Precise Call Graphs for C Programs with Function Pointers

Ana Milanova (`milanova@cs.rpi.edu`)
Department of Computer Science, Rensselaer Polytechnic Institute

Atanas Rountev (`rountev@cis.ohio-state.edu`)
Department of Computer and Information Science, The Ohio State University

Barbara G. Ryder (`ryder@cs.rutgers.edu`)
Division of Computer and Information Sciences, Rutgers University

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 one specific pointer analysis (the FA pointer analysis by Zhang et al.) for the purposes of call graph construction for C programs with function pointers. The analysis is incorporated in a production-strength code-browsing tool from Siemens Corporate Research in which the program call graph is used as a primary tool for code understanding.

The FA pointer analysis uses an inexpensive, almost-linear, flow- and context-insensitive algorithm. To measure analysis precision, we compare the call graph constructed by this analysis with the most precise call graph obtainable by a large category of existing pointer analyses. Surprisingly, for all our data programs the FA analysis achieves the best possible precision. This result indicates that for the purposes of call graph construction, inexpensive pointer analyses may provide precision comparable to the precision of expensive pointer analyses.

Keywords: call graph, function pointers, pointer analysis

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);
```

At line 1 we need to know those variables to which `p` may point in order to determine which variables may be modified by the statement. This information is needed by a variety of applications: for example, if 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.

The *program call graph* is a popular representation of the calling relationships between program procedures: an edge (P_1, P_2) shows that procedure P_1 may call procedure P_2. This information is essential for program comprehension, and can be provided by a variety of software productivity tools. However, such tools face a problem when the program contains indirect calls through function pointers. In this case, some form of *pointer analysis* may be necessary to disambiguate indirect calls. The goal of our research is to investigate such use of pointer analysis in the context of an industrial code-browsing tool.

Precise information about memory accesses and procedure calls is fundamental for static analyses used in software engineering tools and optimizing compilers. Pointer analysis determines the set of memory locations that a given memory location may point to (e.g., the analysis can determine which are the locations that `p` may point to at line 1). In addition, pointer analysis determines which function addresses may be stored in a given function pointer. Because of the importance of such information, a variety of pointer analyses have been developed [11, 12, 9, 5, 1, 22, 20, 25, 18, 13, 7, 4, 6, 3]. These analyses provide different tradeoffs between cost and precision. For example, flow- and context-insensitive pointer analyses [1, 20, 25, 18, 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 traditionally measured with respect to the disambiguation of indirect memory accesses (e.g., the locations that `p` points to at 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. [25, 24] (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. This analysis belongs at the low end of the pointer analysis spectrum with respect to cost and precision.

The FA analysis was implemented in the context of an industrial 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. One of the primary browser features is the display and navigation of call graphs. We worked on extending the tool functionality to extract and display semantic information obtained through static analysis. As part of this work, 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 calls could not be handled—that is, the call graphs did not contain edges representing such calls. 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 "fully precise" call graph. In Section 5 we discuss in detail our definition of "fully precise", but intuitively, this is 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 work 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.

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 8 publicly available realistic C programs, we show that a relatively imprecise and inexpensive pointer analysis produces the fully precise call graph. Therefore, for the purposes of call graph construction in the presence of function pointers, the use of more expensive pointer analyses may be unnecessary.

Outline. The rest of this paper is organized as follows. Section 2 discusses the use of function pointers in C programs. Section 3 provides background on pointer analysis and describes dimensions of analysis cost and precision. Section 4 presents the FA pointer analysis. The notion of fully precise call graph is discussed in Section 5. Section 6 describes our empirical results and the conclusions from these results. Related work is discussed in Section 7. Appendix A provides additional details about our definition of the fully precise call graph.

```
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) {
1   for (i=0; i<num_func; i++)
2     if (strcmp(table[i].name,s)==0)
3       return table[i].func;
4   return NULL; }
int main(int argc,char *argv[]) {
    ...
5   PFB parse_func=find_p_func(argv[1]);
6   if (parse_func)
7     (*parse_func)();
8   else { ... } }
```

Figure 1. Table dispatch.

2. Function Pointers in C Programs

The program call graph provides essential information for software understanding, restructuring, and testing. In the absence of indirect calls, this graph can be easily constructed from the program source code. However, *function pointers* in C enable indirect calls. In real-world C programs, function pointers are often employed as a powerful mechanism for creating compact and flexible code and for extending, customizing, and reusing existing functionality. This section discusses several patterns of function pointer usage in C programs and presents detailed examples that illustrate these typical uses. These examples are representative of the stylistic patterns we encountered in our benchmarks.

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 for the indirect call 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 a widely used idiom in C programs. At run time the functions are often dispatched from the table

```
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. Extensible functionality.

based on user input (e.g., command line option, command line argument, or spreadsheet function). This pattern produces compact code that is easier to understand and maintain, and therefore occurs often in C programs.

2.2. Extensible and Customizable Functionality

Figure 2 shows a fragment from a memory management library. Functions `chunk_fun` and `free_fun` allow library clients to choose the memory allocation and deallocation procedures associated with each `obstack`. Clients could either use default procedures, or could provide other procedures.

The examination of our benchmark programs revealed libraries that define global data structures with function pointer fields. Initially, these fields point to functions that provide default functionality, but clients can redefine this functionality by redirecting the fields to client functions. In this case, function pointers provide a flexible mechanism for extending and customizing reusable code.

2.3. Polymorphic Behavior

In some cases formal parameters are declared as function pointers to allow a 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 the sentence. If `sentence` is invoked from a word counting routine, the processing routine `process` will be counting the words in the sentence. If `sentence` is invoked from a spell checking routine, `process` will be checking for spelling mistakes. This pattern of usage facilitates the creation of reusable code, and is common in many C programs.

3. Pointer Analysis

Since function pointers are commonly used in C programs, software productivity tools should be able to take them into account when constructing the program call graph. To achieve this goal, such tools need to employ *pointer analysis* to identify the functions that could be invoked at indirect calls. Pointer analysis is a popular kind of static analysis that determines which memory locations may be pointed to by a given memory location. Thus, pointer analysis can determine the functions that may be pointed to by a variable `fp`, which allows the disambiguation of indirect calls of the form `(*fp)()`.

In general, the problem addressed by pointer analysis is undecidable [10]. This has led to the development of a wide variety of approximate analyses. All such analyses are conservative (i.e., they are guaranteed to report all possible pointer relationships that could actually occur at run time). Analysis A is more precise than analysis B if the solution computed by A is guaranteed to be a subset of the solution produced by B. More precise analyses are typically more expensive: existing analyses range in complexity from doubly exponential to almost linear. This section describes several important dimensions of the cost/precision spectrum for pointer analysis. A detailed discussion of pointer analysis, dimensions of analysis precision, and an extensive bibliography of existing work appears in [8].

Flow-sensitive vs. flow-insensitive. *Flow-sensitive* analyses take into account the flow of control between program points inside a procedure, and compute separate solutions for these points. These analyses consider the sequence order of program statements. *Flow-insensitive* analyses ignore the flow of control between program points, and therefore, statement execution order. Thus, flow-insensitive analyses are typically less precise and less expensive than flow-sensitive analyses.

Context-sensitive vs. context-insensitive. *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.

Consider the example in Figure 3. A context-insensitive analysis does not separate the different contexts of invocation of function `id`; as a result, the analysis erroneously determines that variable `a` in function `main` points to

```
int *id(int *x) {
  return x;
}

int main() {
  int i,j;
  int *a,*b;
  a = id(&i);
  b = id(&j);
  *a = 0;
}
```

Figure 3. Sample program.

both `i` and `j`. In contrast, a context-sensitive analysis distinguishes between the first and the second invocation of `id`, and correctly determines that `a` points only to variable `i`, whose value is set to zero by the last assignment statement in `main`.

Field-sensitive vs. field-insensitive. Field-sensitive analyses distinguish between different fields of a structure whereas field-insensitive analyses collapse all fields into a single object. For example, for a structure `s` with two pointer fields `f` and `g`, a field-insensitive analysis always reports that `s.f` and `s.g` point to the same set of memory locations, while a field-sensitive analysis maintains separate information for the two fields.

Directional vs. symmetric. This dimension of precision is specific to flow-insensitive analyses. Directional analyses (also referred to as subset-based analyses) treat assignments as unidirectional flow of values while symmetric analyses (referred to as unification-based analyses) treat assignments as bidirectional. For example, consider assignment `x=y` and suppose that `x` may point to `a` and `y` may point to `b`. Given this information, a directional analysis determines that `x` may point to `b`. However, a symmetric analysis infers not only that `x` may point to `b`, but also that `y` may point to `a`.

Analyses that are flow-insensitive, context-insensitive, and symmetric are particularly interesting for analyzing large programs. Because of these properties, analysis implementations can be very efficient: with the appropriate use of UNION-FIND algorithms, the time complexity of such analysis is $O(n\alpha(n,n))$ (i.e., almost linear).

4. FA Pointer Analysis

The FA analysis [25, 24] (**F**low-insensitive **A**lias analysis) is a pointer analysis for C that is flow-insensitive, context-insensitive, field-sensitive, and symmetric. The analysis identifies pairs of *aliases* (i.e., multiple names for the same memory location). For example, after the statement `p=&x`, `*p` and `x` are aliases because they denote the same memory location.

The FA analysis is based on a fast UNION-FIND algorithm whose complexity is almost linear in the size of the program and the size of the produced call graph. Thus, the analysis belongs at the low end of the pointer analysis spectrum with respect to cost and precision. The FA analysis is similar to a popular unification-based pointer analysis by Steensgaard [20]. The most important difference between the two analyses is that the FA analysis is field-sensitive. For example, for a structure `s` with two pointer fields `f` and `g`, Steensgaard's analysis associates a single alias set with `*(s.f)` and `*(s.g)`, while the FA analysis computes distinct alias sets for `*(s.f)` and `*(s.g)`.

Almost all existing pointer analyses are more precise than the FA analysis. For example, one popular pointer analysis is Andersen's analysis [1] which is flow-insensitive, context-insensitive, and directional. Thus, if `x` may point to `a` and `y` may point to `b` after processing statement `x=y`, the FA analysis concludes that `x` and `y` may point to both `a` and `b`, whereas Andersen's analysis concludes that `x` may point to `b` and `a`, and `y` may point to `b`.

The FA analysis handles arrays and pointer arithmetic similarly to other commonly-used pointer analyses. The analysis does not distinguish between different array elements, effectively treating an array as one large variable. Similarly, it ignores pointer arithmetic and treats `*(p+i)` as `*p`.

Since the analysis is field-sensitive, it keeps track of the possible aliases of structure fields. In the presence of casting, it is necessary to identify potentially aliased fields that are declared in different structure types. To do this, the analysis matches up the longest common subsequences of field types; this approach is similar to the technique presented in [23]. The treatment of union types is done in a similar manner. The rules for handling casting and unions are described in detail in [24].

In this section we summarize the key features of the FA analysis; more details are available in [25, 24]. The analysis first computes an equivalence relation, referred to as the *PE equivalence relation* (**P**ointer-related **E**quality). Based on this relation, it is straightforward to identify pairs of potential aliases in the program. During the construction of the PE relation, the analysis resolves indirect calls through function pointers; this information can later be used to construct the program call graph.

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

Figure 4. Sample set of statements.

4.1. PE Equivalence Relation

Memory locations and addresses of memory locations are referred to as *object names*. An object name starts with a variable or a heap name followed by a sequence of applications of left-associative structure field accesses (.field), or right-associative pointer dereferences (*). Auxiliary heap names are created explicitly for dynamically allocated memory (e.g., through calls to `malloc`) and are distinguished by their allocation site. Object names can also be derived by applying the address operator (&) to certain kinds of object names. An object name o is a prefix of an object name o_1 if o_1 is derived from o by applying a field access (.field) or a dereference (*).

The set of object names for the sample statements from Figure 4 is

```
p,&x,x,p->f,*p,&z,z,t
```

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, this 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. For structure types this means that their corresponding fields may have the same values. The PE relation is an equivalence relation (i.e., it is reflexive, symmetric, and transitive), and therefore defines a partitioning 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. Graph nodes correspond to equivalence classes of object names; graph edges are either *dereference edges* labeled with *, or *field edges* labeled with a field identifier. The algorithm for constructing G_{PE} (shown in Figure 5) is defined in terms of several functions:

- *init_equiv_class*(o) initializes a singleton equivalence class which contains object name o.
- *find*(o) returns the equivalence class containing object name o.
- *union*(e_1, e_2) merges equivalence classes e_1 and e_2 and returns the resulting equivalence class.

```
calculate-PE-relation()
1  for each o do
2      init_equiv_class(o)
3      prefix[find(o)] = {}
4  for each o do
5      if o == &o1
6          add (*,o1) to prefix[find(o)]
7      else if o == apply(o1,*)
8          add (*,o) to prefix[find(o1)]
9      else if o == apply(o1,field)
10         add (field,o) to prefix[find(o1)]
11 for each x = y in the program
12     if (find(x) ≠ find(y))
13         MERGE(find(x),find(y))

MERGE(e1,e2)
14 e = union(e1,e2)
15 new_prefix = prefix[e1]
16 for each (a,o) ∈ prefix[e2]
17     if exists (a,o1) ∈ new_prefix
18         if (find(o) ≠ find(o1))
19             MERGE(find(o),find(o1))
20     else
21         new_prefix = new_prefix ∪ {(a,o)}
22 prefix[e] = new_prefix
```

Figure 5. Algorithm for constructing G_{PE}.

- $apply(o,a)$ returns a new object name after applying to name o an accessor a that is a field access or a pointer dereference. For example, $apply(p,*)$ returns $*p$.

Each equivalence class e has a *prefix set* associated with it; intuitively, this set represents the outgoing edges from e in G_{PE}. Set $prefix[e]$ contains a pair (a,o), for some $o \in e'$, if and only if there exists an object name $o_1 \in e$ such that $apply(o_1,a) \in e'$—in other words, $(a,o) \in prefix[e]$ encodes an edge from e to e' labeled with a.

The algorithm for constructing G_{PE} [25] is presented in Figure 5. The first part of the algorithm (lines 1-10) creates the initial G_{PE} graph. A singleton equivalence class is added to the graph for each object name (lines 1-3). Lines 4 through 10 construct the prefix sets for the equivalence classes (i.e., the edges in G_{PE}). For the set of statements in Figure 4, the initial graph contains the following edges:

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

After the initialization phase, the analysis processes all program statements and merges nodes corresponding to expressions that may have the same value (lines 11-13).[1] For example, for each assignment statement, the analysis merges the nodes corresponding to the equivalence classes that contain the object names for the left-hand side and the right-hand side of the assignment. Similarly, for each call site, the equivalence classes of an actual parameter and its corresponding formal are merged. To take into account all possible formal-actual pairs, the analysis resolves indirect calls: for a call through *fp*, each function *func* that is added to the equivalence class containing $*fp$ is considered a potential target of the call. Subsequently, the analysis performs the appropriate merges of equivalence classes of actual arguments at the call through *fp* and the formal parameters of *func*, as well as the return variable of *func* and the left-hand-side of the indirect call.[2] Thus, in addition to G_{PE}, the analysis implicitly constructs the program call graph.

Whenever two merged classes have outgoing edges with the same label, the target nodes are merged recursively. The pseudocode for the recursive merge appears in lines 15 through 21 in Figure 5. For example, in Figure 4, statement `p=&x` results in merging of nodes `p` and `&x`. Nodes `*p` and `x` are also merged 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 `t` and `p` are merged due to the last statement.

The nodes in the final G_{PE} define the PE equivalence relation. Names that are in the same equivalence class correspond to expressions that may have the same value at run time. For the example in Figure 4, the graph contains the following equivalence classes:

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

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

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

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

[1] For simplicity, we present the analysis as if *all* program statements are handled. Analysis implementations need to perform appropriate merges only for statements that can be actually reached from the entry point to the program.

[2] If class e_1 contains object names $*fp_1, \ldots, *fp_n$ that appear at indirect calls, and if e_1 is merged with a class e_2 that contains functions $func_1, \ldots, func_m$, the analysis iterates over each call edge $(*fp_i, func_j)$ and performs the appropriate formal-actual merges. Note that each call edge is processed at most once, because each object name appears in exactly one equivalence class. Thus, the work performed by the analysis is proportional to the size of the call graph, retaining the almost linear complexity.

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

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):

{`table[].func, &func1, &func2`}

{`func1, func2`}

After line 3 in Figure 1 the equivalence classes are

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

{`func1, func2`}

where `ret_find_p_func` is an auxiliary variable that contains the return values of `find_p_func`. At line 5 in Figure 1 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:

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

{`func1, func2, *parse_func`}

From the second class, the analysis infers that the possible targets of the call through `*parse_func` at line 7 are `func1` and `func2`. The final G_{PE} graph contains additional edges (e.g., related to `table[].name`, `argv[]`, `s`, etc.); for brevity, these edges are not shown.

5. Fully Precise Pointer Analysis

After implementing the FA analysis in the code-browsing tool by Siemens Research, our goal was to evaluate the precision of the produced call graphs. To achieve this, we defined a conceptual "fully precise" pointer analysis $\mathcal{P}$. This analysis is a fully flow-sensitive, context-sensitive, field-sensitive pointer analysis that represents a point at the very high end of the design space for pointer analysis. A large number of pointer analyses (or their minor variations) [11, 9, 5, 1, 22, 20, 25, 18, 13, 7, 4, 6, 3] can be considered approximate versions of $\mathcal{P}$—that is, the solution computed by $\mathcal{P}$ is a subset of the solutions computed by these analyses. $\mathcal{P}$ was specifically designed to achieve this high

```
int *id(int *x) {
    return x;
}
int main() {
    int i,j;
    int *a,*b;
1   a = id(&i);
2   b = id(&j);
3   *a = 0;
}
```

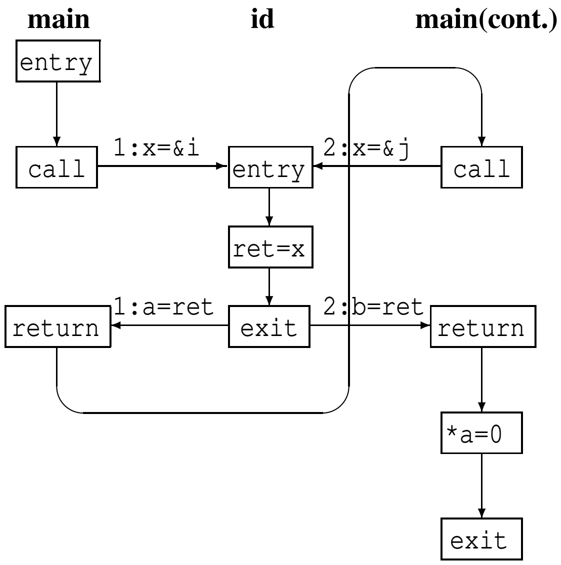

Figure 6. Sample program and its interprocedural control flow graph.

precision. For example, the analysis maintains full information about calling context; for existing analyses, this approach is too expensive (or even impossible in the presence of recursion), and they employ some form of calling context approximation.

The definition of $\mathcal{P}$ provides a theoretical model of the best possible precision that is obtainable with the standard commonly-used pointer analysis technology. Since the FA analysis always produces a call graph that is a superset of $\mathcal{P}$'s call graph, we wanted to determine the difference between the two graphs in order to evaluate the imprecision of the FA analysis and to understand the sources of this imprecision. In this section we present the key features of $\mathcal{P}$; additional details are available in Appendix A.

5.1. Interprocedural Control Flow Graph

Analysis $\mathcal{P}$ is based on an *interprocedural control flow graph (ICFG)* $G = (N, E, n_0)$. The sample program from Figure 3 and its ICFG are shown in Figure 6. G is a directed graph with nodes $n \in N$ representing program statements, edges $e \in E$ representing flow of control between statements, and starting node $n_0 \in N$ representing the entry point of the program.

Each procedure has associated a single *entry node* and a single *exit node*; node n_0 is the entry node of the starting procedure. 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*, the ICFG does not contain edges (call,entry) or (exit,return); such edges are discovered during the analysis.

5.2. Points-to Graphs

Relationships that involve pointer values are represented in $\mathcal{P}$ by *points-to graphs* in which nodes correspond to memory locations and edges represent "points-to" relationships.

Let V be the set of all local variables (including formals), global variables, and heap variables.[3] In a points-to graph, each local variable and heap variable from V can be represented multiple times for different calling contexts of the corresponding enclosing procedure.

To achieve full context sensitivity, analysis $\mathcal{P}$ defines a set of contexts Γ. Each context is a sequence of call sites that represents one possible chain of procedure invocations starting from `main`. For example, if `main` contains a call site s_1 that invokes procedure `p1`, and if `p1` contains a call site s_2 that invokes `p2`, context $(s_1, s_2) \in \Gamma$ represents invocations of `p2` from s_2 when `p1` had been invoked from s_1. In essence, a context $\gamma \in \Gamma$ is an encoding of one possible state of the run-time call stack. For convenience, we define an empty context $\varepsilon \in \Gamma$ representing an empty sequence of call sites.

Points-to graph nodes are pairs $(v, \gamma) \in V \times \Gamma$ representing context copies of variables; we will use v^γ to denote node (v, γ). For a local or a heap variable v, these pairs present different run-time instances of the variable for different invocation contexts γ of the enclosing procedure. A single node v^ε is used to represent a global variable, or a local/heap variable in `main`. A points-to edge $(v_1^{\gamma_1}, v_2^{\gamma_2})$ shows that a memory location represented by the first node may contain the address of a memory location represented by the second node.

Example. For the program in Figure 6, the set of contexts is $\Gamma = \{(1), (2)\}$ and the nodes are i^ε, j^ε, a^ε, b^ε, $x^{(1)}$, $x^{(2)}$, $ret^{(1)}$, and $ret^{(2)}$. Variable *ret* is an auxiliary name created to represent the return value of `id`. As described below, $\mathcal{P}$ computes various points-to graphs for different paths in the ICFG. For example, the points-to graph for path `(main:entry), (1:call), (id:entry), (id:ret=x)` contains edges $(x^{(1)}, i^\varepsilon)$ and $(ret^{(1)}, i^\varepsilon)$.

5.3. Analysis Solution

For each path $p = (n_0, \ldots, n)$ in the ICFG, the analysis computes a points-to graph that represents all points-to relationships that exist at n when the flow of control follows p. To compute this points-to graph, the analysis associates

[3] A heap variable is an auxiliary name representing memory locations allocated on the heap; each allocation site (e.g., call to `malloc`) is represented by a unique heap variable.

a *transfer function* f_{n_i} with each ICFG node n_i. The input of f_{n_i} is a points-to graph *PtG* and a context $\gamma \in \Gamma$. *PtG* represents the points-to relationships that exist immediately before the execution of n_i; context γ represents an invocation context for the procedure containing n_i. The output of f_{n_i} is a new points-to graph PtG' and a new context γ'. PtG' represents the points-to relationships immediately after the execution of n_i, and γ' is an updated calling context. Further discussion of the transfer functions is presented in Appendix A; for brevity, here we omit these details.

To compute the points-to graph corresponding to a path $p = (n_0, n_1, \ldots n_k, n)$, $\mathcal{P}$ applies the transfer functions for all path nodes (except for the last node) in the order defined by the path. Thus, the solution computed at the top of the last node of p is $f_{n_k}(\ldots f_{n_1}(f_{n_0}(\emptyset, \varepsilon))\ldots)$. The final solution for a node n (which represents the points-to relationships *before* node n) is the union of the points-to graphs corresponding to all *realizable paths* from n_0 to n. A realizable path is a path on which every procedure returns to the call site that invoked it [19, 11, 16]; only such paths represent potential sequences of execution steps. For example, path `(main:entry)`, `(1:call)`, `(id:entry)`, `(id:ret=x)`, `(id:exit)`, `(2:return)` is not realizable because procedure `id` does not return to the call site that invoked it.

When $\mathcal{P}$ determines all realizable paths from n_0 to n, it has to take into account paths that involve indirect calls. The initial ICFG only represents direct calls; thus, during the analysis, the current points-to graph is used to infer additional (call,entry) and (exit,return) edges. If node n_i contains an indirect call through pointer *fp*, a path $(n_0, \ldots, n_i)$ is extended with the entry node of any procedure pointed to by the corresponding fp^γ.

6. Empirical Results

Our experiments were performed on a set of 8 realistic C programs, ranging in size from 2652 to 26273 lines of code. The description of the dataset is given in Table I. Each program employs function pointers; the number of indirect calls in the program is shown in the last column of Table I.

In our evaluation we considered every indirect call $x = (*fp)(\ldots)$ in each program, and we determined all functions *func* that were in the PE equivalence class for $*fp$ as computed by the FA analysis. For each such *func*, we manually examined the program source code and attempted to identify a realizable ICFG path (from the entry of `main` to the indirect call) such that the appropriate edge $(fp^\gamma, func)$ was present in the points-to graph computed by $\mathcal{P}$ for that path. In all cases we successfully identified at least one such path. Thus, for all of our data programs, the call graph constructed by the FA analysis was a *subset* of the call graph computed by $\mathcal{P}$; of course, this implies that the two graphs were identical.

Table I. Program description.

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

Our precision evaluation showed that for all data programs the FA analysis constructed the best possible call graph obtainable with the standard pointer analysis technology. This 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.

6.1. Table Dispatch

The case of function dispatch from dispatch tables based on a string is one of 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 $\mathcal{P}$ cannot do better but conclude that all functions in the table can be potentially selected.

This pattern of usage occurs in several of our benchmarks. For example, `find` uses a dispatch table to select a parsing function based on the value of a command line argument (the example in Figure 1 was motivated by `find`). In `less`, structure instances that represent possible command line options are stored in a table, and are selected based on user-provided options. Each instance has associated a handling function accessed through a function pointer field. In a similar manner, `minicom` uses a dispatch table to handle the selection of communication functions.

6.2. Extensible Functionality

We encountered several libraries that used structure fields that 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. Typically, a function pointer field is initialized once and is 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. For this pattern of usage, it is crucial that the FA analysis is field-sensitive (i.e., it is capable of distinguishing between structure fields). To illustrate this point, recall the set of statements in Figure 2. Steensgaard's pointer analysis [20], which is the most popular inexpensive pointer analysis, does not distinguish between structure fields; therefore this analysis will erroneously infer that 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 sets of targets for `chunkfun` and `freefun` are merged by the analysis.

We observed this pattern in several of our benchmarks (e.g., `m4`, `unzip`). For example, `m4` uses an efficient memory management library, and does not override the default memory allocation and deallocation functionality provided by that library.

6.3. POLYMORPHIC BEHAVIOR

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 call site that invokes `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 our benchmarks, in all cases of occurrence of this pattern, the following three conditions are true: (i) the function address is passed directly at the call to `f` (i.e., the call site is of the form `f(&g)`), (ii) the function address is taken only at calls to `f` (i.e., all occurrences of `&g` are of the form `f(&g)`), and (iii) the function pointer formal parameter of `f` is never accessed in `f` except at the indirect calls. Clearly, $\mathcal{P}$ determines that the possible targets of the indirect call are all functions whose addresses are used as parameters at calls to `f`. The FA analysis groups these functions in the PE equivalence class of `*fp`. Because of the conditions specified above, it is easy to see that the equivalence class of formal parameter `*fp` is guaranteed to contain *only* functions whose addresses are taken at calls to `f` (e.g., `g`); thus, the solution for the indirect call is the same as the solution computed by $\mathcal{P}$.

For example, one of our benchmark programs (`072.sc`) uses two polymorphic functions that evaluate a mathematical function whose address is passed as an argument (e.g., *cosine*, *tangent*, etc.). One function handles functions that require two arguments such as *power*, and the other handles functions with a single argument such as *cosine*. In this case, the functions whose addresses are taken at distinct call sites are grouped by the FA analysis into two disjoint sets—the set of two-argument functions and the set of single-argument functions. In `diction`, a polymorphic routine parses each input sentence and then invokes on it a processing function whose address is provided as a parameter. Depending on the desired functionality, the processing function either identifies commonly misused phrases (for the *diction* command), or analyses sentence length and various readability measures (for the *style* command).

7. Related Work

There is a large body of work on pointer analyses for C with varying degrees of precision and cost [11, 9, 5, 1, 22, 20, 25, 18, 13, 7, 4, 6, 3, 17]. Analysis precision is typically evaluated with respect to the disambiguation of indirect memory accesses (e.g. in `*p=1`). The most frequently used metric is the average size of a points-to set [20, 18, 13, 7, 4, 6]. Our work evaluates pointer analysis precision with respect to the disambiguation of indirect calls.

Existing work evaluates only the *relative precision* of different analyses—that is, it compares the solution computed by analysis X with the solution computed by analysis Y. Our work determines the *absolute precision* of the FA analysis, by comparing it with the fully precise analysis $\mathcal{P}$. We believe that such evaluations of absolute precision are more useful and informative.

Work by Murphy et al. [15] studies several commercial tools for call graph extraction for C programs. This work focuses on the complex design and engineering aspects of tool development, and evaluates how design decisions affect the resulting call graphs. One observation presented in the paper is that the majority of commercial call graph extraction tools do not resolve calls through function pointers. Because indirect calls have significant impact on the call graphs of C programs, it is important to study techniques for the disambiguation of indirect calls. Our work suggests that an inexpensive analysis can resolve calls through function pointers precisely, and therefore it can be successfully applied in production-strength software tools.

Work by Antoniol et al. [2] provides a comprehensive study of the impact of function pointers on the call graphs of C programs (e.g., what percentage of all calls are made through function pointers, and how many functions are reachable only along paths that contain indirect calls). Similarly to our work, this study uses pointer analysis to disambiguate indirect calls. The conclusion in [2] 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. Tonella et al. [21] perform reverse engineering on a large software system written in C. Their work clearly shows that function pointers are heavily used in industrial-size software systems and significantly affect the call graph structure. One of the conclusions in [21] is that the disambiguation of indirect calls by using pointer analysis is necessary in the context of software engineering tasks. The goal of our work is to evaluate the precision of the FA analysis in the context of this problem.

Recent work by Mock et al. [14] studies the usage of dynamic points-to data for the purposes of improving program slicing for C. One of the conclusions of this work is that imprecise resolution of indirect calls may significantly affect the precision of the resulting static slices. Our work shows that an inexpensive pointer analysis with certain properties (e.g., field sensitivity) is capable of constructing precise call graphs.

8. Conclusions and Future Work

We evaluated the precision of the FA pointer analysis with respect to the disambiguation of indirect calls in 8 realistic C programs. The results from our experiment indicate that inexpensive analyses such as the FA analysis may provide sufficient precision for the purposes of call graph construction for C programs with function pointers. In this context, the use of more expensive pointer analyses may result in significantly increased cost without any corresponding precision gains.

Clearly, one limitation of this study is that it is unclear whether the results will extend to other data programs. We are particularly interested in obtaining precision results for larger programs and for programs that may exhibit other patterns of function pointer usage. The major difficulty in accomplishing a study of larger programs is the need for manual examination of program code. We are currently developing techniques that can speed up this process. In addition, we are interested in performing dynamic analysis for the purposes of assessing analysis precision, and evaluating the possibility of human error.

Obtaining a more comprehensive description of common patterns of function pointer usage is another interesting area of future research. Such patterns can be used as predictors of the precision of the FA analysis and other pointer analyses. In addition, these patterns may be formalized and classified using techniques from the design patterns community.

Acknowledgements

This research was supported by NSF grant CCR-9900988 and by Siemens Corporate Research. The authors would like to thank the SCAM 2002 and JASE reviewers for their helpful comments.

References

1. Andersen, L.: 1994, 'Program Analysis and Specialization for the C Programming Language'. Ph.D. thesis, DIKU, University of Copenhagen.
2. Antoniol, G., Calzolari, F., and Tonella, P.: 1999, 'Impact of Function Pointers on the Call Graph'. In: *European Conference on Software Maintenance and Reengineering*. pp. 51–59.
3. Cheng, B., and Hwu, W.: 2000, 'Modular Interprocedural Pointer Analysis Using Access Paths'. In: *Conference on Programming Language Design and Implementation*. pp. 57–69.
4. Das, M.: 2000, 'Unification-based Pointer Analysis with Directional Assignments'. In: *Conference on Programming Language Design and Implementation*. pp. 35–46.
5. Emami, M., Ghiya, R., and Hendren, L.: 1994, 'Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers'. In: *Conference on Programming Language Design and Implementation*. pp. 242–257.
6. Fähndrich, M., Rehof, J., and Das, M.: 2000, 'Scalable Context-sensitive Flow Analysis Using Instantiation Constraints'. In: *Conference on Programming Language Design and Implementation*. pp. 253–263.
7. Foster, J., Fähndrich, M., and Aiken, A.: 2000, 'Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C'. In: *Static Analysis Symposium*. pp. 175–198.
8. Hind, M.: 2001, 'Pointer Analysis: Haven't We Solved This Problem Yet?'. In: *Workshop on Program Analysis for Software Tools and Engineering*. pp. 54–61.
9. Hind, M., Burke, M., Carini, P., and Choi, J.,: 1999, 'Interprocedural Pointer Alias Analysis'. *ACM Trans. Programming Languages and Systems* **21**(4), 848–894.
10. Landi, W.: 1992, 'Undecidability of Static Analysis'. *ACM Letters on Programming Languages and Systems* **1**(4), 323–337.
11. Landi, W., and Ryder, B. G.: 1992, 'A Safe Approximation Algorithm for Interprocedural Pointer Aliasing'. In: *Conference on Programming Language Design and Implementation*. pp. 235–248.
12. Ryder, B. G., Landi, W., Stocks, P., Zhang, S., and Altucher, R.: 2001, 'A Schema for Interprocedural Side-Effect Analysis with Pointer Aliasing'. *ACM Trans. Programming Languages and Systems* **23**(1), 105–186. An earlier version available as Rutgers Computer Science Department Technical Report DCS-TR-336.
13. Liang, D., and Harrold, M. J.: 1999, 'Efficient Points-to Analysis for Whole-program Analysis'. In: *Symposium on the Foundations of Software Engineering*. pp. 199–215.
14. Mock, M., Atkinson, D., Chambers, C., and Eggers, S.: 2002, 'Improving Program Slicing with Dynamic Points-to Data'. In: *Symposium on the Foundations of Software Engineering*. pp. 71–80.
15. Murphy, G., Notkin, D., Griswold, W., and Lan, E.: 1998, 'An Empirical Study of Static Call Graph Extractors'. *ACM Trans. on Software Engineering and Methodology* **7**(2), 158–191.
16. Reps, T., Horwitz, S., and Sagiv, M.: 1995, 'Precise Interprocedural Dataflow Analysis Via Graph Reachability'. In: *Symposium on Principles of Programming Languages*. pp. 49–61.
17. Rountev, A., and Chandra, S.: 2000, 'Off-line Variable Substitution for Scaling Points-to Analysis'. In: *Conference on Programming Language Design and Implementation*. pp. 47–56.
18. Shapiro, M., and Horwitz, S.: 1997, 'Fast and Accurate Flow-insensitive Points-to Analysis'. In: *Symposium on Principles of Programming Languages*. pp. 1–14.

19. Sharir, M., and Pnueli, A.: 1981, 'Two Approaches to Interprocedural Dataflow Analysis'. In: S. Muchnick and N. Jones (eds.): *Program Flow Analysis: Theory and Applications*. Prentice Hall, pp. 189–234.
20. Steensgaard, B.: 1996, 'Points-to Analysis in Almost Linear Time'. In: *Symposium on Principles of Programming Languages*. pp. 32–41.
21. Tonella, P., Antoniol, G., Fiutem, F., and Calzolari, F.: 2000, 'Reverse Engineering 4.7 Million Lines of Code'. *Software – Practice and Experience* **30**(2), 129–150.
22. Wilson, R., and Lam, M.: 1995, 'Efficient Context-sensitive Pointer Analysis for C Programs'. In: *Conference on Programming Language Design and Implementation*. pp. 1–12.
23. Yong, S., Horwitz, S., and Reps, T.: 1999, 'Pointer Analysis for Programs with Structures and Casting'. In: *Conference on Programming Language Design and Implementation*. pp. 91–103.
24. Zhang, S.: 1998, 'Practical Pointer Aliasing Analyses for C'. Ph.D. thesis, Rutgers University.
25. Zhang, S., Ryder, B. G., and Landi, W.: 1996, 'Program Decomposition for Pointer Aliasing: A Step towards Practical Analyses'. In: *Symposium on the Foundations of Software Engineering*. pp. 81–92.

Appendix A

As described in Section 5, the fully precise analysis $\mathcal{P}$ associates a transfer function f_n with each node n in the ICFG. The input of f_n is a points-to graph that represents points-to relationships existing immediately before n, and a calling context for the procedure containing n. The output of f_n is a new points-to graph that shows relationships immediately after n, as well as an updated calling context.

The transfer functions for some sample statements are shown in Figure 7.[4] The functions update the input points-to graph by adding new points-to edges and by removing "killed" points-to edges, taking into account the current context. Auxiliary function $killed(G,x)$ removes all points-to edges in G from memory location x. Similarly, $killed\_set(G,S)$ kills all points-to edges in G from every location in set S.

The transfer function for a (call,entry) edge updates the calling context by appending the call site label l to the current invocation string γ. In addition, the function updates the values of formal parameters f_i based on the corresponding actuals a_i. The function for (exit,return) is defined only if the label of the last (call,entry) edge in γ is the same as the label of the current (exit,return) edge; this ensures that information is propagated only along realizable paths. Function *killed_locals* removes all points-to graph nodes that represent local variables whose lifetime is terminated due to the return.

As described in Section 5, the analysis needs to take into account additional (call,entry) and (exit,return) edges that correspond to indirect calls.

[4] For simplicity, the definition of $\mathcal{P}$ in Section 5 only describes node transfer functions. However, this definition can be trivially extended to associate transfer functions with (call,entry) and (exit,return) edges.

$$f(p = \&q, \langle PtG,\gamma\rangle) =$$
$$\quad \langle (p^{\gamma}, q^{\gamma}) \cup killed(PtG, p^{\gamma}), \gamma\rangle$$

$$f(i: p = malloc(...), \langle PtG,\gamma\rangle) =$$
$$\quad \langle (p^{\gamma}, heap_i^{\gamma}) \cup killed(PtG, p^{\gamma}), \gamma\rangle$$

$$f(p = q, \langle PtG,\gamma\rangle) =$$
$$\quad \langle \{(p^{\gamma}, x) \mid (q^{\gamma}, x) \in PtG\} \cup killed(PtG, p^{\gamma}), \gamma\rangle$$

$$f(*p = q, \langle PtG,\gamma\rangle) =$$
$$\quad \langle \{(x, y) \mid (q^{\gamma}, y) \in PtG \wedge (p^{\gamma}, x) \in PtG\} \cup$$
$$\quad killed\_set(PtG, MustPoint(p^{\gamma})), \gamma\rangle$$

$$f(p = *q, \langle PtG,\gamma\rangle) =$$
$$\quad \langle \{(p^{\gamma}, y) \mid (q^{\gamma}, x) \in PtG \wedge (x, y) \in PtG\} \cup$$
$$\quad killed(PtG, p^{\gamma}), \gamma\rangle$$

$$f((call, entry)^l, \langle PtG,\gamma\rangle) =$$
$$\quad \langle PtG \cup \bigcup_{f_i} \{(f_i^{\gamma\oplus l}, x) \mid (a_i^{\gamma}, x) \in PtG\}, \gamma \oplus l\rangle$$

$$f((exit, return)^l, \langle PtG,\gamma\rangle) = \text{if } \gamma \text{ equals } \gamma' \oplus l \text{ then}$$
$$\quad \langle \{(p^{\gamma'}, x) \mid (ret^{\gamma}, x) \in PtG\} \cup$$
$$\quad killed(killed\_locals(PtG, \gamma), p^{\gamma'}), \gamma'\rangle$$

Figure 7. Sample transfer functions.

Thus, for a path $(n_0, n_1, \ldots, n_k, n)$ in which n is an indirect call through *fp*, the points-to graph *PtG* and the context γ produced by $f_{n_k}(\ldots f_{n_1}(f_{n_0}(\emptyset, \varepsilon))\ldots)$ are used to determine all possible targets of the indirect call—that is, any *func* such that $(fp^{\gamma}, func) \in PtG$.