Heterogeneous Fixed Points with Application to Points-to Analysis

Aditya Kanade, Uday Khedker, and Amitabha Sanyal

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay.
{aditya,uday,as}@cse.iitb.ac.in

Abstract. Many situations can be modeled as solutions of systems of simultaneous equations. If the functions of these equations monotonically increase in all bound variables, then the existence of extremal fixed point solutions for the equations is guaranteed. Among all solutions, these fixed points uniformly take least or greatest values for all bound variables. Hence, we call them *homogeneous* fixed points. However, there are systems of equations whose functions monotonically increase in some variables and decrease in others. The existence of solutions of such equations cannot be guaranteed using classical fixed point theory. In this paper, we define general conditions to guarantee the existence and computability of fixed point solutions of such equations. In contrast to homogeneous fixed points, these fixed points take least values for some variables and greatest values for others. Hence, we call them *heterogeneous* fixed points. We illustrate heterogeneous fixed point theory through points-to analysis.

1 Introduction

Many situations can be modeled as solutions of systems of simultaneous equations. If the functions of these equations monotonically increase in all bound variables, then the existence of extremal fixed point solutions for the equations is guaranteed through Knaster-Tarski fixed point existence theorem [14, 17]. Among all solutions, these fixed points uniformly take least or greatest values for all bound variables. Hence, we call them *homogeneous*.

However, there are systems whose functions monotonically increase in some variables and decrease in others. Emami's (intraprocedural) points-to analysis [4] exhibits this behavior. This analysis computes a variant of may and must aliases in terms of points-to abstraction. The aliases that hold along some but *not* along all paths are captured by *possibly* points-to relation. The aliases that hold along all paths are captured by *definite* points-to relation. While their algorithm performs a fixed point computation, the monotonicity of the functions is not obvious. Consequently the existence and computability of the fixed point solutions cannot be assumed.

The definite and possible points-to relations have both positive as well as negative dependences amongst themselves. Such *heterogeneous dependences* are inherent to points-to analysis. If these mutual dependences are *consistent* in a

manner we define later, then the existence of fixed points can be guaranteed. These fixed points called *heterogeneous fixed points* take least values for some variables and greatest values for others. We generalize Knaster-Tarski fixed point existence theorem [14, 17] to heterogeneous fixed points.

In section 2, we show that monotonicity of functions in Emami's points-to analysis is not obvious. We then reformulate points-to analysis so that the heterogeneous dependences can be better understood. In section 3, we identify conditions for consistency of heterogeneous dependences so that the existence of fixed points can be assured. In section 4, we define a property called *heterogeneous monotonicity* which captures the consistency conditions. We also define heterogeneous fixed points and show that the former guarantees the existence of latter. Finally, in section 5, we define the solution of our points-to analysis using heterogeneous fixed point theory.

2 Points-to Analyses

In this section, we show that monotonicity of functions in Emami's points-to analysis is not obvious. We then reformulate the analysis to explicate heterogeneous dependences. A brief overview of Emami's points-to analysis is provided in the appendix.

2.1 Monotonicity Issues in Emami's Points-to Analysis

Emami's points-to analysis [4] computes *points-to relation* between pointer expressions. This relation has elements of the following types:

- *Definite Points-To*. A triple (p_1, p_2, D) holds at a program point if the stack location denoted by p_1 contains address of the stack location denoted by p_2 along *every* execution path reaching that point.
- *Possibly Points-To*. A triple (p_1, p_2, P) holds at a program point if the stack location(s) denoted by p_1 contains address(es) of the stack location(s) denoted by p_2 along *some* execution paths reaching that point.

We abstract the algorithm in [4] as data flow equations. In this paper, we restrict ourselves to intraprocedural analysis and a subset of the language in [4].

The points-to relation at IN of a node i is a confluence of points-to relations at OUT of its predecessors $p_1, \ldots, p_k$.

$$\mathsf{input}_i = \begin{cases} \mathsf{Merge}\left(\mathsf{output}_{p_1}, \ldots, \mathsf{output}_{p_k}\right) & \text{if } i \neq \text{entry} \\ \phi & \text{if } i = \text{entry} \end{cases} \tag{1}$$

where "entry" is the unique entry node of the procedure and the Merge operation [3], defined below, is extended to multiple arguments in an obvious way.

$$\begin{aligned} \mathsf{Merge}(S_1, S_2) = \{(p_1, p_2, D) \mid (p_1, p_2, D) \in S_1 \cap S_2\} \cup \\ \{(p_1, p_2, P) \mid (p_1, p_2, r) \in S_1 \cup S_2 \wedge (p_1, p_2, D) \notin S_1 \cap S_2\} \end{aligned} \tag{2}$$

Note that the definition of Merge excludes the definite information along all paths from being considered as possible points-to information.

Let (x, y, r) hold before an assignment in node i where r is either D or P.

- If x is only likely to be modified, then after the assignment, x may or may not point to y. Hence the definiteness of its pointing to y must be changed to the possibility of its pointing to y. This is captured by the property $\mathsf{changed\_input}_i$ (ref. Appendix A: 24 and 25).
- If x is definitely modified as a side effect of the assignment, then x ceases to point to y. This is captured by the property $\mathsf{kill\_set}_i$ (Appendix A: 26).

An *R-location* represents the variable whose address appears in the rhs. An *L-location* represents the variable which is being assigned this address. Both of the *L-location* and *R-location* depend on the nature of points-to information and can be either *definite* or *possible*.

An assignment generates definite points-to information between its definite L-locations and definite R-locations. All other combinations between its L-locations and R-locations are generated as possibly points-to information. This is captured by the property $\mathsf{gen\_set}_i$ (Appendix A: 27).

Finally, the points-to information at OUT of node i is

$$\mathsf{output}_i = (\mathsf{changed\_input}_i - \mathsf{kill\_set}_i) \cup \mathsf{gen\_set}_i \tag{3}$$

The existence of a fixed point solution requires that all functions in the system of equations should be monotonic in an appropriate lattice. As [4] does not define a partial order over the points-to information domain, we have to assume it. Since the values being computed are sets of points-to triples, we embed them in a lattice with set inclusion as the natural partial order.

Example 1. Let a node i contain the following assignment : $*x = \&y$. Consider the following two cases:

1. Let $\mathsf{input}_i = \{(d, a, D), (x, b, P), (x, c, P)\}$. Then,

$$\mathsf{output}_i = \{(d, a, D), (x, b, P), (x, c, P), (b, y, P), (c, y, P)\}.$$

2. Let $\mathsf{input}'_i = \mathsf{input}_i \cup \{(x, d, P), (b, y, P), (c, y, P)\}$. The resulting output'_i is

$$\mathsf{output}'_i = \{(d, a, P), (x, b, P), (x, c, P), (x, d, P), (b, y, P), (c, y, P), (d, y, P)\}.$$

Clearly, output_i and output'_i are incomparable and

$$\mathsf{input}_i \subseteq \mathsf{input}'_i \not\Rightarrow \mathsf{output}_i \subseteq \mathsf{output}'_i$$

Hence the flow function is non-monotonic w.r.t. set inclusion as partial order. □

As can be seen from the example, an increase in the possible points-to information in input has increased the possible points-to information in output and has decreased the definite points-to information. Similarly, it can be shown that

an increase in the definite points-to information in input results in increase of the definite points-to information in output and decrease of the possible points-to information. This is an instance of heterogeneous dependences. While this behavior is inherent in points-to analysis, the existence of and convergence to a fixed point is not guaranteed in general unless monotonicity of functions can be established.

Consider yet another partial order $\leq$ in which the D/P tags in the points-to triples also determine the ordering.

$$S_1 \leq S_2 \iff ((x,y,P) \in S_1 \implies (x,y,P) \in S_2) \wedge \\ ((x,y,D) \in S_1 \implies (x,y,r) \in S_2, \text{ where } r = D/P)$$

While the flow function could be monotonic under this partial order, Merge still exhibits non-monotonicity.

$$\mathsf{Merge}\,(\{(x,y,D)\},\phi) = \{(x,y,P)\}$$
$$\mathsf{Merge}\,(\{(x,y,D)\},\{(x,y,D)\}) = \{(x,y,D)\}$$

Though $\phi \leq \{(x,y,D)\}$, $\mathsf{Merge}(\{(x,y,D)\},\phi) \not\leq \mathsf{Merge}(\{(x,y,D)\},\{(x,y,D)\})$.

In summary, the monotonicity of functions in Emami's analysis has not been addressed and is not obvious. Consequently the existence and computability of the fixed point solution cannot be assumed.

2.2 May-Must Points-To Analysis

We now reformulate Emami's analysis to explicate the heterogeneous dependences. As is customary in data flow analysis [12, 11], we associate data flow information with IN and OUT of a node. Let $\mathsf{MustIN}_i/\mathsf{MayIN}_i$ be respectively must and may data flow properties at IN of a node i. Let $\mathsf{MustOUT}_i/\mathsf{MayOUT}_i$ be respectively must and may data flow properties at OUT of a node i. Unlike [4], we compute *inclusive* may information implying that MayIN_i and MayOUT_i information also includes points-to information which holds along all paths reaching node i. Our may information corresponds to both definite and possible information whereas our must information corresponds to definite information only.

Since we use separate data flow properties for may and must information, we do not need the third component of points-to triples (D/P). Let U be the universal set containing all type correct points-to pairs $\langle p_1, p_2\rangle$. The lattice of data flow information is $(\wp(U), \subseteq, \cup, \cap, U, \phi)$, where $\wp(U)$ is power set of U and $\subseteq$ is the partial order. Hereafter, we denote this complete lattice by $(\wp(U), \subseteq)$.

The must L-locations and R-locations are represented by $\mathsf{MustL}_i/\mathsf{MustR}_i$ and the may L-locations and R-locations by $\mathsf{MayL}_i/\mathsf{MayR}_i$. Let x be a variable and '&' and '$*$' respectively be dereferencing and referencing operators. The must and may R-locations and L-locations are defined in Table 1.

The points-to analysis is a forward problem as points-to information flows along the control flow of program. The must points-to problem being an all path problem, MustIN of a node is intersection of MustOUT of all its predecessors. In

lhs_i	MustL_i	MayL_i
x	$\{x\}$	$\{x\}$
$*x$	$\{y \mid \langle x, y\rangle \in \mathsf{MustIN}_i\}$	$\{y \mid \langle x, y\rangle \in \mathsf{MayIN}_i\}$

rhs_i	MustR_i	MayR_i
$\&x$	$\{x\}$	$\{x\}$
x	$\{y \mid \langle x, y\rangle \in \mathsf{MustIN}_i\}$	$\{y \mid \langle x, y\rangle \in \mathsf{MayIN}_i\}$
$*x$	$\{z \mid \langle x, y\rangle, \langle y, z\rangle \in \mathsf{MustIN}_i\}$	$\{z \mid \langle x, y\rangle, \langle y, z\rangle \in \mathsf{MayIN}_i\}$

Table 1. Definitions of L-locations and R-locations.

the absence of interprocedural information, MustIN of the entry node is initialized to empty set. The data flow equations for MustIN_i and $\mathsf{MustOUT}_i$ are as follows:

$$\mathsf{MustIN}_i = \begin{cases} \bigcap\limits_{p \in pred(i)} \mathsf{MustOUT}_p & \text{if } i \neq \text{entry} \\ \phi & \text{if } i = \text{entry} \end{cases} \tag{4}$$

$$\mathsf{MustOUT}_i = (\mathsf{MustIN}_i - \mathsf{MustKill}_i) \cup \mathsf{MustGen}_i \tag{5}$$

where $pred(i)$ is set of all predecessors of node i. This is a conventional form of data flow analysis which employs IN, OUT, Gen, and Kill properties. Emami's analysis involves an additional property for the "changed" input set.

Since an assignment potentially updates any of its *may* L-locations, all must points-to pairs from the *may* L-locations are killed. The set of such pairs is denoted by $\mathsf{MustKill}_i$. Further, an assignment generates must points-to pairs between all must L-locations and must R-locations. They are contained in the set $\mathsf{MustGen}_i$.

$$\mathsf{MustKill}_i = \{\langle x, y\rangle \mid \underline{x \in \mathsf{MayL}_i} \ \wedge \ \langle x, y\rangle \in \mathsf{MustIN}_i\} \tag{6}$$

$$\mathsf{MustGen}_i = \{\langle x, y\rangle \mid x \in \mathsf{MustL}_i \ \wedge \ y \in \mathsf{MustR}_i\} \tag{7}$$

The may points-to problem is some path problem and hence, MayIN of a node is union of MayOUT of its predecessors. Again, in the absence of interprocedural information, MayIN of the entry node is initialized to empty set. The data flow equations for MayIN_i and MayOUT_i are as follows:

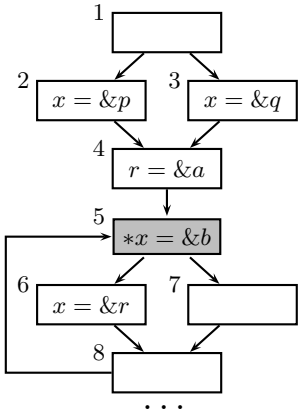
$$\mathsf{MayIN}_i = \begin{cases} \bigcup\limits_{p \in pred(i)} \mathsf{MayOUT}_p & \text{if } i \neq \text{entry} \\ \phi & \text{if } i = \text{entry} \end{cases} \tag{8}$$

$$\mathsf{MayOUT}_i = (\mathsf{MayIN}_i - \mathsf{MayKill}_i) \cup \mathsf{MayGen}_i \tag{9}$$

An assignment kills all may points-to pairs from any *must* L-location of the assignment. The set of such pairs is denoted by $\mathsf{MayKill}_i$. Further, an assignment generates may points-to pairs between all may L-locations and may R-locations. They are contained in the set MayGen_i.

$$\mathsf{MayKill}_i = \{\langle x, y\rangle \mid \underline{x \in \mathsf{MustL}_i} \ \wedge \ \langle x, y\rangle \in \mathsf{MayIN}_i\} \tag{10}$$

$$\mathsf{MayGen}_i = \{\langle x, y\rangle \mid x \in \mathsf{MayL}_i \ \wedge \ y \in \mathsf{MayR}_i\} \tag{11}$$



MustIN_5^1	$\{\langle r, a\rangle\}$
MayIN_5^1	$\{\langle r, a\rangle, \langle x, p\rangle, \langle x, q\rangle\}$
$\mathsf{MustOUT}_5^1$	$\{\langle r, a\rangle\}$
MayOUT_5^1	$\{\langle r, a\rangle, \langle x, p\rangle, \langle x, q\rangle, \langle p, b\rangle, \langle q, b\rangle\}$

After first iteration

MustIN_5^2	$\{\langle r, a\rangle\}$
MayIN_5^2	$\{\langle r, a\rangle, \langle x, p\rangle, \langle x, q\rangle, \langle x, r\rangle, \langle p, b\rangle, \langle q, b\rangle\}$
$\mathsf{MustOUT}_5^2$	ϕ
MayOUT_5^2	$\{\langle r, a\rangle, \langle x, p\rangle, \langle x, q\rangle, \langle x, r\rangle, \langle p, b\rangle, \langle q, b\rangle, \langle r, b\rangle\}$

After second iteration

Fig. 1. How MayIN_i affects MayOUT_i and $\mathsf{MustOUT}_i$

The may-must analysis is *not* a simple combination of the union and intersection data flow analyses. Usually the dependences among the data flow variables are all positive. In this case, there are negative dependences as well.

3 Consistency of Dependences

The nature of the underlying dependences in points-to analysis brought out by may-must formulation is analyzed in this section. We identify the conditions which guarantee existence of fixed points in presence of such dependences.

3.1 Positive and Negative Dependences

From the underlined terms in the data flow equations (4) – (11), it is clear that $\mathsf{MustOUT}_i$ decreases with increase in MayIN_i and MayOUT_i decreases with increase in MustIN_i.

Definition 1. *A variable* x depends *on a variable* y *iff* x *is defined in terms of* y *and there exist at least two distinct values of* y *such that the corresponding values of* x *are distinct, keeping rest of the variables constant.*

If the non-decreasing values of y result in the non-decreasing values of x then x depends *positively* on y. Otherwise, x depends *negatively* on y.

Example 2. Consider the program flow graph shown in Figure 1. To create maximum optimization opportunities, we want the largest set of must points-to pairs and the smallest set of may points-to pairs which together form the solution of may-must analysis. Hence, we initialize the data flow variables as follows:

$$\mathsf{MustIN}_i = \mathsf{MustOUT}_i = U \tag{12}$$

$$\mathsf{MayIN}_i = \mathsf{MayOUT}_i = \phi \tag{13}$$

where U is the universal set containing all type correct points-to pairs.

We compute the data flow properties using round robin iterative method in which nodes are visited in the reverse depth first order. Let P_i^j be a property P at node i in iteration j. The data flow values at node 5 after first and second iterations over the program flow graph are shown in Figure 1.

MustIN_5 remains the same in first and second iterations, but MayIN_5 increases in second iteration which causes $\mathsf{MustOUT}_5$ to decrease. Thus, the dependence of $\mathsf{MustOUT}_i$ on MayIN_i is negative. MayOUT_5 increases in second iteration and hence the dependence of MayOUT_i on MayIN_i is positive. □

Similarly, it can be shown that $\mathsf{MustOUT}_i$ depends positively on MustIN_i and MayOUT_i depends negatively on MustIN_i. The dependences in may-must data flow equations can be summarized as follows:

D1. The dependence of MustIN_i on $\mathsf{MustOUT}_p$, $p \in pred(i)$, is positive (4).
D2. The dependence of $\mathsf{MustOUT}_i$ on MustIN_i is positive but that on MayIN_i is negative (5, 6, and 7).
D3. The dependence of MayIN_i on MayOUT_p, $p \in pred(i)$, is positive (8).
D4. The dependence of MayOUT_i on MayIN_i is positive but that on MustIN_i is negative (9, 10, and 11).

Since not all dependences between the variables are positive, the existence and computability of fixed point solutions of the equations cannot be guaranteed using the classical results [14, 13, 17]. However, as we demonstrate later if the dependences are mutually consistent, the existence of fixed points can be guaranteed.

3.2 Consistency of Dependences

Consistency of dependences can be defined in terms of a dependence graph. Nodes in this graph represent the bound variables. If a variable x depends positively on a variable y, then there is a solid edge from x to y. If x depends negatively on y, then there is a dashed edge from x to y. If a variable x does not depend on y, then there is no edge from x to y.

The *parity* of a path in a dependence graph is even if the path has an even number of dashed edges, otherwise its parity is odd. If all paths between every pair of nodes have the same parity, then the dependences between those variables are consistent. If the parity is even then an increase in one's value leads to an increase in other's and vice versa. If the parity is odd then an increase in one's value leads to a decrease in other's and vice versa. If the paths are of different parities then the mutual influences cannot be determined.

Definition 2. *Dependences in a system of simultaneous equations are consistent iff for every pair of nodes (x, y) contained in a strongly connected component of the dependence graph, all paths between x and y have the same parity.*

For simplicity, we assume that the dependence graph of the variables has a single maximal strongly connected component. The systems that have more than one maximal strongly connected components in their dependence graphs are discussed in [10].

4 Heterogeneous Fixed Points

In the classical setting, monotonicity and fixed points of a set of functions are straightforward generalizations of the corresponding definitions of the individual functions. These generalizations are *uniform*. Hence, we call them *homogeneous*.

To capture systems like may-must data flow equations, we generalize these formulations so that they *need not* be uniform over the components. We call our formulations *heterogeneous*. Here, we present only relevant part of the formulation and associated results. A more detailed treatment can be found in [10]. We now introduce some terminology used.

Let S_n be a system of n ($n > 0$) simultaneous equations in n variables:

$$\begin{aligned} x_1 &= f_1(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, \ldots, x_n) \end{aligned}$$

The functions $f_1, \ldots, f_n$ are called the *component functions* of S_n. Let $\boldsymbol{F}$ be a function defined as $\boldsymbol{F}(\boldsymbol{X}) = \langle f_1(\boldsymbol{X}), \ldots, f_n(\boldsymbol{X})\rangle$ where $\boldsymbol{X} = \langle x_1, \ldots, x_n\rangle$. We call $\boldsymbol{F}$ the *function vector* of S_n. The variables $x_1, \ldots, x_n$ which appear on the left side of the equalities are called the *bound* variables of S_n.

We assume that a bound variable x_i takes values from a finite[1] complete lattice $L_i = (L_i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \top_i, \bot_i)$, where $\sqsubseteq_i$ is the partial order over the set L_i, $\sqcup_i$ and $\sqcap_i$ are respectively join and meet of the lattice, and $\top_i$ and $\bot_i$ are respectively the top and bottom of the lattice. Let $\boldsymbol{L} = (\boldsymbol{L}, \sqsubseteq, \sqcup, \sqcap, \top, \bot) = L_1 \times \cdots \times L_n$. A function f_i has type $\boldsymbol{L} \to L_i$. The function vector $\boldsymbol{F}$ has type $\boldsymbol{L} \to \boldsymbol{L}$.

4.1 Heterogeneous Monotonicity

Consider a partition (P, Q) of the set $\{1, \ldots, n\}$. We define the heterogeneous monotonicity of a function vector with respect to a partition (P, Q) as follows :

Definition 3. *A function vector $\boldsymbol{F} : \boldsymbol{L} \to \boldsymbol{L}$ is* heterogeneously monotonic *(or simply* h-monotonic*) w.r.t. a partition (P, Q) iff*

1. *for an $i \in P$, f_i monotonically increases in x_j, if $j \in P$ and monotonically decreases in x_j, if $j \in Q$ and*
2. *for an $i \in Q$, f_i monotonically increases in x_j, if $j \in Q$ and monotonically decreases in x_j, if $j \in P$.*

If the function vector $\boldsymbol{F}$ is h-monotonic w.r.t. a partition (P, Q), then (P, Q) is called a *valid* partition for the system. The classical monotonicity of a function vector $\boldsymbol{F}$ is a special case of h-monotonicity with $(\{1, \ldots, n\}, \phi)$ and $(\phi, \{1, \ldots, n\})$ as (the only) valid partitions.

[1] For simplicity, we consider finite lattices. More general treatment is available in [10].

A function f_i monotonically increases in a variable x_j iff the variable x_i depends positively on the variable x_j. A function f_i monotonically decreases in a variable x_j iff the variable x_i depends negatively on the variable x_j.

Lemma 1. *There exists a valid partition for a system S_n iff the dependences in S_n are consistent.*

Proof. Let there be a valid partition but the dependences in S_n not be consistent. There exist two variables x_i and x_j such that ρ_1 and ρ_2 are two paths between them and the parity of ρ_1 is even and that of ρ_2 is odd. From the definition of h-monotonicity and composibility of the dependences, due to the dependences along the path ρ_1, i and j should belong to the same set of a partition. Similarly, due to the dependences along the path ρ_2, i and j should belong to the different sets of a partition. This is a contradiction.

Let there be no valid partition but the dependences in S_n be consistent. There exist two variables which can be placed in the same as well as the different sets. Clearly, there exist two paths between them which have different parities. Hence, the dependences in S_n are not consistent. □

If i and j belong to the same set in a valid partition, then x_i and x_j have even parity paths between them in the dependence graph. If i and j belong to different sets in a valid partition, then x_i and x_j have odd parity paths between them in the dependence graph. That is, an increase in the value of a variable in a set can lead only to an increase in the values of the variables in that set and decrease in the values of the variables in the other set.

4.2 Identifying Valid Partitions

We give an algorithm called Even-Odd-Analysis to identify the valid partitions given the dependence graph of a system. This algorithm returns valid partitions iff the dependences in S_n are consistent. Let $D = (V, E)$ be the dependence graph of a system of equations, where V is the set of bound variables and E is the set of edges representing the dependences among the variables.

Let *dependence* be a property of edges. A value 1 of $dependence(\langle x_u, x_v \rangle)$ denotes a solid edge from x_u to x_v while a value -1 denotes a dashed edge from x_u to x_v. Let $membership(x_u)$ denote to which set of the partition the variable x_u belongs. The function Initialize initializes the *dependence* properties according to the monotonicities of the functions and assigns a value 0 to *membership* property of all variables to indicate that their membership in the sets of a valid partition is yet to be determined.

Initialize(D).

```
1.  for each ⟨x_u, x_v⟩ ∈ E
2.      if f_u monotonically increases in x_v then
3.          dependence(⟨x_u, x_v⟩) ←  1 /* solid edge */
4.      else
```

5. $dependence(\langle x_u, x_v \rangle) \leftarrow -1$ /* dashed edge */
6. **for** each $x_u \in V$
7. $membership(x_u) \leftarrow 0$
8. **return**

Let Strongly-Connected-Components be a function which takes a graph and returns the strongly connected components in it as sets of sets of nodes. Select-Node selects an element from a set. Even-Odd-Analysis first initializes the properties explained above. It then selects a node from a strongly connected component and assigns it *membership* in a set. It invokes a function Dft which traverses the graph in depth-first order and determines memberships of nodes iff the dependences are consistent. If Dft returns a 0 then the dependences are inconsistent and there is no valid partition. Otherwise, the sets of the valid partition can be constructed from the *membership* properties of nodes.

Even-Odd-Analysis(D).

1. Call Initialize(D)
2. $SCC \leftarrow$ Strongly-Connected-Components(D)
3. **for** each $C \in SCC$
4. $x_u \leftarrow$ Select-Node(C)
5. $membership(x_u) \leftarrow 1$
6. $success \leftarrow$ Dft(x_u, C)
7. **if** $success = 0$ **then**
8. **print** "No partitions possible for the component $[C]$"
9. **else**
10. $P \leftarrow \{\, u \in \{1, \ldots, n\} \mid x_u \in C \wedge membership(x_u) = \ \ 1 \,\}$
11. $Q \leftarrow \{\, u \in \{1, \ldots, n\} \mid x_u \in C \wedge membership(x_u) = -1 \,\}$
12. **print** "Partitions for component $[C]$ are $([P], [Q])$ and $([Q], [P])$"
13. **return**

Dft takes a node x_u and a strongly connected graph C. For every neighbour x_v of x_u in C, it checks whether x_v has been visited previously. If not then it assigns x_v a membership consistent with the dependence $\langle x_u, x_v \rangle$. If the dependence is 1, then x_v goes to the same set else it goes to the other set. It then calls itself on x_v and C. If failure is returned then it propagates it upwards. Otherwise it analyzes other neighbours of x_u. If x_v has been visited, then it verifies the consistency of membership of x_v w.r.t. the dependence $\langle x_u, x_v \rangle$. If the dependences are inconsistent, it returns a failure, otherwise it goes to next neighbour of x_u. Finally, returns a success status.

Dft(x_u, C).

1. **for** each $x_v \in C$ such that $\langle x_u, x_v \rangle \in E$ /* for every neighbour of x_u */
   /* if unvisited, assign membership consistent with dependence $\langle x_u, x_v \rangle$*/
2. **if** $membership(x_v) = 0$ **then**
3. $membership(x_v) \leftarrow dependence(\langle x_u, x_v \rangle) \times membership(x_u)$
4. $success \leftarrow$ Dft(x_v, C) /* traverse recursively */

5. **if** $success = 0$ **then return** 0 /* propagate failure upwards */
/* if visited, check for inconsistency with the dependence $\langle x_u, x_v \rangle$ */
6. **elseif** $membership(x_v) \neq dependence(\langle x_u, x_v \rangle) \times membership(x_v)$ **then**
7. **return** 0 /* if inconsistent, return the failure status */
8. **return** 1 /* return the success status */

4.3 Relating to Homogeneity

We now relate heterogeneity with classical homogeneity. We construct a lattice

$$\boldsymbol{L}^{(\mathrm{P,Q})} = \left(\boldsymbol{L}^{(\mathrm{P,Q})}, \sqsubseteq^{(\mathrm{P,Q})}, \sqcup^{(\mathrm{P,Q})}, \sqcap^{(\mathrm{P,Q})}, \top^{(\mathrm{P,Q})}, \bot^{(\mathrm{P,Q})} \right)$$

as a product of the component lattices or their *duals*[2] as defined below:

$$\boldsymbol{L}^{(\mathrm{P,Q})} = \boldsymbol{L}_1^{(\mathrm{P,Q})} \times \cdots \times \boldsymbol{L}_n^{(\mathrm{P,Q})}$$
$$\text{where, } \boldsymbol{L}_i^{(\mathrm{P,Q})} = \begin{cases} L_i & i \in P \\ L_i^{-1} & i \in Q \end{cases}$$

where L_i^{-1} is the dual of L_i and (P, Q) is a partition of $\{1, \ldots, n\}$.

Consider a system $S_n^{(\mathrm{P,Q})}$ whose function vector $\boldsymbol{F}^{(\mathrm{P,Q})} : \boldsymbol{L}^{(\mathrm{P,Q})} \to \boldsymbol{L}^{(\mathrm{P,Q})}$ is isomorphic to $\boldsymbol{F}$. The component functions of $S_n^{(\mathrm{P,Q})}$ are $f_1^{(\mathrm{P,Q})}, \ldots, f_n^{(\mathrm{P,Q})}$. The systems S_n and $S_n^{(\mathrm{P,Q})}$ are called *duals* of each other w.r.t. the partition (P, Q).

Lemma 2. *If the systems S_n and $S_n^{(\mathrm{P,Q})}$ are duals of each other w.r.t. a partition (P, Q), then (P, Q) is a valid partition for S_n iff the function vector $\boldsymbol{F}^{(\mathrm{P,Q})} : \boldsymbol{L}^{(\mathrm{P,Q})} \to \boldsymbol{L}^{(\mathrm{P,Q})}$ of $S_n^{(\mathrm{P,Q})}$ is monotonic.*

Proof. We prove forward implication by considering following two cases:

Case 1. Let $i \in P$. By construction, $\boldsymbol{L}_i^{(\mathrm{P,Q})} = L_i$. Since, the function vector $\boldsymbol{F}$ of S_n is h-monotonic w.r.t. the partition (P, Q),

$$\text{if } j \in P,\ a_j \sqsubseteq_j a'_j \implies f_i(x_1, \ldots, a_j, \ldots, x_n) \sqsubseteq_i f_i(x_1, \ldots, a'_j, \ldots, x_n),$$
$$\text{if } j \in Q,\ a_j \sqsupseteq_j a'_j \implies f_i(x_1, \ldots, a_j, \ldots, x_n) \sqsubseteq_i f_i(x_1, \ldots, a'_j, \ldots, x_n),$$

The component functions f_i and $f_i^{(\mathrm{P,Q})}$ are isomorphic. By construction of the lattice $\boldsymbol{L}^{(\mathrm{P,Q})}$,

$$\text{if } j \in P,\ a_j \sqsubseteq_j^{(\mathrm{P,Q})} a'_j \implies f_i^{(\mathrm{P,Q})}(x_1, \ldots, a_j, \ldots, x_n) \sqsubseteq_i^{(\mathrm{P,Q})} f_i^{(\mathrm{P,Q})}(x_1, \ldots, a'_j, \ldots, x_n)$$
$$\text{if } j \in Q,\ a_j \sqsubseteq_j^{(\mathrm{P,Q})} a'_j \implies f_i^{(\mathrm{P,Q})}(x_1, \ldots, a_j, \ldots, x_n) \sqsubseteq_i^{(\mathrm{P,Q})} f_i^{(\mathrm{P,Q})}(x_1, \ldots, a'_j, \ldots, x_n)$$

Hence, for an $i \in P$, $f_i^{(\mathrm{P,Q})}$ monotonically increases in all bound variables.

Case 2. For an $i \in Q$, $\boldsymbol{L}_i^{(\mathrm{P,Q})} = L_i^{-1}$. By arguments similar to the above case, $f_i^{(\mathrm{P,Q})}$ can be shown to be monotonically increasing in all bound variables.

Thus, all component functions of $S_n^{(\mathrm{P,Q})}$ monotonically increase in all variables and hence, $\boldsymbol{F}^{(\mathrm{P,Q})}$ is monotonic. The converse is by an analogous argument. □

[2] Lattices $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ and $(L, \sqsupseteq, \sqcap, \sqcup, \bot, \top)$ are called duals of each other.

Lemma 3. *A system S_n and its dual system $S_n^{(\mathrm{P,Q})}$ w.r.t. any partition (P, Q) of the set $\{1, \ldots, n\}$ have the same solutions.*

Proof. The systems are isomorphic and are defined over the same sets. □

4.4 Heterogeneous Fixed Points

We have assumed that a component lattice L_i is a complete lattice, hence any subset of L_i has a least upper bound (lub) and a greatest lower bound (glb). Let $Fix(\boldsymbol{F})$ be the set of fixed points of $\boldsymbol{F}$. Let $Fix_i(\boldsymbol{F})$ be the set of elements from $\boldsymbol{L}_i$ that belong to some fixed point of $\boldsymbol{F}$.

We define $\mathrm{hfp}^{(\mathrm{P,Q})}(\boldsymbol{F})$ element-wise such that its ith element $\mathrm{hfp}_i^{(\mathrm{P,Q})}(\boldsymbol{F})$ is glb of $Fix_i(\boldsymbol{F})$, if $i \in P$ and lub of $Fix_i(\boldsymbol{F})$, if $i \in Q$.

$$\mathrm{hfp}_i^{(\mathrm{P,Q})}(\boldsymbol{F}) = \begin{cases} \sqcap_i Fix_i(\boldsymbol{F}) \text{ if } i \in P \\ \sqcup_i Fix_i(\boldsymbol{F}) \text{ if } i \in Q \end{cases} \tag{14}$$

We now show that if (P, Q) is a valid partition for a system S_n, then $\mathrm{hfp}^{(\mathrm{P,Q})}(\boldsymbol{F})$ exists and is a fixed point of the function vector $\boldsymbol{F}$. We call this fixed point, a *heterogeneous fixed point* (HFP) of an $\boldsymbol{F}$ w.r.t. a partition (P, Q). From among the fixed point values, it takes element-wise least possible values from the component lattices with subscripts in P and element-wise greatest possible values from the component lattices with subscripts in Q. We now give an existence theorem for heterogeneous fixed points.

Theorem 1 (HFP Existence Theorem). *If the function vector $\boldsymbol{F} : \boldsymbol{L} \to \boldsymbol{L}$ of a system S_n is h-monotonic w.r.t. a partition (P, Q), then*

$$\mathrm{hfp}^{(\mathrm{P,Q})}(\boldsymbol{F}) \in Fix(\boldsymbol{F})$$

Proof. Since the function vector $\boldsymbol{F}$ is h-monotonic w.r.t. a partition (P, Q), from Lemma 2, $\boldsymbol{F}^{(\mathrm{P,Q})} : \boldsymbol{L}^{(\mathrm{P,Q})} \to \boldsymbol{L}^{(\mathrm{P,Q})}$ is monotonic. By Knaster-Tarski fixed point theorem [14, 17], the least fixed point of $\boldsymbol{F}^{(\mathrm{P,Q})}$, $\mathrm{lfp}\left(\boldsymbol{F}^{(\mathrm{P,Q})}\right)$ exists. We can write $\mathrm{lfp}\left(\boldsymbol{F}^{(\mathrm{P,Q})}\right)$ element-wise as:

$$\mathrm{lfp}_i\left(\boldsymbol{F}^{(\mathrm{P,Q})}\right) = \begin{cases} \sqcap_i^{(\mathrm{P,Q})} Fix_i\left(\boldsymbol{F}^{(\mathrm{P,Q})}\right) \text{ if } i \in P \\ \sqcap_i^{(\mathrm{P,Q})} Fix_i\left(\boldsymbol{F}^{(\mathrm{P,Q})}\right) \text{ if } i \in Q \end{cases} \tag{15}$$

From Lemma 3, $Fix(\boldsymbol{F}) = Fix\left(\boldsymbol{F}^{(\mathrm{P,Q})}\right)$. (15) can be rewritten as:

$$\mathrm{lfp}_i\left(\boldsymbol{F}^{(\mathrm{P,Q})}\right) = \begin{cases} \sqcap_i^{(\mathrm{P,Q})} Fix_i(\boldsymbol{F}) \text{ if } i \in P \\ \sqcap_i^{(\mathrm{P,Q})} Fix_i(\boldsymbol{F}) \text{ if } i \in Q \end{cases} \tag{16}$$

From the construction of the dual lattice $\boldsymbol{L}^{(\mathrm{P,Q})}$, $\sqcap_i^{(\mathrm{P,Q})} = \sqcap_i$, if $i \in P$ and $\sqcap_i^{(\mathrm{P,Q})} = \sqcup_i$, if $i \in Q$. From this and (16) and the definition of hfp (14),

$$\mathrm{lfp}_i\left(\boldsymbol{F}^{(\mathrm{P,Q})}\right) = \left\{ \begin{matrix} \sqcap_i Fix_i(\boldsymbol{F}) \text{ if } i \in P \\ \sqcup_i Fix_i(\boldsymbol{F}) \text{ if } i \in Q \end{matrix} \right\} = \mathrm{hfp}_i^{(\mathrm{P,Q})}(\boldsymbol{F})$$

Hence, $\mathrm{hfp}^{(\mathrm{P,Q})}(\boldsymbol{F}) \in Fix(\boldsymbol{F})$. □

For simplicity of exposition, we have proved Theorem 1 by appealing to Knaster-Tarski theorem, but it is also possible to prove it from the first principles.

From the construction of $\boldsymbol{L}^{(\mathrm{P},\mathrm{Q})}$, we already know that

$$\begin{aligned} \bot_i^{(\mathrm{P},\mathrm{Q})} &= \bot_i \ \text{ if } i \in P \\ \bot_i^{(\mathrm{P},\mathrm{Q})} &= \top_i \ \text{ if } i \in Q \end{aligned} \tag{17}$$

The least fixed point $\mathrm{lfp}\left(\boldsymbol{F}^{(\mathrm{P},\mathrm{Q})}\right)$ can be computed iteratively starting with $\bot^{(\mathrm{P},\mathrm{Q})}$ [13]. Hence, the heterogeneous fixed point of a function vector $\boldsymbol{F}$ w.r.t. a valid partition (P, Q) can be defined as follows:

$$\mathrm{hfp}^{(\mathrm{P},\mathrm{Q})}(\boldsymbol{F}) = \boldsymbol{F}^k\left(\bot^{(\mathrm{P},\mathrm{Q})}\right) \tag{18}$$

where $k \in \mathbb{N}$ is the least number such that $\boldsymbol{F}^k\left(\bot^{(\mathrm{P},\mathrm{Q})}\right) = \boldsymbol{F}^{k-1}\left(\bot^{(\mathrm{P},\mathrm{Q})}\right)$.

5 Solution of May-Must Data Flow Analysis

We now apply the heterogeneous fixed point theory to may-must points-to analysis (ref. section 2.2). Consider the data flow equations (4), (5), (8), and (9). If there are n nodes in a program flow graph, there are $4 \times n$ equations forming a system $S_{(4\times n)}$. MustIN_i, $\mathsf{MustOUT}_i$, MayIN_i, and MayOUT_i, where $i \in \{1, \ldots, n\}$ are the bound variables of the system. The corresponding flow functions are denoted by $\mathsf{f}_{\mathsf{MustIN}i}$, $\mathsf{f}_{\mathsf{MustOUT}i}$, $\mathsf{f}_{\mathsf{MayIN}_i}$, and $\mathsf{f}_{\mathsf{MayOUT}_i}$.

For convenience, we abstract the data flow equations as:

$$\begin{aligned} x_{(4\times i-3)} &= f_{(4\times i-3)}\left(x_1, x_2, \ldots, x_{4\times n}\right) \\ x_{(4\times i-2)} &= f_{(4\times i-2)}\left(x_1, x_2, \ldots, x_{4\times n}\right) \\ x_{(4\times i-1)} &= f_{(4\times i-1)}\left(x_1, x_2, \ldots, x_{4\times n}\right) \\ x_{(4\times i-0)} &= f_{(4\times i-0)}\left(x_1, x_2, \ldots, x_{4\times n}\right) \end{aligned}$$

For translating these equations back to may-must points-to analysis, we will use the following mappings:

Bound Variables	Functions
$\mathsf{MustIN}_i \leftrightarrow x_{(4\times i-3)}$	$\mathsf{f}_{\mathsf{MustIN}i} \leftrightarrow f_{(4\times i-3)}$
$\mathsf{MustOUT}_i \leftrightarrow x_{(4\times i-2)}$	$\mathsf{f}_{\mathsf{MustOUT}i} \leftrightarrow f_{(4\times i-2)}$
$\mathsf{MayIN}_i \leftrightarrow x_{(4\times i-1)}$	$\mathsf{f}_{\mathsf{MayIN}_i} \leftrightarrow f_{(4\times i-1)}$
$\mathsf{MayOUT}_i \leftrightarrow x_{(4\times i-0)}$	$\mathsf{f}_{\mathsf{MayOUT}_i} \leftrightarrow f_{(4\times i-0)}$

A component lattice is $L_j = (\wp(U), \subseteq)$. The product lattice of the system is $\boldsymbol{L} = L_1 \times \cdots \times L_{(4\times n)} = (\wp(U), \subseteq)^{(4\times n)}$. A component function is of the type $(\wp(U), \subseteq)^{(4\times n)} \to (\wp(U), \subseteq)$.

Consider the following two sets $P, Q \subseteq \{1, \ldots, (4 \times n)\}$:

$$P = \{(4 \times i - 1),\ (4 \times i - 0) \mid i \in \{1, \ldots, n\}\} \tag{19}$$

$$Q = \{(4 \times i - 3),\ (4 \times i - 2) \mid i \in \{1, \ldots, n\}\} \tag{20}$$

The set P represents variables $\mathsf{MayIN}_i/\mathsf{MayOUT}_i$ and the corresponding functions $\mathsf{f}_{\mathsf{MayIN}_i}/\mathsf{f}_{\mathsf{MayOUT}_i}$. Q represents variables $\mathsf{MustIN}_i/\mathsf{MustOUT}_i$ and the functions $\mathsf{f}_{\mathsf{MustIN}i}/\mathsf{f}_{\mathsf{MustOUT}i}$.

Claim. The function vector $\boldsymbol{F}$ of the system $S_{(4\times n)}$ is h-monotonic w.r.t. the partition (P, Q).

Proof. When a variable x_i does not depend on a variable x_j, then it is safe to assume that f_i either monotonically increases in x_j or monotonically decreases in x_j, i.e. x_i has either a positive or a negative dependence on x_j. The result follows directly from the dependences D1–D4 and the construction of $S_{(4\times n)}$. □

There could be several other valid partitions, but we are interested in this particular partition as we want the largest possible must information and the smallest possible may information, for enabling maximum optimization opportunities. In [10], we give results about the number and the nature of valid partitions for any given system. The desired solution of may-must data flow equations is $\mathrm{hfp}^{(\mathrm{P},\mathrm{Q})}(\boldsymbol{F})$. To compute the solution, we first initialize the variables with corresponding elements in $\bot^{(\mathrm{P},\mathrm{Q})}$ as follows:

$$\begin{aligned} x_{(4\times i-3)} &= \mathsf{MustIN}_i &&= U \\ x_{(4\times i-2)} &= \mathsf{MustOUT}_i &&= U \\ x_{(4\times i-1)} &= \mathsf{MayIN}_i &&= \phi \\ x_{(4\times i-0)} &= \mathsf{MayOUT}_i &&= \phi \end{aligned} \tag{21}$$

With the above initialization, the heterogeneous fixed point solution can be computed by iteratively solving the may-must data flow equations until two consecutive iterations result in same values (ref. (18)).

Further, it can be shown that our analysis and Emami's analysis compute equivalent information. Since our analysis converges, it can be shown that starting with appropriate initializations Emami's analysis also converges [9].

6 Conclusions and Future Work

Many analyses can be modeled as fixed point solutions of systems of simultaneous equations in which the functions may monotonically increase in some bound variables and decrease in others. The classical fixed point theory does not cover such situations. It requires all dependences among the bound variables to be positive. The classical extremal fixed points uniformly take either least values for all variables or greatest values for all variables, where the element-wise comparison is restricted to the set of solutions.

The heterogeneous fixed point theory is a generalization of the classical fixed point theory. It allows positive as well as negative dependences among the variables. We have shown that if the dependences are mutually consistent then the variables can be partitioned into two sets such that two variables belong to the same set iff the dependences between them are positive. This guarantees the existence of a fixed point called heterogeneous fixed point, which depending on

the partition, takes least values for some variables from among all fixed points, and greatest values for others. Our theory also suggests appropriate initialization thereby assuring computability of fixed points. We have applied heterogeneous fixed point theory to explain convergence issues in points-to analysis.

Further work includes exploring applications of heterogeneous fixed points in program analysis, abstract interpretation and semantics [2, 6], and fixed point logics. We would also like to compare the expressiveness of heterogeneous fixed points with other fixed point formulations like mu-calculus [16, 8], generalized inductive definitions [5], etc.

7 Acknowledgments

Authors wish to thank Patrick Cousot for useful comments and suggestions on a related technical report [10] on heterogeneous fixed points. Authors are thankful to Supratik Chakraborty, Bageshri Sathe, and Amey Karkare for some useful discussions.

References

1. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM Press, 1993.
2. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.
3. M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, School of Computer Science, McGill University, Montreal, 1993.
4. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256. ACM Press, 1994.
5. S. Feferman. Formal theories for transfinite iterations of generalized inductive definitions and some subsystems of analysis. In A. Kino, J. Myhill, and R. E. Vesley, editors, *Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo, N.Y. Studies in Logic and the Foundations of Mathematics.*, pages 303–326. North-Holland, 1968.
6. R. Giacobazzi and I. Mastroeni. Compositionality in the puzzle of semantics. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 87–97. ACM press, 2002.
7. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.
8. P. Hitchcock and D. Park. Induction rules and termination proofs. In M. Nivat, editor, *Proceedings 1st Symp. on Automata, Languages, and Programming, ICALP'72, Paris, France, 3–7 July 1972*, pages 225–251. Amsterdam, 1973.

9. A. Kanade, U. Khedker, and A. Sanyal. Equivalence of may-must and definite-possibly points-to analyses. Dept. Computer Science and Engg., Indian Institute of Technology, Bombay, April 2005. `http://www.cse.iitb.ac.in/~aditya/reports/equivalence-points-to.ps`.
10. A. Kanade, A. Sanyal, and U. Khedker. Heterogeneous fixed points. Technical Report TR-CSE-001-05, Dept. Computer Science and Engg., Indian Institute of Technology, Bombay, January 2005. `http://www.cse.iitb.ac.in/~aditya/reports/TR-CSE-001-05.ps`.
11. U. Khedker. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter Data Flow Analysis. CRC Press, 2002.
12. G. A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973.
13. S. C. Kleene. *Introduction to Mathematics*. D. Van Nostrand, 1952.
14. B. Knaster. Une théorème sur les fonctions d'ensembles. *Annales Soc. Polonaise Math.*, 6:133–134, 1928.
15. J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31. ACM Press, 1988.
16. D. Scott and J. W. de Bakker. A theory of programs. Unpublished notes, IBM seminar, Vienna, 1969.
17. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

A Overview of Emami's Points-to Analysis

Several representations have been proposed for capturing the aliasing information [1, 7, 15, 4]. The points-to analysis [4] due to Emami et al. computes *points-to relation* between pointer expressions denoting stack locations. The aliases that hold along some but *not* along all paths are captured by *possibly* points-to relation. If a variable x possibly contains address of a variable y, then it is denoted by (x, y, P). The aliases that hold along all paths are captured by *definite* points-to relation. If a variable x definitely contains address of a variable y, then it is denoted by (x, y, D).

For simplicity of exposition, we consider a subset of the language in [4]. Non-pointer assignments are ignored. The pointer expressions may consist of scalar and pointer variables, referencing operator '&', and dereferencing operator '$*$'. The left-hand side (lhs) of an assignment can be either x or $*x$ and the right-hand side (rhs) can be x, $\&x$, or $*x$, for some variable x. We restrict ourselves to intraprocedural analysis and view the program as a flow graph. The nodes in the graph are either empty or contain a single pointer assignment.

We now abstract the algorithm in [4] as data flow equations. The points-to relation at IN of a node i is a confluence of points-to relations at OUT of its predecessors $p_1, \ldots, p_k$.

$$\mathsf{input}_i = \begin{cases} \mathsf{Merge}\left(\mathsf{output}_{p_1}, \ldots, \mathsf{output}_{p_k}\right) & \text{if } i \neq \text{entry} \\ \phi & \text{if } i = \text{entry} \end{cases} \tag{22}$$

where "entry" is the unique entry node of the procedure and the operation Merge [3], defined below, is extended to multiple arguments in an obvious way.

$$\mathsf{Merge}(S_1, S_2) = \{(p_1, p_2, D) \mid (p_1, p_2, D) \in S_1 \cap S_2\} \cup \{(p_1, p_2, P) \mid (p_1, p_2, r) \in S_1 \cup S_2 \wedge (p_1, p_2, D) \notin S_1 \cap S_2\} \quad (23)$$

Note that the definition of Merge excludes the definite information along all paths from being considered as possible points-to information.

An *R-location* represents the variable whose address appears in the rhs. An *L-location* represents the variable which is being assigned this address. Both of the L-location and R-location depend on the nature of points-to information and could be either *definite* or *possible*. Let L_i denote the set of L-locations of an assignment in node i tagged with D or P appropriately. Let R_i denote the corresponding R-locations. Then,

lhs_i	L_i
x	$\{(x, D)\}$
$*x$	$\{(y, r) \mid (x, y, r) \in \mathsf{input}_i\}$

rhs_i	R_i
$\&x$	$\{(x, D)\}$
x	$\{(y, r) \mid (x, y, r) \in \mathsf{input}_i\}$
$*x$	$\{(z, r_1 \oplus r_2) \mid (x, y, r_1), (y, z, r_2) \in \mathsf{input}_i\}$

where lhs_i and rhs_i are lhs and rhs of the assignment in node i, and $\oplus$ is defined as

$$r_1 \oplus r_2 \triangleq \begin{cases} P & \text{if } r_1 \neq r_2 \\ r_1 & \text{otherwise} \end{cases}$$

Let (x, y, r) hold before an assignment in node i.

- If $(x, P) \in \mathsf{L}_i$, then x may or may not be modified. Thus after the assignment, x may or may not point to y. Hence the definiteness of its pointing to y must be changed to the possibility of its pointing to y. This is captured by (24) and (25) below.
- If $(x, D) \in \mathsf{L}_i$, then x is definitely modified as a side effect of the assignment and x ceases to point to y. This is captured by (26) below.

$$\mathsf{change\_set}_i = \{(x, y, D) \mid \underline{(x, P) \in \mathsf{L}_i} \wedge (x, y, D) \in \mathsf{input}_i\} \quad (24)$$

$$\mathsf{changed\_input}_i = (\mathsf{input}_i - \mathsf{change\_set}_i) \cup \{(x, y, P) \mid (x, y, D) \in \mathsf{change\_set}_i\} \quad (25)$$

$$\mathsf{kill\_set}_i = \{(x, y, r) \mid \underline{(x, D) \in \mathsf{L}_i} \wedge (x, y, r) \in \mathsf{input}_i\} \quad (26)$$

An assignment generates definite points-to information between definite L-locations and definite R-locations. All other combinations between L-locations and R-locations are generated as possibly points-to information.

$$\mathsf{gen\_set}_i = \{(x, y, D) \mid (x, D) \in \mathsf{L}_i \ \wedge \ (y, D) \in \mathsf{R}_i\} \cup \{(x, y, P) \mid (x, r_1) \in \mathsf{L}_i \wedge (y, r_1) \in \mathsf{R}_i \wedge (r_1 \neq D \vee r_2 \neq D)\} \quad (27)$$

Finally, the points-to information at OUT of node i is

$$\mathsf{output}_i = (\mathsf{changed\_input}_i - \mathsf{kill\_set}_i) \cup \mathsf{gen\_set}_i \quad (28)$$