# Precise Call Graph Construction in the Presence of Function Pointers*

Ana Milanova Atanas Rountev Barbara G. Ryder
Rutgers University, New Brunswick, NJ 08903, USA
{milanova,rountev,ryder}@cs.rutgers.edu

**Abstract**

The use of pointers creates serious problems for optimizing compilers and software engineering tools. Pointers enable indirect memory accesses through pointer dereferences, as well as indirect procedure calls (e.g., through function pointers in C). Such indirect accesses and calls can be disambiguated with *pointer analysis*. In this paper we evaluate the precision of a pointer analysis by Zhang et al. [17] for the purposes of call graph construction for C programs with function pointers. The analysis uses an inexpensive, almost-linear, flow- and context-insensitive algorithm. To measure analysis precision, we compare the call graph computed by the analysis with the most precise call graph obtainable by a large category of pointer analyses. Surprisingly, for all our data programs the analysis from [17] achieves the best possible precision. This result indicates that for the purposes of call graph construction, even inexpensive analyses can provide very good precision, and therefore the use of more expensive analyses may not be justified.

## 1 Introduction

In languages like C, the use of *pointers* creates serious problems for optimizing compilers and software engineering tools. Pointers enable *indirect memory accesses*. For example, the statement `*p=1` may indirectly modify all variables that are pointed to by variable `p`. In addition, pointers allow *indirect procedure calls*—for example, if `fp` is a function pointer in C, statement `(*fp)()` may invoke all functions that are pointed to by `fp`.

Precise information about memory accesses and procedure calls is fundamental for a large number of static analyses used in optimizing compilers and software engineering tools. To obtain such information, a variety of *pointer analyses* have been developed [8, 7, 4, 1, 15, 14, 17, 12, 9, 6, 3, 5, 2]. These analyses provide different tradeoffs between cost and precision. For example, flow- and context-insensitive pointer analyses [1, 14, 17, 12, 3] ignore the flow of control between program points and do not distinguish between different calling contexts of procedures. As a result, such analyses are relatively inexpensive and imprecise. In contrast, analyses with some degree of flow- or context-sensitivity are typically more expensive and precise.

The precision of different analyses has been traditionally measured with respect to the disambiguation of indirect memory accesses. However, there has been no work on measuring analysis precision with respect to the disambiguation of indirect procedure calls and its impact on the construction of the program call graph. The goal of our work is to measure the precision of a pointer analysis by Zhang et al. [17] (referred to as the *FA pointer analysis*) for the purposes of call graph construction for C programs with function pointers. The FA analysis is a flow- and context-insensitive analysis with $O(n\alpha(n,n))$ complexity, where $n$ is the size of the program and $\alpha$ is the inverse of Ackermann's function. The analysis is at the lower end of the spectrum with respect to cost and precision, and is comparable to Steensgaard's points-to analysis [14].

The FA analysis was implemented in the context of a source code browser for C developed at the Software Engineering Department of Siemens Corporate Research. The standard version of the browser provides syntactic cross-reference information and a graphical user interface for accessing this information. The PROLANGS group at Rutgers University worked on extending the browser functionality to provide and display semantic information obtained through static analysis. In particular, we implemented the FA pointer analysis and used its output to augment the call graph information provided by the browser. In the standard syntax-based browser version, indirect procedure calls could not be handled. By using the output of the FA analysis, the browser became capable of providing correct and complete information about the program call graph.

To measure analysis precision, for each of our data programs we compared the call graph computed by the FA analysis with the call graph computed using the

*This research was supported by NSF grant CCR-9900988 and by Siemens Corporate Research.

most precise pointer analysis.[1] By comparing these two call graphs, we wanted to evaluate the imprecision of the FA analysis and to gain insight into the sources of this imprecision. Somewhat surprisingly, in all our data programs there was *no difference* between the two call graphs. This result indicates that for the purposes of call graph construction, even analyses at the lower end of the cost/precision spectrum can provide very good precision, and therefore the use of more expensive analyses may not be justified. This finding is particularly interesting because in the context of disambiguating indirect memory accesses, the use of more expensive analysis provides substantial precision benefits.

The rest of this paper is organized as follows. Section 2 describes the FA pointer analysis. The notion of "most precise pointer analysis" is discussed in Section 3. Section 4 describes our empirical results and the conclusions from these results.

## 2 The FA Pointer Analysis

The FA analysis is a pointer alias analysis[2] for C which is flow-insensitive and context-insensitive. A version of FA has been initially defined in the context of program decomposition [17] for programs without unions and type casting. Later it has been extended to handle type casting and unions [16]. The FA analysis is based on fast UNION-FIND data structures and runs in almost linear time. It is a relatively imprecise and computationally inexpensive memory disambiguation technique. The features of FA which distinguish it from other popular unification-based pointer analyses such as [14] are the following:

- It takes into consideration the declared types of variables
- It makes distinction between structure fields
- It is designed to handle unions and type casting

In this section we briefly describe the FA analysis. The algorithm first computes the PE equivalence relation on the names that appear in the program. The PE relation is then used to derive the FA equivalence relation which provides aliasing information.

### 2.1 The PE relation

The memory locations and the addresses of memory locations are referred to as *object names*. The set of object names for the example from Figure 1 is {`p`,`&x`,`x`,`*p`, `p->f`,`z`,`&z`,`tt`}. These are the names that appear syntactically in the program; `*p` appears as a prefix of `(*p).f`; `x` and `z` appear in `&x` and `&z`, respectively.

```
p = &x;
p->f = &z;
tt = p;
```

Figure 1: Example Program

The goal of the PE relation is to group the object names into equivalence classes by value equality. If the members of a given equivalence class are pointers then those pointers may point to the same memory location. Similarly, if the members of an equivalence class are of structure type, then the values of their fields are identical.

The algorithm builds an initial graph $G_{PE}$, where the nodes are the object names and the edges could be *dereference edges* labeled with * or *field edges* labeled with the field identifier. For our example, the initial graph contains the following edges:

$$\{\texttt{p} \xrightarrow{*} \texttt{*p},\ \texttt{*p} \xrightarrow{f} \texttt{p->f},\ \texttt{\&x} \xrightarrow{*} \texttt{x},\ \texttt{\&z} \xrightarrow{*} \texttt{z}\}$$

The algorithm processes each pointer-related statement in the program, and merges the nodes corresponding to the equivalence classes of the object names on the left-hand and right-hand sides of the statement. If there are outgoing edges with the same label, those nodes are subsequently merged recursively. For our example, statement `p=&x` results in merging nodes `p` and `&x`; subsequently, nodes `*p` and `x` are merged as well. Nodes `p->f` and `&z` are merged due to statement `p->f=&z`; no recursive merge follows because `p->f` has no outgoing edges. Similarly, nodes `tt` and `p` are merged due to the last statement.

The nodes in the final $G_{PE}$ are equivalence classes of object names, and the edges are either dereference edges or field edges. The resulting $G_{PE}$ for our program contains the following equivalence classes, represented in the graph as nodes:

{`p`,`&x`,`tt`}, {`*p`,`x`}, {`p->f`,`&z`}, {`z`}

The meaning of this information is that after dereferencing `p` and `tt` might have the same value, that is, they may point to the same memory location. Similarly, `x` and `*p` (which are of structure type) might have their fields containing the same value.

### 2.2 The FA relation

The FA relation is derived from the PE relation in the following way: suppose that two object names $o_1$ and $o_2$ appear in the same equivalence class $n$ in $G_{PE}$. If

[1] The exact definition of the "most precise analysis" will be discussed in Section 3.

[2] *Aliasing* occurs when multiple names refer to the same memory location. For example, after the statement `p=&x`, `*p` and `x` are aliases.

(i) there is a path in $G_{PE}$ which starts from $n$ with a dereference edge and leads to another equivalence class $m$, and (ii) $m$ contains object names $o_3$ and $o_4$ which are derived from $o_1$ and $o_2$ respectively, then $o_3$ and $o_4$ are in the FA relation. It can be proven that if two object names may be aliased at some program point, these names are in the FA relation [16].

For our example, the elements of the $G_{PE}$ equivalence class {`p,&x,tt`} are not aliased because there is no path in $G_{PE}$ which leads to it. The FA equivalence classes are

{`p`}, {`tt`}, {`*p,x`}, {`p->f`}, {`z`}

For example, the meaning of FA equivalence class {`*p,x`} is that `*p` and `x` may be aliased at some program point. Note that the computation of the FA relation also removes object names of the form `&x`.

For the purposes of our investigation, we used the version of FA which handles unions and type casting. Using this version of the analysis, we examined the equivalence class of `*fp` for each function pointer `fp`. This information was used to determine the possible targets of all indirect calls through `fp`.

## 3 The Most Precise Pointer Analysis

In our comparison experiments, we wanted to determine the difference between the call graph computed with the FA analysis and the "best possible" call graph. Our notion of a "most precise analysis" is defined with respect to a specific category of pointer analyses. Each analysis from this category can be defined by a tuple $<G, L, F, M, \eta>$, where:

- $G = (N, E, n_0)$ is a directed graph with node set $N$, edge set $E$ and starting node $n_0 \in N$. For our purposes, $G$ is an interprocedural control flow graph, as defined below.
- $E_+ \subseteq N \times N$ is a set of additional edges that may result from the resolution of indirect calls, as described below.
- $<L, \leq, \wedge>$ is a finite meet semi-lattice [10] with partial order $\leq$ and meet operation $\wedge$.
- $F \subseteq \{f \mid f : L \times E_+ \to L \times E_+\}$ is a function space closed under composition and arbitrary meets. We assume that $F$ is monotone [10].
- $M : N \to F$ is an assignment of transfer functions to the nodes in $G$ (without loss of generality, we assume no edge transfer functions). The transfer function for node $n$ will be denoted by $f_n$.
- $\eta \in L$ is the solution at the bottom of $n_0$.

The program is represented by an *interprocedural control flow graph* (ICFG) [8], which contains control flow graphs for all procedures in the program. Each procedure has associated a single *entry node* (node $n_0$ is the entry node of the starting procedure) and a single *exit node*. Each call statement is represented by a pair of nodes, a *call node* and a *return node*. For each *direct call*, there is an edge from the call node to the entry node of the called procedure, as well as an edge from the exit node of the called procedure to the return node in the calling procedure. For *indirect calls*, $G$ does not contains edges (call,entry) or (exit,return). Such edges are discovered during the analysis, and are accumulated in sets $e \subseteq E_+$.

The lattice elements are points-to graphs representing the current points-to relationships in memory. Each transfer function $f : L \times E_+ \to L \times E_+$ takes as input a points-to graph and a set of currently known (call,entry) and (exit,return) edges at indirect calls. The function produces a new points-to graph by adding new points-to edges and removing "killed" points-to edges. In addition, $f$ produces a set $e \subseteq E_+$ of new (call,entry) and (exit,return) edges at indirect calls.

The sequence of nodes $n_0, \ldots, n_i$ (where $n_0$ is the starting node) is a *path* if and only if $n_0, \ldots, n_{i-1}$ is a path, and

- $(n_{i-1}, n_i) \in E$, or
- $(n_{i-1}, n_i) \in e$, where

$$(l, e) = f_{n_i}(\ldots(f_{n_1}(f_{n_0}(\eta, \emptyset)))\ldots)$$

For each such path $p = (n_0, \ldots, n_i)$, let $f_p = f_{n_0} \circ f_{n_1} \circ \ldots \circ f_{n_i}$. A *realizable path* is a path on which every procedure returns to the call site which invoked it [13, 8, 11]; only such paths represent potential sequences of execution steps. Let $RP(n_0, n)$ be the set of all realizable paths from starting node $n_0$ to any node $n$. For each $n \in N$, the **meet-over-all-realizable-paths** (MORP) solution at $n$ is defined as

$$MORP(n) = \bigwedge_{p \in RP(n_0, n)} f_p(\eta, \emptyset)$$

The MORP solution is the most precise solution computable by any analysis describable in this model. For each of our data programs, we considered all $n \in N$ that represent indirect calls. For each such $n$, we manually computed $MORP(n)$. This allowed us to determine the best possible call graph computable by analyses that belong to the category described above. Since the vast majority of published pointer analyses fall in this category, this "best call graph" is the most precise call graph

| Name | Description | LOC | Indirect Calls |
|---|---|---|---|
| `diction` | GNU diction command | 2652 | 3 |
| `072.sc` | Spreadsheet program | 9192 | 2 |
| `gettext` | GNU gettext command | 12327 | 23 |
| `find` | GNU find command | 15200 | 22 |
| `minicom` | UNIX communication program | 15607 | 6 |
| `m4` | GNU macro processor | 16375 | 17 |
| `less` | GNU less command | 20397 | 4 |
| `unzip` | Extraction utility | 26273 | 307 |

Table 1: Program Description

obtainable with the standard, widely-used pointer analysis technology. By comparing this call graph with the call graph computed by the FA analysis, we evaluated the amount of imprecision of the FA analysis.

## 4 Empirical Results

We have performed preliminary experiments on a set of realistic C programs, ranging in size from 2652 to 26273 lines of code. The description of the dataset is given in Table 1. Each program employs function pointers; the number of indirect calls in the program is shown in the last column of Table 1.

Our comparison of the FA-based call graphs with the best possible call graphs showed *no differences*. This surprising result can be explained with the fact that the usage of function pointers in C programs is simpler than the usage of data pointers; in fact, we observed several stylistic patterns of usage of function pointers. The results from this experiment indicate that inexpensive analyses such as FA may provide sufficient precision for the purposes of call graph construction. In this context, the use of more pointer expensive analyses may not be justified.

## References

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[3] M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[4] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Conference on Programming Language Design and Implementation*, pages 242–257, 1994.

[5] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[6] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, LNCS 1824, pages 175–198, 2000.

[7] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Trans. Programming Languages and Systems*, 21(4):848–894, May 1999.

[8] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Conference on Programming Language Design and Implementation*, pages 235–248, 1992.

[9] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 199–215, 1999.

[10] T. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[11] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[12] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[13] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and

N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[14] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[15] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[16] S. Zhang. *Practical Pointer Aliasing Analyses for C*. PhD thesis, Rutgers University, August 1998.

[17] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Symposium on the Foundations of Software Engineering*, pages 81–92, 1996.