THE DESIGN OF A DATA FLOW ANALYZER

Anita L. Chow
Andres Rudmik
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, Massachusetts 02254

ABSTRACT

This paper presents an efficient inter-procedural data flow analysis algorithm for precisely determining aliases in programs that employ a rich set of parameter passing mechanisms and pointer data types. This approach handles the use of pointers bounded to a data type as in Pascal, as well as unbounded pointers that can point to the same locations to which variables map. In the last step of this approach, the alias information is used to compute data flow information that is required for optimization.

INTRODUCTION

The optimization of large modular programs presents many problems. Typically, separate compilation forces the data flow analyzers to make worst case assumptions at call sites to procedures outside of .the compilation unit. Another problem is that a global inter-procedural data flow analysis must be performed in order to perform effective optimization of these programs. This inter-procedural data flow analysis is greatly complicated by various parameter passing mechanisms and through the use of pointers. The first problem has been solved through the use of a compiling system technology that utilizes a database to store the intermediate code for the entire program [1]. The solution to the second problem is the primary thrust of this paper.

A major complication to the gathering of precise data flow summary information is created when the same memory location can be referred to by different names. These names are called aliases.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-074-5/82/006/0106 $00.75

There are two ways in which aliases can be created. First, aliases are created when the reference parameter-passing mechanisms involved in a procedure call cause two distinct variables to map to the same storage location at the same time. Second, the use of pointer data types also complicates the picture by introducing aliases of variables even in the absence of procedure calls.

In this paper, we describe the design of an inter-procedural data flow analyzer that takes aliases into consideration when computing the data flow summary informaton. In this approach, the presence of aliases in programs that use reference parameters and pointer variables can be precisely determined. If aliases are present, this information is propagated through a program graph to compute procedure side effects.

PREVIOUS WORK

It is well recognized that in order to perform effective program analysis, one must be able to compute the side effects of all call sites and the aliases of all variables in a program. In the past, several inter-procedural data flow analysis algorithms have been developed using different computational strategies to compute data flow information to varying degrees of precision.

An early paper by Spillman [2] describes an inter-procedural data flow analysis technique for computing the "modifies" and "exit" side effects and the aliasing of names. In this approach, an "Expose Matrix" is built, which has a row for each variable, interrupt, and procedure invocation, and a column for each expression whose value may be changed indirectly when a value is assigned to one of the row variables or when an interrupt or procedure call occurs. This matrix can be constructed by a single pass over the program.

The Expose Matrix is then analyzed in two steps. First, the procedures are examined in reverse invocation order and the data variables which have been modified are calculated at each call site, using only the aliases created by the reference parameters at the site. This information is merged as one examines all the procedures called before processing the calling

procedure. If a program has recursive procedures, then iteration is required to compute the side effects. The second step in Spillman's approach is to process the procedures in invocation order, computing the aliasing effects by propagating alias information from the call sites to the called procedures.

There are several shortcomings to this approach. First, the computation of the alias information is imprecise. Second, this approach does not take into account the scoping of variables and the use of the same names in different scopes. A third disadvantage is that the use of the Expose Matrix leads to a space- and time-inefficient algorithm that would be inappropriate for very large programs.

The primary advantages of Spillman's approach are that the Expose Matrix can be built in one pass over the program, and that his techniques can handle a wide range of language features.

The inter-procedural data flow analysis algorithms developed by Allen [3] extract side effect information that is different in kind from that obtained by Spillman. Allen calculates the set of statements in a procedure which modify a variable whose value will be available on exit from a procedure; the set of expressions that use variables that exist at the call site; and the set of variables that may not be defined within the procedure. This kind of information would be useful for performing cross-procedural optimization.

Allen's algorithm propagates information by processing the procedures in reverse invocation order. Each procedure is subjected to global data flow analysis and data flow information is associated with each statement. The premise of her approach is that if the side effects of all called procedures are known, we can determine the side effects of the calling procedure.

Allen and Schwartz [4] have extended this algorithm to handle recursion by making worst case assumptions and then improving on these assumptions through iteration. This approach requires the reprocessing of the set of recursive procedures and can be quite expensive. Rosen [5] shows that a more precise result can be obtained, but at the added expense of more complex analysis. Lomet [6] handles recursive procedures with less precision than Rosen, but still analyzes each procedure only once.

Barth [7] presents a means of computing the "modification" and "reference" side effects for procedures in terms of the data flow relationship that can be computed directly from the local analysis of procedures. He gathers this information in a single pass over the program. His method is easy to implement, and is quite efficient, but is imprecise in the presence of reference parameters in that he does not treat different calls on the same procedure separately.

The effect of pointer assignments in data flow analysis problems has been studied to varying degrees. In the extreme case, this particular problem has been completely ignored. Banning [8] has addressed this problem, but his solution is not entirely satisfactory. He does not distinguish among variable locations pointed to by different pointer variables of the same type. Moreover, he does not treat the problematic case in which pointer variables point to the same locations to which variables map. Weihl [9] explicitly handles pointer variables, but his algorithm is imprecise. This imprecision stems from the fact that Weihl's algorithm does not handle different calls on a procedure properly.

In this paper, we present a data flow analysis algorithm which includes consideration of aliases. Our approach for precisely determining aliases is based on a program model that handles the two aliasing mechanisms in a unified manner.

ENVIRONMENT

The data flow analyzer described in this paper is a component of the CHILL Compiling System (CCS) [1] now being developed at GTE Laboratories. The CCS is designed specifically to support extensive static program analysis in that the intermediate code is stored in a program database making it readily available for further processing. The expected applications will be block-structured programs employing nested procedures; recursion; pass-by-reference, pass-by-value, pass-by-result, and pass-by-value-result parameters, and two kinds of pointer assignments:

(1) a := b, where a and b are pointers to a given type,
(2) a := ADDR(b), where a is a pointer to the same type as b.

The intermediate code (IC) produced by the CCS is a high level flow graph that directly models the control flow of the program. The program graph to be used in the data flow analyzer can be easily extracted from the IC. This IC has been used to build optimizers as described by Lepage [10].

ORGANIZATION OF THE DATA FLOW ANALYZER

The data flow analysis procedure consists of two passes through the intermediate code. In pass 1, alias information is determined. In pass 2, using the alias information, data flow information is determined.

The data flow analyzer consists of three key components that perform the following tasks: alias computation, intra-procedural data flow computation, and inter-procedural data flow computation. These tasks will be described in the following sections. Figure 1 depicts the information flow within the data flow analyzer.

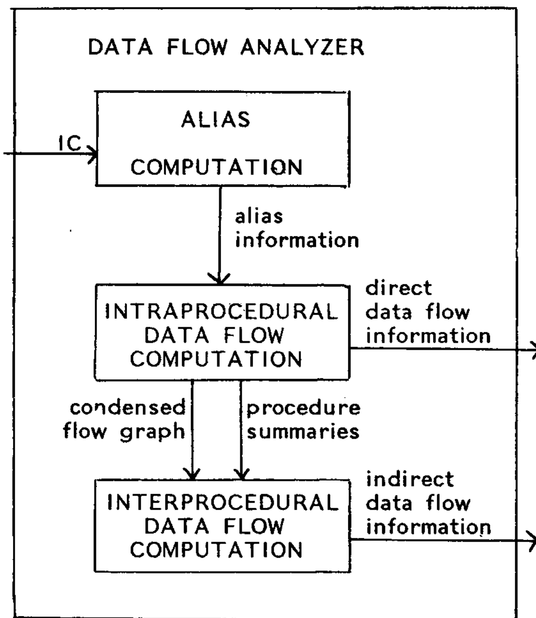


Figure 1. Data flow diagram of the data flow analyzer.

ALIAS COMPUTATION

The alias computation task consists of two steps: (1) constructing a super graph; and (2) determining the alias propagation.

Super Graph

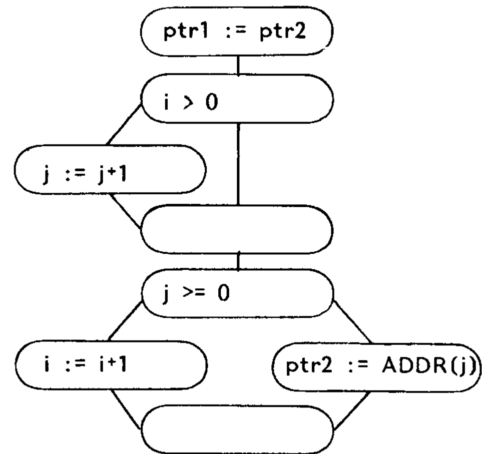
The super graph which models the reference parameter passing mechanism and pointer assignments in a unified manner is constructed as follows. First, a condensed flow graph with unique entry and exit nodes is constructed for each procedure by traversing the intermediate code and extracting only those control structures which contain call sites or pointer assignments. Note that the condensed flow graph contains only that information which is necessary for determining alias propagation. As an illustration of this step, consider Figure 2 which presents

a) a program segment,
b) the intermediate code generated for that segment, and
c) the condensed flow graph derived from the intermediate representation.

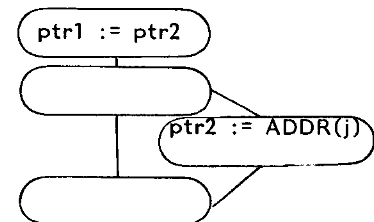
Next, these condensed flow graphs are linked together to form a super graph. Consider a call site s that is calling procedure p which has the following formal parameters: a is a pass-by-reference non-pointer parameter, b is a pass-by-reference pointer parameter, c is a pass-by-value pointer parameter, d is a pass-by-value-result parameter, and e is a pass-by-result parameter. We may write procedure p with the following Pascal-like notation:

```
ptr1 := ptr2;

IF i>0 THEN
   j := j+1;

IF j>=0 THEN
   i := i+1
ELSE
   ptr2 := ADDR(j);
```

(a) program



(b) intermediate code



(c) condensed representation

Figure 2. The condensed representation of a program.

```
/* T1, T2, T3, T4, and T5 are previously
   defined types */
PROCEDURE p (REFERENCE a :  T1;
             REFERENCE b : @T2;
             VALUE c : @T3;
             VALUE-RESULT d : @T4;
             RESULT e : @T5);[1]
BEGIN
 .
 .
 .
END;
```

[1]This order of parameters is chosen for explanatory purposes only. The parameters can appear in any order chosen by the programmer.

Let BIND(s,x) denote the actual parameter bound to the formal parameter x by call site s.

The procedure can then be represented as:

```
PROCEDURE p;
     VAR  a' :  @T1;
          b' : @@T2;
          c  :  @T3;
          d  :  @T4;
          e  :  @T5;

BEGIN
          a' := ADDR(BIND(s,a));
          b' := ADDR(BIND(s,b));
          c  := BIND(s,c);
          d  := BIND(s,d);
             .    } replace all occurrences
             .    } of a with a'@ and
             .    } b with b'@.
          BIND(s,d) := d;
          BIND(s,e) := e;
END;
```

The call site s is represented as two chains of nodes, the prologue chain and the epilogue chain (see Figure 3).

The head of the prologue chain is the entry node of p and the remaining nodes correspond to pointer assignments due to reference, value and value-result parameter passing. All incoming edges to s in the original condensed flow graph are new incoming edges to the head of the prologue chain and all outgoing edges from the entry node in the condensed flow graph of p are outgoing edges from the tail of the chain. The tail of the epilogue chain is the exit node of p and the remaining nodes correspond to pointer assignments due to result and value-result parameter passing. All outgoing edges from s in the original condensed flow graph are outgoing edges from the tail of the epilogue chain and all incoming edges to the exit node in the condensed flow graph of p are incoming edges to the head of the chain. Figure 3 depicts the representation of call site s. In this model, there is no difference in handling recursive calls and simple calls.

We conclude this section by categorizing different nodes in the super graph:

(1) entry nodes,
(2) exit nodes,
(3) assignment nodes of the form a := b, where a, b are pointer variables,
(4) assignment nodes of the form a := ADDR(b), where a is a pointer variable.
(5) control structure nodes containing type (3) or type (4) nodes.

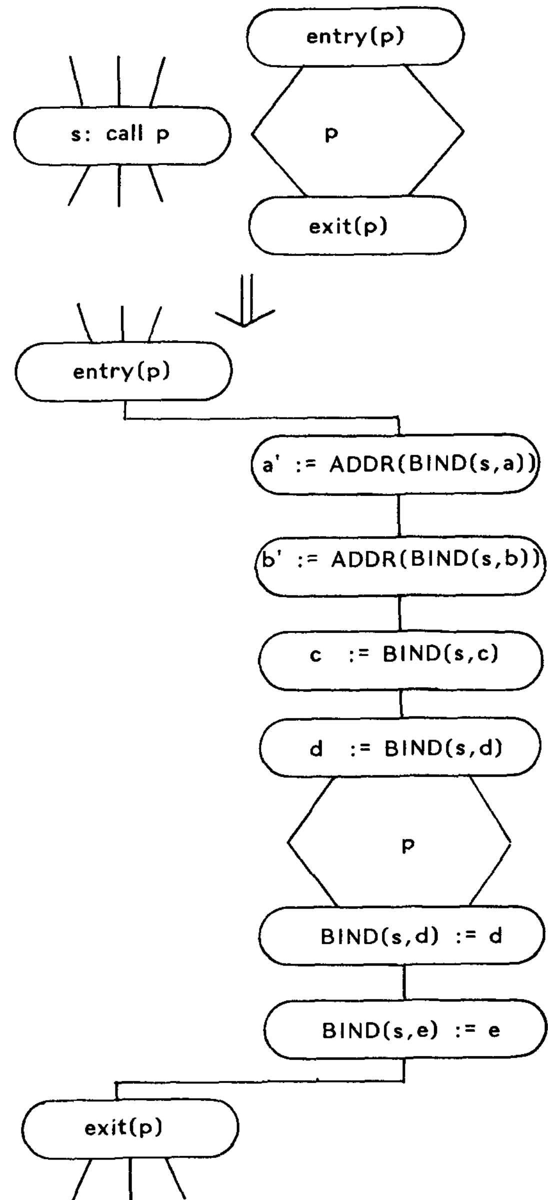


Figure 3. Representation of call site s.

Alias Propagation

We now present a precise method for finding all possible aliases of each variable after execution of a node in the program graph. Before proceeding with this method, we shall define the terms which are crucial to the discussion.

Within the definition of a procedure p, a set of local variables LOCAL(p) is declared. This set includes, FORMAL(p), the formal parameters of

p. The set of variables declared in all ancestors of procedure p are global to p; and it is denoted by GLOBAL(p).

With each pointer variable a in LOCAL(p), we associate a target variable *a which is the storage location accessible through a.

Finally, two variables are aliases at some point in the execution of a program if they are mapped to the same location at that point.

The entity describing the alias relations before or after execution of a node n is an alias fact X which is the set $\{ALIAS(v) \mid v \in V\}$ where V is a set of visible variables, ALIAS is a mapping from V to 2^V and ALIAS(v) is a set of variables which are alias to v before or after execution of the node n. We define a binary operation $^+$ on alias facts: given two alias facts $X_i = \{ALIAS_i(v) \mid v \in V_i\}$ and $X_j=\{ALIAS_j(v) \mid v \in V_j\}$, $X_i{}^+X_j$ is another alias fact $X_k=\{ALIAS_k(v) \mid v \in V_k\}$ such that:

$$ALIAS_k(v) = \begin{cases} ALIAS_i(v) \cup ALIAS_j(v) & \text{for } v \in V_i \cap V_j \\ ALIAS_i(v) & \text{for } v \in V_i - V_j \\ ALIAS_j(v) & \text{for } v \in V_j - V_i \end{cases}$$

and $V_k=V_i \cup V_j$.

Associated with each node in the graph is an aliases propagation function, f_n, which describes how the creation of aliases at a node n depends on the aliasing before entering the node and the type of node:

$$AFexit(n) = f_n\,(AFentry(n)),$$

where AFentry(n) and AFexit(n) are the alias information before and after the execution of node n.

Solving the alias problem entails finding the alias information before entering each node. The solution for this is as follows:

$$AFentry(n) = \sum_{m \text{ is a predecessor of } n} AFexit(m)$$

The problem of finding AFentry(n)'s can be formulated as an instance of a monotone data flow analysis framework [13].

Consider a monotone data flow analysis framework (L, $^+$, F), where L is the set of alias facts, $^+$ is the meet operator defined earlier and F contains operations on L which correspond to deriving AFexit(n) from AFentry(n). We will verify that F is a monotone operation space associated with L in the appendix. We define an instance (G, M) of this framework where G is the super graph of a program, and M:N->F is a function. Then the "meet over all paths" MOP solution to this framework is the same as AFentry(n)'s. There are a number of ways to find this MOP solution [11, 12]. It was determined that the iterative method of Kildall [12] with depth first ordering of nodes would be sufficient, yet simple and straightforward to implement.

We shall define F and M in the remaining of this section. We first show how the alias facts AFexit(n)'s of different types of nodes are derived from AFentry(n)'s and the nodes types.

Let $AFexit(n) = \{ALIAS'(v) \mid v \in V'\}$ and
$AFentry(n) = \{ALIAS''(v) \mid v \in V''\}$.

For n being the entry node of procedure P,
$V'' = V' \cup LOCAL(P) \cap GLOBAL(P)$

$$ALIAS''(v) = \begin{cases} ALIAS'(v) \cap GLOBAL(P) & \text{if } v \in V' \cap GLOBAL(P), \\ \{v\} & \text{if } v \in LOCAL(P). \end{cases}$$

For n being the exit node of procedure P,

$$V'' = V' - LOCAL(P).$$

$$ALIAS''(v) = ALIAS'(v) \cap GLOBAL(P).$$

For n being a node representing a := b,
a is a pointer variable,

$$V'' = V'$$

$$ALIAS''(v) = \begin{cases} ALIAS'(v) \cup \{*c \mid c \in ALIAS'(a)\} & \text{if } v \in ALIAS'(*b) \\ \{*c \mid c \in ALIAS'(a)\} \cup ALIAS'(*b) & \text{if } v \text{ is target var of a ptr var in } ALIAS'(a). \\ ALIAS'(v) & \text{otherwise.} \end{cases}$$

For n being a node representing a := ADDR(b),
a is pointer variable,

$$V'' = V'$$

$$ALIAS''(v) = \begin{cases} ALIAS'(v) \cup \{*c \mid c \in ALIAS'(a)\} & \text{if } v \in ALIAS'(b) \\ \{*c \mid c\ ALIAS'(a)\} \cup ALIAS'(b) & \text{if } v \text{ is a target var, of a ptr var in } ALIAS(a) \\ ALIAS'(v) & \text{otherwise.} \end{cases}$$

Now we can say that F contains operations f_n such that $M(n)=f_n$ and $f_n\ (AFentry(n)) = AFexit(n)$, where n is a node in the super graph. Operations in F are of the form:

$$f(X_i) = \{g_i(v) \mid v \in V_i \cup V'-V''\}$$

for all aliasing facts $X_i = \{ALIAS_i(v) \mid v \in V_i\}$, where g_i is an arbitrary function, $g_i: V \to 2^V$, depending on $ALIAS_i$, and V' and V'' are

constant sets of variables. For node n, the values of V_n', V_n'' and $g_{n,i}$ of $f_n(X_i)$ are as follows.

For n being the entry node of procedure P,

$$V_n' = LOCAL(P)$$
$$V_n'' = \phi$$
$$g_{n,i}(v) = \begin{cases} ALIAS_i(v) \cap GLOBAL(P) & \text{if } v \in V_i \cap GLOBAL(P) \\ \{v\} & \text{if } v \in LOCAL(P) \end{cases}$$

For n being the exit node of procedure P,

$$V_n' = \phi$$
$$V_n'' = LOCAL(P)$$
$$g_{n,i}(v) = ALIAS_i(v) \cap (V_i - LOCAL(P))$$

For n representing a := b where a and b are pointer variables,

$$V_n' = \phi$$
$$V_n'' = \phi$$
$$g_{n,i}(v) = \begin{cases} ALIAS_i(v) \cup \{*c \mid c \in ALIAS_i(a)\} & \text{if } v \in ALIAS_i(*b) \\ \{*c \mid c\ ALIAS_i(a)\} \cup ALIAS_i(*b) & \text{if } v \in \{*c \mid c \in ALIASi(a)\} \\ ALIAS_i(v) & \text{otherwise.} \end{cases}$$

For n representing a := ADDR(b) where a is a pointer variable,

$$Vn' = \phi$$
$$Vn'' = \phi$$
$$g_{n,i}(v) = \begin{cases} ALIAS_i(v) \cup \{*c \mid c \in ALIAS_i(a)\} & \text{if } v \in ALIAS_i(b) \\ \{*c \mid c\ ALIAS_i(a)\} \cup ALIAS_i(b) & \text{if } v \in \{*c \mid c \in ALIAS_i(a)\} \\ ALIAS_i(v) & \text{otherwise.} \end{cases}$$

INTRA-PROCEDURAL DATA FLOW COMPUTATION

The alias information obtained from the alias computation task described above is then used to compute intra-procedural data flow information, excluding consideration of the side effects of procedure calls. Specifically, this component of the data flow analyzer determines, for each procedure p, the side effects of statements including the consideration of aliases. As a by-product, this component also produces a condensed representation of procedure p, which contains only call sites, and

a summary of the direct side effects of p. This information facilitates inter-procedural data flow computation as discussed in the next section.

The algorithm is straightforward. It involves traversing the intermediate code tree of each procedure and obtaining the correct alias facts.

The preliminary version of the data flow analyzer in the CCS computes two flow insensitive side-effects [8]. They are:

DEF(n) which is the set of variables whose values may be changed by executing node n; and

REF(n) which is the set of variables whose values may be referenced by executing n.

The direct side effects are defined as follows.

$$DDEF(n) = \begin{cases} ALIAS(v) & \text{if n is an assignment to some variable v,} \\ \bigcup_{m \text{ nested in } n} DDEF(m) & \text{otherwise.} \end{cases}$$

$$DREF(n) = \begin{cases} ALIAS(v) & \text{if n is a leaf expression referencing some variable v,} \\ \bigcup_{m \text{ nested in } n} DREF(m) & \text{otherwise.} \end{cases}$$

Note that the direct side effects of call sites are null. The summary of the procedure p is just the union of the direct side effects of statements in p. They are:

$$IDEF(p) = \bigcup_{s \text{ is a statement in } p,} DDEF(s)$$

and

$$IREF(p) = \bigcup_{s \text{ is a statement in } p.} DREF(s)$$

INTER-PROCEDURAL DATA FLOW COMPUTATION

Using the intra-procedural data flow information computed with the precise alias information as described in the previous section, we then follow the general approach of Banning [8] and Spillman [2] to determine the side effects of call sites. For each procedure p, we shall find the side effects of p, from which the side effects of every call on p can be derived. This involves constructing the reverse of the call graph of the program with the information provided in the condensed flow graphs which were produced in the intra-procedural analysis phrase and propagating the side-effects of procedures.

The side effects of a procedure p arise from its own action and the side effects of the procedures called in p. The problem of finding these side effects can be formulated as an instance of a monotone data flow analysis framework. In this case, the flow graph is the

reverse of the call graph. We process the procedures in the reverse invocation order. When the side effects of a procedure p is known, for each procedure q which calls p, we compute the side effects of such calls and incorporate them in the direct side effects of q. We then delete procedure p and all the edges from p to q's from the flow graph.

The side effects of the procedure p are available when those of the procedures called in p are known. Thus , for a cycle-free flow graph, it will eventually be reduced to a single node (the main procedure). If the flow graph contains a cycle, we will end up with a subgraph containing procedures whose side effects are not available. To complete in solving this problem, we could use a reduction method like that of Graham [11]. This, however, is unnecessarily complex as these subgraphs tend to be very small. Therefore, we adopt a round-robin version of the iterative algorithm by Kildall [12].

We shall describe formally how to compute definition side effects of call sides and those of procedures; other flow insensitive side effects can be obtained in a similar manner. The set DEF(s) of a call side s which calls procedure p is:

$$DEF(s) = DEF(p) \cap GLOBAL(p) \cup BIND(s, y) \quad y \in DEF(p) \cap FORMAL(p)\}$$

And the definition side effects of a procedure is:

$$DEF(p) = (GLOBAL(p) \cup FORMAL(p)) \cap (IDEF(p) \cup DEF(s)) \quad s \text{ is a call site in } p$$

IMPLEMENTATION AND COMPLEXITY

In this section, we discuss some considerations concerning implementation of the data flow analyzer, a specific implementation that was chosen for the CCS, and its time and space complexities.

The most common way of representing sets of variables in data flow analysis is a vector of bits. When the sets are sparse, this representation becomes space inefficient. However, when the ratio of the size of the set to that of the universal set exceeds 1/50, the bit vector representation out-performs the linked list representation. Here, the universal set is the set of visible variables, of size #VISIBLE. For programs with extensive use of pointers, this ratio can be expected. Moreover, the advantage of bit vectors is that is is easy to perform set operations on sets. Therefore, the alias sets are represented as bit vectors of size #VISIBLE.

The alias fact is a set of #VISIBLE alias sets. Since there are exponential number of possible alias sets and the number of non-trivial alias sets in an alias fact is significantly smaller than #VISIBLE, the alias facts are represented as linked lists of bit vectors. Therefore, the space required by an alias fact is less than $(\#VISIBLE)^2$ bits. The number of alias facts computed by the analyzer is the sum of the number of call sites and the number of pointer assignments. Operations on alias facts require time O(maximum number of non-trivial alias sets in an alias fact).

SUMMARY

The inter-procedural data flow analyzer has been designed as a component tool of the CCS. The use of the program database allows us to perform a complete global data flow analysis without having to make worst case assumptions. The alias and side effect information is also stored in the program database to be used by various optimizers and is available as program documentation.

REFEERENCES

[1] Rudmik, A. and B.G. Moore, "The CHILL Compiling System: Towards a CHILL Programming Environment," Proc. Fourth Int. Conf. Software Engr. for Telecommunication Switching Systems (1981), 187-190.

[2] Spillman, T.C., "Exposing Side-Effects in a PL/I Optimizing Compiler," Proceedings IFIP Conference 1971, North Holland Publishing Co., Amsterdam, 376-381.

[3] Allen, F.E., "Interprocedural Data Flow Analysis," Proceedings IFIP Congress 74, North Holland Publishing Company, Amsterdam, 398-402.

[4] Allen, F.E. and J.T. Schwartz, "Determining the Data Relationships in a Collection of Procedures," Computer Science Report RC 4989, IBM Thomas J. Watson Research Center, (1974).

[5] Rosen, B.K., "Data Flow Analysis for PL/I Programs," Computer Science Report RC 5211, IBM Thomas J. Watson Research Center, Yorktown Heights (1975).

[6] Lomet, D.B., "Data Flow Analysis in the Presence of Procedure Calls," IBM Journal of Research and Development 21, 5 (1977), 559-571.

[7] Barth, J., "An Interprocedural Data Flow Analysis Algorithm," Conf. Rec. Fourth ACM Symposium on Principles of Programming Languages, (1977), 119-131.

[8] Banning, J.P., "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables," Conf. Rec. Seventh ACM Symp. Principles of Programming Languages (1980), 29-41.

[9] Weihl, W.E., "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables," Research Report RC8060, IBM T. J. Watson Research Center, Yorktown Heights, New York, January, 1980.

[10] Lepage, M.T., D.T. Barnard and A. Rudmik, "Optimization of Hierarchical Directed Graphs," Computer Languages 6 (1981), 19-34.

[11] Graham, S.L. and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," J. ACM 23, 1(1976), 172-202.

[12] Kildall, G.A., "A Unified Approach to Global Program Optimization," Conf. Rec. First ACM Symp. Principles of Programming Languages (1974), 194-206

[13] Hecht, M.S., Flow Analysis of Computer Programs, North-Holland Pub. Co., New York (1977).

APPENDIX

The following defines the monotone operation space mentioned in Section 5.0. The definition is from Hecht [13].

Definition. A monotone operation space is a set F of operations on some set L if and only if the following conditions are satisfied:

(1) The meet operator + distributes over every member f in F. That is $f(X_1 + X_2) = f(X_1) + f(X_2)$ for all X_1 and X_2 in L.

(2) There exists an identity operation e in F. That is there exists e F such that $e(X_1)$ for all X_1 in L.

(3) F is closed under composition. That is $f_1 o f_2$ F for all f_1 and f_2 in F, where $f_i o f_2(Xi) = f_1(f_2(Xi))$ for all X_i L.

(4) F is closed under +. That is $f_1 + f_2$ F or all f_1 and f_2 in F, where $f_i + f_2(Xi) = f_1(Xi) + f2(Xi)$ for all X L.

We must show that this definition is satisfied by the set of operations F in the monotone data flow analysis framework that is used to find aliasing facts on entering a node in the super graph. This set of operations are of the form:

$$f_1(X_i) = \{g_{1,i}(v) \mid V\varepsilon\ V_i \cup V'_1 - V_1"\}$$

for all aliasing facts $X_i = \{ALIAS_i(v) \mid v\varepsilon V_i\}$, where V_1' and $V_1"$ are constant sets, and $g_{1,i}$ is an arbitrary function, $g_{1,i} : V \to 2^V$, depending on f_i and $ALIAS_i$. F is shown to be a monotone operation space by demonstrating distributivity of +, closure under +, and closure under composition.

Distributivity of +.

$$f_1\ (X_i + X_j) = f_i(\{ALIAS_i(v) \cup ALIAS_j(v) \mid v\varepsilon\ V_i \cup V_j\})$$
$$= \{g_{1,i\cup j}\ (v) \mid v\varepsilon\ V_i \cup V_j \cup V_1' - V_1"\}$$

where $g_{i,i\cup j}$ is some function depending on f_i, and

$ALIAS_i \cup ALIAS_j$; while

$$f_1(X_i) + f_1(X_j) = \{g_{1,i}(v) \mid v\varepsilon\ V_i \cup V_1' - V_1"\}\ \{g_{1,j}(v) \mid v\varepsilon V_j \cup V_1' - V_1"\}$$
$$= \{g_{1,i}(v) \cup g_{1,j}(v) \mid V\varepsilon V_i \cup V_j \cup V_1' - V_1"\}$$

Therefore, $f_1(X_i + X_j)$ contains $f_1(X_i) + f_1(X_j)$.

Closure under +.

For function f_1, f_2, we show that

$$f_1(X_i) + f_2(X_i) = \{g_{3,i}(v) \mid v\varepsilon\ V_i \cup V_3' - V_3"\}$$

where

$$g_{3,i}(v) = g_{1,i}(v) \cup g_{2,i}(v) \text{ for all } v\varepsilon V.$$
$$V_3' = V_1' \cup V_2'$$
and $$V_3" = V_1" \cup V_2"$$

This is done as follows:

$$f_1(X_i) + f_2(X_i) = \{g_{1,i}(v) \mid V\varepsilon\ V_i \cup V_1' - V_1"\} + \{g_{2,i}(v) \mid v\varepsilon\ V_i \cup V_2' - V_2"\}$$
$$= \{g_{1,i}(v) \cup g_{2,i}(v) \mid v\varepsilon V_i \cup (V_1' \cup V_2') - (V_1" \cup V_2")\}$$

Closure under composition.

For function f_1 and f_2, we show that

$$f_1(f_2(Xi)) = \{g_{3,i}(v) \mid v\varepsilon V_i \cup V_3' - V_3"\}$$

where

$g_{3,i}(v) = g_{1,j}(v)$;

$g_{1,j}$ depends only on $ALIAS_i$, f_1 and $g_{2,i}$;

$V_3' = V_1' \cup V_2'$; and $V_3" = V_1" \cup V_2"$

$$f_1(f_2(X_i)) = f_1(\{g_{2,i}(v) \mid v\ V_i \cup V_2' - V_2"\})$$

Let $ALIAS_j(v) = g_{2,i}(v)$ for $v\varepsilon\ V_i \cup V_2' - V_2"$

Then,

$$f_1(f_2(X_i)) = f_1(\{ALIAS_j(v) \mid v\varepsilon\ V_i \cup V_2' - V_2"\})$$
$$= \{g_{3,j}(v) \mid v\varepsilon\ V_i \cup V_2' \cup V_1' - (V_2" \cup V_1")\}.$$