ISSN:0975-9646

STAR: A Comparison of Points-To Analysis Algorithms

Kanchan Arora

M.Tech, Indraprastha Institute of Information Technology
New Delhi, India

Abstract— **Pointer analysis is a static program analysis which aims to determine memory locations that a pointer variable can refer to. Pointer Analysis has a rich literature and variety of applications. In this paper, firstly an introduction to pointer analysis is presented and then four algorithms: *S*teensgaard, Refinemen*T*-based, *A*ndersen, *R*inard's algorithm is analysed deeply and finally comparison of these four analyses based on certain parameters is provided.**

Keywords— **Points-to Analysis, Flow-Sensitivity, Context-Sensitivity, Pointers, Program Analysis**

I. Introduction

One of the most important challenges which computer science is facing today is the size and complexity of modern software. With the increase in size and complexity, software is becoming more difficult to understand, more difficult to promise correct, and more difficult to optimize. Program analysis is a key tool to manage this complexity. It has been used for such diverse purposes as model checking, security analysis, error-checking, compiler-optimization [7], hardware synthesis, software refactoring and parallelization.

This paper describes the contributions of a set of research works and papers. In the second section some terminologies are shown to help understand program analysis and pointer analysis. In second section, all the selected analyses are analysed. In the third section, comparison of all the described analysis based on some facets is presented. In the fourth section, conclusions about each analysed algorithm are stated.

II. Terminologies

Program analysis is the process of analysing the behaviour of programs for its correctness and robustness. Analysis should be efficient as it aims at ensuring that the program does what it is supposed to do and at the same time reducing the resource usage. It becomes more difficult to analyse when a program contains indirection. Indirection can be indirect data-flow or indirect control-flow. All types of indirections are implemented by using pointers and that is where pointer analysis comes into picture. The objective of pointer-analysis is to resolve this indirection by computing *points-to-sets* for each program entity. Points-to-set is the set of all the memory locations that can be indirectly referenced by that entity. For example in the program shown in Fig.1, the points to relation is { pp→p, p→b, qq→q, q→b } where → stands for points-to.

```
main()
{
int a, b,*p,*q,**pp,**qq;
p=&a;
q=&b;
pp=&p;
qq=&q;
*pp=&b;
return 0;
}
```

Fig.1 Example program for points-to analysis

The indirection present in the program must be resolved as precisely as possible. Here precision means the points-to sets should be as small as possible. The more precisely the indirection is solved, the more effective program analysis will be.

Pointer analysis, like most static analyses, is an un-decidable problem [1]. It is complex and multi-dimensional. Dimensions of pointer analysis are flow sensitivity, context-sensitivity and definiteness which are explained below:

A. Flow-Sensitivity

A flow-sensitive[10] program analysis computes for each program point what memory locations pointer expressions may refer to whereas flow-insensitive[9] pointer analysis computes what memory locations pointer expressions may refer to, at any time in program execution. Table 1 shows an example program to clearly differentiate between the two.

TABLE I
FLOW-SENSITIVITY

Flow Insensitive	**Flow Sensitive**

int main(void)
{
int x, y, *p;
p = & x;
/* (p ->x),(p->y) */
foo(p);
p = & y
/*(p ->x),(p->y)*/
}
	int main(void)
	{
	int x, y, *p;
	p = & x;
	/* (p ->x) */
	foo(p);
	p = & y
	/*(p->y)*/
	}

B. Context-Sensitivity

Analysis is context-sensitive [2],[8] if each invocation is kept separated from other invocations whereas analysis is context-insensitive if all the calling contexts are merged and analysed together. Let us examine the following code:

```
int g;
int main()
{
int a;
C1: foo(&g,1)
C2: foo(&a,3)
}
foo(int * p, int q)
{
int r;
*p=q;
 r = *p+ 5;
 }
```

Fig. 2 Example program for context-sensitivity

Analysis of this code will be p = &g; q = 1; p= &a; q = 3;*p = q; r = g+5. When we analyze context-insensitively, the interaction of p=&g from call site C1 and q=3 from call site C3 produces two spurious results g=3 and r=8. Evidently context-sensitivity is crucial for precise pointer analysis.

Flow- and context-sensitivity are independent of each other; an analysis can be either flow-sensitive or flow-insensitive and at the same time either context-sensitive or context-insensitive.

C. Definiteness

Points-to analysis (Alias analysis) can be must-alias or may-alias. *May-Alias* is aliasing that may occur during execution. *Must-Alias* is aliasing that will definitely occur during execution. For example, consider the section of code that shown in Fig. 3 accesses members of structures:

```
x.a = 1;
y.a = 2;
i = x.a + 3;
```

Fig. 3 Example for alias analysis

There are following three possible cases:

1. The variables x and y cannot alias (i.e. never point to the same memory location).
2. The variables x and y must alias (i.e. always point to the same memory location).
3. It cannot be determined at compile time that whether x and y alias or not.

If x and y cannot alias, then i = x.a + 3; can be changed to i = 4. If x and q must alias, then i = x.a + 3; can be changed to i = 5 because x.a + 3 =y.a + 3. In both cases, we are able to perform optimizations from the alias knowledge. On the other hand, if it is not known if x and y alias or not, then no optimizations can be performed and the whole of the code must be executed to get the result. Two memory references are said to have a *may-alias* relation if their aliasing is unknown.

III. Analyses Selected

This section describes the four selected analyses and key features of algorithms used are highlighted.

A. Andersen Analysis

Andersen analysis [3] is a context-insensitive and flow-insensitive points-to analysis. The algorithm used by Andersen is briefly explained below:

1) The algorithm examines statements that create pointers, one by one in any order as the algorithm is flow insensitive.

2) Each statement updates the points-to graph if it can create new points-to relationships.

3) Six kinds of statements are considered:

- p = &a;
- p = q;
- p = *r;
- *p = &a;
- *p = q;
- *p = *r;

4) For each statement points to graph is updated in the following way:

a) p = &a: an arc from p to a is added which shows that p can point to a.(Fig. 4)

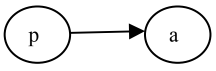

Fig. 4 Points-to graph for p=&a

b) *p = q:* Arcs from p to everything q points to are added in the graph. If new arcs from q are later added, corresponding arcs from p are also added (iterative algorithm).(Fig. 5)

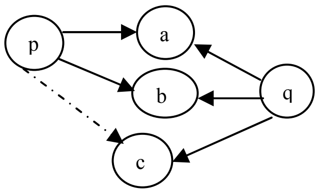

Fig. 5 Points-to graph for p=q

c) p = *r;

Let S be all the nodes r points to. Let T be all the nodes members of S point to. Arcs from p to all nodes in T are added. If later pointer assignments increase S or T, new arcs from p mare also added.(Fig. 6)

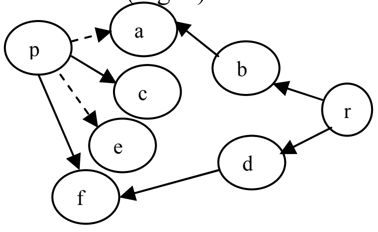

Fig. 6 Points-to graph for p=*r

d) *p = &a;

An arc to node a from all nodes which p points to is added. If new arcs from p are later added, new arcs to a are added.(Fig. 7)

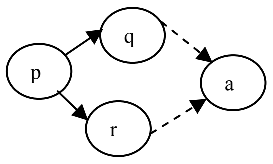

Fig.7 Points-to graph for *p=&a

e) *p = q;

Nodes pointed to by p are linked to all nodes pointed to by q. If later pointer assignments add arcs from p or q, relevant arcs are also added.(Fig. 8)

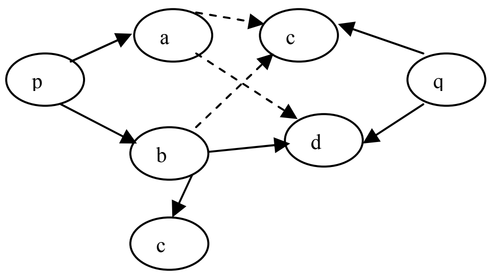

Fig. 8 Points-to graph for *p=q

f) *p = *r;

Let S be all the nodes r points to. Let T be all the nodes members of S point to. Arcs from all nodes p points to all nodes in T are added. If later pointer assignments increase S or T or link new nodes to p, new arcs are added.(Fig. 9)

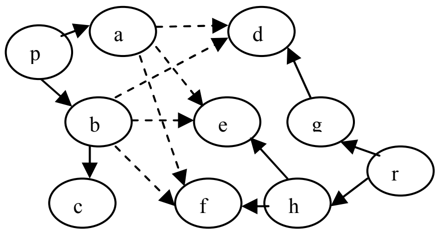

Fig. 9 Points-to graph for *p=*r

Andersen's Analysis is precise but slow. It can require O(n^3) time (where n is the number of nodes in the points-to graph). For example statement like p = *q can force the algorithm to visit n^2 nodes (q may point to n nodes and each of these nodes may point to n nodes). The number of pointer statements analyzed can be O(n), leading to an O(n^3) execution time. Andersen's analysis for large programs is complex. Therefore, it should be used for small programs.

B. Steensgaard Analysis

Since Andersen's analysis was non-linear in time, a fast and accurate analysis which runs in linear time for large programs was required. Steensgaard offered such algorithm (PTA). PTA (Points-To Analysis) algorithm used Andersen's approach only with the difference that merging of nodes takes place if any pointer can reference both.

In Andersen's Analysis points-to graph of p=&a; p=&b may be as shown in Fig. 10

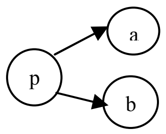

Fig. 10 Andersen's points-to graph

But in Steensgaard analysis points-to graph will be:

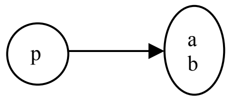

Fig.11 Steensgaard's points-to graph

Steensgaard's Algorithm is sometimes less accurate than Andersen's Algorithm. For example in Fig. 12, the points-to graph, created by Andersen's algorithm, shows that p may point to a or b whereas q may only point to b:

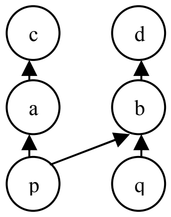

Fig. 12 Andersen's graph

In Steensgaard's Algorithm the points-to graph will be:

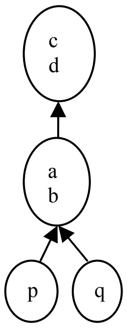

Fig. 13 Steensgaard's graph

Points-to graph in Fig. 13 is incorrectly showing that q may point to both a and b but it actually points to only b.

This algorithm is linear time algorithm because for statements like p = *q can't make the algorithm visit n^2 nodes, because multiple nodes referenced by the same pointer are always merged. Therefore using this algorithm, execution time of O(n.α(n)) which is essentially linear in n can be achieved. By Steensgaard analysis very large programs can be analyzed in linear time without too much of a loss in precision.

C. Refinement-Based Analysis

Refinement based analysis [6] is a context-sensitive, flow-insensitive and demand-driven analysis [11]. A context-sensitive analysis is an analysis that considers the calling context when analyzing the target of a function call. In the example code shown in Fig. 14, a context-sensitive analyzer analyses f() (at least) twice in this program, because it is called from two call sites. This makes it precise, as the effects of f() are quite different each time.

```
int a,b;
int *x;
void f(void)
{
  ++*x;
}
int main()
{
 x = &a;
 f();
 x = &b;
 f();
}
```

Fig. 14 Example code

A context-sensitive analysis can infer that a==1 and b is unchanged after the first call, and that both a and b are 1 after the second call. Context-sensitivity also makes the analysis expensive. A context-insensitive analysis would only analyze f() once, and would typically only produce information like “f() modifies a or b, thus after any call to f(), the contents of both these variables are unknown”.

Problem with context sensitive analysis is that it is costly as precise and deep-context analyses may explode in complexity. So there is a need of analysis which focuses on preciseness in the code where it matters. Such analysis need to handle following three key language features:

1) Assignments
2) Method Calls
3) Heap Accesses

In the points-to graph, edges entering and exiting a method call are labelled with open and close parentheses specific to the call site. A path in graph with mismatched call parentheses corresponds to an unrealizable control flow path. A balanced-parentheses language can be used to filter out such paths. Match edges are those edges which connect matched field parentheses from source of open to sink of close. Match edges can be used to skip sub-paths.

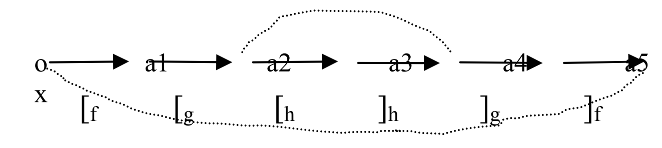

Fig. 15 Points-to graph with match edges

Match edges can be removed to refine heaps and calls. Removal of such edges is named as refinement by Bodik. After refinement the points to graph is shown in Fig. 16:

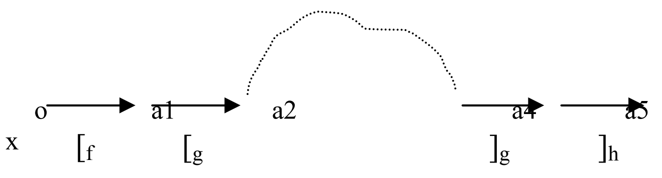

Fig. 16 Points-to graph after refinement

Refinement based analysis requires less memory as it is demand-driven and flow-insensitive. It is more precise and easy to implement.

D. Rinard Analysis

While analysing multi-threaded programs, certain questions about what location will be written by which statement and what a particular pointer will be pointing to in one thread and what it will point to after all the threads are executed need to be answered. Consider the example code in Fig. 17.

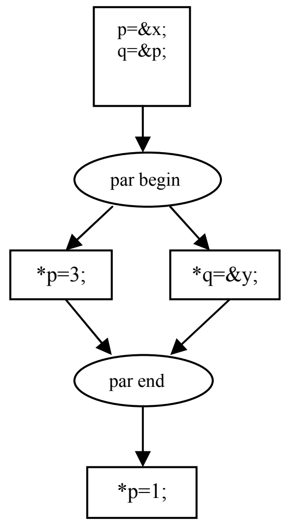

Fig. 17 Example control flow graph for multithreaded code

Analysis that answers what location is written by *p=3 and *p=1 is needed. Therefore, analysis which analyse interaction between concurrent threads is required. Points-to graph at each program point is as shown below:

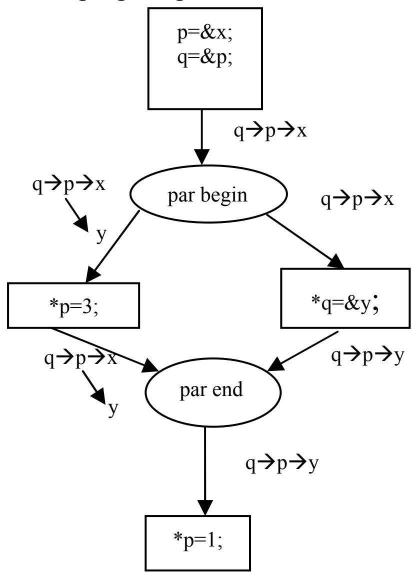

Fig. 18 Naive approach for multithreaded code

One possible solution to analyze interactions between concurrent threads is to analyse all possible inter-leavings and merge results. It fails because of the exponential complexity i.e. for n threads with S1...Sn statements number of inter-leavings=(S1+.....+Sn)!/(S1!....Sn!).

Rinard[5] introduced the concept of interference. Interference means adding points-to edges created by other concurrent threads. Rinard's Analysis is flow sensitive and context-sensitive data flow analysis. At each program point data flow analysis will generate a triple <C,I,E> as data flow information where C is the current points-to relationships ,I is the interference information from other threads and E is the edges created by the current thread. Rinard's Analysis for example code shown in Fig 17 is shown below:

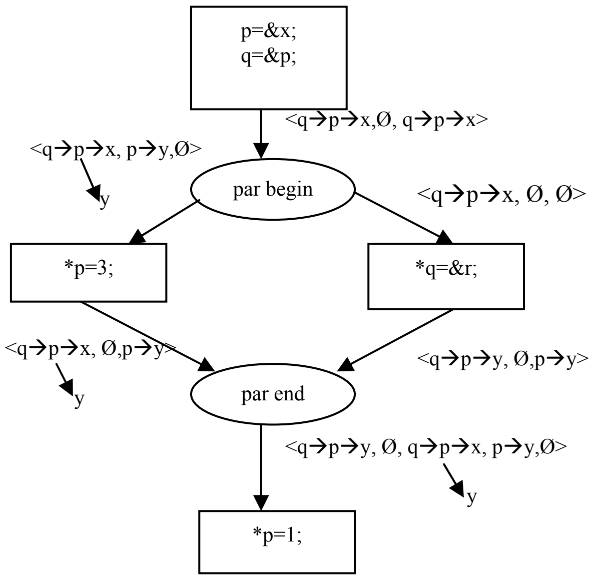

Fig. 19 Rinard's Analysis for multithreaded code

It approximates conservatively all possible analysis of concurrent thread statements.

IV. Comparison

In this section, behavior of algorithms described in section III is compared with each other along certain parameters as shown in the Table II.

According to the granularity of the precision of the points-to relationship, implementation approach can be with or without flow sensitivity and context-sensitivity. A fully flow and context sensitivity usually cost too much time and memory on large programs. Analysis provided by Andersen and Steensgaard are both context and flow-insensitive while refinement-based analysis is context-sensitive and flow – insensitive. Rinard's analysis takes into account both flow and context sensitivity.

There is always a trade-off between scalability and precision in pointer analysis. A precise analysis is more often not scalable. Andersen and Rinard's analysis is not scalable but precise. Steensgaard's analysis whereas is scalable but not precise. Bodik's analysis [6] is both precise and scalable.

Andersen and Steensgaard gave the analysis for C programming language. Bodik's demand-driven analysis is Java-oriented. Rinard's algorithm is suitable for language which support multi-threading.

TABLE II
COMPARISON TABLE

Comparison Parameters	Behaviour of Algorithm			
	Andersen	**Steensgaard**	**Refinement-Based**	**Rinard**
Algorithm Type	Subset-based algorithm	Unification-based(PTA)	Demand-driven	Inference-based
Time Complexity	O(n^3) where n is the number of nodes in points-to graph(non-linear)	O(n)-linear	For greater than 1,60,000 Statements,time taken is less than 13 minutes	Polynomial-time
Preciseness	Precise	Less precise than Andersen	Precise	Precise
Flow Sensitivity	Flow-insensitive	Flow-insensitive	Flow-insensitive	Flow-sensitive
Context-sensitivity	Context-insensitive	Context-insensitive	Context-sensitive	Context-Sensitive
Definiteness	May-Analysis	May-analysis	Refinement-based	May-analysis
Memory efficient	No	Yes	Yes(For greater than 1,60,000 Statements,memory required is less than 35 MB)	Yes
Scalable	No	Yes	Yes	No
Language Dependency	C language	C language	Java language	Java

V. Conclusions

This paper describes a comparative study of four pointer analyses. The major conclusions about these analyses are:

1) Andersen analysis [3] is a precise analysis but non-linear in both time and space requirements.
2) Steensgaard analysis [4] is a linear time analysis but provides spurious results.
3) Refinement-based analysis [6] is memory efficient as it is demand-driven and flow-insensitive. It is more precise and easy to implement.
4) Rinard's analysis[5] provides a solution to approximate conservatively all possible analysis of concurrent thread statements.

Acknowledgment

I dedicate this study to my family who have always supported me and believed that I could do it. I would like to thank all my friends for their invaluable help and moral support.

References

[1] W. Landi, Undecidability of static analysis, ACM Letters on Programming Languages and Systems, 1(4):323–337, 1992.

[2] J. Zhu, Towards scalable flow and context sensitive pointer analysis. In Design Automation Conference, 2005. Proceedings, 42nd, pages 831-836, 2005.

[3] Lars Ole Andersen, Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, May 1994 .

[4] Bjarne Steensgaard, Points-to analysis in almost linear time, In Symposium on Principles of Programming Languages (POPL), pages 32–41,1996.

[5] A. Salcianu and M. Rinard, Pointer and escape analysis for multithreaded programs. ACM SIGPLAN Notices, 36(7):23, 2001.

[6] Manu Sridharan and Rastislav Bodik, Refinement-based context-sensitive points-to analysis for java. SIGPLAN Not., 41(6):387-400, 2006.

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley, August 2006.

[8] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. PLDI*, pages 1–12, La Jolla, CA, June 1995.

[9] Susan Horwitz, Precise flow-insensitive may-alias analysis is np-hard, Transactions on Programming Languages and Systems, 19(1):1–6, 1997. ISSN 0164-0925.

[10] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In Proc. International Static Analysis Symposium (SAS), pages 57–81, 1998.

[11] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodik. Demand-driven points-to analysis for Java. In Richard P. Gabriel, editor, Proc.20th OOPSLA, pages 59–76, San Diego, CA, October 2005. ISBN 1-59593-031-0.