

General Flow-Sensitive Pointer Analysis and Call Graph Construction

Endre Horváth*, István Forgács**, Ákos Kiss*, Judit Jász*
and Tibor Gyimóthy*

*University of Szeged Aradi Vértanúk tere 1. 6720 Szeged, Hungary {hendre, akiss, jasy, gyimi} @inf.u-szeged.hu	**4D Soft Ltd. Soroksári út 18. 1095 Budapest, Hungary forgacs@4dsoft.hu
---	---

Abstract. Pointer analysis is a well known, widely used and very important static program analyzing technique. After having studied the literature in this field of research we found that most of the methods approach the problem in a flow-insensitive way, i.e. they omit the use of the control-flow information. Our goal was to develop a technique that is flow-sensitive and can be used in the analysis of large programs. During this process we have found that our method can give more accurate results if we build the call graph and compute the pointer information at the same time. In this paper we present two of our algorithms for pointer analysis and give some examples to help their comprehension. In the future these algorithms are planned to form the basis of a flow- and context-sensitive method used in the impact analysis of real life applications.

1 Introduction

Pointer analysis is a well known and widely used static program analyzing technique. The point of the analysis is to compute which variables use the same memory locations and which pointers may point to the same memory address. We have studied the relevant literature in this field (in a survey written in Hungarian language [6]) and found that most of the methods use a flow-insensitive approach to the problem. This results a faster, but less accurate algorithm. In the future we want to use the algorithm for impact analysis of large (over million lines of code), real-life C++ applications, so we cannot afford to lose accuracy. Similarly most of the algorithms studied can have problems with running time if they are used to analyze large programs.

With this knowledge we decided not to use any of the techniques studied, but to develop our own method. Our algorithm is a general, language independent, flow-sensitive approach to the problem, which can be used for the analysis of object-oriented programs. The base idea is a forward slicing technique, which ensures maximum accuracy possible. A further proven advantage of the algorithm is its short execution time even for large programs.

During the development of the algorithm we discovered that an interprocedural pointer analysis can only be successful if it builds the call graph at the same time. For this reason we present our algorithm in two steps. First we give an intraprocedural algorithm that computes the pointer information for a given function. This algorithm is flow-sensitive and context-insensitive, as any algorithm that focuses on only one function. Second we present our final results in the form of an interprocedural algorithm, which computes the pointer information and builds the call graph in parallel.

The rest of the paper is organized in the following way. In the next section we will introduce the intraprocedural algorithm and give an example to help its comprehension. In Section 3 we will present the interprocedural algorithm. Next, in Section 4, we will discuss some related work on points-to analysis. Finally, in Section 5, we will draw some conclusions from our results and will outline the directions of our future work.

```

1 algorithm intraprocedural_pointer_analysis(f)
2   List = MemAlloc(f)
3   while List is not empty do
4      $u \in List, List = List \setminus \{u\}$ 
5      $PtrInf^* = \bigcup_{v \in Prev(u)} PtrInf(v)$ 
6     if p gets a value at u then
7       for each  $\langle P, v \rangle \in PtrInf^*$  pair do
8          $P = P \setminus \{p\}$ 
9         if  $\exists q \in P : q \in Use(p, u)$  then
10           $P = P \cup \{p\}$ 
11        endif
12      endfor
13      if  $u \in MemAlloc(f)$  then
14         $PtrInf^* = PtrInf^* \cup \{\langle Def(u), u \rangle\}$ 
15      endif
16    endif
17    if  $PtrInf^* \neq PtrInf(u)$  then
18       $List = List \cup Next(u)$ 
19    endif
20     $PtrInf(u) = PtrInf^*$ 
21  endwhile
22 algorithm end

```

Fig. 1. Intraprocedural pointer analysis algorithm

2 Intraprocedural Pointer Analysis

The first step of the algorithm is to search for the memory allocation points in the program, and also for the statements where a pointer variable is set to point

```

procedure  $f()$ 
begin
   $x_1 = \text{new A}();$  (1)
   $x_2 = \text{new B}();$  (2)
  if (...) then begin (3)
     $x_3 = x_1;$  (4)
     $x_1 = \text{new A}();$  (5)
  end else begin
     $x_3 = x_2;$  (6)
     $x_1 = \text{new B}();$  (7)
  end;
   $x_4 = x_3;$  (8)
end;

```

Fig. 2. An example for the intraprocedural algorithm

to an invalid memory location (in different languages this can be expressed with *nil*, *null*, *NULL* or *0*). For these statements we set the initial pointer information to be $\{\langle(p), u\rangle\}$, where u is the number of the statement and p is the variable getting a value at that statement. (The algorithm stores the pointer information as a set of pairs like $\langle\text{variable set}, \text{number}\rangle$, where a pair gives the set of variables that may point to the same object after a statement and the number of the statement where the object was created).

Once we found the memory allocation points we can propagate this information through the statements of the program. The pointer information of a statement depends on two things: 1.) the information at the statements that precede the current statement in the control-flow of the program and 2.) the semantics of the statement itself. First we compute the union of the information propagated from the statement's ancestors and then we modify this result according to the statement. If a pointer gets a new value at the statement, we remove the pointer from each set of variables and add it to those sets that contain the variables which were used in the definition of the pointers new value. We let the algorithm iterate until it reaches its fixpoint.

The pseudo code of the algorithm is given in Fig. 1. The following notation is used in the code:

- f is a procedure or a function,
- u and v are arbitrary statements in the program,
- $List$ is a list or a set, which contains the statements to be analyzed,
- p and q are arbitrary pointer variables,
- P is the set of pointer variables that point to the same location,
- $MemAlloc(f)$ is a set that contains those statements of function f that are memory allocations or assign the *null* value to a variable,
- $Def(u)$ is the (pointer) variable that gets a value at statement u ,
- $Use(p, u)$ is the set of variables that appear in the definition of pointer p at statement u ,

- $Prev(u)$ is the set of statements that precede statement u in the control flow of the program,
- $Next(u)$ is the set of statements that succeed statement u in the control flow of the program,
- $PtrInf(u)$ contains the pointer information associated with statement u (it is empty at the start and it contains the required information after the execution of the algorithm).

To help the comprehension of the algorithm we present a small example with only a few variables and one conditional statement. This can be seen in Fig. 2. In the beginning $List = MemAlloc(f) = \{1, 2, 5, 7\}$, because the memory allocation points are at 1, 2, 5 and 7. Also $PtrInf(i) = \emptyset$ for $i = 1, \dots, 8$. We can see the necessary information and the results of the algorithm in three tables. First in the table in Fig. 3 we give the statements that precede and succeed each statement in the control flow of the example. Second we show the iteration of the algorithm in Fig. 4. Finally the table in Fig. 5 holds the pointer information for the example.

number of the statement – u	$Prev(u)$	$Next(u)$
1	\emptyset	{2}
2	{1}	{3}
3	{2}	{4,6}
4	{3}	{5}
5	{4}	{8}
6	{3}	{7}
7	{6}	{8}
8	{5,7}	\emptyset

Fig. 3. Control flow of the example in Fig. 2

3 Iterprocedural Analysis

The only way to compute precise pointer information is to propagate the necessary information across function boundaries in the program. There is no problem if the program contains only simple function calls because in this case we always know where to propagate the information. This task becomes harder if the program contains function calls whose target is not obvious (i.e. in the case of function pointers or polymorphic function calls). In this case we need to update the call graph as we compute the pointer information so the possible targets of such function calls become clear.

The first step of the algorithm is to find the possible entry points of the program (it is usually one function, like *main*, and all the static initialization blocks). Thereafter this we start the iteration of the algorithm. When we find

current statement – u	$PtrInf(u)$	$List$
1	$\{ \langle x_1, 1 \rangle \}$	$\{2, 5, 7\}$
2	$\{ \langle x_1, 1 \rangle, \langle x_2, 2 \rangle \}$	$\{5, 7, 3\}$
5	$\{ \langle x_1, 5 \rangle \}$	$\{7, 3, 8\}$
7	$\{ \langle x_2, 7 \rangle \}$	$\{3, 8\}$
3	$\{ \langle x_1, 1 \rangle, \langle x_2, 2 \rangle \}$	$\{8, 4, 6\}$
8	$\{ \langle x_1, 5 \rangle, \langle x_2, 7 \rangle \}$	$\{4, 6\}$
4	$\{ \langle (x_1, x_3), 1 \rangle, \langle x_2, 2 \rangle \}$	$\{6, 5\}$
6	$\{ \langle x_1, 1 \rangle, \langle (x_2, x_3), 2 \rangle \}$	$\{5, 7\}$
5	$\{ \langle x_3, 1 \rangle, \langle x_2, 2 \rangle, \langle x_1, 5 \rangle \}$	$\{7, 8\}$
7	$\{ \langle x_1, 1 \rangle, \langle x_3, 2 \rangle, \langle x_2, 7 \rangle \}$	$\{8\}$
8	$\{ \langle (x_3, x_4), 1 \rangle, \langle x_2, 2 \rangle, \langle x_1, 5 \rangle, \langle x_1, 1 \rangle, \langle (x_3, x_4), 2 \rangle, \langle x_2, 7 \rangle \}$	\emptyset

Fig. 4. The iteration of the algorithm, step-by-step, on the example in Fig. 2

number of the statement – u	$PtrInf(u)$
1	$\{ \langle x_1, 1 \rangle \}$
2	$\{ \langle x_1, 1 \rangle, \langle x_2, 2 \rangle \}$
3	$\{ \langle x_1, 1 \rangle, \langle x_2, 2 \rangle \}$
4	$\{ \langle (x_1, x_3), 1 \rangle, \langle x_2, 2 \rangle \}$
5	$\{ \langle x_3, 1 \rangle, \langle x_2, 2 \rangle, \langle x_1, 5 \rangle \}$
6	$\{ \langle x_1, 1 \rangle, \langle (x_2, x_3), 2 \rangle \}$
7	$\{ \langle x_1, 1 \rangle, \langle x_3, 2 \rangle, \langle x_2, 7 \rangle \}$
8	$\{ \langle (x_3, x_4), 1 \rangle, \langle x_2, 2 \rangle, \langle x_1, 5 \rangle, \langle x_1, 1 \rangle, \langle (x_3, x_4), 2 \rangle, \langle x_2, 7 \rangle \}$

Fig. 5. The pointer information for each statement for the example in Fig. 2

a function call we compute the possible targets for that call from the pointer information known so far. Although this is not necessary for direct function calls, the targets of indirect function calls can only be determined from the parallel points-to analysis. If the iteration reaches a new function we initialize the information for that function and it becomes part of the iteration.

We also have to define how we are propagating the information at function calls. We split each call into two parts (i.e. two nodes in the control-flow graph), a call point and a return point. This way the predecessors of a function’s entry point are the call points from where we could have called the function. Likewise the successors of the function’s exit point are the possible return points of the function.

We present the pseudo code of the algorithm in Fig. 6. We use the same formalism as before with the following extensions:

- F is the set of functions and procedures to be analyzed,

```

1 algorithm interprocedural_pointer_analysis( $F$ )
2    $List = \bigcup_{f \in \text{EntryPoints}} \text{MemAlloc}(f)$ 
3    $FuncList = \text{EntryPoints}$ 
4   while  $List$  is not empty do
5      $u \in List, List = List \setminus \{u\}$ 
6      $PtrInf^* = \bigcup_{v \in \text{Prev}(u)} PtrInf(v)$ 
7     if  $p$  gets a new value at  $u$  then
8       for each  $\langle P, v \rangle \in PtrInf^*$  pair do
9          $P = P \setminus \{p\}$ 
10        if  $\exists q \in P : q \in \text{Use}(p, u)$  then
11           $P = P \cup \{p\}$ 
12        endif
13      endfor
14      if  $u \in \text{MemAlloc}(f)$  then
15         $PtrInf^* = PtrInf^* \cup \{\langle \text{Def}(u), u \rangle\}$ 
16      endif
17      else if  $u$  is the entry point of a function call then
18         $List = List \cup (\bigcup_{f \in \text{Target}(u, PtrInf^*) \setminus FuncList} \text{MemAlloc}(f))$ 
19         $FuncList = FuncList \cup \text{Target}(u, PtrInf^*)$ 
20      endif
21      if  $PtrInf^* \neq PtrInf(u)$  then
22         $List = List \cup \text{Next}(u)$ 
23      endif
24       $PtrInf(u) = PtrInf^*$ 
25    endwhile
26 algorithm end

```

Fig. 6. Interprocedural pointer analysis algorithm

- EntryPoints is the set of functions that can be the entry points of the program,
- FuncList is the set of functions found during the iteration,
- $\text{Target}(u, PtrInf)$ is the set of functions that are the potential targets of the function call at statement u according to the pointer information $PtrInf$.

The algorithm builds the call graph in an implicit way. After the iteration the $\text{Target}(u, PtrInf(u))$ expression will give us the set of potentially called functions at an arbitrary statement u .

Again, to help the comprehension of the algorithm we present a simple example in Fig. 7 with four functions. In Fig. 8 we give the statements that precede and succeed each statement in the control flow of the example. We divided the call points in two parts labeled N_h for the call and N_v for the return. Function *main* is the only entry point of the program, so $\text{EntryPoints} = \text{FuncList} = \{\text{main}\}$ and according to this $List = \text{MemAlloc}(\text{main}) = \{10, 11\}$. In the beginning, $PtrInf(i) = \emptyset$ for $i = 1, \dots, 12$ (this includes both parts of the split points). We show the iteration of the algorithm in Fig. 9. and finally the table in Fig. 10 holds the pointer information for the example.

```

procedure f()           procedure h()
begin
   $x_3 = x_1;$            (1)    $x_3 = x_2;$            (8)
  g();                 (2)    $x_5 = \text{new B}();$    (9)
   $x_4 = x_1;$            (3)   end;
  h();                 (4)
   $x_5 = x_2;$            (5)
end;

procedure g()           procedure main()
begin
   $x_1 = x_2;$            (6)    $x_1 = \text{new A}();$    (10)
   $x_4 = \text{new A}();$    (7)    $x_2 = \text{new B}();$    (11)
end;                               f();           (12)
                                   end;

```

Fig. 7. An example for the interprocedural algorithm

number of the statement – u	Prev(u)	Next(u)
1	{12 _h }	{2 _h }
2 _h	{1}	{6}
2 _v	{7}	{3}
3	{2 _v }	{4 _h }
4 _h	{3}	{8}
4 _v	{9}	{5}
5	{4 _v }	{12 _v }
6	{2 _h }	{7}
7	{6}	{2 _v }
8	{4 _h }	{9}
9	{8}	{4 _v }
10	∅	{11}
11	{10}	{12 _h }
12 _h	{11}	{1}
12 _v	{5}	∅

Fig. 8. Control flow of the example in Fig. 7

current statement - u	$PtrInf(u)$	List
10	$\{ \langle x_1, 10 \rangle \}$	{11}
11	$\{ \langle x_1, 10 \rangle, \langle x_2, 11 \rangle \}$	{12 _h }
12 _h	$\{ \langle x_1, 10 \rangle, \langle x_2, 11 \rangle \}$	{1}
1	$\{ \langle (x_1, x_3), 10 \rangle, \langle x_2, 11 \rangle \}$	{2 _h }
2 _h	$\{ \langle (x_1, x_3), 10 \rangle, \langle x_2, 11 \rangle \}$	{7, 6}
7	$\{ \langle x_4, 7 \rangle \}$	{6, 2 _v }
6	$\{ \langle x_3, 10 \rangle, \langle (x_1, x_2), 11 \rangle \}$	{2 _v , 7}
2 _v	$\{ \langle x_4, 7 \rangle \}$	{7, 3}
7	$\{ \langle x_4, 7 \rangle, \langle x_3, 10 \rangle, \langle (x_1, x_2), 11 \rangle \}$	{3, 2 _v }
3	$\{ \langle \emptyset, 7 \rangle \}$	{2 _v }
2 _v	$\{ \langle x_4, 7 \rangle, \langle x_3, 10 \rangle, \langle (x_1, x_2), 11 \rangle \}$	{3}
3	$\{ \langle \emptyset, 7 \rangle, \langle x_3, 10 \rangle, \langle (x_1, x_2, x_4), 11 \rangle \}$	{4 _h }
4 _h	$\{ \langle x_3, 10 \rangle, \langle (x_1, x_2, x_4), 11 \rangle \}$	{9, 8}
9	$\{ \langle x_5, 9 \rangle \}$	{8, 4 _v }
8	$\{ \langle \emptyset, 10 \rangle, \langle (x_1, x_2, x_3, x_4), 11 \rangle \}$	{4 _v , 9}
4 _v	$\{ \langle x_5, 9 \rangle \}$	{9, 5}
9	$\{ \langle (x_1, x_2, x_3, x_4), 11 \rangle, \langle x_5, 9 \rangle \}$	{5, 4 _v }
5	$\{ \langle \emptyset, 9 \rangle \}$	{4 _v }
4 _v	$\{ \langle (x_1, x_2, x_3, x_4), 11 \rangle, \langle x_5, 9 \rangle \}$	{5}
5	$\{ \langle (x_1, x_2, x_3, x_4, x_5), 11 \rangle, \langle \emptyset, 9 \rangle \}$	{12 _v }
12 _v	$\{ \langle (x_1, x_2, x_3, x_4, x_5), 11 \rangle \}$	\emptyset

Fig. 9. The iteration of the algorithm, step-by-step, on the example in Fig. 7

number of the statement - u	$PtrInfo(u)$
1	$\{ \langle (x_1, x_3), 10 \rangle, \langle x_2, 11 \rangle \}$
2 _h	$\{ \langle (x_1, x_3), 10 \rangle, \langle x_2, 11 \rangle \}$
2 _v	$\{ \langle x_4, 7 \rangle, \langle x_3, 10 \rangle, \langle (x_1, x_2), 11 \rangle \}$
3	$\{ \langle x_3, 10 \rangle, \langle (x_1, x_2, x_4), 11 \rangle \}$
4 _h	$\{ \langle x_3, 10 \rangle, \langle (x_1, x_2, x_4), 11 \rangle \}$
4 _v	$\{ \langle (x_1, x_2, x_3, x_4), 11 \rangle, \langle x_5, 9 \rangle \}$
5	$\{ \langle (x_1, x_2, x_3, x_4, x_5), 11 \rangle \}$
6	$\{ \langle x_3, 10 \rangle, \langle (x_1, x_2), 11 \rangle \}$
7	$\{ \langle x_4, 7 \rangle, \langle x_3, 10 \rangle, \langle (x_1, x_2), 11 \rangle \}$
8	$\{ \langle (x_1, x_2, x_3, x_4), 11 \rangle \}$
9	$\{ \langle (x_1, x_2, x_3, x_4), 11 \rangle, \langle x_5, 9 \rangle \}$
10	$\{ \langle x_1, 10 \rangle \}$
11	$\{ \langle x_1, 10 \rangle, \langle x_2, 11 \rangle \}$
12 _h	$\{ \langle x_1, 10 \rangle, \langle x_2, 11 \rangle \}$
12 _v	$\{ \langle (x_1, x_2, x_3, x_4, x_5), 11 \rangle \}$

Fig. 10. The pointer information for each statement for the example in Fig. 7

4 Related Work

There are several methods developed to compute the pointer information for a given program. Although the speed and the accuracy of some algorithms is very good, we did not see any technique that gives a language independent, flow- and context-sensitive solution to the problem.

Hind et al. [5] and Jász et. al [6] (in hungarian) summarize the results in this field. Hind [3] describes related issues and remaining open problems associated with pointer analysis. Yur et al. [13] give a flow- and context-sensitive algorithm, which was developed for the analysis of C programs. Hind et al. [4] and Marx et al. [7] give a flow-sensitive, but context-insensitive solution for the problem. Andersen's [1] context-sensitive algorithm is a benchmark in this field, but Wilson et al. [12] and Whaley et al. [11] present good context-sensitive techniques also. Bacon and Sweeney [2] use the pointer analysis to handle the virtual function calls in the C++ code. Milanova et al. [9] measure the performance of technique called *Flow-insensitive Alias analysis* and also Milanova et al. [8] define object sensitivity to support the pointer analysis for object oriented languages.

5 Conclusions and Future Work

We presented two language-independent, flow-sensitive algorithms to compute the pointer information for a program. Our empirical tests show that the algorithms can be used quite well. Not only that we can determine the precise pointer information, but we can also build a call graph that is more accurate than the ones computed without the points-to information. Unfortunately, we did not implement the algorithms for any particular programming language yet, so we cannot present our empirical results in the terms of numbers.

The goal of our future work is to develop a system for the impact analysis of large, real-life programs. The algorithms presented in this paper are a part of this future system. The developments will include:

- **making the algorithm context sensitive:** by making the algorithm context-sensitive we will be able to distinguish the different call sites of a function and so we can make the pointer information more precise.
- **making the algorithm incremental:** if only minor changes are made to the program we do not have to compute all the information from the beginning, but will be able to reuse some data from previous runs.
- **focusing on the special properties of the C++ language:** there are only a few methods that can compute pointer information for a C++ program. Since nowadays C++ is one of the most popular programming languages we would like to give help to developers (and compilers) in the computation of points-to information for real-life C++ programs.

References

1. Andersen., L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (1996) 324–341
3. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), Snowbird, UT (2001)
4. Hind, M., Burke, M., Carini, P., Choi, J.D.: Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems* **21** (1999) 848–894
5. Hind, M., Pioli, A.: Which pointer analysis should I use? In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM Press (2000) 113–123
6. Jász, J., Horváth, E., Kiss, A., Forgács, I., Gyimóthy, T.: Pointeranalízis (2004)
7. Marx, D.I.S., Frankl, P.G.: Path-sensitive alias analysis for data flow testing. *Software Testing, Verification & Reliability* **9** (1999) 51–73
8. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press (2002) 1–11
9. Milanova, A., Rountev, A., Ryder, B.: Precise call graph construction in the presence of function pointers. Technical report, DCS-TR-442, Rutgers University (2001)
10. Tonella, P., Antoniol, G., Fiutem, R., Merlo, E.: Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In: Proceedings of the 19th international conference on Software engineering, ACM Press (1997) 433–443
11. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, ACM Press (2004) 131–144
12. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: SIGPLAN Conference on Programming Language Design and Implementation. (1995) 1–12
13. Yur, J., Ryder, B.G., Landi, W.A.: An incremental flow- and context-sensitive pointer aliasing analysis. In: Proceedings of the 21st international conference on Software engineering, IEEE Computer Society Press (1999) 442–451