Precise Call Graph Construction in the Presence of Function Pointers*

Ana Milanova Atanas Rountev Barbara G. Ryder
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
{milanova,rountev,ryder}@cs.rutgers.edu

Abstract

The use of pointers presents serious problems for software productivity tools for software understanding, restructuring, and testing. 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. [20, 19] for the purposes of call graph construction for C programs with function pointers. The analysis is implemented in the context of a production-strength code-browsing tool from Siemens Corporate Research. 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 [20, 19] 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 software productivity tools that use some form of semantic code analysis for the purposes of software understanding, restructuring, and testing. Pointers enable *indirect memory accesses*. For example, consider the following sequence of statements:

```
1  *p = 1;
2  write(x);
```

The first statement contains an indirect memory access through pointer `p`. At line 1 we need to know where `p` may point to in order to determine what variables may be modified by the statement. This information is needed by a variety of applications: for example, when slicing with respect to statement 2, a slicing tool needs to determine whether statement 1 should be included in the slice. 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`. Such indirect calls significantly complicate the interprocedural flow of control in the program.

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. The goal of *pointer analysis* is to determine the set of memory locations that a given memory location may point to. For example, for the sample statements from above, pointer analysis can determine what are the locations that `p` may point to. In addition, pointer analysis determines which function addresses may be stored in a given function pointer. Because of the importance of such points-to information, a variety of analyses have been developed [10, 9, 5, 1, 18, 17, 20, 15, 11, 7, 4, 6, 3, 14, 8]. These analyses provide different tradeoffs between cost and precision. For example, flow- and context-insensitive pointer analyses [1, 17, 20, 15, 4] 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 more precise.

The precision of different analyses has been tradition-

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

ally measured with respect to the disambiguation of indirect memory accesses (e.g., what locations does `p` point to in statement `*p = 1`). 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. [20, 19] (referred to by its authors 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 α is the inverse of Ackermann's function. The analysis is relatively imprecise and inexpensive, and is comparable to Steensgaard's points-to analysis [17] in terms of cost and precision.

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 "most precise" call graph. In Section 4 we discuss in detail our definition of "most precise", but intuitively, this was the best call graph that could be computed by a wide variety of existing pointer analyses (including analyses that are theoretically more precise than the FA analysis, and substantially more expensive in practice). 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. 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 existing empirical evidence shows that for the purposes of disambiguating indirect memory accesses (e.g., in `*p = 1`), the use of more expensive analyses provides substantial precision benefits.

```
typedef int (*PFB)();

struct parse_table {
    char *name;
    PFB func; };

int func1() { ... }
int func2() { ... }

struct parse_table table[] = {
    {"name1", &func1},
    {"name2", &func2} };

PFB find_p_func(char *s) {
    for (i=0; i<num_func; i++)
      if (strcmp(table[i].name,s)==0)
        return table[i].func;
    return NULL; }

int main(int argc, char *argv[]) {
    ...
    PFB parse_func=find_p_func(argv[1]);
    if (parse_func)
      (*parse_func)();
    else { ... } }
```

Figure 1. Table dispatch.

Contributions The contributions of our work are the following:

- We present the first empirical study of pointer analysis precision with respect to disambiguation of indirect procedure calls and call graph construction.
- On a set of eight publicly available realistic C programs, we show that a relatively imprecise and inexpensive pointer analysis produces the most precise call graph. Therefore, for the purposes of call graph construction, the use of more expensive analyses may not be justified.

Outline The rest of this paper is organized as follows. Section 2 presents several examples of function pointer usage in C. Section 3 describes the FA pointer analysis. The notion of most precise call graph is discussed in Section 4. Section 5 describes our empirical results and the conclusions from these results. Related work is discussed in Section 6.

2 Usage Patterns for Function Pointers in C

This section presents several examples that illustrate typical uses of function pointers in C programs. These examples are representative of the stylistic patterns we encountered in our benchmarks.

```
struct _chunk { ... };
struct obstack {
    struct _chunk *chunk;
    struct _chunk *(*chunkfun) ();
    void (*freefun) (); };
void chunk_fun(struct obstack *h,
               void *f) {
    h->chunkfun=
        (struct _chunk *(*)())f; }
void free_fun(struct obstack *h,
              void *f) {
    h->freefun = (void (*)()) f; }
int main() {
    struct obstack h;
    chunk_fun(&h,&xmalloc);
    free_fun(&h,&xfree); ...  }
```

Figure 2. Extendable functionality.

2.1 Table Dispatch

Consider the example in Figure 1. Table `table` maps a name to a function address. Function `find_p_func` takes a name as an argument and returns the address of the function that corresponds to that name in the map. Therefore, the function invoked at run time at the indirect invocation site at line 7 is either `func1` or `func2`, depending on the value of the first command line argument. Storing function addresses in large function tables is widely used. At run time the functions are typically dispatched from the table based on user input (e.g., command line option, command line argument, or spreadsheet function).

2.2 Extendable and Customizable Functionality

Figure 2 shows part of a memory management library. Functions `chunk_fun` and `free_fun` allow the library client to choose memory allocation and deallocation routines initially associated with each `obstack`. The library client may also redefine `chunkfun` and `freefun` of an `obstack` at any time.

We also observed libraries that define global data structures with function pointer fields. These fields are initialized to point to functions that provide default functionality. The library provides an interface which allows the client to redefine or extend this functionality by changing the values of the function pointer fields.

2.3 Polymorphic Behavior

In some cases formal parameters are declared as function pointers in order to allow the enclosing function to behave in a polymorphic manner. For example, the goal of function

```
void sentence(FILE *f,void (*process)())
```

may be to read all sentences from a given file, parse each sentence, and then process that sentence. If `sentence` is invoked from a word counting routine, the processing routine `process` will be counting the words in a sentence. If `sentence` is invoked from a spell checking routine, `process` will be checking for spelling mistakes.

3 The FA Pointer Analysis

The FA analysis [20, 19] is a pointer alias analysis[1] for C that is flow-insensitive and context-insensitive. *Flow-sensitive* analyses take into account the flow of control between program points inside a procedure, and compute separate solutions for these points. *Flow-insensitive* analyses ignore the flow of control between program points, and therefore can be less precise and more efficient than flow-sensitive analyses. *Context-sensitive* analyses distinguish between the different contexts under which a procedure is invoked, and analyze the procedure separately for each context. *Context-insensitive* analyses do not separate the different invocation contexts for a procedure, which improves efficiency at the expense of some possible precision loss.

The FA analysis is based on an algorithm that uses a fast UNION-FIND data structure and has almost-linear time complexity. The analysis is a relatively imprecise and computationally inexpensive approach for memory disambiguation, and belongs at the low end of the pointer analysis spectrum with respect to cost and precision. This analysis is similar to a popular unification-based pointer analysis by Steensgaard [17]. The most important difference between the FA analysis and Steensgaard's analysis is that the FA analysis is able to distinguish between structure fields. For example, if there is a structure with two pointer fields `f` and `g` and if `s` is a variable of that structure type, Steensgaard's analysis will always determine that `s.f` and `s.g` point to the same memory locations, because it associates a single alias set with `*(s.f)` and `*(s.g)`. The FA analysis computes distinct alias sets for `*(s.f)` and `*(s.g)`.

In this section we briefly describe the FA analysis at a high level; we do not intend to provide all details of its design and implementation. The analysis is described in detail in [20, 19].

The algorithm first computes an equivalence relation, referred to in [20, 19] as the *PE (Pointer-related Equality) equivalence relation*. The PE relation is then used to derive another equivalence relation, referred to as the *FA (Flow-insensitive Alias) equivalence relation*. The FA relation provides aliasing information.

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

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

Figure 3. Sample set of statements.

3.1 The PE Equivalence Relation

Memory locations and addresses of memory locations are referred to as *object names*. For example, the set of object names for the statements from Figure 3 is

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

These are the names that appear syntactically in the program. Name `*p` is added because it appears as a prefix of `(*p).f`. Names `x` and `z` appear in `&x` and `&z`, respectively.

If a pair of object names is in the PE relation, that means that the expressions denoted by these names may have the same value at run time. For example, for pointers this means that the two pointers may point to the same memory locations. The relation defines a partition of the set of object names into equivalence classes. To compute the relation, the analysis builds a graph referred to as the G_{PE} graph. The nodes in this graph are equivalence classes of object names; the edges are either *dereference edges* labeled with *, or *field edges* labeled with a field identifier.

In the initial G_{PE} graph there is a singleton equivalence class for each object name. For our sample set of statements, the initial graph contains the following edges:

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

The analysis processes all program statements and merges nodes corresponding to expressions that may have the same value. For example, for each assignment, the analysis merges the two nodes corresponding to the equivalence classes that contain the left-hand side and the right-hand side of the assignment. If there are outgoing edges that have the same label, their target nodes are merged recursively. For our example, after processing `p=&x` the analysis merges nodes `p` and `&x`. Nodes `*p` and `x` are merged as well because there are outgoing edges labeled `*` from `p` to `*p` and from `&x` to `x`. 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} graph define the PE equivalence relation. The graph for our example contains the following equivalence classes, represented as graph nodes:

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

Example 1 Recall the set of statements in Figure 1. The initial G_{PE} graph contains the following edges:

$$\texttt{table[]} \xrightarrow{\text{name}} \texttt{table[].name}$$

$$\texttt{table[]} \xrightarrow{\text{func}} \texttt{table[].func}$$

$$\texttt{\&func1} \xrightarrow{*} \texttt{func1} \qquad \texttt{\&func2} \xrightarrow{*} \texttt{func2}$$

When `table` is initialized, singleton equivalence classes $\{$`table[].func`$\}$ and $\{$`&func1`$\}$ are merged first. Then nodes $\{$`table[].func,&func1`$\}$ and $\{$`&func2`$\}$ are merged. Because there are outgoing edges with the same label from these nodes to nodes $\{$`func1`$\}$ and $\{$`func2`$\}$ respectively, $\{$`func1`$\}$ and $\{$`func2`$\}$ are merged as well. As a result of the initialization of `table`, the analysis creates the following equivalence classes (connected with a dereference edge):

$$\{\texttt{table[].func, \&func1, \&func2}\}$$

$$\{\texttt{func1, func2}\}$$

After line 3 the created equivalence classes are

$$\{\texttt{ret\_find\_p\_func, table[].func,}$$

$$\texttt{\&func1, \&func2}\}$$

$$\{\texttt{func1, func2}\}$$

where `ret_find_p_func` contains the return values of `find_p_func`. At line 5 the equivalence class which contains `ret_find_p_func` is merged with the singleton class $\{$`parse_func`$\}$. As a result of the recursive merge, the following equivalence classes are produced:

$$\{\texttt{ret\_find\_p\_func, table[].func,}$$

$$\texttt{\&func1, \&func2, parse\_func}\}$$

$$\{\texttt{func1, func2, *parse\_func}\}$$

The G_{PE} graph contains additional nodes and edges (e.g., related to `argv` and `s`); for brevity, the rest of the graph is not shown.

3.2 The FA Relation

The FA relation is derived from the PE relation in the following manner. Suppose that two object names o_1 and o_2 belong to the same equivalence class $n \in G_{PE}$. If n has an outgoing edge labeled with `*` and if the target node of that edge contains object names o'_1 and o'_2 that are derived from o_1 and o_2, then the pair (o'_1, o'_2) is in the FA relation. For example, for the statements in Figure 3, `p` and `&x` belong to node $\{$`p, &x, tt`$\}$, which has an outgoing dereference edge whose target is node $\{$`*p, x`$\}$; therefore, the pair (`*p`,`x`) is in the FA relation. In general, the relation holds for any pair of object names o'_1 and o'_2 that belong to the same equivalence class $m \in G_{PE}$ and for which (i) there is a path in

G_{PE} from n to m labeled $a_1, a_2, \ldots, a_k$ such that at least one a_i is a dereference, and (ii) o'_1 and o'_2 are derived from $o_1 \in n$ and $o_2 \in n$ by applying $a_1, a_2, \ldots, a_k$.

Given the G_{PE} graph, it is straightforward to compute the FA relation. It can be proven that if two object names may be aliased at some program point, they are guaranteed to be in the FA relation [20, 19]. For our example, the FA equivalence classes are:

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

The meaning of FA equivalence class {`*p,x`} is that `*p` and `x` may be aliased at some program point.

For the purposes of our investigation, we examined the FA equivalence class of `*fp` for each function pointer `fp` in the program. This information was used to determine the possible targets of all indirect calls through `fp`.

Example 2 Recall Example 1 from Section 3.1. After constructing the G_{PE} graph, the analysis computes the following FA equivalence classes:

{`ret_find_p_func`}, {`table[].func`},
{`parse_func`},
{`*parse_func, func1, func2`}

Therefore, the possible targets of the indirect call at line 7 in Figure 1 are `func1` and `func2`.

4 The Most Precise Pointer Solution

In our comparison experiments, we wanted to determine the difference between the call graph computed by the FA analysis and the best possible call graph (i.e., the call graph computed from the most precise pointer analysis solution). Our notion of a "most precise solution" is defined with respect to a specific category of pointer analyses. More precisely, we defined a formal model that describes a wide variety of existing pointer analyses (flow-sensitive/flow-insensitive, context-sensitive/context-insensitive). In essence, this model characterizes the vast majority of the standard commonly-used pointer analysis technology. Inside this model, we define the notion of a most precise pointer solution. Any analysis that fits in this model computes a solution that is either the same as, or an over-approximation of the most precise solution. To simplify the presentation, we omit the formal description of our model (the details can be found in [12]), and we only describe the major points of the model and the corresponding most precise solution.

We consider analyses that represent the program using an *interprocedural control flow graph (ICFG)* $G = (N, E, n_0)$ which contains control flow graphs for all procedures in the program. The set of nodes N represents all program statements and the set of edges E represents the flow of control between these statements. 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 contain edges (call,entry) or (exit,return); such edges are discovered during the analysis.

The model uses *rules*, where each rule represents the meaning of an individual program statement. Each rule takes as input a points-to graph (i.e., a set of points-to edges (x, y), representing the fact that memory location x contains the address of memory location y). The rule produces a new points-to graph by adding new points-to edges and removing "killed" points-to edges. In addition, the rule may add newly discovered edges (call,entry) and (exit,return) at indirect calls.

A *realizable path* in the ICFG is a path on which every procedure returns to the call site that invoked it [16, 10, 13]; only such paths represent potential sequences of execution steps. Note that some of these paths are not apparent in the initial ICFG (because of indirect calls), and are only discovered during the analysis.

For each realizable path $p = (n_0, \ldots, n_i)$, let G_p be the points-to graph which results from applying the rules for all nodes in the path in the order specified by the path. 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 *most precise solution* (MPS) at n is defined as

$$MPS(n) = \bigcup_{p \in RP(n_0,n)} G_p$$

Intuitively, with respect to the category of pointer analyses that fit the above model, the most precise solution $MPS(n)$ represents the best possible approximation of the points-to relationships that are created at run time on possible execution paths to node n. Thus, $MPS(n)$ is at least as precise as the solutions computed by the majority of existing pointer analyses (including analyses that, at least theoretically, are significantly more precise than the FA analysis described earlier).

For each of our data programs, we considered all $n \in N$ that represent indirect calls. For each such n, we examined the program source code and we manually computed $MPS(n)$. The resulting "best possible call graph" is the most precise call graph obtainable with the standard, widely used pointer analysis technology.

Name	Description	LOC	Indirect Calls
`diction 0.8`	GNU diction command	2652	3
`gdbm 1.8.0`	GNU database routines	5577	1
`072.sc 6.1`	Spreadsheet program	9192	2
`find 4.1`	GNU find command	15200	22
`minicom 1.83.0`	UNIX communication program	15607	6
`m4 1.4`	GNU macro processor	16375	17
`less 3.40`	GNU less command	20397	4
`unzip 5.40`	Extraction utility	26273	307

Table 1. Description of data programs.

5 Empirical Results

We performed experiments on a set of eight realistic C programs, ranging in size from 2652 to 26273 lines of code. The description of the data programs 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.

For each data program, we compared the FA-based call graph with the best possible call graph. Our comparison showed *no differences* between the two graphs. 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; as discussed in Section 2, we observed several stylistic patterns of function pointer usage.

The case of function dispatch from a dispatch table based on a string is the most frequently occurring pattern of function pointer usage (recall the example in Figure 1). The string that is used to select the function from the table is either (i) evaluated at run time, (ii) determined based on a command line argument or option, or (iii) determined based on interactive user input. Therefore, even using the most precise pointer solution, we cannot do better but conclude that all functions in the table can be potentially selected.

We encountered several libraries that used structure fields to store function pointers. Although the libraries provide functionality for changing the default functions pointed to by the function pointers, this functionality is not used by the library clients—that is, the function pointer fields are initialized once and are not modified later in the code. As a result, the FA analysis is able to conclude that the points-to set associated with each function pointer field is a singleton, and therefore the achieved precision is the same as in the most precise solution. For this pattern of usage, it is crucial that the FA analysis is able to distinguish between structure fields. To illustrate this point, recall the example in Figure 2. Steensgaard's points-to analysis [17], which is a popular analysis similar to the FA analysis, does not distinguish between structure fields within the same structure. Therefore this analysis will erroneously infer that the possible targets at indirect calls through `h.chunkfun` are `xmalloc` and `xfree`. The same imprecision occurs at indirect calls through `h.freefun`. The imprecision is due to the fact that the points-to sets of `chunkfun` and `freefun` are merged because the analysis does not distinguish the fields in structure `obstack`.

Finally, consider the following example which summarizes another frequently used pattern:

```
void f(void (*fp)()) {...(*fp)()...}
```

Suppose that there is a path from the entry node of the program to a given call site for `f`, and there is a path from the entry node of `f` to the indirect call site `(*fp)()` (otherwise the indirect call would be dead code). For all such cases in our benchmarks, the function address is taken at the actual call site—that is, the call has the form `f(&g)`. Clearly, in the most precise pointer solution, the points-to set of `fp` contains all functions `g` whose addresses are used as actuals at calls to `f`. The FA analysis adds these functions to the FA equivalence class of `*fp`, and for our benchmarks these are the only elements of that equivalence class.

The results from this experiment indicate that inexpensive pointer analyses such as the FA analysis may provide sufficient precision for the purposes of call graph construction. In this context, the use of more expensive pointer analyses may not be necessary.

6 Related Work

There is a large body of work on various pointer analyses for C with different degrees of cost and precision [10, 9, 5, 1, 18, 17, 20, 15, 11, 7, 4, 6, 3, 14, 8]. Traditionally, the precision of these analyses has been evaluated with respect to the disambiguation of indirect memory reads and writes (e.g., in `*p=1`). Our work evaluates the precision of pointer analysis with respect to indirect procedure calls and call graph construction.

Existing work makes only *relative* evaluation of analysis precision (i.e., how does the solution computed by analysis X differ from the solution computed by analysis Y). Our work evaluates the *absolute* precision of the FA analysis, by comparing it with the best precision that could be achieved with the standard pointer analysis technology.

Previous work by Antoniol et al. [2] provides a comprehensive study of the impact of function pointers on the call graphs of C programs. Similarly to our work, this study uses pointer analysis to determine the possible targets of indirect calls. The goal of the work in [2] is to evaluate the impact of function pointers on the call graphs of C programs. The conclusion of this study is that indirect calls deeply affect the structure of the call graph, and therefore pointer analysis should be employed to take into account such calls. The goal of our work is to evaluate the precision of the FA analysis in the context of this problem. Our results indicate that precise call graph construction is possible even with inexpensive pointer analyses.

7 Conclusions and Future Work

We present an empirical evaluation of the precision of the FA pointer analysis [20, 19] with respect to disambiguation of indirect calls in eight realistic C programs. Our results show that in the context of this problem, a relatively imprecise and inexpensive pointer analysis such as the FA analysis is capable of achieving the best precision obtainable with the standard pointer analysis technology. These findings indicate that pointer analyses at the lower end of the cost/precision spectrum may be sufficiently precise for the purposes of call graph construction in the presence of function pointers.

In our future work we would like to reconfirm the results from this study for more programs and for programs that are larger than the ones in our current data set. To compute the most precise pointer solution for larger programs, we are considering techniques for filtering out parts of the program that are irrelevant with respect to the generation and propagation of function pointer values. Finally, we would like to investigate approaches for adapting the FA analysis for incomplete programs (e.g., libraries), and to evaluate empirically the precision of the resulting call graphs.

References

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

[2] G. Antoniol, F. Calzolari, and P. Tonella. Impact of function pointers on the call graph. In *European Conference on Software Maintenance and Reengineering*, pages 51–59, 1999.

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

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

[5] 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.

[6] 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.

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

[8] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA. In *Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

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

[10] 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.

[11] 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.

[12] A. Milanova, A. Rountev, and B.G. Ryder. Precise call graph construction in the presence of function pointers. Technical Report DCS-TR-442, Rutgers University, 2001.

[13] 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.

[14] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Conference on Programming Language Design and Implementation*, pages 47–56, 2000.

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

[16] 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.

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

[18] 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.

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

[20] 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.