge f_{10}. Observe that all edges in Δ_f represent classical points-to facts except f_9. We need the pointees of y from the callers of f to reduce f_9. □

For recursive calls, the Δ for a callee may not have been computed because of a cycle in the call graph. This is handled in the usual manner [9,21] by over-approximating initial Δ that computes $\top$ for *may* points-to analysis (which is $\emptyset$). Such an initial GPG, denoted $\Delta_\top$ (Definition 3), kills all points-to relations and generates none. $\Delta_\top$ is not expressible as a GPG and is not a natural $\top$ element of the meet semi-lattice [9] of GPGs. The identity GPGΔ_{id} represents an empty set of edges because it does not generate or kill points-to information. For more details, please see [4].

6 Computing Points-To Information Using GPGs

Recall that the points-to information is represented by a memory M. We define two operations to compute a new memory M' using a GPG or a GPG edge from a given memory M.

- An *edge application* $[\![e]\!]M$ computes memory M' by incorporating the effect of GPG edge $e \equiv x \xrightarrow{i,j} y$ in memory M. This involves inclusion of edges described by the set $\left\{w \xrightarrow{1,0} z \mid w \in M^{i-1}\{x\},\ z \in M^{j}\{y\}\right\}$ in M' and removal of edges by distinguishing between a strong and a weak update. The edges to be removed are characterized much along the lines of $\mathit{callkill}$.
- A *GPG application* $[\![\Delta]\!]M$ applies the GPG Δ to M and computes the resulting memory M' using edge application iteratively.

Let $\mathit{PT}_{\boldsymbol{v}}$ denote the points-to information at program point $\boldsymbol{v}$ in procedure f. Then, $\mathit{PT}_{\boldsymbol{v}}$ can be computed by *(a)* computing *boundary information* of f (denoted BI_f) associated with Start_f, and *(b)* computing the points-to information at $\boldsymbol{v}$ from BI_f by incorporating the effect of all paths from Start_f to $\boldsymbol{v}$.

BI_f is computed as the union of the points-to information reaching f from all of its call points. For the main function, BI is computed from static initializations. In the presence of recursion, a fixed point computation is required for computing BI.

If $\boldsymbol{v}$ is Start_f, then $\mathit{PT}_{\boldsymbol{v}} = \mathit{BI}_f$. For other program points, $\mathit{PT}_{\boldsymbol{v}}$ can be computed from BI_f in the following ways; both of them compute identical $\mathit{PT}_{\boldsymbol{v}}$.

(a) *Using statement-level flow function (Stmt-ff):* Let $\mathit{stmt}(\boldsymbol{u},\boldsymbol{v})$ denote the statement between $\boldsymbol{u}$ and $\boldsymbol{v}$. If it is a non-call statement, let its flow function $\delta(\boldsymbol{u},\boldsymbol{v})$ be represented by the GPG edge $\boldsymbol{n}$. Then $\mathit{PT}_{\boldsymbol{v}}$ is computed as the least fixed point of the following data flow equations.

$$\mathit{In}_{\boldsymbol{u},\boldsymbol{v}} = \begin{cases} [\![\Delta(\mathit{Start}_q, \mathit{End}_q)]\!]\mathit{PT}_{\boldsymbol{u}} & \mathit{stmt}(\boldsymbol{u},\boldsymbol{v}) = \mathit{call}\ q \\ [\![\boldsymbol{n}]\!]\mathit{PT}_{\boldsymbol{u}} & \text{otherwise} \end{cases}$$

$$\mathit{PT}_{\boldsymbol{u}} = \bigcup_{\boldsymbol{u} \in \mathit{pred}(\boldsymbol{v})} \mathit{In}_{\boldsymbol{u},\boldsymbol{v}}$$

(b) *Using GPGs:* $\mathit{PT}_{\boldsymbol{v}}$ is computed using GPG application $[\![\Delta(\mathit{Start}_f, \boldsymbol{v})]\!]\mathit{BI}_f$. This approach of $\mathit{PT}_{\boldsymbol{v}}$ computation is oblivious to intraprocedural control flow and does not involve fixed point computation for loops.

Our measurements show that the *Stmt-ff* approach takes much less time than using GPGs for PT_v computation. This may appear surprising because the *Stmt-ff* approach requires an additional fixed point computation for handling loops which is not required in case of GPGs. However, using GPGs requires more time because the GPG at $\boldsymbol{v}$ represents a cumulative effect of the statement-level flow functions from $Start_f$ to $\boldsymbol{v}$. Hence the GPGs tend to become larger with the length of a control flow path. Thus computing PT_v using GPGs for multiple consecutive statements involves redundant computations.

Bypassing of BI. Our measurements show that using the entire BI of a procedure may be 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 *bypassing* has been successfully used for data flow values of scalars [15,16]. GPGs support this naturally for pointers with the help of upwards exposed versions of variables. An upwards exposed version in a GPG indicates that there is a use of a variable in the procedure which requires pointee information from the callers. Thus, the points-to information of such a variable is relevant and should be a part of BI. For variables that do not have their corresponding upwards exposed versions in a GPG, their points-to information is irrelevant and can be discarded from the BI of the procedure, effectively bypassing its calls.

7 Implementation and Measurements

We have implemented GPG based points-to analysis in GCC 4.7.2 using the 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 3.40 GHz. Figure 9 provides the empirical data.

Our method eliminates local variables using the SSA form and GPGs are computed only for global variables. Eventually, the points-to information for local variables is computed from that of global variables and parameters. Our implementation over-approximates an array by treating it as a single variable and maintains its information flow-insensitively. Heap memory is approximated by maintaining indirection lists of field dereferences of length 2 (see [4]). Unlike the conventional approaches [25,27,28], our summary flow functions do not depend on aliasing at the call points. The actually observed number of aliasing patterns (column S in Fig. 9) suggests that it is undesirable to indiscriminately construct multiple PTFs for a procedure.

Columns A, B, P, and Q present the details of the benchmarks. Column C provides the time required for the first phase of our analysis i.e., computing GPGs. The computation of points-to information at each program point has four variants (using GPGs or *Stmt-ff* with or without bypassing). Their time measurements are provided in columns D, E, F, and G. Our data indicates

Program	kLoC	# of pointer stmts	Time for GPG based approach (in seconds)					Avg. # of pointees per pointer					Avg. # of pointees per dereference		
			GPG Constr.	computing points-to info				GPG			GCC	LFCPA			
				GPG NoByp	GPG Byp	*Stmt-ff* NoByp	*Stmt-ff* Byp	G/NoByp (per stmt)	G/Byp (per stmt)	L+Arr (per proc)	G+L+Arr (per proc)	G+L+Arr (per stmt)	GPG	GCC	LFCPA
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*	*O*
lbm	0.9	370	0.10	0.22	0.21	0.26	0.28	1.31	1.42	2.21	17.74	0.05	1.09	2.25	1.50
mcf	1.6	480	75.29	33.73	30.05	1.25	0.91	18.73	6.10	10.48	34.74	1.22	4.25	2.57	0.62
libquantum	2.6	340	6.47	10.23	1.95	8.21	1.85	139.50	22.50	1.11	4.49	3.34	1.50	2.93	0.83
bzip2	5.7	1650	3.17	11.11	8.71	4.73	3.30	43.39	8.38	1.89	31.46	0.94	1.72	2.94	0.33
milc	9.5	2540	7.36	6.08	5.89	4.29	5.61	21.15	16.32	4.52	14.06	31.73	1.18	2.58	1.61
sjeng	10.5	700	9.36	39.66	25.75	14.75	7.56	445.22	64.81	3.07	2.68	-	0.98	2.71	-
hmmer	20.6	6790	38.23	51.73	14.86	31.32	13.50	43.49	5.85	6.05	59.35	1.56	1.04	3.62	0.91
h264ref	36.1	17770	208.47	1262.07	199.34	457.26	74.62	219.71	9.24	16.29	98.84	-	0.98	3.97	-
gobmk	158.0	212830	652.78	3652.99	1624.46	1582.62	1373.88	11.98	1.73	6.34	4.08	-	0.65	3.71	-

Program	# of call sites	# of procs.	Proc. count for different buckets of # of calls (reuse of GPGs)				# of procs. requiring different no. of PTFs based on the no. of aliasing patterns						# of procs. for different sizes of GPG in terms of the number of edges						# of procs. for different % of context ind. info. (for non-empty GPGs)				# of inconclusive compositions
							Actually observed				Predicted												
			2-5	5-10	10-20	20+	2-5	6-10	11-15	15+	2-5	15+	0	1-2	3-4	5-8	9-50	50+	< 20	20-40	40-60	60+	
	P	*Q*	*R*				*S*				*T*		*U*						*V*				*W*
lbm	30	19	5	0	0	0	8	0	0	0	13	0	13	4	2	0	0	0	3	0	0	3	0
mcf	29	23	11	0	0	0	0	0	0	0	4	0	10	5	2	3	2	1	5	1	1	6	1
libquantum	277	80	24	11	4	3	7	3	1	0	14	4	42	10	7	12	9	0	20	12	1	5	0
bzip2	288	89	35	7	2	1	22	0	0	0	28	2	62	13	4	5	5	0	26	0	0	1	1
milc	782	190	60	15	9	1	37	8	0	1	35	25	157	11	19	2	7	0	6	10	9	14	3
sjeng	726	133	46	20	5	6	14	3	1	3	10	14	99	20	6	3	5	0	3	4	10	17	0
hmmer	1328	275	93	33	22	11	62	5	3	4	88	32	167	56	20	15	15	2	54	20	11	23	4
h264ref	2393	566	171	60	22	16	85	17	5	3	102	46	419	76	23	15	30	3	54	13	27	53	8
gobmk	9379	2697	317	110	99	134	206	30	9	10	210	121	1374	93	8	1083	97	42	41	1192	39	51	0

Fig. 9. Time, precision, size, and effectiveness measurements for GPG Based Points-to Analysis. Byp (Bypassing), NoByp (No Bypassing), *Stmt-ff* (Statement-level flow functions), G (Global pointers), L (Local pointers), Arr (Array pointers).

that the most efficient method for computing points-to information is to use statement-level flow functions and bypassing (column G).

Our analysis computes points-to information flow-sensitively for globals. The following points-to information is stored flow-insensitively: locals (because they are in the SSA form) and arrays (because their updates are conservative). Hence, we have separate columns for globals (columns H and I) and locals+arrays (column J) for GPGs. GCC-PTA computes points-to information flow-insensitively (column K) whereas LFCPA computes it flow-sensitively (column L).

The second table provides measurements about the effectiveness of summary flow functions in terms of *(a)* compactness of GPGs, *(b)* percentage of context independent information, and *(c)* reusability. Column U shows that GPGs are empty for a large number of procedures. Besides, in six out of nine benchmarks, most procedures with non-empty GPGs have a significantly high percentage of context independent information (column V). Thus a top-down approach may involve redundant computations on multiple visits to a procedure whereas a bottom-up approach may not need much work for incorporating the effect of a callee's GPG into that of its callers. Further, many procedures are called multiple times indicating a high reuse of GPGs (column R).

The effectiveness of bypassing is evident from the time measurements (columns E and G) as well as a reduction in the average number of points-to pairs (column I).

We have compared our analysis with GCC-PTA and LFCPA [11]. The number of points-to pairs per function for GCC-PTA (column K) is large because it is partially flow-sensitive (because of the SSA form) and context-insensitive. The number of points-to pairs per statements is much smaller for LFCPA (column L) because it is liveness-based. However LFCPA which in our opinion represents the state of the art in fully flow- and context-sensitive exhaustive points-to analysis, does not seem to scale beyond 35 kLoC. We have computed the average number of pointees of dereferenced variables which is maximum for GCC-PTA (column N) and minimum for LFCPA (column O) because it is liveness driven. The points-to information computed by these methods is incomparable because they employ radically dissimilar features of points-to information such as flow- and context-sensitivity, liveness, and bypassing.

8 Related Work

Section 1 introduced two broad categories of constructing summary flow functions for pointer analysis. Some methods using placeholders require aliasing information in the calling contexts and construct multiple summary flow functions per procedure [25,28]. Other methods do not make any assumptions about the calling contexts [12,13,20,23,24] but they construct larger summary flow functions causing inefficiency in fixed point computation at the intraprocedural level thereby prohibiting flow-sensitivity for scalability. Also, these methods cannot perform strong updates thereby losing precision.

Among the general frameworks for constructing procedure summaries, the formalism proposed by Sharir and Pnueli [21] is limited to finite lattices of data flow values. It was implemented using graph reachability in [14,18,19]. A general technique for constructing procedure summaries [5] has been applied to unary uninterpreted functions and linear arithmetic. However, the program model does not include pointers.

Symbolic procedure summaries [25,27] 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 (as mentioned in Sect. 1). We overcome this problem using generalized points-to facts. Saturn [6] also creates summaries that 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 [28]. 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.

9 Conclusions and Future Work

Constructing bounded summary flow functions for flow and context-sensitive points-to analysis seems hard because it requires modelling unknown locations accessed indirectly through pointers—a callee procedure's summary flow function is created without looking at the statements in the caller procedures. Conventionally, they have been modelled using placeholders. However, a fundamental problem with the placeholders is that they explicate the unknown 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 generalized points-to graph (GPG) whose edges track indirection levels and represent generalized points-to facts. A simple arithmetic on indirection levels allows composing generalized points-to facts to create new generalized points-to facts with smaller indirection levels; this reduces them progressively to classical points-to facts. Since unknown locations are left implicit, no information about aliasing patterns in the calling contexts is required allowing us to construct a single GPG per procedure. GPGs are linearly bounded by the number of variables, are flow-sensitive, and are able to perform strong updates within calling contexts. Further, GPGs inherently support bypassing of irrelevant points-to information thereby aiding scalability significantly.

Our measurements on SPEC benchmarks show that GPGs 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]). We expect to scale the method to still larger programs by *(a)* using memoisation, and *(b)* constructing and applying GPGs incrementally thereby eliminating redundancies within fixed point computations.

Observe that a GPG edge $x \xrightarrow{i,j} y$ in M also asserts an alias relation between $M^i\{x\}$ and $M^j\{y\}$ and hence GPGs generalize both points-to and alias relations.

The concept of GPG provides a useful abstraction of memory involving pointers. The way matrices represent values as well as transformations, GPGs represent memory as well as memory transformers defined in terms of loading, storing, and copying memory addresses. Any analysis that is influenced by these operations may be able to use GPGs by combining them with the original abstractions of the analysis. We plan to explore this direction in the future.

Acknowledgments. We are grateful to anonymous reviewers for incisive comments which helped in improving the paper significantly. The paper has benefited from the feedback of many people; in particular, Supratik Chakraborty and Sriram Srinivasan gave excellent suggestions for improving the accessibility of the paper. Our ideas have also benefited from discussions with Amitabha Sanyal, Supratim Biswas, and Venkatesh Chopella. The seeds of GPGs were explored in a very different form in the Master's thesis of Shubhangi Agrawal in 2010.

References

1. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
2. Dillig, I., Dillig, T., Aiken, A.: Sound, complete and scalable path-sensitive analysis. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008. ACM, New York (2008)
3. Feng, Y., Wang, X., Dillig, I., Dillig, T.: Bottom-up context-sensitive pointer analysis for Java. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 465–484. Springer, Heidelberg (2015). doi:10.1007/978-3-319-26529-2_25
4. Gharat, P.M., Khedker, U.P.: Flow and context sensitive points-to analysis using generalized points-to graphs. CoRR (2016). arXiv:1603.09597
5. Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 253–267. Springer, Heidelberg (2007)
6. Hackett, B., Aiken, A.: How is aliasing used in systems software? In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT 2006/FSE-14. ACM, New York (2006)
7. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001. ACM, New York (2001)
8. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
9. Khedker, U.P., Sanyal, A., Sathe, B.: Data Flow Analysis: Theory and Practice. Taylor & Francis (CRC Press, Inc.), Boca Raton (2009)

10. Khedker, U.P., Karkare, B.: Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 213–228. Springer, Heidelberg (2008)
11. Khedker, U.P., Mycroft, A., Rawat, P.S.: Liveness-based pointer analysis. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 265–282. Springer, Heidelberg (2012)
12. Li, L., Cifuentes, C., Keynes, N.: Precise and scalable context-sensitive pointer analysis via value flow graph. In: Proceedings of the 2013 International Symposium on Memory Management, ISMM 2013. ACM, New York (2013)
13. Madhavan, R., Ramalingam, G., Vaswani, K.: Modular heap analysis for higher-order programs. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 370–387. Springer, Heidelberg (2012)
14. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical extensions to the IFDS algorithm. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 124–144. Springer, Heidelberg (2010)
15. Hakjoo, O., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for C-like languages. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, 11–16 June 2012
16. Hakjoo, O., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, UK, 09–11 June 2014
17. Padhye, R., Khedker, U.P.: 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 2013. ACM, New York (2013)
18. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995. ACM, New York (1995)
19. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. In: Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development, TAPSOFT 1995. Elsevier Science Publishers B. V., Amsterdam (1996)
20. Shang, L., Xie, X., Xue, J.: On-demand dynamic summary-based points-to analysis. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO 2012. ACM, New York (2012)
21. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, Chap. 7 (1981)
22. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: 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 2005. ACM, New York (2005)
23. Sălcianu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
24. Whaley, J., Rinard, M.: 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 1999. ACM, New York (1999)

25. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1995 (1995)
26. Yan, D., Guoqing, X., Rountev, A.: 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 2012. ACM, New York (2012)
27. Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008. ACM, New York (2008)
28. Zhang, X., Mangal, R., Naik, M., Yang, H.: Hybrid top-down and bottom-up interprocedural analysis. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014. ACM, New York (2014)