A SURVEY OF INTERPROCEDURAL DATA FLOW ANALYSIS TECHNIQUES

by: Barbara Ryder

DCS-TR-85

Department of Computer Science
Rutgers, The State University of New Jersey
New Brunswick, New Jersey 08903

December, 1979

Table of Contents

Section Page

I. Introduction 1

II. Language Model 1

III. Data Flow Analysis 2

III.A May vs Must Data Flow Information 3

III.B Sensitivity to Flow of Control 5

III.C Precision 7

III.D Interval Analysis Algorithm 7

III.E Iterative Algorithm 9

IV. Previous Efforts 10

IV.A Spillman 11

IV.B Allen 12

IV.C Allen and Schwartz 13

IV.D Rosen 14

IV.E Lomet 16

IV.F Barth 18

IV.G Banning 20

IV.H Sharir and Pnueli 21

V. Open Questions 25

References 28

I. Introduction

This technical report surveys recent work in the area of interprocedural data flow analysis. We summarize previous work in the area, present the programming language models used and categorize the "goodness" of the solutions of each method. In this paper we assume the reader is familiar with recursive, procedural languages (e.g., ALGOL) and classical data flow analysis algorithms and problems (e.g., finding def-to-use chains using interval analysis or the general iterative algorithm). First, we discuss the basic language model used by the majority of the methods. Second, we consider the data flow problems tackled and describe their characteristics; we also give short descriptions of the interval analysis and general iterative algorithms. Third, we present each method in terms of its major goal, language model restrictions, data flow problems addressed, alias handling and strengths and/or weaknesses. Fourth, we discuss open questions in the area.

II. Language Model

The language model used by previous investigators is a block structured, recursive language which allows procedure parameters, label variables, and pointer variables. We note that the language model presented here resembles a subset of PL/I. Aliasing of variables exists; that is, there is the possibility that two distinct

variable names, in the same execution environment, refer to the same storage location. We say the variables are "aliases" of one another. Aliasing arises through the association of reference parameters with actual arguments in a sequence of procedure references (Ryder, 1979) or through the use of pointers. Recently, suggestions for handling the latter form of aliasing have been made, but no implemented versions of these ideas have been reported (Weihl, 1979) (Cousot/Cousot, 3/1977). Aliasing also occurs at explicit programmer direction through the use of language constructs for storage sharing (e.g., EQUIVALENCE statements in FORTRAN). Only the non-explicit forms of aliasing were addressed by the methods described in this paper. Due to aliasing, the data flow effects of a statement can extend beyond the variables which textually appear in that statement.

III. Data Flow Analysis

The optimizations being considered by the methods presented in this paper include: code motion, dead code elimination, common subexpression elimination, and constant propagation. These optimizations depend in part on the solution of the reaching definitions, live uses and available expressions data flow problems (Hecht, 1977). Previous to the work considered here, the occurrence of a procedure reference at program point p, served to "kill" all active definitions and/or uses at p. The study of interprocedural data flow

analysis is motivated by the desire to be able to improve this overestimate of the effect of the procedure reference in order to better optimize code containing procedure references.

The information we seek in order to solve the above optimization problems can be categorized according to two general criteria. First, we can determine whether the data flow information describes "possible" or "certain" program behavior. Second, we can consider the sensitivity of the information to program flow.

III.A **May vs Must Data Flow Information**

The first criterion for categorizing data flow information refers to the difference between "may" and "must" information. "May" information refers to the modification, use or preservation of the value of a variable on some execution path in the program (i.e., "possible" information). "Must" information refers to the modification, use or preservation of the value of a variable on all execution paths in the program (i.e., "certain" or "always" information).

For example, code motion requires "may" information; we need to guarantee that we break no use-to-def links by moving a definition or use out of a loop. A use-to-def link exists between a use of a variable at program point p and a definition of the variable at program point q if there is a definition-free path for that variable from q to p. Def-to use links are defined analogously (Hecht, 1977).

Common subexpression elimination requires "must" information; there must be a calculation of the subexpression on every path reaching the use, before the additional calculation represented by the use can be eliminated.

In Figure 1. below, the value of variable x "must" be preserved through procedure P, whereas the value of variable y "may" be preserved since it is modified on some execution paths through P. Similarly, the value of variable cnt "must" be modified by P whereas the value of variable y "may" be modified by P.

```
procedure P (real x,y);
begin integer cnt;
cnt:=0;                  Node 1 in flowgraph

while x <= y do          Node 2

     y:=y/2;
     cnt:=cnt+1;
endwhile;                Node 3

print (cnt);
end P;                   Node 4
```

Figure 1. An ALGOL Example

For "may" information, a safe solution is an overestimate, since this information involves a weak statement about the program. For "must" information, a safe solution is an underestimate, since this information involves a strong statement about the program.

III.B Sensitivity to Flow of Control

The second criterion for categorizing data flow information is its sensitivity to program flow, which affects how we can collect the information. Consider the flowgraph G for procedure P. Define the set MOD(n) for each node n in G to be the set of variables whose values may be modified by execution of the statements in the basic block corresponding to n. A basic block is a sequence of statements in a program which correspond to straight line code (Hecht, 1977). Then MOD(P), the set of variables possibly modified by execution of P, can be computed as the union of the MOD(n) sets for all the nodes in G. We call MOD a flow insensitive relation; that is, the MOD effect of the whole procedure is the union of the effects of its "parts" (i.e., its basic blocks).

Flow sensitive relations cannot be calculated in this manner. Consider how to find the set of variables used in P before being first defined in P, USE(P). We define USE(n) to be the set of variables used at each node in the flowgraph before being defined at that node. However, USE(P) is not the union of the USE(n) sets. USE(P) is useful in checking for undefined variables at compile time but clearly we must do more work than set union to calculate this relation.

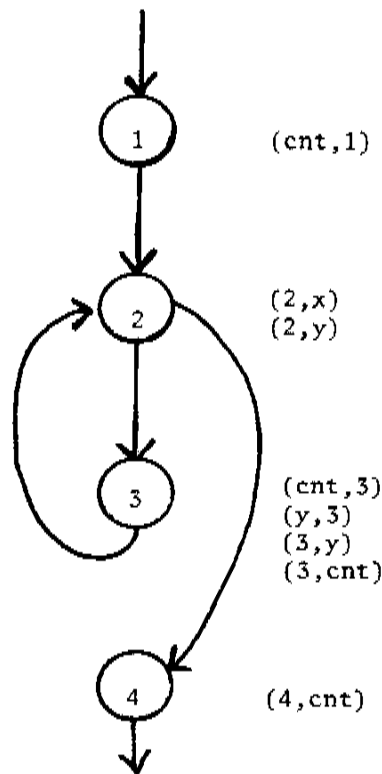

Figure 2. Flowgraph of procedure P

We can illustrate the difference between USE and MOD using the program in Figure 1. and its flowgraph in Figure 2. Let (x,i) represent a definition of variable x at node i in the flowgraph. Let (j,y) represent a use of variable y at node j which is not preceeded by a definition of y at j. Let "+" represent set union.

Then:

```
        MOD(P) = MOD(1) + MOD(2) + MOD(3) + MOD(4)
               ={(cnt,1)} + 0 + {(cnt,3) (y,3)} + 0
               ={(cnt,1) (cnt,3) (y,3) }
but
        USE(P) = {(2,x) (2,y) (3,y)}
and     S      = USE(1) + USE(2) + USE(3) + USE(4)
               ={(2,x) (2,y) (3,y) (3,cnt) (4,cnt)}
therefore
        USE(P) ~= S.
```

III.C Precision

In our discussions below, we refer to the data flow information as "precise" using Banning's definition (Banning, 1978). A method calculates the data flow effect of a reference in a manner "precise up to symbolic execution" if it is able to determine all variables which may or must be affected in the designated way under the assumption that all execution paths through any procedures invoked as a result of this reference are equally likely. In addition, the determination of the path actually executed must be independent of the reference itself.

III.D Interval Analysis Algorithm

There are two major data flow analysis algorithms: the interval analysis algorithm and the iterative algorithm. Interval analysis was developed by Allen and Cocke (Allen/Cocke, 1977). Their algorithm uncovers the nested loop structure of the flow graph of a program. Its goal is to discern the def-to-use links in the graph. The algorithm decomposes the flow graph uniquely into disjoint regions called intervals. They have the following properties.

i) An interval can only be entered through its header node.

ii) The header node of an interval dominates all nodes in the interval. If x and y are nodes in the flow graph, x dominates y if all paths from the entry node of the flow graph to y must pass through x.

iii) The header node is contained in all cycles in the interval.

iv) The order in which nodes are added to an interval, called interval order, is a linear ordering embedding the partial order of the digraph containing those nodes. Interval order is not unique.

v) The interval constructing algorithm can be applied to the derived graph, whose nodes are the intervals of the flow graph and whose edges represent paths from one interval to another in the flow graph. If this process of interval/derived graph construction is iterated and the final derived graph obtained is a single node, the original flow graph is called reducible (Allen/Cocke, 1977).

Figure 3. presents the derived graph sequence for procedure P given in Figure 1.

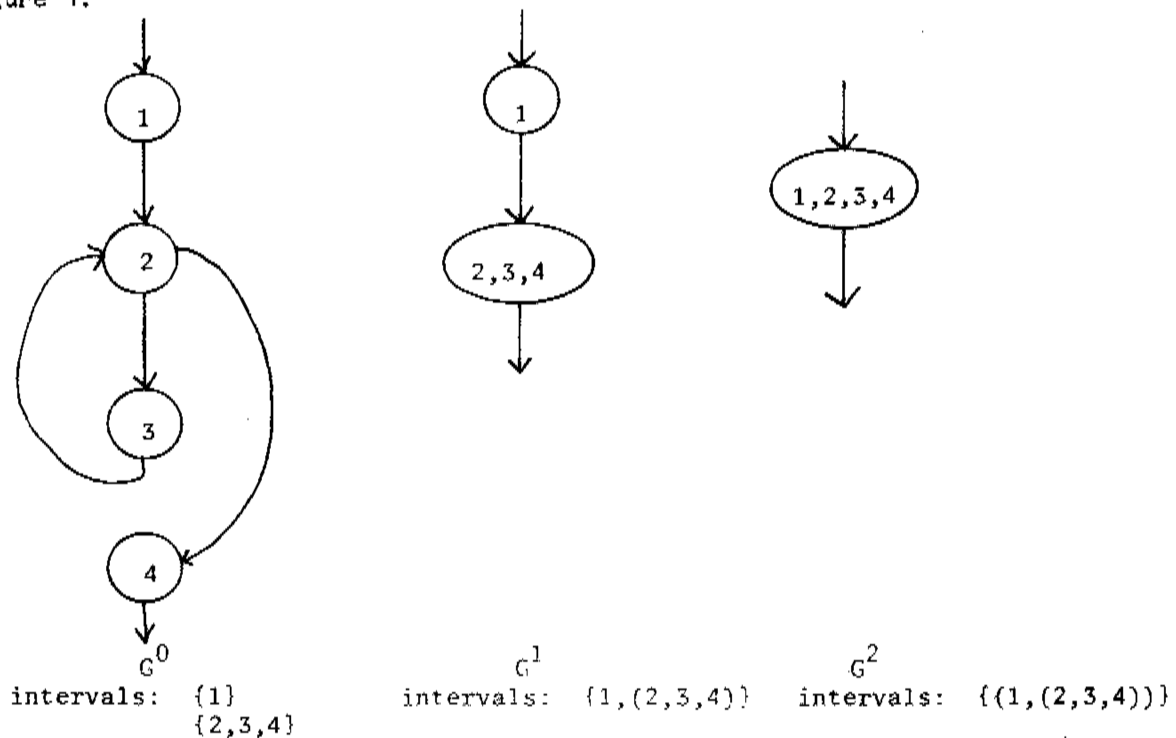

Figure 3. Derived Graph Sequence for Procedure P

The interval algorithm is a two stage process, involving a

data collection pass and a propagation pass. In each pass the entire sequence of derived graphs is processed. In the first pass, the data flow information due to program flow within an interval is collected. In the second pass, data flow information from outside an interval or from cycles within the interval, collected in the first pass, is propagated through the interval.

The interval analysis algorithm was extended to perform interprocedural data flow analysis on recursive programs (Allen, 1974) (Allen/Schwartz, 1973). Other algorithms were developed which share the notion of decomposition of the flow graph into components and solving data flow problems using the decomposition; these algorithms are often referred to as elimination algorithms (Graham/Wegman, 1976) (Hecht, 1977) (Tarjan, 1974).

III.E **Iterative Algorithm**

The general iterative algorithm was developed by Kildall in 1973 (Kildall, 1973). A lattice model is used in which lattice elements correspond to data flow information at each node in the flow graph. The way in which data flow information is transformed by program flow is represented by functions in a corresponding functional framework. A set of simultaneous equations is formed involving the lattice elements and functions which transform them. These equations describe the data flow problem whose solution is a fixed point of the equations. This solution is found iteratively for lattices of finite length (i.e., all chains in the lattice are finite) (Hecht, 1977).

The efficiency of the iterative algorithm can be improved by ordering the iteration, as in the round robin and node listing techniques (Hecht, 1977). It has also been optimized for reducible flow graphs (Tarjan, 1976). Recently, the lattice model technique has been extended to ascertain data types in weakly typed languages and to perform interprocedural data flow analysis (Jones/Muchnick, 1976) (Kaplan, 1978) (Tenenbaum, 1974) (Miller, 1979) (Schwartz, 1975) (Sharir/Pnueli, 1978).

Now that we have categorized the data flow information we wish to collect, presented our programming language models and highlighted the major intraprocedural data flow algorithms, we shall summarize the previous efforts in interprocedural data flow analysis.

IV. Previous Efforts

The first group of methods we shall present were associated with the experimental PL/I compiler project at IBM Yorktown Heights. The work of Spillman, Allen, Schwartz, Lomet and Rosen will be presented. Second, we consider the work of Barth and Banning, who were most concerned with refining data flow information in the presence of recursion and aliasing while restricting themselves to one pass over the program text. Third, we present the work of Sharir and Pnueli who mathematically modelled the previous methods and suggested an altogether new approach which can be worked into an inexpensive,

approximation technique. In addition, we discuss Sharir's recent effort at generalizing Allen's method.

To summarize the individual contributions of the IBM "group": Spillman worked on discovering aliasing and interprocedural links in the program; Allen extended her intraprocedural interval analysis to interprocedural analysis; Schwartz and Allen gave an approximate technique for handling recursion; Rosen presented a precise though unwieldy analytic method, capable of distinguishing between different references to a procedure; and Lomet compared the latter two approaches on recursive programs and presented an approximation method which falls somewhere between these two.

IV.A Spillman

Spillman (Spillman, 1971) worked on the problem of discovering aliases and obtaining the set of values possibly assumed by procedure and label parameters in the program. His effort was directed at a subset of PL/I which contained recursion, label and procedure parameters, and pointers. He addressed only the question "can this variable be modified?", which involves may, flow insensitive information. His method examined the program text once with procedures occurring in arbitrary order on input. Information was extracted and stored in a bit matrix. He calculated the calling relations among the procedures and derived an invocation order for them, in which a procedure preceeded all other procedures it might

call during execution of the program. Two passes over the bit matrix were made, in order to find all the effects which result from each reference (in the reverse invocation order pass) and all possible aliases of formal parameters, especially procedure parameters (in the invocation order pass). In case of recursion, he iterated the passes until the matrix stabilized. Iteration was also necessary in the case of a reference to a procedure parameter which itself contained procedures or procedure parameters as actual arguments.

In considering the strengths and weaknesses of Spillman's method, we must remember that it is the earliest work presented here (1971) and dealt with the most general language structures of all the methods. He obtained a broad range of alias and modification information in one pass over text. However, there was imprecision in his alias information; he obtained some aliases which could never occur during execution of the program because he accumulated aliases for formal parameters over all invocations of a procedure. In addition, he ignored issues of scope and multiple "copies" of a local variable in a recursive procedure.

IV.B Allen

Allen was interested in solving the reaching definitions and live uses data flow problems in an interprocedural setting, using her intraprocedural interval analysis (Allen, 1971),(Allen/Cocke, 1977). She used Spillman's work on aliasing and initially formulated an

approach for a restricted subset of PL/I without recursion, procedure parameters, label variables or pointers (Allen, 1974). She used Spillman's method to construct the call graph of the program, a digraph whose nodes are the procedures of the program and whose edges correspond to possible procedure references. Her intraprocedural algorithm applied to the call graph determined the set of definitions reaching each procedure, while only examining each procedure once, processing them in reverse invocation order. Her processing order insured that the full data flow implications of a reference was available when it was encountered. An obvious weakness of this method was that it could not handle recursion since a cycle in the call graph precluded the determination of an invocation order.

IV.C **Allen and Schwartz**

Allen and Schwartz (Allen/Schwartz, 1973), (Schwartz, 1974) extended the above algorithm to handle recursion. When a recursive subgraph in the call graph G was encountered, a minimal set of edges E' was found such that E(G) - E' was acyclic. Each edge (x,y) in E' was removed, a new copy of node y was made y', and a new edge (x,y') added obtaining the acyclic call graph G'. For each y' added, they initialized the data flow information associated with y' with a safe estimate of the information associated with y. Then they followed Allen's algorithm described above. After processing y, they had a better estimate for the information at y'. They either re-initialized

y' and iterated the process until it stabilized at a solution, or chose to stop iterating at some point and accepted an approximate solution. Schwartz (Schwartz, 1974) claimed they need only perform a few iterations to obtain "accurate enough" information for most programs.

A weakness of this method stemmed from the imprecision of Spillman's aliasing information, on which it depended; to review, Spillman calculated the aliases of parameters by the union of alias sets over all references to a procedure. Although the Allen/Schwartz method can handle flow sensitive data flow information, it settled for an approximate solution to all problems in the presence of recursion. Lomet (Lomet, 1977) discussed the fact that their "solution" on the recursive subgraph may not be as precise as possible because of the necessity to start the iteration with safe estimates of the data flow information.

IV.D **Rosen**

Rosen (Rosen, 1975) developed the most precise method to date for solving the "may" data flow problems (i.e., "may be modified, may be preserved, and may be used"). We illustrate his approach by considering the "may be modified" data flow problem; that is, MOD(p) is the set of variables which may be modified by execution of procedure p. For all procedures p, all possible alias sets e, and all exits d of p, Rosen computed MOD((p,e),d), the set of variables

possibly modified by an execution of procedure p under the aliases e which returns to the calling procedure through exit d. He calculated MOD((p,e),d) in terms of local flow in p and the MOD((p',e'),d') sets resulting from references in p to p' with alias sets e' which return to p through exit d'. This yielded a nonlinear set of simultaneous equations in these MOD variables which were solved with iterative techniques. He then calculated the modification effects of a reference by unioning appropriate MOD((p,e),d) sets.

The programming language model used by Rosen was basically a subset of PL/I with recursion and reference parameters. No procedure parameters, label variables or pointers were allowed. The alias sets which could arise during execution of the program were assumed known. When the code for some procedure was unavailable, he assumed a "worst case" effect.

The greatest strength of Rosen's method was that it provided precise results on flow sensitive and flow insensitive data flow information. It provided data flow effects specific to each reference rather than yielding information from a union over all references. He avoided the re-examination of program text by codifying the effects of a procedure into equation form.

An obvious weakness of this method was the large number of possible alias sets under which a procedure might be referenced. The number of unknowns in his equations could grow quite large. In addition, he avoided discussing aliasing when, in order to determine

the aliases under which a procedure is referenced, he needed all the aliases of procedures which reference it. This calculation required a separate pass; in the presence of recursion, iteration would be necessary.

Lomet (Lomet, 1977) contrasted the handling of recursion by Rosen and Allen/Schwartz. Rosen iterated to the most precise data flow information by starting with an unsafe initial estimate. This necessitates iterating Rosen's equations until they stabilize to insure the safety of the use of his results. By contrast, in the Allen/Schwartz scheme, all intermediate results are safe.

Recently, Rosen expanded and optimized his original algorithm (Rosen, 1979). He presented an example as motivation for the general iterative approach which he used and also compared his method with others mentioned here, stressing the language independent formulation of his method.

IV.E Lomet

Lomet (Lomet, 1977) presented an approximation technique for the calculation of "may" data flow information in the presence of aliases in which, unlike Rosen's method, the domain of all possible alias sets is not considered. However, the data flow effects of a particular call site can be distinguished, unlike the Allen/Schwartz method. He was concerned with summary data flow information, the effect of a procedure reference on the data flow in the calling

procedure, and with local data flow, the effect of a procedure reference on data flow in the called procedure.

Lomet was concerned with the "may be modified, may be preserved and may be used" data flow information. His programming language model was PL/I with restrictions similar to Rosen: no label variables, pointers or procedure parameters. He assumed the possible alias sets as given; he defined these so that only variables within the lexical scope of a block had aliases within the block. Thus he assumed he could differentiate between different "copies" of a local variable in a recursive procedure.

Lomet developed a model for approximating summary data flow with aliasing, using calculations of the data flow assuming no aliasing, combined with the pairwise aliasing of variables due to references. For recursion, Lomet used either the Allen/Schwartz or the Rosen method; in the latter case, he used no alias sets in the calculations, preventing the potential explosion in the number of variables in Rosen's method. For local data flow, he presented an approximation technique using the Allen/Schwartz interval analysis algorithm for reaching definitions. He used the local data flow results to generate the summary data flow information and showed how to distinguish the summary data flow effects of different references to the same procedure.

Lomet's goal was to obtain more refined information than Allen/Schwartz while avoiding the possible combinatorial explosion of

Rosen; he achieved this goal. He proved that he obtained "may be modified" information precisely. He contrasted the previous two methods in their approach to recursion. Allen/Schwartz made a pessimistic, safe assumption about recursive references and their method always iterated to a safe solution, which may be imprecise. Rosen made an optimistic, unsafe assumption about recursive references and achieved the most precise solution, but his method must be iterated to a fixed point to insure safety of the use of the information.

The next two efforts in interprocedural data flow analysis were primarily concerned with obtaining as much information as possible in the presence of recursion and aliasing while using only one pass over program text.

IV.F Barth

Barth (Barth, 1977) (Barth, 1978) defined the direct data flow relations, those calculated from the program text disregarding the effect of procedure references, and the alias relations, as binary relations on the set of procedures and variables in the program. He considered parameters to be variables. He then gave formulae involving these relations, their composition, intersection and transitive closure, which calculated the summary data flow effects of a procedure (i.e., the direct effects and effects of procedures called).

Barth's programming language model was a recursive, block structured language, which had reference and value parameters. He allowed no procedure parameters or pointers. He was interested in "may be modified, may be used and must be defined" data flow information. He considered the calculation of data flow information with no aliasing and then with aliasing introduced by the use of reference parameters. In the former, he used a relation SCOPE, to distinguish between different copies of a local variable of a recursive procedure, using the notion of procedure nesting level to define the level of local variables. In the latter case, he used two relations to describe all possible argument/parameter aliases resulting from a sequence of procedure references and accompanying parameter associations.

The strengths of his method were that it operated in one pass over program text, utilized a compact data representation (i.e., bitstring) and performed data flow analysis which was very precise in the calculation of "may be modified" information in programs without reference parameters. He handled recursion without re-examining program text and distinguished between different copies of local variables of recursive procedures by using his notion of SCOPE. He was able to prove which sets of variables may be modified by a particular call site under the assumptions of call-by-value or call-by-reference. He obtained a complexity result for his method which shows that asymptotically, his algorithm in the "no recursion, no reference

parameters" case is the "best" possible for summary data flow information, his algorithm performs asymptotically like a transitive closure algorithm. He had a PASCAL implementation of his algorithm. The main weakness of his method was that he calculated aliases imprecisely by collecting aliases across all references to a procedure and thus he allowed some aliases which could never occur during program execution.

IV.G **Banning**

Banning (Banning, 1979)(Banning, 1978) allowed only aliases which could occur along some execution path in the program. He set up an information propagation problem (Graham/Wegman, 1976) to solve for summary data flow information in a program and was able to derive local data flow information for a particular call site from the summary information combined with his alias relations. His programming language model was similar to Barth's. He also had a PASCAL implementation of his algorithm. Banning searched for the same data flow information as Barth, "may be modified, may be used, and must be preserved". Banning calculated the sets of aliases which exist in a particular programming environment using a straightforward recursive algorithm.

A strength of Banning's method was its precision on "may be modified" information with aliasing, but it was imprecise on flow sensitive data flow information. He claimed that to do this

precisely, one cannot use a one pass method on arbitrarily ordered input. He distinguished the different alias sets associated with different references. He also distinguished data flow effects associated with different procedure exits. He contrasted his method with a variant of Allen/Schwartz technique and found the latter compared favorably on most of the PASCAL programs he encountered. His elimination algorithm formulation enabled him to avoid the re-examination of program text in the presence of recursion.

An additional weakness of his method was that it did not handle procedure parameters, in general; he allowed references to procedure parameters which could contain only value parameters. Within a procedure, the data flow effects of a reference to a formal parameter was associated with the union of the effects of all procedures which can be associated with the formal parameter in the course of program execution. This is an overestimate of the effect of one reference. He also failed to handle pointers.

IV.H Sharir and Pnueli

The work of Sharir and Pnueli included an elimination method model of previous techniques and an additional method from which they derived an approximate algorithm currently under investigation (Sharir/Pnueli, 1978). Sharir and Pnueli were interested in a procedural, recursive programming language with no procedure variables or parameters, no aliases and no external procedures. They assumed

for their analysis that the call graph of the program was available as input to the analysis program. They used the classical lattice framework description of data flow problems to be able to consider data flow problems in general (Hecht, 1977).

Sharir and Pnueli presented a model of the functional approach to interprocedural data flow analysis which they claimed described most previous methods. A procedure was considered a "black box" which had its own particular effects on data. They formulated an elimination method which solved for the summary data flow effects of a procedure. This is valid with recursion for distributive, bounded frameworks and yields the "meet over all valid interprocedural paths" solution. For a finite lattice L, the associated function space is bounded, and they give an iterative workset algorithm, equivalent to the above algorithm.

Their second method used a "super" flow graph of the program built from the flow graphs of all the procedures in the program with the possible interprocedural call and return edges added(Sharir, 1977). To trace data flow on this flow graph, they associated with the data flow information, a call string which documented the calling history of the path along which the information had been propagated. These call strings determined along which valid interprocedural paths, information could be propagated; during this process, the call string was updated as needed. If there was no recursion, they showed their algorithm converged to the "meet over all

valid interprocedural paths" solution for a distributive framework. For a finite lattice L, they bounded the maximum length call string which must be examined in order to achieve this solution; the bound was expressed in terms of the size of L and the number of call sites in the program.

Finally, they developed an approximate call string algorithm usable for infinite lattices or nondistributive frameworks, (e.g., constant propagation). They encoded the call strings in a manner which decreased the size of the call string space which must be considered, but also sacrificed some precision, since it allowed possible propagation along invalid interprocedural paths. They proved that this approach leads to an underestimate of the "meet over all valid interprocedural paths" solution (i.e., safe data flow information). The degree of difference between the approximate solution and the actual solution depends upon how many invalid paths are allowed by the encoding; how to select an encoding to minimize this difference is a question for investigation.

The strength of the Sharir/Pnueli approach is that they presented a new view of interprocedural data flow analysis which can be used for general data flow problems expressible in terms of a lattice. Unfortunately, they did not compare their work to others in terms of performance, accuracy etc.. Also, they used a restricted language model and gave no data on their experiences with any implementation.

More recently, Sharir developed and implemented in the SETL optimizer, an algorithm which represents a generalization of Allen's interval algorithm presented here(Sharir, 1979). He handled "bitvector" data flow problems in the presence of recursion and/or irreducible flow graphs. Essentially, a depth-first spanning tree of the call graph of the program is used to determine a processing order among its procedures. Some procedures may be reprocessed iteratively to gather information from recursive subgraphs of the call graph, but the number of necessary iterations is bounded.

The first step in the algorithm involves analyzing each procedure in processing order. For recursive programs, this step is iterated until the information associated with each procedure entry stabilizes. During this step, each procedure p is analyzed by an intraprocedural interval-based algorithm. The processing order insures that the data flow effects of a procedure reference in p can be determined when it is encountered during analysis of p. In the second step, the algorithm propagates data flow information intraprocedurally to each node in each flow graph, using the interval structure from the first step. Sharir presented the algorithm for forward data flow problems, but maintained that backward problems can also be handled. Unfortunately, no comparisons to previous methods were made, nor were there any remarks about implementation experience.

V. Open Questions

This section summarizes the issues in interprocedural data flow analysis which were not addressed by the methods described above. There are two programming language constructs, procedure parameters and pointers, which no previous method has handled accurately. In addition, precise estimates of flow sensitive data flow information have not been obtained. There is a need for empirical comparison of these methods in terms of performance and for their analytic comparison in terms of complexity. Finally, there is the open question of the utility of this information in terms of the optimizations performed on actual programs in a high level language (e.g., PL/I, ALGOL 68). The answer to this question can be used to justify or oppose the additional effort necessary to obtain more accurate information.

In large, "practical" scientific software packages, the handling of procedure parameters is normally more important than the handling of recursion. Procedure parameters are plentiful whereas recursion, when it is used, appears in a very straightforward manner, not in the complexity considered by Barth, for example (Barth, 1977). Barth and Banning disregarded procedure parameters because their one pass criterion would be violated by trying to handle them. A reference to a procedure parameter which itself contains procedure parameters or a procedure as arguments, necessitates this re-examination of text. We can argue that this criterion is unsuitable for programs which are

being optimized before being used repeatedly for large processor runs; for such software, more accurate optimizations even at increased cost are worthwhile. Recently, Weihl at IBM Yorktown Heights has developed a method for obtaining an overestimate of the call graph of a program. The actual call graph is a subgraph of the graph he calculated (Weihl, 1979). The accuracy of the approximation is dependent upon the pattern of programmer usage of procedure parameters; further study is needed to determine the utility of this algorithm.

Within the question of handling procedure parameters is the question of how much information concerning procedure text is really necessary to perform the data flow analyses considered here, that is, to characterize the effects of a procedure and its references. This has bearing on the number of times the actual program text must be examined. This question also pertains to the handling of data flow information in the presence of external procedures (e.g., separately compiled procedures).

Pointer variables complicate the alias calculations greatly. Their usage needs to be categorized to see if some restricted usage is "easy" to handle, while avoiding issues of undecidability. Weihl's work also addressed pointer aliasing and value flow in the presence of pointers(Weihl, 1979). His algorithm gathered information in one pass over the program, disregarding flow of control, which resulted in the overestimates obtained. Further work will determine how useful and accurate the derived information actually is.

Little attempt has been made to compare the performance of these methods on a high level language such as PL/I. Likewise there has been little attempt to analyze the complexity of the methods, though Barth presented the asymptotic complexity of his method. In analyzing their performance and complexity we must keep in mind their accuracy on various data flow problems. In this regard the Sharir/Pnueli methods are most interesting for comparison purposes because of the generality of their formulation of the data flow problems considered.

There is a need for comparing approximation techniques on flow sensitive data flow information with precise techniques, in terms of the tradeoff of accuracy vs. additional effort. There is also is the question of the importance of the optimizations which use the flow sensitive information.

References

Allen, F. and Cocke, J., "A Program Data Flow Analysis Procedure", Communications of the ACM, Vol. 19, 1976.

Allen, F. and Schwartz, J. T., "Determining the Data Relationships in a Collection of Procedures", 1973, unpublished communication.

Allen, F., "A Basis for Program Optimization", IFIPS Proceedings 1971.

Allen, F., "Interprocedural Data Flow Analysis", IFIPS Proceedings 1974.

Banning, J., A Method for Determining Side Effects of Procedure Calls, Ph.D. thesis, Computer Science Department, Stanford University, November 1978.

Banning, J., "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables", Proceedings of POPL 1979, pp. 29-41.

Barth, J., A Practical Interprocedural Data Flow Analysis Algorithm and its Applications, Computer Science Department, University of California at Berkeley, May 1977.

Barth, J., "A Practical Interprocedural Data Flow Analysis Algorithm", Communications of the ACM, September 1978.

Cousot, P. and Cousot, R., "Static Determination of Dynamic Properties of Generalized Type Unions", SIGPLAN Notices, March 1977, pp. 77-94.

Graham, S. and Wegman, P., "Fast and Usually Linear Algorithms for Global Flow Analysis", Journal of the ACM, January 1976.

Hecht, M., Flow Analysis of Computer Programs, Elsevier North-Holland, 1977.

Jones, N. and Muchnick, S., "Binding Time Optimization in Programming Languages", Proceedings of 1976 POPL Conference, pp. 77-94.

Kaplan, M., "General Scheme for the Automatic Inference of Variable Types", Proceedings of 1978 POPL Conference, pp. 60-75.

Kildall, G., "A Unified Approach to Global Program Optimization", Proceedings of 1973 POPL Conference, pp. 194-206.

Lomet, D., "Data Flow Analysis in the Presence of Procedure Calls", IBM Research and Development Journal, November 1977.

Miller, T., "Type Checking in an Imperfect World",Proceedings of 1979 POPL Conference, pp. 237-243.

Rosen, B., "Data Flow Analysis for Procedural Languages", Journal of the ACM, April 1979, pp. 322-344.

Rosen, B., "Data FLow Analysis for Recursive PL/I Programs", IBM Research Report RC-5211, January 1975.

Ryder, B. G., "Constructing the Call Graph of a Program", IEEE Transactions on Software Engineering, May 1979, pp. 216-225.

Schwartz, J. T., "Interprocedural Optimization", SETL Newsletter no.134, July 1, 1974.

Schwartz, J. T., "Optimization of Very High Level Languages--I: Value Transmission and its Corollaries", Computer Languages, Vol. 1, 1975, pp. 161-194.

Sharir, M. and Pnueli, A., "Two Approaches to Interprocedural Data

Flow Analysis", Computer Science Department Technical Report Number 002, New York University, September 1978.

Sharir, M., "Interprocedural Analysis of Global Variable Usage", 1979, unpublished.

Sharir, M., "On Interprocedural Flow Analysis",SETL Newsletters nos. 187 April 1977, 187A May 1977.

Spillman, T., "Exposing Side-Effects in a PL/I Optimizing Compiler", IFIPS Proceedings 1971.

Tarjan, R. E., "Iterative Algorithms for Global Flow Analysis", Computer Science Department Technical Report, Stanford University, February 1976.

Tarjan, R. E., "Testing Flow Graph Reducibility", Journal of Computer and System Sciences, Vol. 9, 1974, pp. 355-365.

Tenenbaum, A., Type Determination for Very High Level Languages, Ph.D. thesis, Courant Institute in New York University,October 1974.

Weihl, W., "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables", unpublished summary of POPL 1980 paper.