Proc. Estonian Acad. Sci. Eng., 2005, **11**, 4, 286–295

General flow-sensitive pointer analysis and call graph construction

Endre Horváth[a], István Forgács[b], Ákos Kiss[a], Judit Jász[a] and Tibor Gyimóthy[a]

[a] Department of Software Engineering, University of Szeged, Aradi Vértanúk tere 1, 6720 Szeged, Hungary; {hendre,akiss,jasy,gyimi}@inf.u-szeged.hu
[b] 4D Soft Ltd., Telepy u. 24, 1096 Budapest, Hungary; forgacs@4dsoft.hu

Received 4 August 2005, in revised form 30 September 2005

Abstract. Pointer analysis is a well known, widely used and very important static program analysis 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 do not use 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 this 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, to be used in the impact analysis of real life applications.

Key words: pointer analysis, flow-sensitive, interprocedural.

1. INTRODUCTION

Pointer analysis is a well known and widely used static program analysis 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 [1]) and found that most of the methods use a flow-insensitive approach to the problem. This results in a faster, but less accurate algorithm. There are several methods developed to compute the pointer information for a given program. Although the speed and the accuracy of some algorithms are very good, we did not

see any technique that gives a language-independent, flow- and context-sensitive solution to the problem.

In his works Hind summarizes the results in this field [2] and describes related issues and open problems, associated with pointer analysis [3]. Yur et al. [4] give a flow- and context-sensitive algorithm, which was developed for the analysis of C programs. Hind et al. [5] and Marx et al. [6] give a flow-sensitive, but context-insensitive solution for the problem. Andersen's context-sensitive algorithm [7] is a benchmark in this field, but Wilson et al. [8] and Whaley et al. [9] also present good context-sensitive techniques. Bacon and Sweeney [10] use the pointer analysis to handle the virtual function calls in the C++ code. Milanova et al. [11] measure the performance of a technique called *Flow-insensitive Alias Analysis* and define object sensitivity to support the pointer analysis for object-oriented languages [12].

With the knowledge, gained from our survey, we decided not to use any of the techniques cited above, but to develop our own method. In the future we want to use the algorithm for impact analysis of large (over million lines of code), real-life C++ applications; thus we cannot afford to lose accuracy. 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 basic idea is a forward-slicing technique, which ensures maximum possible accuracy. An additional 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 introduce the intraprocedural algorithm and give an example to help its comprehension. In Section 3, we present the interprocedural algorithm. Finally, in Section 4, we draw some conclusions from our results and outline the directions of our future work.

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 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 $\langle$variable set, number$\rangle$, where a pair gives the set of variables that may

point to the same object after a statement and to the number of the statement, where the object was created).

Once we have 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 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 analysed,
- 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 the function f that are memory allocations or assign the *null* value to a variable,

```
1  algorithm intraprocedural_pointer_analysis(f)
2      List = MemAlloc(f)
3      while List is not empty do
4          u ∈ List, List = List \ {u}
5          PtrInf* = ⋃_{v∈Prev(u)} PtrInf(v)
6          if p gets a value at u then
7              for each ⟨P, v⟩ ∈ PtrInf* pair do
8                  P = P \ {p}
9                  if ∃q ∈ P : q ∈ Use(p, u) then
10                     P = P ∪ {p}
11                 endif
12             endfor
13             if u ∈ MemAlloc(f) then
14                 PtrInf* = PtrInf* ∪ {⟨Def(u), u⟩}
15             endif
16         endif
17         if PtrInf* ≠ PtrInf(u) then
18             List = List ∪ Next(u)
19         endif
20         PtrInf(u) = PtrInf*
21     endwhile
22 algorithm end
```

Fig. 1. Intraprocedural pointer analysis algorithm.

- *Def*(u) is the (pointer) variable that gets a value at the statement u,
- *Use*(p, u) is the set of variables that appear in the definition of the pointer p at the statement u,
- *Prev*(u) is the set of statements that precede the statement u in the control flow of the program,
- *Next*(u) is the set of statements that succeed the statement u in the control flow of the program,
- *PtrInf*(u) contains the pointer information associated with the 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 $i = 1, \ldots, 8$. We can see the necessary information and the results of the algorithm in three tables. First, in Table 1 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 Table 2. Finally Table 3 holds the pointer information for the example.

```
procedure f()
begin
   x1 = new A();          (1)
   x2 = new B();          (2)
   if (...) then begin    (3)
      x3 = x1;            (4)
      x1 = new A();       (5)
   end else begin
      x3 = x2;            (6)
      x1 = new B();       (7)
   end;
   x4 = x3;               (8)
end;
```

Fig. 2. An example of the intraprocedural algorithm.

Table 1. Control flow of the example in Fig. 2

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}	$\emptyset$

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

Current statement u	$PtrInf(u)$	$List$
1	$\{< x_1, 1 >\}$	$\{2, 5, 7\}$
2	$\{< x_1, 1 >, < x_2, 2 >\}$	$\{5, 7, 3\}$
5	$\{< x_1, 5 >\}$	$\{7, 3, 8\}$
7	$\{< x_2, 7 >\}$	$\{3, 8\}$
3	$\{< x_1, 1 >, < x_2, 2 >\}$	$\{8, 4, 6\}$
8	$\{< x_1, 5 >, < x_2, 7 >\}$	$\{4, 6\}$
4	$\{< (x_1, x_3), 1 >, < x_2, 2 >\}$	$\{6, 5\}$
6	$\{< x_1, 1 >, < (x_2, x_3), 2 >\}$	$\{5, 7\}$
5	$\{< x_3, 1 >, < x_2, 2 >, < x_1, 5 >\}$	$\{7, 8\}$
7	$\{< x_1, 1 >, < x_3, 2 >, < x_2, 7 >\}$	$\{8\}$
8	$\{< (x_3, x_4), 1 >, < x_2, 2 >, < x_1, 5 >, < x_1, 1 >, < (x_3, x_4), 2 >, < x_2, 7 >\}$	$\emptyset$

Table 3. The pointer information for each statement for the example in Fig. 2

Number of the statement u	$PtrInf(u)$
1	$\{< x_1, 1 >\}$
2	$\{< x_1, 1 >, < x_2, 2 >\}$
3	$\{< x_1, 1 >, < x_2, 2 >\}$
4	$\{< (x_1, x_3), 1 >, < x_2, 2 >\}$
5	$\{< x_3, 1 >, < x_2, 2 >, < x_1, 5 >\}$
6	$\{< x_1, 1 >, < (x_2, x_3), 2 >\}$
7	$\{< x_1, 1 >, < x_3, 2 >, < x_2, 7 >\}$
8	$\{< (x_3, x_4), 1 >, < x_2, 2 >, < x_1, 5 >, < x_1, 1 >, < (x_3, x_4), 2 >, < x_2, 7 >\}$

3. INTERPROCEDURAL 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; thus 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). After this we start the iteration of the algorithm. When we find 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 entry point are the call points, from where we could have called the function. Likewise the successors of the function exit point are the possible return points of the function.

We present the pseudo code of the algorithm in Fig. 3. We use the same formalism as before with the following extensions:

- F is the set of functions and procedures to be analysed,
- *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,
- *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 *Target*$(u, \mathit{PtrInf}(u))$ expression will give us the set of potentially called functions at an arbitrary statement u.

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

Fig. 3. Interprocedural pointer analysis algorithm.

Again, to help the comprehension of the algorithm we present in Fig. 4 a simple example with four functions. In Table 4 we give the statements that precede and succeed each statement in the control flow of the example. We divided the call points into two parts, labelled N_h for the call and N_v for the return. Function $main$ is the only entry point of the program, so $EntryPoints = FuncList = \{main\}$ and according to this $List = MemAlloc(main) = \{10, 11\}$. In the beginning, $PtrInf(i) = \emptyset$ for $i = 1, \ldots, 12$ (this includes both parts of the split points). We show the iteration of the algorithm in Table 5 and finally Table 6 holds the pointer information for the example.

```
procedure f()                    procedure h()
begin                            begin
  x3 = x1;          (1)            x3 = x2;          (8)
  g();              (2)            x5 = new B();     (9)
  x4 = x1;          (3)          end;
  h();              (4)
  x5 = x2;          (5)
end;

procedure g()                    procedure main()
begin                            begin
  x1 = x2;          (6)            x1 = new A();     (10)
  x4 = new A();     (7)            x2 = new B();     (11)
end;                               f();              (12)
                                 end;
```

Fig. 4. An example for the interprocedural algorithm.

Table 4. Control flow of the example in Fig. 7

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	$\emptyset$	$\{11\}$
11	$\{10\}$	$\{12_h\}$
12_h	$\{11\}$	$\{1\}$
12_v	$\{5\}$	$\emptyset$

Table 5. The iteration of the algorithm, step by step, on the example in Fig. 7

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

Table 6. The pointer information for each statement for the example in Fig. 7

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

4. 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 efficiently. 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 the following:

- **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 help the developers (and compilers) in the computation of points-to information for real-life C++ programs.

REFERENCES

1. Jász, J., Horváth, E., Kiss, A., Forgács, I. and Gyimóthy, T. *Pointeranalízis*. University of Szeged, Szeged, 2004.
2. Hind, M. and Pioli, A. Which pointer analysis should I use? *ACM SIGSOFT Softw. Eng. Notes*, 2000, **25**, 113–123.
3. Hind, M. Pointer analysis: Haven't we solved this problem yet? In *Proc. 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, 2001.
4. Yur, J., Ryder, B. G. and Landi, W. A. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proc. 21st International Conference on Software Engineering*. Los Angeles, 1999, 442–451.
5. Hind, M., Burke, M., Carini, P. and Choi, J.-D. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 1999, **21**, 848–894.
6. Marx, D. I. S. and Frankl, P. G. Path-sensitive alias analysis for data flow testing. *Softw. Test. Verif. Reliab.*, 1999, **9**, 51–73.
7. Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
8. Wilson, R. P. and Lam, M. S. Efficient context-sensitive pointer analysis for C programs. *ACM SIGPLAN Notices*, 1995, **30**, 1–12.
9. Whaley, J. and Lam, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Notices*, 2004, **39**, 131–144.

10. Bacon, D. F. and Sweeney, P. F. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Notices*, 1996, **31**, 324–341.
11. Milanova, A., Rountev, A. and Ryder, B. Precise call graph construction in the presence of function pointers. In *Proc. Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*. Montreal, 2002, 155.
12. Milanova, A., Rountev, A. and Ryder, B. G. Parameterized object sensitivity for points-to and side-effect analyses for Java. *ACM Trans. Softw. Eng. Methodol.*, 2002, **14**, 1–11.

Juhtimisvoogu arvestav üldine viidaanalüüs ja juhtimisvoograafi koostamine

Endre Horváth, István Forgács, Ákos Kiss, Judit Jász ja Tibor Gyimóthy

Viidaanalüüs on staatilise programmianalüüsi tuntud tehnika. Valdkonna uurimisel on leitud, et enamik viidaanalüüsi meetodeist ei arvesta analüüsides programmi juhtimisvoos antud teavet. Eesmärgiks on luua selline staatilise programmianalüüsi tehnika, mis arvestaks juhtimisvoogu (oleks n-ö juhtimisvootundlik) ja oleks rakendatav suurte programmide analüüsil. On täheldatud, et niisugune analüüsimeetod annab täpsemaid tulemusi siis, kui samaaegselt koostatakse juhtimisvoograaf ja tehakse viidaanalüüs. Artiklis on esitatud koos selgitavate näidetega kaks sellist viidaanalüüsi algoritmi. Esitatud algoritmid on aluseks tulevasele juhtimis- ja kontekstitundlikule viidaanalüüsimeetodile.