McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Hybrid Points-to Analysis

Alida Segal

ACAPS Project No. 1.3
December 19, 1997

1 Abstract

This paper reports on the design, partial implementation and expected results of the hybrid approach between FP (fast points-to analysis) and CSP (context-sensitive points-to analysis) models to collect alias information. FP computes a class equivalence memory model that estimates the possible targets of pointers at compile time, in almost linear time in the size of the call graph. CSP, on the other hand, gathers approximate points-to relationships between pairs of stack locations over all contexts of the invocation graph. CSP may in the worst case run in exponential time in the size of the call graph. The new hybrid approach reuses FP knowledge of points-to information to initialize the CSP initial estimate of the alias output information for recursive functions, and has the potential to reduce the number of iterations these functions are processed. This strategy results in a faster CSP algorithm that collects accurate points-to information with initial knowledge gathered by the FP analysis on recursive functions. Other attempts presented in this paper, tried to combine the FP and CSP models and resulted in a hybrid approach in which FP classifies functions as Selected or Excluded depending on their ability to modify or leave intact the alias information, respectively.

List of Contents

1 Abstract 1

1 List of Contents 2

2 Introduction and Motivation 3

3 Background and Related Work 4
3.1 CSP Overview 4
3.2 FP Overview 5

4 Specific Problem Statement 7

5 Solution Strategy..... 8

6 Experimental Framework.....10

7 Other Results 11

8 Expected Testbed and Environment.....12

9 Detailed Results and their Analysis.....13

10 Conclusion and Future Work.....14

11 References 15

List of Contents

1 Abstract 1

1 List of Contents 2

2 Introduction and Motivation 3

3 Background and Related Work 4
3.1 CSP Overview 4
3.2 FP Overview 5

4 Specific Problem Statement 7

5 Solution Strategy..... 8

6 Experimental Framework.....10

7 Other Results 11

8 Expected Testbed and Environment.....12

9 Detailed Results and their Analysis.....13

10 Conclusion and Future Work.....14

11 References 15

2 Introduction and Motivation

Alias analysis plays an important role in optimizing and parallelizing compilers. Two or more expressions that denote the same memory address, are aliases of one another. The alias relationship can be introduced by the address-of operator (e.g. &a), by the single pointer reference (e.g. *a), and by the multi-pointer references (e.g. **a). The ambiguities and side effects introduced by the alias existence are particularly important, since one or two variables reference the same memory location, and may purposely, or inadvertantly, change the value held in that memory cell. The presence of pointers makes data-flow analysis more complex, since they cause uncertainty regarding what is defined and used. The only safe assumption about an unkown pointer p is to assume that an indirect assignment through a pointer p can potentially change any variable. Another assumption that has to be made is that any use of the data pointed to by a pointer, e.g., x=*p, can potentially use any variable. These assumptions result in more live variables and reaching definitions than is realistic and fewer available expressions than is realistic. As for assignments with pointer variables, in the presence of procedure calls, there's no need to make the worst-case assumption - that everything can be changed - provided, that the set of variables a procedure might change can be computed. For a language that permits recursive procedures, the data available to a procedure consists of the globals and its own locals, in such a way that parameters may be passed by reference. A good approximation to the variables a procedure can change are the globals and the parameters to the function. The basic idea is to determine how each procedure influences the *gen, kill, use*, or *def* information of the others, then to compute the data flow information for each procedure independently. In this context, the points-to analysis information is used to resolve the complexities introduced by the pointer accessed memory locations in that it builds the read and write sets of a program. The immediate benefits of definite points-to analysis as implemented in the McCAT C compiler, are to provide killing information, pointer replacement, insight into the dependence analysis for parallelization purposes and reduction in memory loads and stores. Other advantages of points-to analysis information are in call graphs construction with function pointers.

A practical example of the use of points-to information to derive the read and write sets of statements S1, S2, S3, is featured next. The notation {p, a, D} means that p points to a definitely, for the statement p = &a.

int *p, a, b;	Points-to Information	Read Sets	Write Sets
S1: a= 1;	{}	{}	{a}
S2: p = &a;	{p, a, D}	{a}	{p}
S3: b = *p;	{p, a, D}	{p,a}	{b}

Table 1. Points-to Information, Read, and Write Sets for S1, S2, S3

3 Background and Related Work

McCAT implements two strategies in dealing with points-to analysis : (1) fast linear points-to analysis (FP)[4,5,6], and (2)context-sensitive interprocedural points-to analysis (CSP)[2,3]. FP computes a memory model that estimates the possible targets of pointers at compiler time, imprecisely, and runs in linear time in the size of the program, whereas CSP gathers accurate, definite points-to information in time exponential in the size of the call graph. The FP approach is based on Steensgaard's [4,5] inference algorithm in which the points-to analysis is obtained from the storage model constructed for the program and refined by a set of rules on basic statements. The CSP Model [2,3] estimates the possible and definite relationships between abstract stack locations in the context of invocation graphs, and will be covered in more detail, in the next paragraphs.

3.1 CSP Overview

Definition 1: Abstract stack location x definitely points-to abstract stack location y, in a specific invocation context, if x and y each represent exactly one real stack location and stack location x contains the address of the real stack location of y. The use and notation for this rule are :

```
x = &y                                  {(x, y, D)}
```

Definition 2: Abstract stack location x possibly points-to abstract stack location y, in a specific invocation context, if it is possible that one of the real stack locations of x contains the address of one of the real stack locations of y. The use and notation for this rule are :

```
if (expression is true)
        x = &y                          {(x, y, P)}
else
        x = &z                          {(x, z, P)}
```

Definition 3: A points-to set S at a program point p , is a safe approximation, if for all pairs of stack locations, where stack_i holds x, and stack_j holds y, comply to the following rules:

```
(i) if stack_i points-to stack_j on all valid execution paths to p,
then S contains either {(x, y, D)} or {(x, y, P)}.
(ii) if stack_i points-to stack_j on some, but not all valid execution
paths, then S contains {(x, y, P)}.
(iii) if S contains {(x, y, D)}, then stack_i must points-to stack_j on
all execution paths.
```

```
The L-location refers to the variable reference itself, thus for x we have :
                L-location : {(x, D)}

The R-location refers to the stack locations pointed to by the variable reference,
such that if x points-to y with relationship D, then we have :
                R-location : {(y, D)| (x, y, D) belongs to S}
```

The CSP basic rules for points-to analysis process the changes to the input by generating new kill sets, new gen sets, and sets that go from definite to possible. If and else statements merge the output alias information of both if and else bodies. While loops get the input as the initial alias estimate, process the body and merge the input and output into a new input, iteratively, until the last estimate of alias information and the new merged input converge to the same value. Points-to analysis for procedure calls are measured in terms of all invocation paths in the invocation graph and is a simple depth-first traversal of the program call structure starting with main. The rules used by the interprocedural analysis can be found in CSP [3], in the Compositional Interprocedural Rules for Points-to Analysis. A variation of the interprocedural rules dealing with the hybrid approach described in this paper is found in section 5 of Solution Strategy. The concept around function pointers is to build an invocation graph of the program and leave it incomplete at the points of function pointer call. Next points-to analysis is performed on the invocation graph, and where there is a call though a function pointer, we find all functions it can point-to , and the invocation graph is updated with them. Next each function pointed to is analyzed again in this context. Finally the output points-to analysis is computed by merging the outputs of all pointed to functions in the invocation graph. For a more in-depth treatment of the Basic Analysis Rules for Points-to Analysis read CSP [3].

3.2 FP Overview

This model tracks the dataflow of alias information with a dynamic storage model that shows all its possible configurations at runtime. An abstract memory location is associated with every variable in the program and is referred to as an Equivalence Class Representative (ECR). Pointers create new sets of ECR obtained from joining the corresponding ECR fields of the variables who point-to the same data. A graphical example of an ECR is shown next in Figure 1. The *Declare Node* has the name of the variable, the ECR field has the memory address of the variable, the REF field shows the list of variables this node points-to, and LAM is the function signature of the node. The FP analysis consists of joining the ECR's related to variables that may point to the same memory location during the lifetime of the program.

Several rules help shape the storage model. These rules impose to join certain abstract memory locations of type ECR, LAM, or REF.

In the case of X = &Y the REF(X) component is joined with the ECR(Y) location.

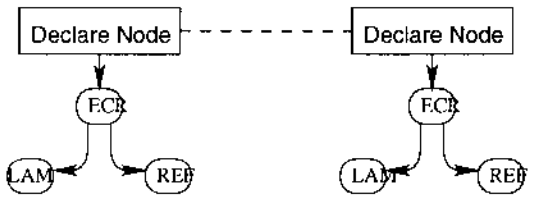

Figure 1: Assignment of an ECR Structure

For statement X = Y the rule implies to join REF(X) with REF(Y).

The rule for X = *Y is to join REF(X) with REF(REF(Y)).

Statement *X = Y is handled by joining REF(REF(X)) with REF(Y).

Other compound statements are processed as a combination of the 4 cases already listed. Function pointers are handled by the LAM component of the structure, which is divided into the LAM Parameters (function parameters) and LAM Return (function return value). In the case of function pointers the join operation is applied between the corresponding LAM Parameters for the actual and formal parameters. At the place the function returns to the call site, the LAM Return component of the function is joined with the LAM Return of the variable that is instantiated by the function.

The FP call graph is evaluated with a worklist containing the nodes in the graph, which collects their read, write and points-to information, as shown previousely in Table 1, section 2. According to Olivar's methodology, a node in the graph is processed by following the procedure ProcessNode(node), outlined in the next paragraph.

```
ProcessNode(node)
{
                merge(info(node), info(parents));
                take_out_from_worklist(node);
                if (info(parents) == CHANGED && children(node) == IN_LIST)
                       put_back_into_worklist(node);
}
```

The algorithm creates a worklist of the nodes in the graph with a DFS search. For every node in the worklist, it merges node's information with that of the parents, and it takes it out of the list. If some of the parents information changed and some of the node's chidren are still in the worklist, the node is placed right back into the list. For more details and information about the FP approach as implemented in McCAT, read Olivar's paper [6].

4 Specific Problem Statement

Large benchmarks are not suitable targets for accurate CSP analysis, since their algorithm's time is potentially, exponential in the size of the graph. A combined strategy between FP and CSP could prove to be beneficial in saving processing time for the CSP analysis, according to Olivar's results [6]. These results indicate that FP analysis is almost linear and compares favorably with CSP, as Table 2. shows next. Olivar's table [6] summarizes the execution times of points-to analysis under FP and CSP, for several bechmarks and indicates that benchmark travel can run 192.83 more time under CSP than under FP. Moreover, benchmark backprop takes 5.75 more time in collecting alias information under CSP than under FP. Furthermore, benchmark livc, takes 286.8 more time executing the CSP analysis than the FP one.

Benchmark	Size (lines of code)	FP Time (sec)	CSP Time (sec)	Ratio (CSP/FP)
backprop	243	0.04	0.23	5.75
exptree	348	0.04	0.56	11.2
anagram	477	0.05	1.52	30.4
travel	851	0.06	11.57	192.83
ks	668	0.07	2.63	37.57
chomp	428	0.08	4.68	58.5
rubik	445	0.08	7.20	90
stanford	865	0.11	1.28	11.63
clinpack	1037	0.09	2.94	32.6
compress	1206	0.14	5.50	39.28
fi	1296	0.09	2.49	26.6
sim	1340	0.12	9.58	79.83
livc	1759	0.25	71.70	286.8
config	2036	0.36	14.9	41.38
circuit	2392	0.14	3.78	27
yacr2	2734	0.26	17.85	68.65
ear	3089	0.35	8.32	23.32
cq	4753	0.47	26.11	55.55

Table 2: FP and CSP execution times for the computation of points-to information

These results have inspired me to attempt to use the good features of FP in order to improve CSP's time analysis. The main FP features of interest are the points-to information knowledge that may be reused by the CSP analysis as the first rough estimate for the interprocedural alias information.

The effects of procedure calls to the points-to analysis under CSP, is measured in terms of all invocation paths in the invocation graph and is a simple depth-first traversal of the program call structure starting with main. In the case of recursive functions, the strategy under CSP is to approximate all possible unrollings, terminating the depth-first traversal each time the

function name is the same as one of the ancestors. The CSP Compositional Interprocedural Rules for Points-to Analysis are outlined in the next page with minor modifications on lines 26 and 32, that implement the new strategy in dealing with all the possible unrollings of recursive functions.

The approach proposed by this paper is to let FP run to completion and collect alias information classified as interprocedural points-to analysis. Next, this alias information may be reused under CSP as the initial output estimate for the points-to information for recursive functions. Currently, CSP algorithm for recursive functions initializes the first alias estimate for the output node of a function in the invocation graph to Bottom, and then it enters a do loop that evaluates the body of the function and the results are saved in the computed alias output for the recursive function. The initial alias output estimate and the computed alias output for the function are merged together and the do while loop is exited when the two outputs converge to the same value. The estimated and computed points-to outputs may take a couple of iterations until they converge.

The strategy proposed by this algorithm has the potential to reduce the number of iterations the recursive body functions are processed and may provide a faster convergence method by using the points-to information under FP for recursive functions to approximate the estimate of the CSP stored output.

5 Solution Strategy

This paper proposes a new strategy that combines the FP and CSP models to obtain a faster and more precised analysis that uses the best features from both approaches. The next algorithm - process_call_use_FP_results(), contains the changes that are to be applied to the CSP Compositional Interprocedural Rules for Points-to Analysis in the context of the new hybrid approach.

```
1 fun process_call_use_FP_results(Input, actualList, formalList, node, funcBody)
2 { (funcInput, mapInfo) = map_process(Input, formalList, actualList);
3 case 'node'
4 { "Ordinary":
5    if (funcInput == node.storedInput)       /* already computed */
6                       return(unmap_process(Input, node.storedOutput, mapInfo));
7    else  /* compute output, store input and output */
8    {                  funcOutput = process_stmt(funcBody, funcInput, node);
9                       node.storedINput = funcInput;
10                      node.storedOutput = funcOutput;
11                      return(unmap_process(Input, node.storedOutput, mapInfo));
12   }
```

```
13   "Approximate":
14   recursivNode = node.recursivEdge; /* get partner recursive node in call graph */
15   if (isSubsetOf(funcInput, recursivNode.storedInput))
16                       return(unmap_process(Input, recursivNode.storedOutput, mapInfo));
17   else /* put this input in the pending list */
18   {                   addToPendingList(funcInput, recursivNode.pendingList);
19                       return(Bottom);
20   }
21   "Recursive":
22   if (funcInput == node.storedInput)       /* already computed */
23                       return(unmap_process(Input, node.storedOutput, mapInfo));
24   else  /* get initial input and output estimate */
25   {                   node.storedInput = funcInput;
26                       node.storedOutput = node.storedFPOutput;
27                       node.pendingList = {}; done = false;
28   }do  /* process the body  of the node */
29   {                  funcOutput = process_stmt(funcBody.node, node.storedInput, node);
30            if (node.pendinList != {})       /* if unresolved inputs, merge input*/
31            {          node.storedInput = Merge(node.storedInput,pendingListInputs);
32                       node.storedOutput = node.storedFPOutput;
33                       node.pendingList = {};
34            }else if (isSubsetOf(funcOutput, node.storedOutput))
35                       done = true;
36            }else  /* merge outputs and try again */
37                      node.storedOuput = Merge(node.storedOuput,funcOutput);
38   } while (not done);
39   node.storedInput = funcInput; /* reset stored input to initial input */
40   return(unmap_process(Input, node.storedOutput, mapInfo));
41  }}
```

Figure 2: New Compositional Interprocedural Rules for Points-to Analysis

The oulined algorithm incorporates the new approach for the Compositional Interprocedural Rules for Points-to Analysis, in which the FP points-to information is reused under CSP to initialize the first estimate of the node stored Output. The function process_call_use_FP_results() attempts to estimate the points-to information for every node in the invocation graph, and receives initially some alias input, the actual and formal parameters, the node and the body of the function. It tries to classify the node as *Ordinary, Approximate, or Recursive*, and preprocesses the node alias information based on its type. The input points-to set for the called procedure inherit the points-to information at the calling site, such that formal parameters inherit the points-to relationships from the corresponding actual parameters. Nodes whose input and output alias information was computed, return to the calling function and unmap their

parameters. *Ordinary* nodes, whose results were not computed, have the body of the function evaluated for alias information and the resulting outputs are stored, returned and unmapped to the calling site. The *Recursive* and *Approximate nodes* are used together to implement the accurate points-to analysis resolution for recursive functions. *Approximate* nodes use the current stored output approximation for the function, whereas *Recursive* nodes compute the effect of points-to information on the particular function. The *Recursive* nodes are processed iteratively, and their do while loop is exited when the input and output stored alias information is general enough and can not be changed by any more function evaluation. *Recursive* nodes under the new strategy get their first estimate for the output alias information from the points-to information gathered under the FP method for the same node, as shown on line 26.

```
line 26 :       node.storedOutput = node.storedFPOutput;
```

The do while loop iterates as long as the alias information obtained from processing the node is not contained in the output points-to information stored previousely for the node. FP alias knowledge may be reused by CSP recursive nodes to refine and converge the alias outputs, in a fewer number of trials. If the computed alias output is not contained or convergent to the stored alias output, the two are merged and stored into an updated stored output for the node. The process of iteration and alias refinement continues a couple of times until the outputs converge. The new approach claims to reduce the number of iterations executed inside the do while loop by reusing the FP alias knowledge for the node, and therefore results in a faster algorithm.

6 Experimental Framework

The McGill Compiler Architecture Testbed, McCAT Compiler [8,9] is the proposed experimental framework for this project since it has robust and working implementations of the FP and CSP models. The McCAT Compiler takes a C source code program and translates it into a high level tree representation called *First*.

The next stage of McCAT has *First* structure transformed into a simplified Abstract Syntax Tree called *Simple*. Several optimizing iterations are applied to *Simple* that deal with program restructuring, function inlining, loop unrolling, pointer alias analysis, generated constant propagation, dependence analysis and high level loop parallelization. The *Simple* abstract tree structure is made of nodes that represent the statements of a program with all the relationships and logic inherent in it. The simplified *Simple* representation is fed into a new structure called *Last* that optimizes the tree with register allocation, instruction schedulling, software pipelining and loop transformation that are better suited for a specific platform and environment. The CSP and FP implementations are found in McCAT under directories : alias, and rw-sets. The files belonging to the FP analysis usually start with the suffix 'eq'.

7 Other Results

A combined strategy between FP and CSP models was initially, suggested by Olivar [6], Her approach was to let FP run first and examine whether a function modifies the pointer information of the entire program. Functions changing the pointer information are marked Selected and those that don't affect the storage model, are classified as Excluded. The Selected functions need to be processed further by CSP in order to achieve accurate results of their alias information. In this context, Olivar [6] has provided a table that summarizes for several benchmarks the number of Selected and Excluded functions, encountered in them.

Benchmark	Total Functions	Selected Functions	Excluded Functions
c4	13	10	3
lharc	91	88	3
bintree	17	13	4
chomp	22	21	1
cq	40	34	6
live	82	81	1
puzzle	17	15	2
stamford	48	42	6
eqntott	63	59	4
espresso	80	79	1
whetstone	7	6	1
ft	42	38	4
yacr2	58	54	4
cur	95	84	11
sc	151	149	2
li	358	354	4
ks	13	12	1
compress	17	16	1

Table 3. FP Classification of Functions as Selected or Excluded

As the results of Table 3. show, the number of Excluded functions is small for most of the benchmarks, and the degree of speedup CSP could gain from working on the Selected functions only, is not remarkable. Rakesh Ghiya has implemented the approach described above, and the results were not encouraging, such that the new implemented hybrid approach did not show noticeble speedup.

8 Expected Testbed and Environment

The approach proposed in this paper is to reuse the FP assessment of alias information in the CSP model as the initial approxiamtion for the alias stored output of recursive functions. Usually under CSP, the alias stored output of recursive functions is initially approximated to BOTTOM, suggesting that nothing is known about it. In case FP is executed at first for a particular program, a great deal of knowledge is gained inexpensively, which can be reused by the CSP model to make better approximations in relationship to the initial storead alias output of recursive functions. The FP runs in almost linear time to the number of nodes in the call graph, and its results are sound and insigtful. The suitability of FP results is emphasized in Olivar's Read and Write Sets Table [6] which compares the Read and Write sets of several benchmarks for FP and CSP, and the findings are quite similar.

Some of the tools and low level implementations details that comprise the McCAT testbed and environment for alias collection will be covered in the next paragraphs. The points-to analysis information is stored in McCAT CSP model in the alias matrix *al_mat* which is of type *AL_BLK*. A function's alias matrix information is retrieved as follows:

```
AL_BLK *al_mat;                          /* alias matrix variable declaration */
al_mat = AL_FUNC_MATRIX(func_decl_node); /* get a function's matrix  */
```

The alias matrix has a value of 1 for any pair of row-column related variables, and a 0 value for unrelated variables. In order to get the index of a variable X in the matrix al_mat and the variables it points-to the following notation is used :

```
INDEX_INFO_INDEX(X);                     /* get index of X in alias matrix */
INDEX_INFO_FIELDS(X);                    /* get the fields X points-to */
```

In order to print the alias pairs use the procedure *print_al_mat(al_mat)* and the format of the output will be (a, b):, for any sure pair of aliases a, b. The format of the output for maybe aliases a, b, is (a, b)?:.

Invisible variables that can be modified by a function, and are not available in its scope, but through functions parameters by reference, are stored into the map-info information saved in the call graph. To access the map-info, the items needed are : the cg_node (the related call graph node), the callee index (the index of the callee variable in the alias matrix), and callee_al_blk (the alias block of the callee). See the use and notation of al_get_map_info_from_cg() next:

```
IMPORT_REL_VAR *al_get_map_info_from_cg(CG_NODE*,int,AL_BLK*);
REL_VAR *rel_var;
rel_var = al_get_map_info_from_cg(cg_node, callee_index, callee_al_blk);
```

For more information on the experimental testbed and environment read [8,9].

9 Detailed Results and their Analysis

The time alotted to this project was not sufficient to implement the entire solution strategy, therefore a mathematical proof will be provided instead to validate the correctness and suitability of this approach to reduce the CSP time complexity for recursive functions, in the context of a portion the pseudocode of the Compositional Interprocedural Rules for Points-to Analysis.

The body of the do while loop was copied for convenience in the next paragraphs. The do while loop is repeated until the condition 'not done' on line 38 becomes false, and the boolean variable is updated only once on line 35 in case the computed alias output is included into the stored output. Every time through the loop, the body of the function is evaluated again and its stored alias output refined. At first the new approach initializes the first approximation for the stored alias output of recursive functions to the final FP stored alias output for the same function.

The Trial and Error approach to estimate the alias information adopted by the original algorithm, has been modified to include a knowledgable approximation provided by the FP analysis, which although imprecise, is quite similar to the CSP analysis. Every time in the loop, if there is a pending list of unresolved inputs, the inputs are merged and stored into an updated input. If the computed stored alias output is different and not included in the stored output approximation, the two outputs are merged and stored back into an updated stored output. This process stops when the alias information can not be changed anymore by new merges, or function evaluations. This refined and general output can be reached in fewer iterations, if the initial Trial and Error guess is made with the alias information obtained from the FP analysis. The complexity of the CSP analysis is in part attributed to the number of iterations the do while loop is repeated for recursive functions. A better approximation for the initial stored output obtained from the FP analysis is bound to reduce the number of iterations the recursive stored outputs are refined, and therefore it results in a faster running CSP analysis.

```
28  }do  /* process the body  of the node */
29  {                  funcOutput = process_stmt(funcBody.node, node.storedInput, node);
30            if (node.pendinList != {})       /* if unresolved inputs, merge inputs */
31            {          node.storedInput = Merge(node.storedInput,pendingListInputs);
32                       node.storedOutput = node.storedFPOutput;
33                       node.pendingList = {};
34            }else if (isSubsetOf(funcOutput, node.storedOutput))
35                       done = true;
36            }else  /* merge outputs and try again */
37                      node.storedOuput = Merge(node.storedOuput,funcOutput);
38  } while (not done);
```

10 Conclusion and Future Work

This paper presented a new combined design strategy between FP and CSP analyses that has the potential to reduce the time complexity of programs with recusive calls, analyzed under the CSP model. The solution strategy outlined, modifies the Compositional Interprocedural Rules for Points-to Analysis to take advantage of the alias FP results. The FP output information for recursive nodes is used as the first approximation for the stored CSP output. This approach is very promissing and may lower the CSP time analyis. Future work should try to implement this new strategy under the McCAT testbed, and measure the speedup of the CSP analysis.

11 References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers - Principles, Techniques, and Tools. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montreal, Quebec, July, 1993.

[3] Maryam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural point-to analysis in the presence of function pointers. In proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, pages 242-256, Florida, June2-24, 1994. SIGPLAN Notices, 29(6), June 1994.

[4] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In Proceedings of the 1996 International Conference on Compiler Construction, pages 136-150, 1996.

[5] Bjarne Steensgaard. Points-to analysis in almost linear time. In Proceedings of the Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 32-41, 1996.

[6] Guirlyn Olivar. Fast Points-to and Side-effect Analysis for the McCAT C Compiler. Master's thesis, McGill University, Montreal, Quebec, April, 1997.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. Introduction to Algorithms. MIT Press, Cambridge, MA, 1994.

[8] Laurie J. Hendren, Guang R. GAO, Chandrika Mukerji, and Bhama Sridharam. Introducing McCAT - The McGill Compiler-Architecture Testbed. ACAPS Technical Memo 27, School of Computer Science, McGill University, Montreal, Quebec, September 1991.

[9] Laurie J. Hendren, and Bhama Sridharam. The SIMPLE AST - McCAT compiler. ACAPS Technical Memo 36, School of Computer Science, McGill University, Montreal, Quebec, October 1992.

[10] Sean Zhang, Barbara G. Ryder, and William Landi. Program Decomposition for pointer aliasing: A step towards practical analyses. Proceedings of teh 4th Symposium on the Foundations of software Engineering, 1996.

[11] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. Laboratory of Computer Science Research Technical Report, Number LCSR-TR-201, 1993.

[12] Rakesh Ghiya. Interprocedural analysis in the presence of function pointers. ACAPS Technical Memo 62, School of Computer Science, McGill University, Montreal, Quebec, December 1992.