Data Flow Analysis of Applicative Programs^(*)

Micha Sharir
Department of Mathematical Sciences
Tel Aviv University
Ramat-Aviv, Tel Aviv 69978, Israel

Abstract: A precise and efficient data-flow analysis algorithm, for analysis of programs involving procedures with call-by-value parameter passing, is presented and analyzed. It is adapted to handle analysis of applicative programs written in a subset of LISP, but is equally suited to analyze programs written in many other languages, such as ADA, APL and SETL.

1. Introduction.

The bulk of data-flow analysis theory deals with intraprocedural analysis, i.e. it analyzes each procedure in the program to be analyzed in separation from the other procedures, and so concerns itself with data-flow materialized during execution of the static environment initiated at entry to the procedure under consideration. Interprocedural data-flow is usually either completely ignored in such an approach (by making worst-case assumptions concerning the effect of procedure calls on the data-flow), or else overestimated using a variety of techniques (See, e.g., [Bar], [Ban]). Moreover, most of these interprocedural techniques concern themselves with "summary" data-flow propagation from inner procedures to outer ones, but not the other way around. Additional problems caused by procedure variables and by the possible 'aliasing' of variables caused by call-by-reference transfer of parameters, make it difficult to obtain sharp interprocedural flow information (See Rosen [Ro₁] for such an attempt, although his approach deals only with "summary" data-flow analysis.) A recent study of Myers [My] overcomes all these difficulties, but the algorithm presented there is not very efficient, especially because the problem (in the presence of aliasing) is shown to be NP-complete. See also Jones [Jo] for a formal general approach to analysis of λ -expressions.

In this paper we will follow the recent interprocedural analysis technique proposed by Sharir and Pnueli [SP, section 3], and later refined by Schwartz and Sharir [SS]. This technique belongs to the class of elimination techniques discussed by Rosen [Ro₂], in that it

(*) Part of this work has been supported by ONR grant N00014-75-C-0571 while the author has been visiting at Courant Institute.

computes data-flow functions that summarize flow effects through progressively more complex program parts, and only then computes data-flow attributes by appropriate applications of these functions. This technique allows full integration of interprocedural analysis with intraprocedural analysis in a simple and natural manner, which also provides precise interprocedural information, and admits efficient realizations for the common subclass of 'bitvectoring' problems [SS].

However, the model assumed in [SP] and [SS] is too weak, mainly because it ignores the stacking/unstacking mechanism of recursive function calls, and is oriented toward sharper analysis of global variables. We will describe in this paper an appropriate generalization of this model which is capable of accurate tracing of data-flow realized by this mechanism, assuming conventional call-by-value binding strategy, which can also be dynamic.

In Section 2 we will present a generalized approach to data-flow analysis via elimination, using second-order functionals which operate on the space of relevant data-flow functions. This generalized approach will then be used in Section 3 to obtain a concrete algorithm for performing data-flow analysis of the bit-vectoring class on applicative programs written in a subset of LISP, which includes most of the standard LISP features, including simple side-effects (such as SETQ), but excludes certain features which make the analysis much more complicated, e.g. destructive operations (such as RPLACA, RPLACD), functional arguments, and intraprocedural branching operations (like the GO primitive). (Nevertheless, it is a relatively straightforward matter to modify our algorithm along the lines suggested in [Ro₃] or [SS] so that it can handle general flow patterns induced by GO. Our algorithm is based on the technical approach of Rosen [Ro₂] [ro₃] that analyzes a program by a simple tree-walk of its parse-tree. Section 4 contains an analysis of the algorithm, including a proof of its correctness. Concluding remarks follow in Section 5.

Our insistence on an applicative program model is not essential, and has been made to facilitate focussing on a single application domain which can benefit most from our approach. We can of course adopt our technique to a procedural program model, to obtain sharp interprocedural data-flow analysis algorithms which are refined versions of the algorithms given in [Ro₂], [SS] etc. In particular, programming languages such as SETL, APL, ADA, in which parameters are passed by value, seem most appropriate to benefit from our algorithm.

2. A Generalized Data Flow Framework.

Standard data-flow analysis theory makes use of the notion of an algebraic framework ([kil], [KU], [Ro₂]), defined as a pair (L, F) , where L is a well-founded meet-semilattice, with the meet operation denoted as $\wedge$, and where F is a space of monotone functions acting on L which include the identity map and which is closed under functional composition and functional (pointwise) meet, also denoted as $\wedge$. A program to be analyzed is represented by a flow graph G whose nodes are certain program sections, and whose edges indicate possible flow of control from one such section to another. This approach allows us to write data-flow equations that describe the relationships between data attributes at (entries of) adjacent program sections, and then to solve these equations to obtain a solution (in L) of the data-flow problem. A typical set of such equations has the following form

$$x_r \leq a_0 \in L$$

$$x_n = \wedge \{f_{(m,n)}(x_m) : (m,n) \in G\}, \quad n \text{ a node in } G$$

where $x_n \in L$ is the data value that we seek at entry to the program section n , where $f_{(m,n)} \in F$ is a data-flow function describing the change in attributes as control advances from the start of m , through m , to the start of n , and where $a_0 \in L$ is an initial data value describing the initial execution state at the entry r of the whole program. A systematic generalization of such equations can be found in [Ro₂].

If the space F has certain favorable properties, the above data-flow equations can be solved by elimination (See [Ro₂]), in which data flow functions are computed for progressively more complex flow patterns until the above equations become trivially solvable. However, use of elimination for interprocedural analysis faces a basic problem of having to assign a data-flow function to each 'basic' (or 'atomic') flow of control, including procedure calls (which are conventionally treated as being atomic in the calling procedure), and this is not a priori possible, especially in the presence of recursive calls. In this section we present an accurate interprocedural elimination technique, which follows the technique presented in [SP], [SS], but which uses a more realistic model for procedure calls. We begin by generalizing the notion of data-flow frameworks as follows: Define a second-order data-flow framework to be a triple (L, F, H) , where (L, F) is a conventional data-flow framework, where the space F is assumed to be a well-founded meet semilattice, with the pointwise functional meet

$\wedge$ taken as the semilattice operation, and such that (F, H) is also a conventional data-flow framework, that is H is a space of 'second-order' functionals operating on F that satisfies the standard requirements in a data-flow framework. Obviously, any data-flow framework (L, F) for which F is well-founded can be made into a second-order framework by adding to it a singleton space of functionals containing only the identity functional id_F .

As in [SS], we assume that each procedure p in the program is represented by a separate flow graph G_p having a unique entry node r_p and a unique exit node e_p . Each procedure call c is represented as a separate node of the flow graph of the calling procedure. In addition to the standard initial association of a data-flow function $f_{(m,n)} \in F$ with each edge (m,n) in the flow graphs, we also associate with each call node c a data-flow functional $h_c \in H$. These functionals are used to describe the flow through a procedure call in terms of the flow through the called procedure (see below). Also, a pair of data-flow functions $\tau_c, \rho_c \in F$ is associated with each such call c , to describe respectively the change in data as control enters the procedure called from c (such as caused e.g. by parameter binding, initialization of local variables etc.) and the change in data as control returns to c from the called procedure. Note that we do not assume that, given a call c to some procedure p , $h_c(f) = \rho_c \circ f \circ \tau_c$ (but see a concluding remark in section 5).

As in [SP], we formulate the data-flow equations to be solved in terms of data-flow functions (in F) rather than in terms of data states (in L). Specifically, for each procedure p and each node n of G_p , let $\varphi_n \in F$ denote the effect on data attributes of the advance of control from the entry of p to the start of n . Then we want to solve the following equations:

$$(2.1) \quad \varphi_{r_p} = \text{id}, \quad (\text{assuming no looping to } r_p \text{ occurs}),$$

$$\varphi_n = \wedge \{g_{(m,n)} \circ \varphi_m : (m,n) \in G_p\}, \quad \text{for each node } n \neq r_p$$

where

$$g_{(m,n)} = \begin{cases} f_{(m,n)}, & \text{if } m \text{ is not a call node} \\ h_m(\varphi_{e_q}), & \text{if } m \text{ is a call node, calling} \\ & \text{procedure } q \end{cases}$$

(Note that φ_{e_q} is the data-flow effect of executing the whole procedure q .) Our approach thus differs from [SP] in that it computes

the flow through a call to a procedure by some transformation (by a functional in H) of the inner flow within this procedure.

The theory of [SP, section 3] extends easily to this new set of equations. The basic results are (we omit the proofs):

Proposition 2.1: Equations (2.1) possess a maximal fixpoint which can be obtained by (a finite sequence of) successive approximations, starting with the following initial values:

$$\varphi_n = \begin{cases} \text{id,} & \text{if } n = r_p \text{ for some procedure } p \\ f_\Omega, & \text{otherwise} \end{cases}$$

where f_Ω denotes a special largest element in F , defined so as to map each $x \in L$ to a special largest element Ω in L .

The maximal fixpoint of (2.1) possesses certain properties. To formulate these properties we first introduce some notations.

Definition: A balanced interprocedurally valid path is an execution path π leading from the entry r_p of a procedure p to a node n of G_p , defined as an ordered tree, the root of which having as children (from left to right) the edges of G_p through which π passes and each edge from a call node c appearing in π is an inner node of the tree, such that the subtree rooted at that edge is a balanced interprocedurally valid path from the entry r_q of the procedure q called from c to its exit e_q . The actual program execution described by π is obtained by a left-to-right preorder walk of π . For each node n , let $\text{BIVP}(n)$ denote the set of all balanced interprocedurally valid paths leading to n from the entry of the procedure containing n . (A similar definition can be found in [Ro₁].) For each node n and each $\pi \in \text{BIVP}(n)$ we define $f_\pi \in F$ as follows: Let the children edges of the root of π be $[e_1, e_2, \dots, e_m]$; then

$$f_\pi = g_{e_m} \circ g_{e_{m-1}} \dots \circ g_{e_1}$$

where

$$g_e = \begin{cases} f_e & \text{(i.e. the data-flow map associated with } e\text{),} \\ & \text{if } e \text{ is not a call-edge} \\ h_c(f_{\pi_1}) & \text{if } e \text{ is a call edge going out of a call} \\ & \text{node } c, \text{ whose subtree is } \pi_1 \end{cases}$$

Proposition 2.2: If F is a distributive over L and H is distributive over F , then the maximal fixpoint of (2.1) satisfies, for

each node n ,

$$\varphi_n = \Lambda\{f_\pi : \pi \in \text{BIVP}(n)\}$$

Having computed the data-flow functions φ_n , we next return to the problem of computing an entry data-state (in L) x_n for each node n in the programs' flow graph. It suffices to compute x_{r_p} for all procedure entries r_p , since for any procedure p and any node $n \neq r_p$ of G_p we have

$$(2.3) \quad x_n = \varphi_n(x_{r_p})$$

The equations that the x_{r_p} 's should satisfy are as follows:

$$(2.4) \quad x_{r_{\text{main}}} = a_0 \in L \quad (\text{initial data state; } \underline{\text{main}} \text{ is the main program, and is assumed to be nonrecursive})$$

$$x_{r_p} = \Lambda\{\tau_c(\varphi_c(x_{r_q})) : c \text{ a call to } p \text{ from a procedure } q\}$$

It is easy to see that Equations (2.4) have a maximal fixpoint which can be computed by standard data-flow analysis methods (see, for instance, [SP] and [SS]). Moreover, this fixpoint has a property similar to that stated in [SP]. To formulate this property, let us first introduce the following

Definition: An interprocedurally valid execution path π leading to a node n is a concatenation of balanced interprocedurally valid paths $\pi_1 \pi_2 \dots \pi_k$ for which there exist corresponding sequences of procedures $p_1, p_2, \dots, p_k$ and of call nodes $c_1, c_2, \dots, c_{k-1}$, such that $p_1 = \underline{\text{main}}$; c_j is a node of G_{p_j} and calls p_{j+1} , for each $j < k$;

$\pi_j \in \text{BIVP}(c_j)$ for each $j < k$, and $\pi_k \in \text{BIVP}(n)$. π thus describes an execution flow from the entry of the main program up to some node n , and $c_1, c_2, \dots, c_{k-1}$ is the sequence of procedure calls initiated but not yet terminated during that execution. For each node n let $\text{IVP}(n)$ denote the set of all interprocedurally valid paths leading to n . Let $\text{CS}(\pi)$ denote the set of all calling sequences $c_1, c_2, \dots, c_i$ of the above form that corresponds to initial segments of π . Let $\pi \in \text{IVP}(n)$ for some node n . Define $f_\pi \in F$ as follows: Let π be decomposed as above; then

$$f_\pi = f_{\pi_k} \circ \tau_{c_{k-1}} \circ f_{\pi_{k-1}} \dots \circ \tau_{c_1} \circ f_{\pi_1}$$

Proposition 2.3: If F and H are distributive, then the maximal fixpoint of Equations (2.3) and (2.4) satisfies

$$x_n = \Lambda\{f_\pi(a_0) : \pi \in \text{IVP}(n)\}$$

for each node n . That is, our solution coincides with the 'meet over all interprocedurally valid paths' solution (compare with an analogous intraprocedural property stated e.g. in [He]).

Remark: The theory presented in this section can be easily extended to apply to backward data-flow analysis, such as live variables analysis. We will not concern ourselves however in this paper with this type of analysis and will continue to consider only forward data-flow problems.

3. Data Flow Analysis of Applicative Programs

Having stated a general approach to interprocedural analysis, we turn next to consider a specific application: data-flow of the 'bitvectoring' class for applicative programs of the LISP style. A data-flow framework of the bitvectoring class is defined as follows [SS]. Let E be some finite set of boolean attributes. Define $L = 2^E$ with the meet operation taken as set intersection. The space F is defined to be isomorphic to a subset of $L \times L$, so that each $f \in F$ is represented by a unique pair $[A_f, B_f] \in L \times L$ such that $A_f \supseteq B_f$ and for each $x \in L$

$$f(x) = (A_f \cap x) \cup B_f$$

It is well known that F contains the identity function id, is closed under functional compositions and meets, is distributive and idempotent.

Next we make this framework into a second-order one by augmenting it by a space H of functionals over F defined as an isomorphic image of L , such that for each $h \in H$ there corresponds an element $W_h \in L$, and for each $f \in F$, $h(f)$ is defined so that

$$[A_{h(f)}, B_{h(f)}] = [A_f \cup W_h, B_f - W_h]$$

It can easily be checked that for $\alpha \in E$

$$(h(f)(x))_\alpha = \begin{cases} x_\alpha, & \alpha \in W_h \\ f(x)_\alpha, & \alpha \notin W_h \end{cases}$$

Thus W_h denotes a set of attributes for which the effect of a procedure p on them is to be 'forgotten' at a call c to p ; the effect

of p on other attributes is left unchanged at c . This choice is well suited for handling (dynamic) call-by-value parameter binding. For example, in an analysis of reaching definitions [Hel], the set W_h representing a functional h associated with a call to a procedure p , can be taken to be the set of all definitions of variables appearing as formal parameters or local variables of p .

Lemma 3.1: (a) The space H contains the identity functional and is closed under functional compositions and meets.

(b) Each functional $h \in H$ is distributive over meet in F , and is therefore monotone increasing.

(c) Each $h \in H$ is idempotent.

(d) Each $h \in H$ is distributive over composition in F , that is

$$h(g \circ f) = h(g) \circ h(f) \text{ , } g, f \in F.$$

Next we describe the semantic program model assumed in this section, and its flow-graph representation. We consider LISP-like programs with the following features excluded: functional arguments, destructive operations (such as RPLACA) computed left-hand sides (such as SET) and explicit branches (such as GO). As mentioned earlier, the last restriction is the least essential, and is made to obtain a simple version of the algorithm, whereas admission of the first two restrictions would change significantly the whole nature of the algorithm. (Note however that side effects are allowed.) We assume conventional left-to-right call-by-value binding strategy.

Each program in our model is thus fully structured, with only the following control flow structures: recursion, conditional statements and sequencing. Such a program P is best represented by its parse tree which induces a hierarchical flow graph representation of P of the sort described by Rosen [Ro₂]. Specifically, each procedure p defined at the top level has its own parse tree T_p , and the main program consists of a sequence of invocations of these procedures. To each inner node n of such a parse tree there corresponds a control flow pattern which has a unique entry point and a unique exit point and which is defined in terms of the flow within the children of n . Call nodes are either nodes in which (user-defined) functions are invoked, or else nodes denoting a PROG block (so that they 'call' themselves).

Our data-flow analysis algorithm has the same structure as the algorithm given in [SS, section 4]. It consists of the following phases:

I. Call-graph analysis. As in [SS], the call graph CG of the program being analyzed is decomposed into its strongly-connected components $S_1, S_2, \dots, S_m$, which are topologically sorted in the reverse postorder of a depth-first spanning tree of CG. Procedures within each such component are also sorted in reverse post-order. In addition, the loop interconnectedness parameter d_S of S , defined as the maximal number of back-edges lying on an acyclic path contained in S , is estimated from above by the number d'_S of back-edge target nodes in S . This phase can of course be performed only once, prior to any data-flow analysis.

II. Elimination Phase. This phase essentially solves Equations (2.1), by iteratively processing procedures in the following way. (recursive procedures p may have to be (partially) reprocessed several times, until convergence of φ_{e_p} is achieved): Iterate once over all strongly

ly connected components S_i of the call graph in postorder; for each such S_i iterate over its procedures in postorder till convergence, but not more than $3d'_{S_i} + 1$ times, applying to each procedure the elimination process described below. Convergence (within S_i) is attained when all data-flow summary functions, $\varphi_{e_p}, p \in S_i$, do not change their values obtained at the preceding iteration. Thus each nonrecursive procedure p , has to be processed just once, whereas each simply recursive p may have to be analyzed 4 times. (An example where 3 times may not be sufficient is given in the next section.)

To process a procedure p , Walk the parse tree T_p of p . For each node n of T_p compute two data-flow functions $\psi_n, \hat{\psi}_n \in F$ such that ψ_n describes the data-flow through the flow subgraph induced by n , from its entry to its exit, and $\hat{\psi}_n$ describes the data-flow through the flow-graph induced by the parent of n in T_p , from the entry of this graph up to the entry of n . The computation of these functions is accomplished using a priori available formulae that depend on the nature of the control structure involved, and are rather similar to those given e.g. in [Ro₃]. For example, suppose that n is a function application and that its children are the function name f and its arguments $a_1 \dots a_k$. Then

$$\begin{aligned}\psi_n &= h_n(\varphi_f) \circ \psi_{a_k} \circ \dots \circ \psi_{a_2} \circ \psi_{a_1}; \\ \hat{\psi}_{a_j} &= \psi_{a_{j-1}} \circ \dots \circ \psi_{a_2} \circ \psi_{a_1}, \quad j = 1 \dots k;\end{aligned}$$

where φ_f is the (current value of the) data-flow function describing

the data-flow through f .

Additional reprocessings of recursive procedures p may be short-cut, e.g. by reanalyzing only call nodes whose flow-effect has changed since the last iteration, and ancestors of such nodes.

It will be shown in the next section that this process indeed yields the φ_n functions appearing in (2.1), and that in fact one has, for each node n whose ancestors in its tree are, in descending order, $r_p = n_1, n_2, \dots, n_k = n$,

$$(3.1) \quad \varphi_n = \hat{\psi}_{n_k} \circ \hat{\psi}_{n_{k-1}} \dots \circ \hat{\psi}_{n_2}$$

III. Entry-Data Analysis. In this phase Equations (2.4) are solved. To this end we first compute φ_c by the preceding formula, for each call node c , and then solve (2.4) using standard data-flow analysis techniques (e.g. the one given in [SS]).

IV. Propagation Phase. In this final step data is propagated from each procedure's entry to any node within it, using the auxiliary functions $\hat{\psi}_n$. This is done by a simple top down tree walk. For each node n visited, x_n is computed using the formula $x_n = \hat{\psi}_n(x_m)$ where m is the father of n , for which x_m has already been computed. This concludes the description of the algorithm.

4. Analysis of the Algorithm.

In this section we prove that the iteration order through procedures and its bounds are sufficient for all functions computed by the elimination phase of the algorithm given in Section 3 to converge to their final values. The proofs use the special structure of bitvectoring frameworks. As in [SS], we have

Lemma 4.1: (cf. [SS, Lemma 4.1]): Suppose that the elimination phase of our algorithm has already processed procedures $p_1, p_2, \dots, p_k$, and is now analyzing $p_{k+1} = p$. Let n be a node of G_p and $\Pi \in \text{BIVP}(n)$. If the reverse sequence of each calling sequence in $\text{CS}(\Pi)$ is a (not necessarily contiguous) subsequence of $p_1, p_2, \dots, p_k$, then, at the end of the current processing of p , we will have $f_\Pi \geq \varphi_n^{k+1}$, where φ_n^{k+1} denotes the current value of φ_n , as computed by (3.1).

Lemma 4.2: Let n be a node in some procedure and let $\Pi \in \text{BIVP}(n)$. (a) Suppose that some calling sequence in $\text{CS}(\Pi)$ contains a procedure p four times. Then

$$f_\Pi \geq f_{\Pi_1} \wedge f_{\Pi_2} \wedge f_{\Pi_3}$$

where Π_i is a subpath of Π obtained by executing the i -th call to p in the way the $(i+1)$ -th call is executed, thereby skipping one level of recursion (Figure 4.1 shows the tree representation of Π , Π_1 , Π_2 and Π_3 .)

Fig. 4.1 - Path shortening

(b) Suppose that some calling sequence in $CS(\Pi)$ contains two occurrences of one procedure p followed by three occurrences of another procedure q , or the other way around. Then

$$f_{\Pi} \geq f_{\Pi_1} \wedge f_{\Pi_2} \wedge f_{\Pi_3}$$

where Π_1 is the subpath of Π obtained by executing the first call to p as the second call to p , where Π_2 is the subpath obtained by executing the first call to q as the second one, and where Π_3 is obtained by executing the second call to q as the third one.

(c) Finally, suppose that some calling sequence in $CS(\Pi)$ contains two occurrences of a procedure p , followed by two occurrences of a second procedure q , followed by two occurrences of a third procedure r . Then

$$f_{\Pi} \geq f_{\Pi_1} \wedge f_{\Pi_2} \wedge f_{\Pi_3}$$

where Π_1 is the subpath obtained by shortcutting Π between the two calls to p , Π_2 is the subpath obtained by shortcutting Π between the two calls to q , and Π_3 is obtained by shortcutting Π between the two calls to r .

Proof: Omitted.

Define a tripled node listing of the call graph CG to be a sequence S of procedures with the property that each path in CG which (excluding its initial node) is a concatenation of three acyclic paths, is a subsequence of S .

Theorem 4.3: Let S be any tripled node listing for the call graph CG, and suppose that the elimination phase of the algorithm given in Section 3 processes procedures in the reverse of their order in S . Then, at the end of one iteration through S , all computed data-flow functions will have their final desired value.

Proof: Let $S = [p_1, p_2, \dots, p_k]$. It follows easily from lemma 4.1 that at the end of one iteration through S we have, for each node n ,

$$(4.1) \quad \varphi_n = \Lambda \{ f_{\Pi} : \Pi \in \text{BIVP}(n) \text{ such that each calling sequence in } \text{CS}(\Pi) \text{ is a subsequence of } S \}$$

It therefore suffices to prove that for each $\Pi \in \text{BIVP}(n)$, f_{Π} is greater than or equal to the right-hand side of (4.1). Let such a Π be given, and assume that it does not appear in (4.1), i.e. there exists a calling sequence C in $\text{CS}(\Pi)$ which is not a concatenation of three acyclic paths in CG. It is easily checked that this implies that at least one of the following cases occurs:

- (a) C contains four occurrences of a procedure p ;
- (b) C contains three occurrences of a procedure p followed by two occurrences of another procedure q ;
- (c) C contains two occurrences of a procedure p followed by three occurrences of another procedure q ;
- (c) C contains two occurrences of a procedure p followed by two occurrences of another procedure q followed by two occurrences of a third procedure r .

In any of these cases we can apply lemma (4.2) to obtain an inequality

$$f_{\Pi} \geq f_{\Pi_1} \wedge f_{\Pi_2} \wedge f_{\Pi_3}$$

for three shorter paths $\Pi_1, \Pi_2, \Pi_3 \in \text{BIVP}(n)$. The path-shortening argument can then apply to these paths repeatedly until an inequality

$$f_{\Pi} > \bigwedge_{\xi} f_{\xi}$$

is obtained, in which all paths ξ appear in the right-hand side of (4.1). This proves the theorem.

Remark: If the meet in (4.1) is empty, $\varphi_n = f_{\Omega}$. This indicates that n is unreachable, e.g. because any attempt to reach n involves an infinite recursion.

As a corollary we obtain immediately a correctness proof for our algorithm.

Theorem 4.4: The iteration order through procedures used in the algorithm given in Section 3 is the reverse of a tripled node listing for the call graph. Consequently, the elimination phase of this algorithm yields the final desired values of all data-flow functions computed there.

Proof: This iteration order is in fact the reverse of the sequence

$$S = \bigwedge_{i=1}^m (3d! S_i + 1) * S_i$$

(where summation stands for concatenation, and multiplication by an integer stands for replication). The proof that S is a tripled node listing is then completely analogous to that of lemma 4.5 and theorem 4.6 of [SS].

Remark: Comparing the performance bounds of our algorithm with similar bounds for the algorithm given in [SS] for their weaker model, we see that our algorithm requires extra processings of recursive procedures. For instance, in the case of a simple recursive cycle ($d = 1$), our algorithm may require four processings of each procedure in that cycle to attain convergence, whereas the algorithm of [SS] does not require more than three processings. The following example shows that three processings may not be sufficient for our algorithm to converge.

Example: Consider an available expressions analysis, analyzing the availability of the single expression $e = a + a$. Let p and q be two mutually recursive procedures such that p is parameterless and q has a as its single formal parameter. Suppose that p and q have the following structure

Fig. 4.2.

where all unmarked edges do not affect the availability of e . In this case, execution of p may result in e being killed. However, this fact is rather subtle and the shortest execution path that demonstrates this behaviour is the following one:

Start p via c_1 (thereby making e unavailable); start p via c_2 ; start q (via c_4); execute p via the center edge (This makes e available, but it is a different incarnation of e , since it refers to the value of a bound to it at the invocation of q , and not to its original value.); finish q and return to c_2 . (This restores the status of e before entering q); finish p and return to c_1 ; finish p . (Thus e gets killed along this path.)

To trace this path three processings of p are required. The values of φ_{e_p} at the end of each processing will be $\underline{1}$, $\underline{id}$ and $\underline{0}$ respectively. (Note that in this case we choose $h_{c_1} = h_{c_2} = h_{c_3} = h_0$,

since these calls do not involve any binding/unbinding effects, and $h_{c_2} = h_1$, since this call will neutralize any effect on e caused by q .) A fourth processing is then required to propagate the final effect of p to nodes such as n in Fig. 4.2. (Here $h_0 = id_F$, and h_1 maps each f to id .)

5. Conclusion

The contributions of this paper are threefold:

(a) A general data-flow analysis technique for a relatively strong

model of procedural and applicative programs, using a new notion of second-order data-flow frameworks. This technique has the advantage of being uniform in the sense that it performs inter-procedural analysis simultaneously with intraprocedural analysis, and does not require a separate information-gathering pass such as required by some former techniques ([Ban], [Bar]). Our technique can take proper account of the procedure call mechanism and is shown to yield sharp information.

(b) A special-purpose elimination algorithm derived from the general technique. This algorithm performs data-flow analysis of the bitvectoring class for applicative programs having LISP-like semantics.

(c) A detailed analysis of the algorithm given in (b), which demonstrates some of the complications that interprocedural flow may create, even for the simple bitvectoring frameworks.

Remarks: 1) We have not discussed applications of our techniques to program optimization. This is discussed in an expanded version of this paper. For instance, a recent result of Pettorossi [Pe] on space optimization of applicative programs is obtained as an application of live-variable analysis using our approach. Other applications such as common subexpression elimination, including expressions involving function calls, are also discussed.

2) It is perhaps surprising that we had to use second-order functionals to capture the data-flow effect of procedure calls. Indeed, using the theory of Cousot and Cousot [CC], data-flow frameworks can be viewed as coarse homomorphic images of a fully detailed algebraic structure in which program states and state transitions are recorded in detail. In this structure the effect of a procedure call can be expressed by composition of the flow effect of the procedure body with the actions done at the beginning and at the end of the call. However, the homomorphic image of this transformation can not in general be expressed in a similar manner, because of the coarse approximative structure of a data-flow framework. Hence the need for second-order functionals (and possibly for more complex functionals in subtler cases) to express the behaviour of calls in a data-flow context.

REFERENCES

- [AU] A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley 1977.
- [AC] F.E. Allen and J. Cocke, "A Program Data-Flow Analysis Procedure", CACM 19(1976), 137-147.
- [Ban] J.P. Banning, "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables", Proc. 6th POPL Conference (1979), 29-41.
- [Bar] J. Barth, "A Practical Interprocedural Data-Flow Analysis Algorithm", CACM 21(1978), 724-736.

- [CC] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks", Proc. 6th POPL Conference (1979), 269-282
- [GW] S.L. Graham and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis", JACM (1976), 172-202.
- [He] M.S. Hecht, "Flow Analysis of Computer Programs", Elsevier North-Holland 1977.
- [Jo] N.D. Jones, "Flow Analysis of Lambda Expressions", This proceedings
- [KU] J.B. Kam and J.D. Ullman, "Monotone Data Flow Analysis Frameworks", Acta Informatica 7(1977), 309-317.
- [Ki] G.A. Kildall, "A Unified Approach to Global Program Optimization", Proc. 1st POPL Conference (1973), 194-206.
- [My] E.W. Myers, "A Precise Interprocedural Data Flow Algorithm", Proc. 8th POPL Conference (1981), 219-230.
- [Pe] A. Pettorossi, "An Algorithm for Reducing Memory Requirements in Recursive Programs Using Annotations", Proc. International Workshop on Program Construction, Bonas, France (1980).
- [Ro₁] B.K. Rosen, "Data-Flow Analysis for Procedural Languages", JACM 26(1979) 322-344.
- [Ro₂] B.K. Rosen, "Monoids for Rapid Data-Flow Analysis", SIAM J. Computing 9(1980), 159-196.
- [Ro₃] B.K. Rosen, "High-Level Data Flow Analysis", CACM 20(1977), 712-724.
- [SS] J.T. Schwartz and M. Sharir, "Design for Optimizations of the Bitvectoring Class", Courant Comp. Sci. Rept. #17, New York University 1979.
- [SP] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Data-Flow Analysis", to appear in "Program Flow Analysis", (S.S. Muchnick, N.D. Jones Eds.) Prentice-Hall, 1981.