Grenze International Journal of Engineering and Technology, June Issue

Comparative Study of Andersen's and Steensgaard's Approaches of Pointer Analysis

Amroz Siddiqui[1], Mritunjay Ojha[2], Dakshayani G[3] and Kaliappan Yadav[4]
[1]Fr. C. Rodrigues Institute of Technology, Department of Computer Engineering, Vashi, Navi Mumbai, India
[2-4]Fr. C. Rodrigues Institute of Technology, Department of Computer Engineering, Vashi, Navi Mumbai, India
Email: amroz@fcrit.ac.in, {mritunjay.ojha, dakshayani}@fcrit.ac.in, yadav.kaliappan@fcrit.onmicrosoft.com

***Abstract*— Pointer analysis refers to finding out memory locations that are accessed in many ways. It is a static program analysis that gives the set of objects a pointer variable refers to. We present a comparative study of widely used various set of techniques that focus on, for a pointer, the possible set of memory location it will point to at runtime. More specifically, it is a compile time technique that resolves the issue of mapping of pointer variables to memory locations. This analysis has a variety of applications from bug detection, memory leak identification to performing other static analysis. Related techniques are points-to and alias analysis. Our work in this paper is a comparative study of many of the previous notable works related to pointer analysis.**

***Index Terms*— Pointer Analysis, Points-to Analysis, Alias Analysis, May and Must Analysis**

I. Introduction

In programming languages, like C and C++, we have variables that stores values determined by their types, for example, a char type variable will store a char literal, an int type variable will store an int literal, similarly, a pointer variable stores the address of another variable determined by the type, that is, an int pointer will store the address of a variable of type int, a float pointer will store address of a variable of type float. Pointer variables usage in programming languages like C and C++ is unavoidable because of various reasons. Besides, accessing values at a memory location, manipulating strings and array structures, dynamically allocating memories, it also creates notoriously beautiful mathematical expressions that not only makes the program efficient, robust and in many cases secure, but also gives an unknown satisfaction to the programmer for creating elegant code. However, this elegance comes with a price. Uninitialized pointers causing segmentation fault, memory leaks creating security holes, and wrong usage of pointer arithmetic are some of the reasons that may demotivate programmers to use them. A compile time study that gives a set of memory locations pointed to by each pointer in the program may give an insight to the structure of mapping between pointer variables and the memory locations they point to, and possibly lead to minimizing of errors due to wrong usage. This study is called pointer analysis.
Pointer Analysis is hard, but essential for enabling many compiler optimizations. Alias analysis is undecidable [1]. Flow insensitive alias analysis is NP-hard [2]. Points-to analysis is undecidable [3]. The issue raised in [4], after more than twenty years, is still not resolved. Nevertheless, engineering view of analysis is more dominant and promising than the science view. Instead of aiming for clean and precise abstractions, we can go for building short and simple approximations and maintain a balance between precision and efficiency. The fundamental fact is that the requisite context information and the useful points-to-information is sparse and small [5]. A majority of

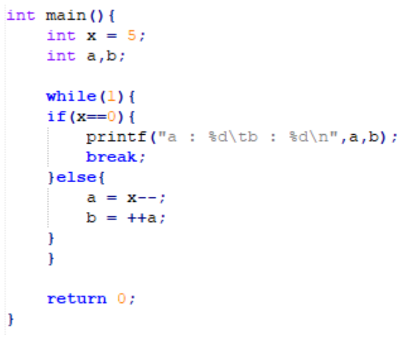
points-to pairs calculated by existing algorithms are never used by any analysis or transformation because they involve dead variables [5]. Liveness and points-to information are mutually dependent. The authors define points-to information only for live pointers. And for pointer indirections, define liveness information using points-to information.

II. Literature studies

A fundamental data structure of pointer analysis is a set of memory locations that a variable may point to during runtime is called as *points-to* set. This simple framework is the basis of one of the early work by Andersen [6], stated as a subset constraints, that is, if *A* and *B* are *points-to* sets then, we can say that all elements from *A* are to be added to *B*. Another early work is by Steensgard [7] which is *unification based*, that is, it involves merging or unifying some points-to sets, and it uses equality constraints. Recent works of [8] introduces a framework that helps the implementation of pointer analysis algorithms, along with comparison of different algorithms. It is a tool that views the source code and abstract syntax tree as an XML representation. Another work presented in [9] is a coarse grained context sensitive and thread sensitive approach. The algorithm presented in that paper uses inter-thread data flow analysis, and not intra-thread. It represents calling contexts as sequences of thread starts. Yet another approach in [10] proposes a probability based technique that can calculate the probability of every mapping of pointer variable and its memory location at each program point. The probability of each relationship is given by the measurable description for each relationship. Graph rewriting representation of flow sensitive pointer analysis in [11] is a parallel approach using Intel Threading Building Blocks. In [12] another approach, refining based on CFL-reachability problem is used to compute new constraints for any given points-to relationship. The authors claim that they present a technique that can extract new constraints from the CFL-paths which they call as sequencing relations, which can be used to refine pointer analysis and other program analysis. They claim that based on the new constraints they can build a points-to analysis which is demand-driven and context-sensitive. In [13], the authors present a static analysis tool which uses pointer analysis algorithm to find memory defects. Their algorithm computes individual solution for each program point while being flow, context, field, and quasi path sensitive. They represent the function's behaviour by using function summaries which makes their analysis context sensitive.

III. program analysis

Before we delve into specific techniques in pointer analysis, let us understand the basis of all kinds of program analysis. A program state is defined as values of all variables in a program. This includes variables associated with stack frames below the top of the run-time stack and also associated with heaps. An execution of a program is defined as a series of transformation of program state. That is, each execution of some intermediate code transforms input state to a new output state. The input state is associated with the program point before the statement and output state is associated with the program point after the statement. A group of sequential instructions, with no branching and looping, form what is called as a basic block. A collection of basic blocks associated with a program under consideration form a flow graph. A sequence of program points is the path in a flow graph that a program can take under execution. Paths can also be identified by the basic block a program traverses during execution. From the possible program states at each point, we find out the information that we need for a particular data flow analysis. Consider the following C code and its equivalent control flow graph as generated by *fdump-tree-all* option of gcc, the gnu C compiler on Linux, as shown in Fig. 1:

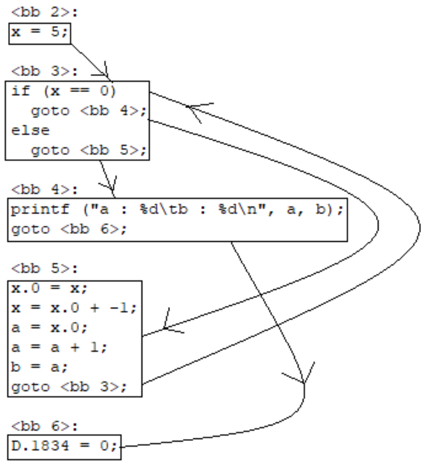


```
int main(){
    int x = 5;
    int a,b;

    while(1){
    if(x==0){
        printf("a : %d\tb : %d\n",a,b);
        break;
    }else{
        a = x--;
        b = ++a;
    }
    }

    return 0;
}
```

(a)



```
<bb 2>:
x = 5;

<bb 3>:
if (x == 0)
  goto <bb 4>;
else
  goto <bb 5>;

<bb 4>:
printf ("a : %d\tb : %d\n", a, b);
goto <bb 6>;

<bb 5>:
x.0 = x;
x = x.0 + -1;
a = x.0;
a = a + 1;
b = a;
goto <bb 3>;

<bb 6>:
D.1834 = 0;
```

(b)

Fig. 1. (a) The example C code. (b) Its equivalent control flow graph generated by fdump-tree-all option of gcc, the gnu C compiler on Linux

TABLE 1. TRANSFER FUNCTIONS

	Forward Propagation	Backward Propagation
Transfer Functions	$OUT[s] = f_s\,(IN[s])$	$IN[s] = f_s\,(OUT[s])$

A. *The Data-flow Abstraction*

We observe that one execution path, with respect to the basic blocks' entry and exit is, *bb2, bb3, bb4, bb6.* There are other paths also, incorporating both conditional and unconditional jumps. Namely, paths from *bb3* to *bb4* and *bb5* are result of conditional jumps, whereas paths from *bb4* to *bb6* as well as *bb5* to *bb3* are unconditional jumps. When data-flow analysis is applied, a *data-flow value* is associated with every program point that represents an abstraction of the set of all possible program states that can be observed for that point. The set of data-flow values is the domain for this application. The set of data-flow values before and after a statement *s* are denoted by *IN[s]* and *OUT[s]* respectively. The data flow problem is to find a solution to a set of constraints on the *IN* and *OUT* of *s*, for all statements *s*. The sets of constraints are of two types, one that is based on transfer functions and other on flow of control.

The relationship between data-flow values before and after a statement is known as transfer function. When the information propagates forward along the execution path, the transfer function of a statement *s*, f_s, takes a data flow value before the statement and produces a new data flow value after the statement. Similarly, when the information propagates backward in the opposite direction of the execution path, the transfer function takes a data flow value after the statement and produces a new data flow value before the statement. Both transfer functions are shown in Table 1 above. The second set of constraints on data flow values is derived from the flow of control. If a basic block *B* consists of statements $s_1, s_2, \ldots, s_n$ in that order, then the control flow value of s_i is the same as the control value of $s_{i+1.}$ That is, $IN[s_{i+1}] = OUT[s_i]$ for all $i = 1, 2, \ldots, n\text{-}1$

When the control flow is between basic blocks, then the statement is replaced by basic block identifier in the equation, and depending on the direction of the flow we derive the constraints as follows:

- Suppose block B contains statements $s_1, s_2, \ldots, s_n$ in that order.
- If s_1 is the first statement of the block B, then $IN[B] = IN[s_1]$
- If s_n is the last statement of the block B, then $OUT[B] = OUT[s_n]$
- Transfer function, f_B is the composition of the transfer function, f_{si} of each statement s_i

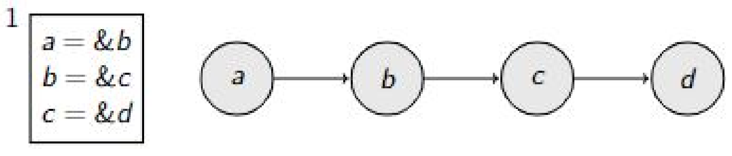
Thus, we have the following constraints as shown in Table II.

TABLE II. TRANSFER FUNCTIONS ON BASIC BLOCKS

	Forward Propagation	Backward Propagation
Transfer Functions	$OUT[B] = f_B\,(IN[B])$	$IN[B] = f_B\,(OUT[B])$
	$IN[B] = \cup_{P\text{ is a predecessor of }B}\, OUT\{P]$	$OUT[B] = \cup_{S\text{ a successor of }B}\, IN[S]$

B. *Andersen’s Approach to Pointer Analysis*

For every pointer in a program, what memory locations does it points to, is the fundamental problem to solve in pointer analysis. Anderson’s pointer analysis algorithm is one of the old and famous approach that is also called as inclusion based algorithm, because of the fact that it translates the input program with statements of the form “p = q” to constraints of the form “q’s points to set is a subset of p's points-to set”. The analysis is flow-insensitive and context-insensitive, meaning that it just completely ignores the control-flows in the input program and considers that all statements can be executed in arbitrary order. An example of Anderson’s approach flow-insensitive points-to analysis is as show in Fig. 2. Anderson’s approach is also known as Inclusion Based Approach.



(a) (b)

Fig. 3. The points-to graph generated on analysis of basic block 1. (a) The basic block 1 (b) The points-to graph.

In simple terms, it means the following:

- a points to b
- b points to c
- c points to d

The constraints generated on the points-to set are given in the Table III below.

Continuing further, when we move on to the next block, that is block 2, we observe the following:

- The objects that c points to, should point to objects that b points to.
- That is, d should point to c

The constraints will be modified to, as shown in Table IV above. And the points-to graph looks like, as shown in Fig. 4.

Moving further, in block 3, we observe that:

- The pointer c should point to the objects that b points to.
- That is, c should point to c.

And thus, we get the new constraints, as shown in Table V, and the new points-to graph as shown in Fig. 5.

TABLE III: CONSTRAINTS ON THE POINTS-TO SET

$\boldsymbol{P_a} \supset \{\boldsymbol{b}\}$
$\boldsymbol{P_b} \supset \{\boldsymbol{c}\}$
$\boldsymbol{P_c} \supset \{\boldsymbol{d}\}$

TABLE IV: AFTER ANALYSIS OF BLOCK 2

$\boldsymbol{P_a} \supset \{\boldsymbol{b}\}$
$\boldsymbol{P_b} \supset \{\boldsymbol{c}\}$
$\boldsymbol{P_c} \supset \{\boldsymbol{d}\}$
$\forall \boldsymbol{x} \in \boldsymbol{P_c}, \boldsymbol{P_x} \supset \boldsymbol{P_b}$

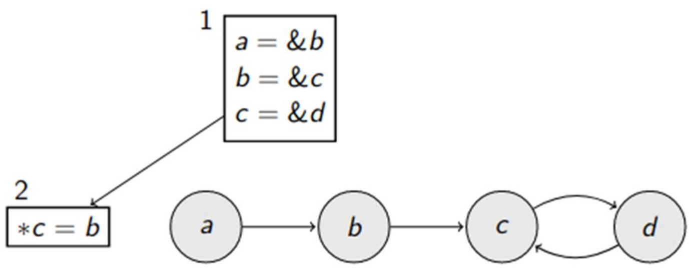


Fig. 4. The points-to graph generated on analysis of basic block 2.

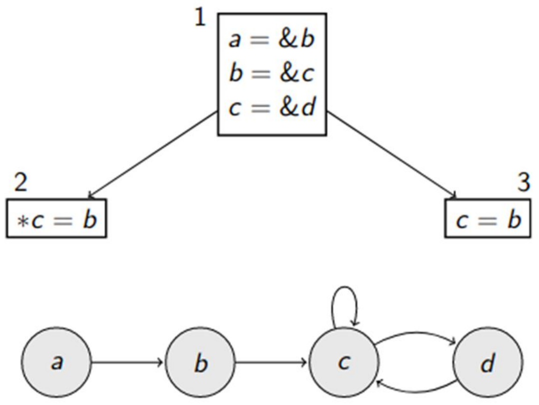


Fig. 5. The points-to graph generated on analysis of basic block 3.

TABLE V: AFTER ANALYSIS OF BLOCK 3

$P_a \supset \{b\}$
$P_b \supset \{c\}$
$P_c \supset \{d\}$
$\forall x \in P_c, P_x \supset P_b$
$P_c \supset P_b$

Finally, in block 4, we observe that:

- The pointer b should point to a also

And the constraints and the points-to graph becomes, as shown in Table 6 and Fig. 6 respectively.
From the constraints 3, 4 and 5 of Table 6, we infer that:

- The pointer c and the objects that it points to should point to objects that b points to
- That is, d and c should point to a
- The objects pointed to by c should point to objects pointed to by b
- That is, a should point to itself and c

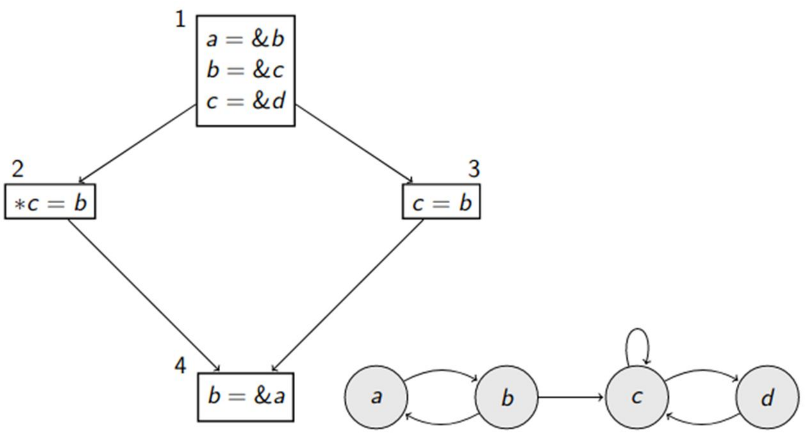
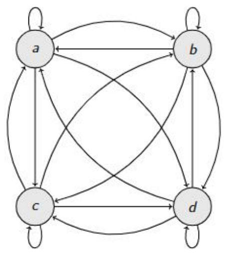


Fig. 6. The points-to graph generated on analysis of basic block 4.

TABLE VI: AFTER ANALYSIS OF ALL BLOCKS


$P_b \supset \{a\}$

The final points-to graph, as a result of Andersen's Flow Insensitive Points-to Analysis, also known as Inclusion Based approach is as shown in Fig. 7:

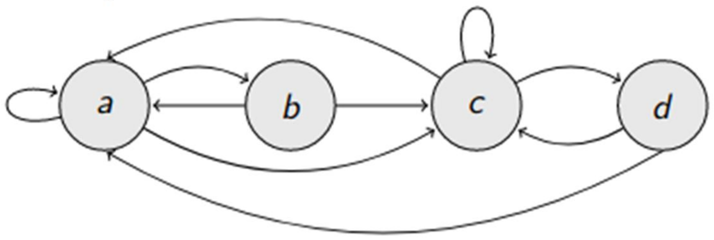


Fig. 7. The final points-to graph based on the Andersen's approach for the example code of Fig. 2

C. Steensgaard's Approach to Pointer Analysis

Steensgaard's approach, also known as Equality Based Approach, is also a constrained based analysis, but it uses equality constraints instead of subset constraints, it is less precise than Andersen's approach, but comparatively more scalable. The fundamental underlying principle in Steensgaard's approach is that:

- Equivalent Locations: Treat all elements a pointer points to as equivalent locations.
- Transitive closure: Elements of all equivalent locations become equivalent.

Conceptually, it restricts every node to only one outgoing edge. That is, if *"p points to x"* and *"p points to y"* then *"merge x and y"*. Thus, the effective additional constraints on the ongoing example is as shown in the table 7, below:
This implies that a, b, c and d are equivalent. And the complete graph as a result of Steensgaard's analysis is as shown below in Fig. 8.

TABLE VII: EFFECTIVE ADDITIONAL CONSTRAINTS AFTER STEENSGAARD'S ANALYSIS

$P_a \supset \{b\}$
$P_b \supset \{c\}$
$P_c \supset \{d\}$
$\forall x \in P_c, P_x \supset P_b$
$P_c \supset P_b$
$P_b \supset \{a\}$
Unify(a, b)
Unify(a, c)
Unify(a, d)

Fig. 8. Steensgaard's points-to graph for the example code of Fig. 2

III. SUMMARY AND CONCLUSION

Andersen's algorithm is a constraint on subset based approach whereas Steensgaard's algorithm is a unification based approach. Consider an assignment of the form *a = b*. In Andersen's approach, we observe that a constraint is added on the subset that says that *"targets of b must be subset of targets of a"*. Graphs of such constraints are called *“inclusion constraints graphs”*. It enforces unidirectional flow from *b* to *a*. On the other hand, Steensgaard’s approach merges equivalence classes, which means *“targets of a and b must be identical”*. It assumes bidirectional flow from *b* to *a* and vice versa. There are in between solutions as in [14], which is a unification based pointer analysis with directional assignment that exploits the semantics of C programming language. It uses Andersen’s approach for top pointers and Steensgaard’s elsewhere. There are other well-known analysis also, like *alias analysis*, which is a kind of a client of points-to analysis. If points-to analysis tells us what objects does each pointer points to, then alias analysis answers the question that can two pointers point to the same location? Once we have performed points-to analysis, it is trivial to compute alias analysis. Two pointers *x* and *y* may alias if $points\text{-}to(x) \cap points\text{-}to(y) \neq \varphi$. Analysis that takes into consideration flow, may and must analysis and its variants are yet another set of analysis to go for. In this survey paper we have studied two approaches through examples. Procedure calls involving pointer variables add more complexity, which is yet to be studied. Implementing this algorithm as one pass of gcc is another goal to be achieved.

REFERENCES

[1] G. Ramalingam, "The Undecidability of Aliasing," ACM Trans. Program. Lang. Syst., p. 1467–1471, 1994.

[2] S. Horwitz, "Precise Flow-Insensitive May-Alias Analysis is NP-Hard," ACM Trans. Program. Lang. Syst., pp. 1-6, 1997.

[3] V. T. Chakaravarthy, "New Results on the Computability and Complexity of Points--to Analysis," in Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, 2003.

[4] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?," in Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, Utah, USA, 2001.

[5] U. P. Khedkar et al, "Liveness-Based Pointer Analysis," in Static Analysis, Berlin, Heidelberg, 2012.

[6] L. O. Andersen, Program analysis and specialization for the c programming language, University of Copenhagen: Master's thesis, May 1994.

[7] B. Steensgaard, "Points-to analysis in almost linear time," in 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '96, New York, NY, USA, 1996.

[8] N. C. Zyrianov V. et al., "srcptr: A framework for implementing static pointer analysis approaches," in 27th International Conference on Program Comprehension (ICPC), Montreal Quebec Canada, 2019.

[9] X. B. Qian J., "Thread-sensitive pointer analysis for inter-thread dataflow detection," in 11th International Workshop on Future Trends of Distributed Computing Systems (FTDCS '07), Sedona, AZ, USA, 2007.

[10] P. S. Chen et al., "Interprocedural probabilistic pointer analysis," in IEEE Transactions on Parallel and Distributed Systems, Fukuoka, Japan, 2004.

[11] V. Nagaraj et al., "Parallel flow-sensitive pointer analysis by graph-rewriting," in Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, UK, 2013.

[12] X. Sun et al., "Refining the Pointer Analysis by Exploiting Constraints on the CFL-Paths," in 20th Asia-Pacific Software Engineering Conference (APSEC), Bangkok, Thailand, 2013.

[13] Y. Wang et al., "TsmartGP: A Tool for Finding Memory Defects with Pointer Analysis," in 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 2019.

[14] M. Das, "Unification-based pointer analysis with directional assignments," in Proceedings of the 2000 {ACM} {SIGPLAN} Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, 2000.