McGill University
School of Computer Science

Interprocedural Analysis in the Presence of Function Pointers

ACAPS Technical Memo 62
Dec 28, 1992

Rakesh Ghiya

School of Computer Science, McGill University
318 McConnell Engineering Building
3480 University Street
Montreal, Quebec, CANADA
H3A 2A7
ghiya@cs.mcgill.ca

Advanced Compilers, Architectures and
Parallel Systems Group

Contents

1 Introduction and Motivation 1

2 The Algorithm 2

3 Salient Features 4

4 Related Work 10

5 Conclusions and Future Work 11

6 Acknowledgements 12

Abstract

The extant techniques for interprocedural analysis do not consider programs with function pointers, alternatively termed as function variables or procedure variables. The presence of function pointers introduces a new level of difficulty in interprocedural analysis by forcing dynamic binding of function calls. Different functions can be invoked from a function pointer call site, in different executions of the program. Thus the interprocedural structure of the program becomes execution dependent. However, complete knowledge of the interprocedural structure of the program is essential for the optimizing phase of a compiler, as all interprocedural analyses require it.

In this paper, we address the problem of obtaining the best possible safe static approximation of the interprocedural structure of the program, under these circumstances, and provide a flow-analytic solution for the same. We have considered C as the language under analysis, and implemented the algorithm in the framework of our McCAT compiler.

1 Introduction and Motivation

As the goals of program flow analysis become more ambitious, increasing emphasis is being laid on collecting sharper static information about programs. In this context, interprocedural analysis has a crucial role to play, as avoidance of flow overestimation can bring in significant improvements in the precision of information. The basic requirements for any interprocedural analysis are:

• A representation of the interprocedural structure of the program to help determine the program points, to which flow information from call sites and function exit points, needs to be propagated, while performing the analysis. It is also used to store information about the nature of the function call : recursive or non-recursive. Typical examples include : ICFG[LR92] and Invocation-Graph[Ema93].

• A mechanism to translate flow information from the context of one procedure to that of another and to ensure the flow of information along realizable interprocedural paths[LR92, Ema93].

In general, the interprocedural representation of the program can be obtained by a simple textual pass over the program. However, in the presence of function pointers, it becomes a non-trivial exercise. Function pointers are special pointer variables, which point to functions, and are dereferenced to call the pointed-to functions. Unlike a normal function call, a call through a function pointer cannot be bound to a unique function at compile time. The function invoked from such a call site, depends on the address contained in the function pointer, when program execution reaches that point. Thus different functions can be invoked from the same call site in different executions of the program. This possibility renders interprocedural structure to be a dynamic property of the program, and makes its static determination a challenging issue.

Procedure inlining is often proposed as a viable alternative to interprocedural analysis. However, even procedure inlining is not feasible to perform, if the function invokable from the call site is not exactly known at compile time. Thus presence of function pointer calls makes interprocedural analysis both hard and essential.

The simplest safe approximation would be to assume that a function pointer call can lead to the invocation of any function in the program. We can not restrict this set using the type information of the function pointer, because of type-casting being allowed in C. Even the number and type of arguments being passed cannot be used for this purpose because C permits passing of variable number of arguments to functions.

The flow information computed by using this conservative strategy would be considerably inaccurate. Every extra function considered invokable from the call site, would introduce substantial imprecision in the information for both the calling and the called function. We provide some statistics in the next section to substantiate this claim.

In general, for any flow analysis problem, there is a trade off between cost and precision. With worst case assumptions, the information obtained is very imprecise, but the cost incurred is minimum. However, in this case, we have to do more work for making safe worst case assumptions, as each function in the program needs to be analyzed in the context of the call.

The way out of this situation is to develop a flow-analytic algorithm for static approximation of the interprocedural structure of the program, which under the given circumstances varies dynamically.

In section 2 we present such an algorithm for building the interprocedural representation, and demonstrate it on example programs. In section 3 we highlight the salient features of the algorithm, and provide some statistics to demonstrate its effectiveness over the naive approach suggested above. We also discuss its theoretical complexity. Related work in this area is discussed in section 4, while in section 5 we draw our conclusions and describe the scope for further work.

2 The Algorithm

First we give a description of the invocation-graph, the interprocedural representation used by the McCAT compiler for facilitating interprocedural analysis. Figure 1 shows a program and its invocation-graph. Every node in the invocation-graph (except main) corresponds to a call site in the program. The child nodes of any node in the invocation-graph represent the functions that can be invoked from the parent function. A node can be either ordinary, recursive($\star$) or approximate(A), depending on the nature of the function call, made from the given call site. An approximate node represents, one level unwinding of recursion. For more details, see[Ema93, Sri92].

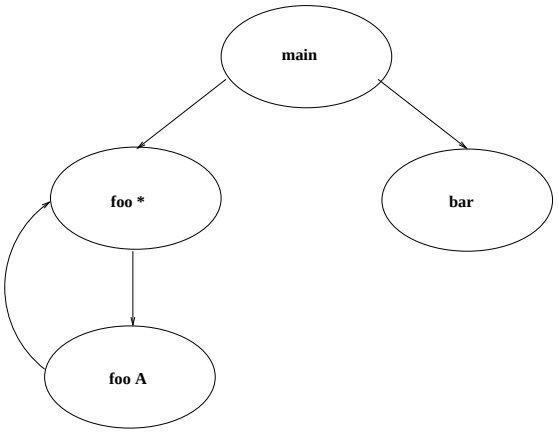
The central idea behind our algorithm is that the set of functions invokable from a function pointer call site is identical to the set of aliases of the function pointer at the program point above the call site. If we know all the functions a function pointer can be aliased to, at the entry of a call site, then the invocation-graph can be effectively constructed and different interprocedural analyses like constant propagation, reaching definitions etc. can be efficiently conducted.

However, alias analysis itself needs to be interprocedural, in order to give precise and useful information. How do we get the invocation-graph for alias analysis itself? The solution lies in building the invocation-graph while performing alias analysis, as described below.

We first build the invocation-graph, and leave it incomplete, at the points a function pointer call is encountered. Then we perform the alias analysis on this incomplete invocation-graph, and, on encountering a call through a function pointer, find all the functions it can be aliased to, according to the current alias information. The invocation-graph is accordingly updated. The alias analyser assumes all these functions to be invokable from the call site under consideration, and merges their output alias sets, to compute the alias set at the exit of the function pointer call site.

```
int a,b,c;
int *pa,*pb,*pc;

foo()
{
   .
   .
   pa = &a;
   if (cond)
      foo();

}

bar()
{
   .
   .
   pb = &b;
}

main()
{
   .
   .
   pc = &c;
   foo();
   .
   .
   bar();
   .
   .
}
```

(a) Example Program



(b) Invocation-Graph

Figure 1: An example program with its invocation-graph

A more formal description of the algorithm is as follows :

1. Build the incomplete invocation-graph : proceed normally till a function pointer call is encountered, as described in [Ema93]. At that point, stop growing the invocation-graph further.

2. Give each node representing a function pointer call, a supernode status. These are the nodes, to which new nodes will be added later on in the analysis. The supernode status will also let other analyses know, that a function pointer call has been encountered and some special processing needs to be done.

3. Begin alias analysis with the incomplete invocation-graph as the representation of the interprocedural structure of the program.

4. During alias analysis, whenever a function pointer call is encountered, say *fp()*, find all the functions fp is

may-aliased to, at the given program point, at this instant. Let these functions be f_1, f_2, ...,f_k. Let the input alias set to the statement fp() be A*in*.

5. Update the invocation-graph, by adding each function f_1, f_2, ..., f_k as a child node of the supernode fp, provided it is not already attached there. Then further grow the invocation-graph for each newly attached node, till completion or till a function pointer call is encountered.

6. Assume, each of these functions is invokable from the call site under consideration. Find the output alias set A*outj* for each potential function call f_j with input alias set as A*in*. Merge all the alias sets A*outj* ($1 \leq j \leq k$) to obtain the output alias set A*out* for the function call fp(). This is the appropriate strategy, as in an execution of the program, the call statements fp() can result in invocation of any of the functions f_1, f_2, ... f_k.

7. Resume alias analysis.

We provide two examples to demonstrate how this algorithm works. Consider the program given in Figure 2. Figure 2(a) shows the original program, and the incomplete invocation-graph obtained by a simple textual pass over the program is provided in Figure 3(a). Figure 2(b) provides the program with alias information computed till the function pointer call *fp()* is encountered at program point A. According to the current information, fp has two aliases: foo and bar. The invocation-graph is accordingly updated, as shown in Figure 3(b). Next, function *foo* is analysed with the input alias set as {(fp,foo) ,(pc,c)} and function bar with the input alias set as {(fp,bar), (pc,c)}. Note that this set is not {(fp,foo)?, (fp,bar)?, (pc,c) } for both the functions, as each function is visited and analysed with the basic assumption that fp is aliased to its name. While analysing the function foo, another function pointer call *fp()* is encountered. The invocation-graph is again updated and the call to foo in main through fp() becomes a recursive call[Ema93]. Finally, the alias information at program point B is obtained by merging the alias output sets from the two possible function calls, as shown in Figure 2(c). The final invocation-graph is shown in Figure 3(c).

Figure 4 demonstrates the algorithm for a program where a function pointer call is made inside a while loop, and the alias set of the function pointer and hence the invocation-graph is achieved through a fix-point computation. Figure 5 shows the different states of the invocation-graph as it is being updated.

3 Salient Features

The invocation-graph obtained from the above algorithm provides much more precise information than the one constructed by making conservative assumptions. We provide some statistics in Figure 6 for four small sample programs. It can be noticed, that the naive approach not only reports an overestimated number of aliases, but it also converts a substantial number of must-aliases to may-aliases, which is a substantial loss of valuable information. This happens because of validation of several unrealizable program paths, by the naive algorithm, for conducting interprocedural flow analysis.

Moreover, no extra work is required in incrementally building the invocation-graph as described in our algorithm. The only thing we do is to utilise the alias information available at a program point at the appropriate time. It is now an agreed fact that precise alias analysis is required to gain effective results from any other flow analysis for languages like C. If alias analysis is to be performed, our algorithm adds no extra cost to the analysis phase of

```
int a,b,c;
int *pa,*pb,*pc;
int (*fp)();

foo()
{
   .
   .
   pa = &a;
   if (cond)
      fp();
}

bar()
{
   .
   .
   pb = &b;
}

main()
{
   .
   .
   pc = &c;
   if (cond)
      fp = foo;
   else
      fp = bar;
   fp();
   .
   .
}
```

(a) original program

```
int a,b,c;
int (*fp)(),*pa,*pb,*pc;

foo()
{
   .
   .
   pa = &a;
   if (cond)
      fp();
}

bar()
{
   .
   .
   pb = &b;
}

main()
{
   .
   .
   pc = &c;
   if (cond)
      fp = foo;
   else
      fp = bar;
/*-- POINT A: --------*/
/*-- (fp,foo)?,
(fp,bar)?, (pc,c)-----*/
   fp();
   .
   .
}
```

(b) alias sets before resolving the call fp() at program point A

```
int a,b,c;
int (*fp)(),*pa,*pb,*pc;

foo()
{
   .
   .
   pa = &a;
   if (cond)
   /*-- POINT C: --*/
   /*-- (fp,foo), (pc,c),
       (pa,a)-------*/
       fp();
   /*-- POINT D: --*/
   /*-- (fp,foo), (pc,c),
         (pa,a) -----*/
}

bar()
{
   .
   .
   /*-- POINT E: --*/
   /*-- (fp,bar),
      (pc,c) -----*/
   pb = &b;
   /*-- POINT F:-*/
   /*-- (fp,bar),
   (pc,c), (pb,b)-*/
}

main()
{
   .
   .
   pc = &c;
   if (cond)
      fp = foo;
   else
      fp = bar;
   /*-- POINT A: --------*/
   /*-- (fp,foo)?, (fp,bar)?,
         (pc,c)-----*/
   fp();
   /*-- POINT B: ---------*/
   /*-- (fp,foo)?, (fp,bar)?,
    (pc,c), (pb,b)?, (pc,c)?-----*/
   .
   .
}
```

(c) final alias sets

Figure 2: An example program demonstrating the algorithm

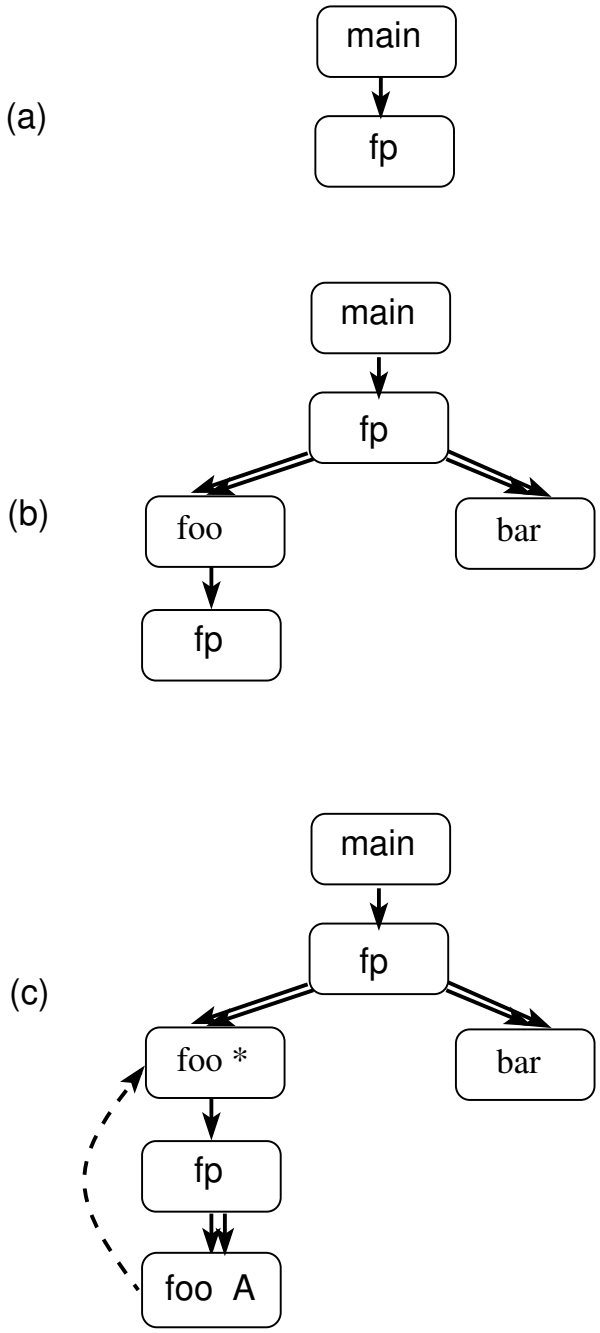


Figure 3: Invocation-Graph construction stages for the program in Figure 2

```
int a,b,c;
int (*fp)();
int *pa,*pb,*pc;

foo()
{
   .
   .
   pa = &a;
   if (cond)
   fp = bar;
}

bar()
{
   .
   .
   pb = &b;
}

main()
{
   .
   .
   pc = &c;
   fp = foo;
   while (cond)
   fp();
   .
   .
}
```

(a) original program

```
int a,b,c;
int (*fp)(),*pa,*pb,*pc;

foo()
{
   .
   .
   pa = &a;
   if (cond)
   fp = bar;
}

bar()
{
   .
   .
   pb = &b;
}

main()
{
   .
   .
   pc = &c;
   fp = foo;
   /*-- POINT A: ---------*/
   /*-- (fp,foo), (pc,c)-*/
   while (cond)
   fp();
   .
   .
}
```

(b) alias sets before resolving the call fp() at program point A

```
int a,b,c;
int (*fp)(),*pa,*pb,*pc;

foo()
{
   .
   .
   /*-- POINT C: ---------*/
   /*-- (fp,foo), (pc,c),
   (pa,a)?, (pb,b)? -----*/
   pa = &a;
   if (cond)
   fp = bar;
   /*-- POINT C: ------------*/
   /*-- (fp,bar)?, (fp,foo)?,
   (pc,c), (pa,a), (pb,b)?--*/
}

bar()
{
   .
   .
   /*-- POINT E: -------*/
   /*-- (fp,bar), (pc,c),
    (pa,a)?, (pb,b)? --*/
   pb = &b;
   /*-- POINT F: -------*/
   /*-- (fp,bar), (pc,c),
    (pa,a)?, (pb,b)----*/
}

main()
{
   .
   .
   pc = &c;
   fp = foo;
   /*-- POINT A: ----------*/
   /*-- (fp,foo), (pc,c)--*/
   while (cond)
   fp();
   /*-- POINT B: --------------*/
   /*-- (fp,bar)?, (fp,foo)?,
    (pc,c), (pa,a)?, (pb,b)?--*/
   .
   .
}
```

(c) alias sets after fix-point computation for the while loop

Figure 4: An example program with a function pointer call inside a loop

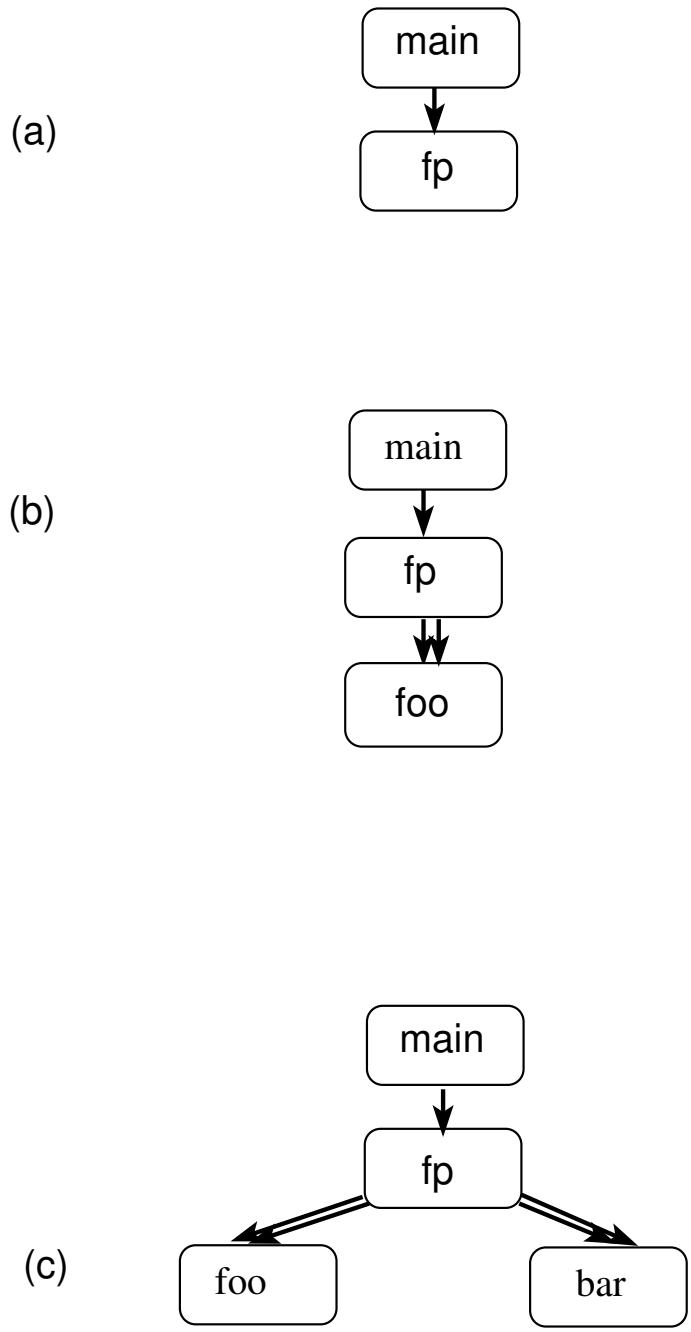


Figure 5: Invocation-Graph construction stages for the program in Figure 4

program	may-alias	must-alias	Total
fpt1.c	50	69	119
fpt2.c	85	99	184
fpt3.c	119	125	244
fpt4.c	269	69	338

(a) Conservative Algorithm's Results

program	may-alias	must-alias	Total
fpt1.c	8	75	83
fpt2.c	14	141	155
fpt3.c	58	155	213
fpt4.c	104	156	260

(b) Our Algorithm's Results

Figure 6: Naive Algorithm v/s Clever Algorithm

the compiler, while making significant contributions to the improvement in the precision of flow information being collected, by all the interprocedural analyses.

However, in the presence of function pointers, computing precise alias information is NP-hard. We don't give a formal proof here, but informally demonstrate why it can be so. Most flow analysis problems become NP-hard, when path information is needed to obtain precise information. The same situation arises here. Consider the following scenario :

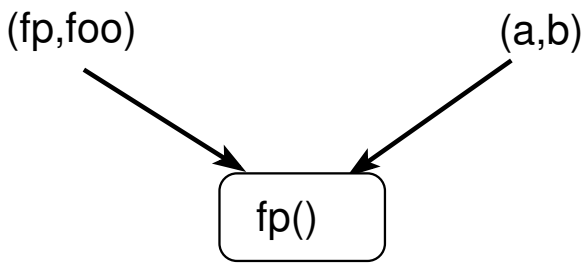


Figure 7: Alias pairs coming from different paths

The alias pairs (fp,foo) and (a,b) come to the fp() node from different paths in the program. The two aliases can never hold together at the entry of the fp() node. So while computing the effect of the potential call to foo() from this call site, the alias pair (a,b) should not be propagated to its entry node. But while sitting at the fp() node, we need path information to know, if the alias pairs (fp,foo) and (a,b) hold on the same path to this node, or they come from different paths. Hence, precise information can not be obtained, efficiently.

4 Related Work

Research has not been reported so far that directly focuses on the problems of interprocedural analysis in the presence of function pointers. However, some work done on analysis of object-oriented languages closely relates to our work, as the virtual function feature provided in object-oriented languages poses similar problems.

A virtual function is a function defined with the same name in every class of an inheritance hierarchy, but with a possibly different implementation for each class. This provides a neat class interface to the client. The function invoked by a virtual function call, say p→foo(), depends on the type (class) of the receiver (the object pointed to by p). Most object-oriented languages support dynamic typing(smalltalk), while the ones which provide static typing(C++), permit implicit and explicit type conversions. For these reasons, a virtual function call is also dynamically bound like a function pointer call in C.

Pande [Pan92] addresses this problem in a fashion, similar to our approach. However, instead of finding all aliases of p, he collects all possible types of the object pointed to by p. This type analysis can be done independently of alias analysis, if only single level pointers are used. With multiple level pointers, the type of a pointer can be changed by a type-casting assignment to another pointer variable which is its alias. In this case, he needs to perform type analysis, alias analysis and invocation-graph construction, all together. He uses ICFG to represent the

interprocedural structure of the program and uses the conditional approach [LR92] to do interprocedural analysis. The algorithm is elegant for single level pointers, but for multiple level dereferences, it becomes complicated.

Sudholt and Steigner [SS92] also present a solution to this problem. They introduce the concept of representants. For each set of functions associated with a particular virtual function, they devise a representant function, which represents the cumulative flow information for all these functions. Every call to a virtual function, is then bound to the corresponding representant function, for purposes of interprocedural flow analysis. The mechanism for devising the representant has not been explained in their paper. Moreover, it remains doubtful, if this concept is feasible for flow-sensitive interprocedural analysis.

Vitek, Horspool and Uhl[VHU92] use abstract interpretation techniques for compile-time analysis of object-oriented languages. They assume the existence of a function *classof*(), which returns all possible classes of an abstract object. However, the implementation of this function is not defined in their paper, which is the issue under consideration.

5 Conclusions and Future Work

In this paper we have addressed the problems of interprocedural analysis in the presence of function pointers, and presented an efficient algorithm for the construction of the invocation-graph under these circumstances. We have also demonstrated that alias analysis provides the basic information needed by our algorithm and hence is a necessity for effectively performing any interprocedural analysis in the presence of function pointers.

The statistics provided also shows that with conservative assumptions about function pointer calls, the alias information becomes very approximate, which in turn can introduce imprecision in results obtained from other analyses. Most importantly, we have shown that our algorithm for invocation-graph construction does not require any extra work. It only needs to introduce minor modifications in the the alias analysis phase. We have also suggested a new strategy for conducting flow analyses that exchange information. Instead of attempting to find the appropriate order in which to perform them, we conduct them simultaneously, where one analysis receives the information from the other, as and when it needs it, and as and what is available at that instant.

We have implemented the algorithm in the framework of our McCAT C compiler, and collected the statistics provided in the paper. We plan to run our algorithm on more programs that use function pointers in a greater variety of ways, and gather more robust statistics.

There is a major scope for improvement in our algorithm, for analysis of programs that use arrays of function pointers. Screen monitoring programs and programs for real time systems, particularly use these constructs. Typically, the array is statically initialized to a set of functions, where each function corresponds to a specific action to be taken, according to the input from the environment. On receiving an input, the program computes the appropriate array index, and invokes the function pointed to by the corresponding function pointer.

However, currently our alias analysis technique assumes the whole array to be aliased to a location, if its address is assigned to any of its entries. For function pointers, it amounts to assuming that the set of functions associated with a function pointer call fp[i]() consists of all the functions pointed to by any of the entries of the array fp[]. This assumption is quite conservative and can introduce considerable imprecision in the flow information.

The situation can be improved by using constant propagation information, about the array indices and making the alias analysis more accurate. Whenever, the value of an index can not be pinned down to a constant, it can still be constrained to a range of values by generalized constant propagation. This would pay big dividends, because if the invokable function set is curtailed by even one function, a considerable amount of imprecision is being reduced. However, constant propagation also needs to be an interprocedural analysis in order to be effective. Hence it too has to be performed together with alias analysis and invocation-graph construction.

Presently we are working on the conceptual formalization of the issues involved in using constant propagation information for better interprocedural analysis. We soon intend to implement the improved algorithm in the framework of our McCAT compiler.

6 Acknowledgements

I would like to thank Maryam for her help with the implementation. Discussions with her also helped in making the ideas presented in this paper more concrete.

References

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C Compiler. Master's thesis, McGill University, August 1993.

[Hen90] Laurie J. Hendren. Parallelizing Programs with Recursive Data Structures. PhD thesis, Cornell University, January 1990.

[LR92] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Symposium on Programming Language Design and Implementation, 1992.*

[VHU92] J. Vitek, R. Horspool and J. Uhl. Compile-time analysis of object-oriented programs. In *Proceedings of the Workshop on Compiler Construction, 1992.*

[Pan92] Hemant Pande. Static analysis of C++ : Type determination and aliasing. Technical Report , TRDDC, Pune, India, December 1992.

[Sri92] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's Thesis, McGill University, June 1992.

[SS92] Mario Sudholt and Christoph Steigner. On interprocedural data flow analysis for object-oriented languages.In *Proceedings of the Workshop on Compiler Construction, 1992.*