A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures

Neil D. Jones

Department of Computer Science
Aarhus University
DK-8000 Aarhus C, Denmark

Steven S. Muchnick

Computer Research Center
Hewlett-Packard Company
1501 Page Mill Road, Bldg. 28B
Palo Alto, California 94304

ABSTRACT

A new approach to data flow analysis of procedural programs and programs with recursive data structures is described. The method depends on simulation of the interpreter for the subject programming language using a retrieval function to approximate a program's data structures.

1. Introduction

In this paper we present a new approach to data flow analysis of programs with recursive data structures and an application of the method to interprocedural flow analysis. The basic approach is similar to that used in the first part of [JoM81] to analyze LISP-like structures, but is significantly more flexible and economical. It depends on the use of tokens to designate the points in a program where recursive data structures are created or modified (and hence to approximate their values) and a retrieval function to finitely represent the interrelationships among the tokens and data values.

In the application to interprocedural flow analysis, we consider an interpreter for a recursive programming language (specialized to execute any particular program) as a program with a recursive data structure, namely the stack of procedure activation records. The great flexibility in the choice of token sets and lattices to approximate the data values of the programming language give us a tremendous range of degrees of exactness of the resulting data flow information. Thus we can produce anything from summary data flow information [Bar78] to exact (though not effectively computable) computation descriptions. The choice of basing the approach on an interpretive model of program semantics makes it comparatively easy to flow analyze such nonstructural features as goto's and multilevel escapes from procedures, in contrast to the contortions necessary if denotational semantics is used.

In section 2 we discuss the general topic of flow analyzing programs with recursive data structures and in sections 3 & 4 we give an example which applies our basic approach to a language which manipulates binary lists. Section 5 begins the discussion of interprocedural flow analysis by presenting a simple model language with recursive procedures and multilevel escapes from procedures. Section 6 presents a general flow analyzer for the example language which may be parameterized to analyze programs for various properties and to varying degrees of exactness. Section 7 gives some examples of specific analyzers constructed from the general one and applications of them to some example programs. Section 8 gives some remarks on making the approach practical. Finally, section 9 gives some conclusions and suggestions for further work in this area.

1. Programs With Recursive Data Structures

In this section we provide an intuitive introduction to the approximation method by showing how to flow analyze a simple flowchart language which manipulates binary lists. For each input program the analysis method (implicitly) produces, for each list-valued variable λ and each program point q, a regular set of binary trees which is guaranteed to include every value which λ may assume at point q during execution.

A related and more formalized approach to approximating programs of this sort using extended regular tree grammars is found in [JoM81]. The method developed here is more powerful in that it yields an ordinary regular tree grammar with fewer nonterminals (namely, one per constructor operation, rather than one per program point), and provides a natural way to model the atomic data manipulated by the program, as well as to obtain descriptions of the list values. The method is described here in an intuitive way by means of an example. Sections 5 through 7 contain a more rigorous development of the method in the context of interprocedural flow analysis and [Jon80] applies the method to flow analysis of the lambda calculus. The method is parameterized by the choice of the set of *tokens*, which can be varied so that the result of the flow analysis ranges from an exact (but not always effectively obtainable) description of the set of program computations through less exact but practically more useful descriptions. The computational complexity of the method also varies with the choice of token set, increasing with the exactness of the descriptions obtained. In all cases the description is "safe", meaning informally that every reachable program state is accounted for in the description (a formal definition is given below).

Most flow analysis methods are (more or less formally) based on an abstract interpretation of the program to be analyzed - that is to say, the program is in essence executed over abstract data sets which model the concrete data used in actual executions. Papers expressing this view explicitly include [Sin72], [Cou77a] and [Cou77b]. For example, [Cou77a] analyzes a flowchart program whose variables range over a set A of atomic values by first defining the *static semantics* of the flowchart. This maps each control point q (= arc in the flowchart) into the set of environments $E(q)$ which may hold when control reaches that point (an environment is a function e mapping program variables to values). Flow analysis is then viewed as the problem of effectively obtaining a safe approximate description of $E(q)$ for each arc q. (A great many flow analysis problems may be viewed in this way, but not all — "history-sensitive" problems such as available expressions require more elaborate machinery to formalize.)

Typically the collection $P(E)$ of all sets of environments is viewed as a lattice with subset inclusion as its ordering and is modeled by a smaller lattice A' which is usually required to satisfy the finite chain condition so that approximations can be obtained in finite time. Notice that from this viewpoint the effect of several flowchart arcs converging at a node is most naturally modeled by the lattice join operation $\sqcup$ in contrast to [Hec77] and [Kil73]'s use of lattice meet.

The approximation of lattice L_1 by lattice L_2 is naturally described by an *abstraction* function $abs: L_1 \to L_2$ and a *concretization* function $conc: L_2 \to L_1$. Cousot [Cou79] has given arguments, strengthened by Nielson [Nie80], that for the purposes of flow analysis $<abs, conc>$ should be an *adjoined* pair of functions, i.e. for all $l_1 \in L_1$ for all $l_2 \in L_2$

$$abs(l_1) \sqsubseteq l_2 \text{ if and only if } l_1 \sqsubseteq conc(l_2)$$

We will use two approximation lattices in our examples, one for parity checking and the other for constant propagation. The lattice A_{eo} of Figure 1 can be used to approximate the parity of sets of integers with the aid of the adjoined pair of functions $abs: P(N) \to A_{eo}$ and $conc: A_{eo} \to P(N)$ defined by

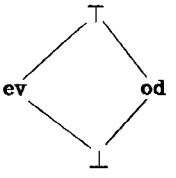


Figure 1. Approximation Lattice for Parity Checking

$$abs(X) = \begin{cases} \bot & \text{if } X = \phi \\ \textbf{ev} & \text{if all } x \in X \text{ are even} \\ \textbf{od} & \text{if all } x \in X \text{ are odd} \\ \top & \text{otherwise} \end{cases}$$

and

$$conc(X') = \text{if } X' = \bot \text{ then } \phi \text{ elif } X' = \textbf{ev} \text{ then } \{0,2,4,6,\ldots\} \text{ elif } X' = \textbf{od} \text{ then } \{1,3,5,7,\ldots\} \text{ else } N$$

The lattice A_{con} in Figure 2 can be used for constant propagation. Suitable abstraction and concretization functions are $abs: P(N) \to A_{con}$ and $conc: A_{con} \to P(N)$ defined by

$$abs(X) = \text{if } X = \phi \text{ then } \bot \text{ elif } X = \{a\} \text{ then } a \text{ else } \top$$

$$conc(\bot) = \phi,\ conc(a) = \{a\},\ conc(\top) = N$$

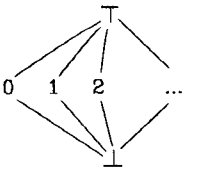


Figure 2. Approximation Lattice for Constant Propagation

Suppose we are given an n-ary operation $op: A^n \to A$ and a lattice A' approximating $P(A)$ via an adjoined pair abs, $conc$. An operation $op': (A')^n \to A'$ is said to be a *safe approximation to* op if for all $a_1', \ldots, a_n' \in A'$

$$abs\{op(a_1,\ldots,a_n) \mid a_i \in conc(a_i') \text{ for } i = 1,\ldots,n\} \sqsubseteq op'(a_1',\ldots,a_n')$$

For example, one safe approximation on A_{eo} to the addition operation on N has definition table

+'	⊥	**ev**	**od**	⊤
⊥	⊥	**ev**	**od**	⊤
ev	**ev**	**ev**	**od**	⊤
od	**od**	**od**	**ev**	⊤
⊤	⊤	⊤	⊤	⊤

This definition can easily be extended to include heterogeneous operations $op: A_1 \times \cdots \times A_n \to A_0$.

1. A Language for List Manipulation

Consider a flowchart language with variables of two types: atomic variables $\alpha_1, \alpha_2, \ldots$ ranging over a set A (e.g. the natural numbers), and binary list variables $\lambda_1, \lambda_2, \ldots$. The set of binary lists can be defined inductively as $L ::= A + L \times L$, using McCarthy's abstract syntax [McC63].

The program points of the flowchart are the arcs q in its arc set Q. The nodes are labeled with commands of the following sorts:

$\alpha_i :=$ atomexp	atomic assignment
$\lambda_i :=$ atomexp	atom-to-list constructor
$\lambda_i := cons(\lambda_j, \lambda_k)$	list-to-list constructor
$\lambda_i := \lambda_j.hd$, $\lambda_i := \lambda_j.tl$	selectors
$\alpha_i := \lambda_j$	list-to-atom conversion
if test	tests

Figure 2 contains an example program.

The semantics of such programs may be expressed operationally in terms of *states* $\sigma = <a, e>$ where $a \in Q$

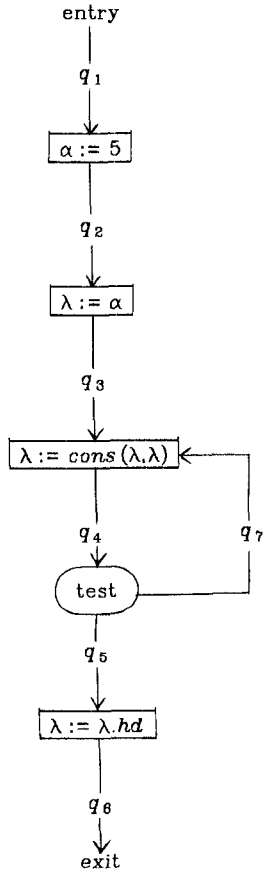


Figure 2. Example Program

is an arc and e is an environment mapping the variables to their current values; the set of all states is thus $\Sigma = Q \times E$ where $E = Var \rightarrow A + L$ is the set of all environments. Rules defining the transition relation $\sigma_1 \Rightarrow \sigma_2$ from one state to another are straightforward and so are omitted in this informal overview. An example computation of the program in Figure 2 might be as follows, where, for brevity, we write $<q,\alpha,l>$ in place of "$\sigma = <q,e>$ where $e(\alpha) = \alpha$ and $e(\lambda) = l$":

$$<q_1,0,0> \Rightarrow <q_2,5,0> \Rightarrow <q_3,5,5> \Rightarrow <q_4,5,<5,5>>$$

$$\Rightarrow <q_7,5,<5,5>> \Rightarrow <q_4,5,<<5,5>,<5,5>>> \Rightarrow$$

$$<q_5,5,<<5,5>,<5,5>>> \Rightarrow <q_6,5,<5,5>>$$

2. Constructing the Data Flow Analyzer

Let σ_0 be the initial program state and $\Rightarrow^*$ be the reflexive transitive closure of $\Rightarrow$. The task at hand is to approximate finitely the set $S \subseteq \Sigma$ of reachable states:

$$S = \{\sigma \mid \sigma_0 \Rightarrow^* \sigma\}$$

Atomic values are easily approximated by using the abstraction function; the main difficulty is representing the values of list-valued variables since their structures may be arbitrarily complex. The approach used here is as follows:

1. A set T of tokens to provide local representations of list structures is chosen. In this instance $T = T_{ATOM} + T_{CONS} + \{nil\}$ is an appropriate choice, where
   T_{ATOM} = {a | a is the exit arc from an atom-to-list constructor statement}
   T_{CONS} = {a | a is the exit arc from a list-to-list constructor statement}.
2. A state $\sigma = <q,\alpha,l>$ in Σ is represented by a triple $<q,\alpha',t>$ where α' is an element of A' approximating α and $t \in T$ is a token which locally represents the list l.
3. The entire set $S \subseteq \Sigma = Q \times A \times L$ is represented by a pair $\delta = <S',r>$ where S' contains representations of all states in S, and r is a (partial) *retrieval function* $r: T \rightarrow A' + P(T \times T)$. Given token t, $r(t)$ may be used to reconstruct the value(s) of the list(s) locally described by t.

To explain how δ represents S, we first show how r represents a list l by a token t, written $t \approx_r l$. The *r-representation relation* $\approx_r \subseteq T \times L$ is defined inductively by

a) $t \approx_r \alpha$ if $\alpha \in A$, $\in A'$ and $\alpha \in conc(r(t))$

b) $t \approx_r <l_1,l_2>$ if $l_1,l_2 \in L$ and there is a pair $<t_1,t_2> \in r(t)$ such that $t_1 \approx_r l_1$ and $t_2 \approx_r l_2$

For example, $T_{ATOM} = \{q_3\}$ and $T_{CONS} = \{q_4\}$ in the program in Figure 2. Suppose now that $r(q_3) = 5$ and $r(q_4) = \{<q_3,q_3>,<q_4,q_4>\}$. Then $q_3 \approx_r 5$, i.e. q_3 r-represents 5, and similarly q_4 r-represents <5,5>, <<5,5>,<5,5>>, <<5,5>,<<5,5>,<5,5>>>, etc.

Let $\delta = <s',r>$. The *state representation relation* $\approx_\delta \subseteq S' \times S$ is naturally defined by $<q',\alpha',t> \approx_\delta <q,\alpha,l>$ if and only if $q = q'$, $\alpha \in conc(\alpha')$ and $t \approx_r l$. Finally, we may take "$\delta = <S',r>$ represents S " to mean that for each $\sigma \in S$ there exists a $\sigma' \in S'$ such that $\sigma' \approx_\delta \sigma$.

The task of flow analysis is to obtain effectively from the given program a representation δ of S. Note that any such representation must have the following two properties, which form the basis for our flow analysis method:

a) S' contains some σ_0' with $\sigma_0' \approx_\delta \sigma_0$, i.e. a description of the start state

b) If $\sigma_1 \in S$, $\sigma_1 \Rightarrow \sigma_2$ and $\sigma_1' \approx_\delta \sigma_1$ for some $\sigma_1' \in S'$ then $\sigma_2' \approx_\delta \sigma_2$ for some $\sigma_2' \in S'$.

Flow analysis can be done by abstractly interpreting the program. We begin with $\delta_0 = <\{\sigma_0'\},r_0>$, where r_0 is the empty retrieval function (i.e. $r_0(\alpha) = \perp$ and $r_0(\lambda) = \phi$). Suppose now that $\delta_k = <S_k',r_k>$ is known, and suppose inductively that for all $i \leq k$, $\sigma_0 \Rightarrow^i \sigma$ implies that $\sigma' \approx_{\delta_k} \sigma$ for some $\sigma' \in S_k'$. Now δ_{k+1} may be obtained by adding elements to the S_k' or $r_k(t)$ sets (or both) so that $\sigma_1 \Rightarrow \sigma_2$ and $\sigma_1' \approx_{\delta_k} \sigma_1$ implies $\sigma_2' \approx_{\delta_{k+1}} \sigma_2$ for some $\sigma_2' \in S_{k+1}'$.

The following table shows the effect of abstractly interpreting the program in Figure 2:

State σ in Σ	Simulated σ' in Σ'	Retrieval function r
$<q_1,?,?>$	$<q_1,\perp,nil>$	
$<q_2,5,?>$	$<q_2,5,nil>$	
$<q_3,5,5>$	$<q_3,5,q_3>$	$5 \in r(q_3)$
$<q_4,5,<5,5>>$	$<q_4,5,q_4>$	$<q_3,q_3> \in r(q_4)$
$<q_7,5,<5,5>>$	$<q_7,5,q_4>$	
$<q_4,5,<<5,5>,<5,5>>>$	$<q_4,5,q_4>$	$<q_4,q_4> \in r(q_4)$
$<q_5,5,<<5,5>,<5,5>>>$	$<q_5,5,q_4>$	
$<q_6,5,<5,5>>$	$<q_6,5,q_3>$ and $<q_6,5,q_4>$	

Note that the transitions from program point q_5 to q_6 simulate the statement $\lambda := \lambda.hd$ by consulting the retrieval function value $r(q_4) = \{<q_3,q_3>,<q_4,q_4>\}$. Since this has two members, two possible new values for λ are obtained.

Note, further, that the approximation is not perfect, since the set of trees actually constructed at q_4 contains only complete binary trees, while $\{l \mid q_4 \approx_r l\}$ contains many others.

We conclude this section with the following observations about the above development:

1. This process may be formalized as follows: First, define the set $\Delta = P(\Sigma') \times R$ of all approximate descriptions, where $R = T \to A' + P(T \times T)$ is the set of retrieval functions. Second, define the natural ordering $\sqsubseteq$ on Δ by $\delta_1 = <S_1,r_1> \sqsubseteq \delta_2 = <S_2,r_2>$ if $S_1 \subseteq S_2$, $r_1(t) \subseteq r_2(t)$ for all $t \in T_{CONS}$ and $r_1(t) \sqsubseteq r_2(t)$ for $t \in T_{ATOM}$, and prove that Δ has no infinite ascending chains provided T is finite and A' has no infinite ascending chains. Third, define the abstract interpretation via a *simulation function* $f: \Delta \to \Delta$. Finally, prove that f is continuous with respect to $\sqsubseteq$ and that $\sigma_1 \Rightarrow \sigma_2$ and $\sigma_1' \approx_\delta \sigma_1$ implies $\sigma_2' \approx_{f(\delta)} \sigma_2$ for some $\sigma_2' \in S_2'$ where $f(\delta) = <S_2',r_2>$. This implies f has a least fixed point $\delta = <S',r>$ and that $\sigma_0 \Rightarrow^* \sigma$ implies $\sigma' \approx_\delta \sigma$ for some $\sigma' \in S'$.
2. The exact choice of the token set T is clearly inessential to the simulation process. All that is required is that whenever a constructor $\lambda_i := cons(\lambda_j,\lambda_k)$ is simulated, the token t chosen to represent λ_i's value is such that a pair $<t_j,t_k>$ representing the λ_j, λ_k values is added to $r(t)$. A larger set T can yield more precise simulation, while a smaller T will give more rapid convergence. In general, T should contain some information about the program's state, as in our choice of the labels of list constructor nodes.

   An exact simulation is obtained by letting T contain, as well, the value of the entire list structure and letting $A' = A$. We consider this a very desirable property of any flow analysis method — that it can be "tuned" to do an exact program execution if desired — since this implies that all semantic information about the program's runtime behavior is potentially available.
3. The retrieval function r can be thought of as a representation of a regular tree grammar [Eng75], [Tha73] with nonterminal set T. The productions are

   a) $t \to a$ for each $a \in conc(r(t))$ if $t \in T_{ATOM}$

   b) $t \to \overset{\wedge}{t_1\ t_2}$ for each $<t_1,t_2> \in r(t)$ if $t \in T_{CONS}$

   It is easily verified that $t \approx_r l$ if and only if $t \to^* l$ by this set of productions. However, we will not use this formulation here because the lattice ordering of A' and its relationship to A is not naturally representable in the usual conception of tree grammars.
4. The regular tree grammars obtained from r seem to be closely related to those obtained by normalizing the extended regular tree grammars of [JoM81], generating essentially the same approximations. However, the grammars derived from r have considerably fewer nonterminals than those of [JoM81] and the present formulation can also handle lattice structures describing atomic values.

3. A Simple Language with Procedures

In this section we discuss a simple procedural language which we shall take as the subject of our interprocedural flow analysis in the next section. The language is purposefully simple to allow us to concentrate on what is new in our approach, namely the flexibility inherent in the use of tokens and retrieval functions, rather than to obscure it with considerations of complex parameter passing methods, scope rules and aliasing. It is hoped that it will be clear from the way we pass from the actual semantics of the language to the approximate flow analysis semantics that such issues can be handled straightforwardly in our approach. The ability of this approach to handle complex control flow is illustrated by multilevel escapes from procedures, a difficult case for previous interprocedural analysis methods.

A program P will comprise a main program $p0$ and a collection of procedures $p1$, ..., pn statically nested directly within the main program (the main program will generally be described as if it were one of the procedures). Each procedure p has a set of local variables, denoted Loc_p, and a set of call-by-value parameters Par_p. The main program is distinguished in that $Par_{p0} = \phi$ and $Glb = Loc_{p0}$ is taken as the set of global variables of the whole program. Thus, within procedure p, the set of accessible variables is $Glb \cup Loc_p \cup Par_p$ and we require (for simplicity) that, for any $p \neq p0$, these three sets be pairwise disjoint.

Each procedure is represented by a flowchart of extended basic blocks [Ken76] (i.e. single-entry, multiple-exit). In particular, a procedure p is a directed graph $<B_p,Q_p,s_p,t_p,eb_p>$ with nodes $b \in B_p$ labeled by extended basic blocks, arcs $q \in Q_p$, two functions s_p, t_p: $Q_p \to B_p$ giving the source and target nodes of each arc and eb_p the distinguished entry block. The functions s_p and t_p specify the endpoints of each arc: q runs from $s_p(q)$ to $t_p(q)$.

The contents of a node $b \in B_p$ may be any single-entry, multiple-exit intraprocedural action (modeled by its transfer and exit selection functions), a call, a return or an interprocedural escape. So as to specify the escapes unambiguously, we stipulate that the Q_p be pairwise disjoint. The (single) arc exiting eb_p is denoted en_p, i.e. $s_p(en_p) = eb_p$.

We specify the semantics of individual instructions by first associating with each procedure its set of *global environments* $E_p = Glb \cup Loc_p \cup Par_p \to Val$, which map the accessible variables to their current values. The semantics of an individual instruction in procedure pi are specified via the following auxiliary functions on

environments:

1. For each (single-entry, multiple-exit) intraprocedural action (e.g. an assignment or a conditional or an extended basic block) there is a *transfer function* $f:E_{pi} \to E_{pi}$ and an *exit selection function* $exit:E_{pi} \to \{1,...,m\}$. The transfer function specifies how the intraprocedural action transforms the environment and the exit selection function specifies the exit arc taken from the instruction.
2. For each call instruction $call\ pj(ex_1,...,ex_m)$ in procedure pi, there is a function $proccall_{ij}:E_{pi} \to E_{pj}$ to set up the environment at the entry to pj defined by

$$proccall_{ij}(e_i)(var) = \begin{cases} e_i(var) & \text{if } var \in Glb \\ eval(ex_k,e_i) & \text{if var is the } k\text{th formal parameter of } pj \\ \perp & \text{if } var \in Loc_{pj} \end{cases}$$

3. For each pair of procedures pi, pj there is a function used for returns and escapes to restore the saved local variable values, $combine_{ij}:E_{pi} \times E_{pj} \to E_{pi}$ defined by

$$combine_{ij}(e_i,e_j)(var) = \begin{cases} e_j(var) & \text{if } var \in Glb \\ e_i(var) & \text{if } var \in Loc_{pi} \cup Loc_{pi} \end{cases}$$

We specify the semantics of programs by an interpreter whose total state is a stack $\sigma = \langle q_1,e_1\rangle\langle q_2,e_2\rangle \ldots \langle q_n,e_n\rangle$ of control point-environment pairs in $\Sigma = (Q \times E)^*$. A control point is an arc $q \in Q$ where

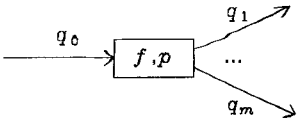
$$Q = \bigcup_{i=0}^{n} Q_{pi}$$

The current control point and environment are q_1 and e_1, and the remainder of the stack contains saved pairs of return addresses and environments for procedures which have been entered but not yet returned from. (Actually, we could restrict e_j for $j > 1$ to hold only the values of local variables and parameters, but we choose not to do so to simplify the notation.) Thus if q_j labels an arc in procedure pi, we have $e_j \in E_{pi}$.

The initial state of the interpreter is $\sigma_0 = \langle en_0,e_0\rangle$, where $e_0 \in E_0$ satisfies $e_0(var) = ?$ for all var and "?" is some (unspecified) initial value.

The transition relation of the interpreter $\Rightarrow \subseteq \Sigma \times \Sigma$ is as follows:

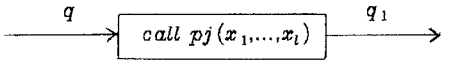
1. for an intraprocedural action



with transfer function $f:E \to E$ and exit selection function $exit:E \to \{1,...,m\}$, we have

$$\langle q_0,e\rangle \S \sigma \Rightarrow \langle q_{exit(e)},f(e)\rangle \S \sigma$$

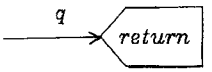
2. for a call instruction



occurring in procedure pi, we have

$$\langle q,e\rangle \S \sigma \Rightarrow \langle en_{pj},proccall_{ij}(e)\rangle \S \langle q_1,e\rangle \S \sigma$$

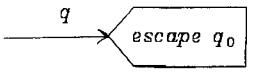
3. for a return instruction from pj



we have

$$\langle q,e\rangle \S \langle q_1,e_1\rangle \S \sigma \Rightarrow \langle q_1,combine(e,e_1)\rangle \S \sigma$$

4. for an escape instruction



with $q_0 \in Q_{pi}$, we have

$$\langle q,e\rangle \S \sigma \Rightarrow esc(q,e,\sigma)$$

where $esc:Q \times E \times \Sigma \to \Sigma$ pops the call stack to the appropriate level, as given by

$$esc(q,e,\langle q_1,e_1\rangle \S \sigma) = \text{if } q_1 \in Q_{pi} \text{ then } \langle q_0,combine(e,e_1)\rangle \S \sigma \text{ else } esc(q,e,\sigma)$$

and $esc(q,e,\varepsilon) = \varepsilon$.

As an example of the above, consider the program in Figure 3 (as the exit selection functions are all trivial they are omitted). Note that the control flow of this program is given by the infinite sequence of arcs $abe(ghc)^\infty$ and that the first few states in its execution are

$$\begin{aligned} \langle a,\{w \to ?\}\rangle &\Rightarrow \langle b,\{w \to 2\}\rangle \\ &\Rightarrow \langle e,\{w \to 2,x \to 1\}\rangle\langle c,\{w \to 2\}\rangle \\ &\Rightarrow \langle g,\{w \to 2,y \to 3,z \to 1\}\rangle\langle f,\{w \to 2, x \to 1\}\rangle\langle c,\{w \to 2\}\rangle \\ &\Rightarrow \langle h,\{w \to 4,y \to 3,z \to 1\}\rangle\langle f,\{w \to 2, x \to 1\}\rangle\langle c,\{w \to 2\}\rangle \\ &\Rightarrow \langle c,\{w \to 4\}\rangle \\ &\Rightarrow \langle g,\{w \to 4,y \to 5,z \to 1\}\rangle\langle d,\{w \to 4\}\rangle \\ &\Rightarrow \langle h,\{w \to 6,y \to 5,z \to 1\}\rangle\langle d,\{w \to 4\}\rangle \\ &\Rightarrow \langle c,\{w \to 6\}\rangle \\ &\ldots \end{aligned}$$

4. An Approximate Interpreter for our Example Language

Our purpose in this section is to construct an approximate interpreter for our simple procedural language, parameterized in such a way that the choice of token set will determine the exactness of the interprocedural flow analyzer so produced and its computational efficiency as well. In particular, the range of behaviors

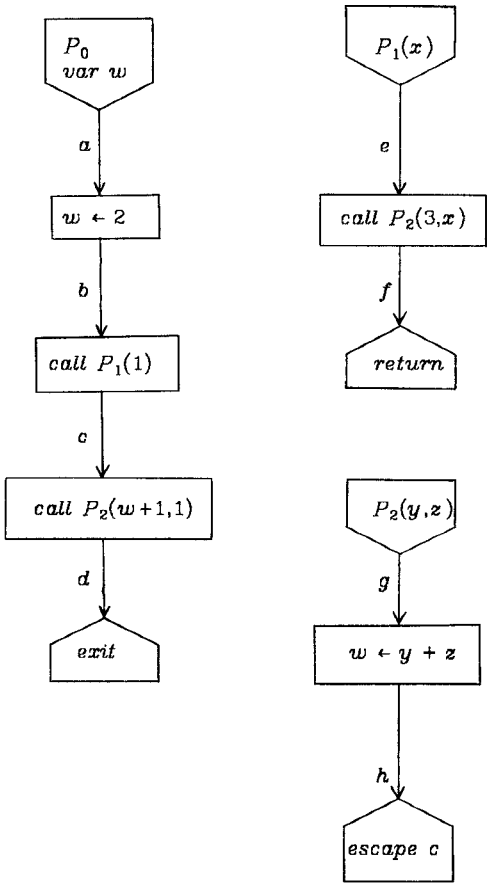


Figure 3. An example program

should include everything from summary data flow analysis [Bar78] to exact execution descriptions.

As a starting point, we assume given

1. An approximation lattice E_{pi}' for each set E_{pi} of environments for procedure pi; for example, to analyze parity of variable values, set $E_{pi}' = Glb \cup Loc_{pi} \cup Par_{pi} \to A_{so}$.
2. For each instruction "*call* $P_j(...)$" in procedure P_i, a safe approximation $proccall_{ij}' : E_{pi}' \to E_{pj}'$ to its parameter setup function $proccall_{ij}$.
3. For each pi and pj, a safe approximation $combine_{ij}' : E_{pi}' \times E_{pj}' \to E_{pi}'$ to $combine_{ij}$ (typically quite analogous to $combine_{ij}$).
4. A function *token* such that if the current control arc of a state description σ' is the entry to a call instruction, then $token(\sigma')$ is a token representing the new stack. Further discussion of the choice of token sets will be deferred until after the construction of the approximate interpreter.

As in the analysis of programs with recursive data structures presented above, an algorithm will be described to find a safe computation description δ of the set of all reachable program states. A state $\sigma = \langle q_1,e_1\rangle...\langle q_m,e_m\rangle$ with $q_1 \in Q_{pi}$ will be described by a triple $\sigma_1' = \langle q_1,e_1',t\rangle$ in $\bar{\sigma}$, where e_1' in E_{pi}' represents e_1 and $t = token(\sigma_2')$. Here σ_2' describes the state just before entry to the current procedure (t is some distinctive initial value t_0 if σ is in the main program). No explicit retrieval function r will be used, but the $r(t)$ used above will correspond intuitively to $\{\sigma' \in \delta \mid token(\sigma') = t\}$.

A *state description* is by definition an element σ' of

$$\Sigma' = \sum_{i=0}^{n} (Q_{pi} \times E_{pi}' \times T)$$

A computation description will be a subset δ of Σ'. However, not every subset of Σ' is suitable for approximation purposes. In particular, two state descriptions $\langle q,e_1',t\rangle$ and $\langle q,e_2',t\rangle$ in the same δ logically can be replaced by the single description $\langle q,e_1' \sqcup e_2',t\rangle$ (and should be for reasons of efficiency and termination). Further, we need a lattice structure on computation descriptions, so that, as in the preceding examples, a minimal safe description δ can be obtained by beginning with the initial state description and abstractly interpreting the program until no new state descriptions can be added. Consequently we define the following:

1. The set of all *computation descriptions* is $\Delta =$

$\{\delta \subseteq \Sigma' \mid \langle q,e_1',t\rangle \in \delta$ and $\langle q,e_2',t\rangle \in \delta$ implies $e_1' = e_2'\}$

2. The order relation $\sqsubseteq$ is given by $\delta_1 \sqsubseteq \delta_2$ iff for all $\langle q,e_1',t\rangle \in \delta_1$ there exists $\langle q,e_2',t\rangle \in \delta_2$ with $e_1' \sqsubseteq e_2'$. This ordering, in fact, makes Δ isomorphic to the lattice of partial functions

$$\sum_{i=0}^{n} (Q_{pi} \times T \to E_{pi}')$$

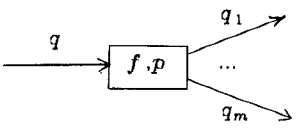
under the natural (pointwise) ordering on function spaces.

The desired computation description is, by definition, the smallest δ with respect to $\sqsubseteq$ which satisfies the simulation rules given below. Each rule is a closure property, specifying that if certain state descriptions σ_1', σ_2', ... are in δ then $\{\sigma'\} \sqsubseteq \delta$ must hold for some additional state description σ'. Existence and uniqueness of such a δ follows from the fact that the rules may be viewed collectively as defining a continuous function $\Theta:\Delta \to \Delta$ on the lattice of computation descriptions. Consequently, by Kleene's constructive version of the Tarski-Knaster fixed point theorem Θ has a unique least fixed point, namely

$$\delta = \bigsqcup_{i=0}^{\infty} \Theta^i(\phi)$$

In general, $\Theta^m(\phi)$ describes the result of applying the simulation rules m times to an initially empty computation description. The simulation rules are as follows:

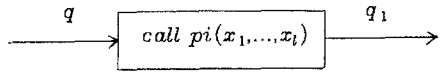
0. Let the initial program state be $\langle en_0,e_0\rangle$. Then $\langle en_0,e_0',t_0\rangle \in \delta$, where $e_0' = abs\{e_0\}$.
1. For an intraprocedural action



with transfer function $f:E_{pi} \to E_{pi}$ and exit selection function $exit:E_{pi} \to \{1,...,m\}$, if $\langle q,e',t\rangle \in \delta$ then $\{\langle q_i,e_i',t\rangle\} \sqsubseteq \delta$ for $i = 1,...,m$ where $e_i' =$

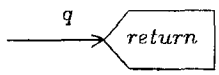
$abs\{f(e) \mid e \in conc(e') \text{ and } exit(e) = i\}$.

2. For a call instruction



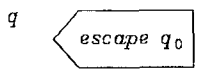
occurring in procedure pj, if $\sigma' = \langle q,e',t\rangle \in \delta$ then $\{\langle en_{pi},proccall'(e'),token(\sigma')\rangle\} \subseteq \delta$.

3. For a return instruction in procedure pj



if $\langle q,e_j',token(\sigma')\rangle \in \delta$ where $\sigma' = \langle q_1,e_1',t_1\rangle$ is in δ and q_1 is an entry arc to an instruction of the form "*call* $pi(...)$" in P_j with exit arc q_2 then $\{\langle q_2,combine_{ij}(e_i',e_j'),t\rangle\} \subseteq \delta$.

4. For an escape from procedure pj to label q_0 in procedure pi



if $\sigma' = \langle q,e_j',t\rangle \in \delta$ and $\sigma' \rightarrow \langle q_1,e_i',t\rangle$ then $\{\langle q_0,combine_{ij}(e_i',e_j'),t_1\rangle\} \subseteq \delta$ where

$\langle q_1,e_i',t_1\rangle \rightarrow \langle q_1,e_i',t_1\rangle$ if $q_1 \in Q_{pi}$ and $\sigma' \in \delta$
$\langle q_1,e_i',token(\sigma')\rangle \rightarrow \sigma_1'$ if $q_1 \notin Q_{pi}$, $\sigma' \in \delta$ and $\sigma' \rightarrow \sigma_1'$

5. Some Examples of Interprocedural Analysis

As an example of our approach to interprocedural flow analysis, we first analyze the program in Figure 3 to determine the parity of the values of its variables. Given an environment $E: Var \rightarrow N$ mapping some set of variables to integer values, we define $E': Var \rightarrow A_{eo}$. A safe approximation $combine_{ij}'$ to $combine_{ij}$ is essentially analogous to $combine_{ij}$, and safe approximations to the various $proccall_{ij}$ functions are straightforward. For example, we could use the following for "*call* $P_2(w+1,1)$" in Figure 3:

$$proccall_{ij}'(e_0')(w) = e_0'(w)$$

$$proccall_{ij'(e_0')}(w) = \text{if } e_0'(w) = \textbf{ev} \text{ then } \textbf{od} \text{ elif } e_0'(w) = \textbf{od} \text{ then } \textbf{ev} \text{ else } e_0'(w)$$

$$proccall_{ij}'(e_0')(z) = \textbf{od}$$

As tokens we take t_0 plus the entry arcs to "*call*" instructions, so $token(\langle q,e',t\rangle) = q$. We assume that initially $e_0'(x) = \bot$ for all global variables x. The following table shows how surface state descriptions accumulate as we perform the analysis:

Computation description: $\delta = \{\sigma_1',...,\sigma_8'\}$ where

$\sigma_1' = \langle a,\{w \rightarrow \bot\},t_0\rangle$	
$\sigma_2' = \langle b,\{w \rightarrow \textbf{ev}\},t_0\rangle$	
$\sigma_3' = \langle e,\{w \rightarrow \textbf{ev},x \rightarrow \textbf{od}\},b\rangle$	$token(\sigma_2') = b$
$\sigma_4' = \langle g,\{w \rightarrow \textbf{ev},y \rightarrow \textbf{od},z \rightarrow \textbf{od}\},e\rangle$	$token(\sigma_3') = e$
$\sigma_5' = \langle h,\{w \rightarrow \textbf{ev},y \rightarrow \textbf{od},z \rightarrow \textbf{od}\},e\rangle$	
$\sigma_6' = \langle c,\{w \rightarrow \textbf{ev}\},t_0\rangle$	
$\sigma_7' = \langle g,\{w \rightarrow \textbf{ev},y \rightarrow \textbf{od},z \rightarrow \textbf{od}\},c\rangle$	$token(\sigma_6') = c$
$\sigma_8' = \langle h,\{w \rightarrow \textbf{ev},y \rightarrow \textbf{od},z \rightarrow \textbf{od}\},c\rangle$	

Note that we have succeeded in determining the parity of all variables.

For our second example we do a constant propagation analysis on the same program (Figure 3). The set of environments $E: Var \rightarrow N$ will be modelled by the lattice $E': Var \rightarrow A_{con}$, and the $combine_{ij}'$ functions are again analogous to the $combine_{ij}$. The $proccall_{ij}$ functions are easily approximated: for example, for the instruction "*call* $P_2(w+1,1)$" we have

$$proccall_{ij}'(e')(w) = e'(w)$$

$$proccall_{ij}'(e')(y) = \text{if } e'(w) \notin \{\bot,\top\} \text{ then } e'(w) + 1 \text{ else } e'(w)$$

$$proccall_{ij}'(e')(z) = 1$$

The following table shows the results of doing the first seven steps of the constant propagation analysis with $token(\langle q,e',t\rangle) = q$:

Computation description (first 7 iterations):

$$\delta = \bigsqcup_{i=0}^{7} \Theta^i(\phi) = \{\sigma_1', \ldots, \sigma_8'\} \text{ where}$$

$\sigma_1' = \langle a,\{w \rightarrow \bot\},t_0\rangle$	
$\sigma_2' = \langle b,\{w \rightarrow 2\},t_0\rangle$	
$\sigma_3' = \langle e,\{w \rightarrow 2,x \rightarrow 1\},b\rangle$	$token(\sigma_2') = b$
$\sigma_4' = \langle g,\{w \rightarrow 2,y \rightarrow 3,z \rightarrow 1\},e\rangle$	$token(\sigma_3') = e$
$\sigma_5' = \langle h,\{w \rightarrow 4,y \rightarrow 3,z \rightarrow 1\},e\rangle$	
$\sigma_6' = \langle c,\{w \rightarrow 4\},t_0\rangle$	
$\sigma_7' = \langle g,\{w \rightarrow 4,y \rightarrow 5,z \rightarrow 1\},c\rangle$	$token(\sigma_6') = c$
$\sigma_8' = \langle h,\{w \rightarrow 6,y \rightarrow 5,z \rightarrow 1\},c\rangle$	

At this point $\langle c,\{w \rightarrow 6\},t_0\rangle$ is to be added to δ. Since this triple and σ_6' differ only in their environment components, they are combined. This happens twice more and the final description which results is given by the following table:

Computation description:

$$\delta = \bigsqcup_{i=0}^{\infty} \Theta^i(\phi) = \{\sigma_1', \ldots, \sigma_8'\} \text{ where}$$

$\sigma_1' = \langle a,\{w \rightarrow \bot\},t_0\rangle$	
$\sigma_2' = \langle b,\{w \rightarrow 2\},t_0\rangle$	
$\sigma_3' = \langle e,\{w \rightarrow 2,x \rightarrow 1\},b\rangle$	$token(\sigma_2') = b$
$\sigma_4' = \langle g,\{w \rightarrow 2,y \rightarrow 3,z \rightarrow 1\},e\rangle$	$token(\sigma_3') = e$
$\sigma_5' = \langle h,\{w \rightarrow 4,y \rightarrow 3,z \rightarrow 1\},e\rangle$	
$\sigma_6' = \langle c,\{w \rightarrow \top\},t_0\rangle$	
$\sigma_7' = \langle g,\{w \rightarrow \top,y \rightarrow \top,z \rightarrow 1\},c\rangle$	$token(\sigma_6') = c$
$\sigma_8' = \langle h,\{w \rightarrow \top,y \rightarrow \top,z \rightarrow 1\},c\rangle$	

Thus we have determined that both x and z have the constant value 1, while w and z are runtime-variable.

6. Choice of Tokens

The possible choices of token sets vary over quite a wide range. At one extreme we can get an exact simulation, and hence derive from

$$\delta = \bigsqcup_{n=0}^{\infty} \Theta^n(\phi)$$

an exact description of the entire computation. One way to obtain this is to use $\Delta = P(\Sigma')$ with the subset ordering, let E' be the flat lattice $E' \cup \{\bot,\top\}$ and define $token(\sigma') = \sigma'$ for all $\sigma' \in \Sigma'$. Clearly such a δ is not in general effectively obtainable, but it does show that there is no intrinsic limit to the flow information obtainable by our method.

We have chosen in our examples to let T be t_0 plus all the program's call arcs, thus differentiating calls from different places to the same procedure. Using procedure entry arcs instead gives smaller computation descriptions by combining information from different calls to the same procedure.

The "functional approach" of [ShP81] can be modelled by using as tokens the input argument descriptions. To see this, suppose procedure pi has arc n and that e_0' describes the environment at entry time. The appearance of a triple $<n,e',e_0'>$ in δ can be expressed in the terminology of [ShP81] by

$$\varphi_{(r_{pi},n)}(e_0') = e'$$

The present approach is more flexible for several reasons, including the flexibility in the choice of tokens, the fact that both local and global variables can be accomodated and the absence of limitations with respect to possible control flows.

The "call-strings approach" of [ShP81] also appears to be achievable by an appropriate choice of T.

7. Some Remarks on Practicality

As presented above, our method is highly general but potentially quite expensive in practice. Clearly, its complexity will vary with the choice of token set and there is some leeway for judicious choices there. Another consideration is that to implement the modelling of returns and escapes efficiently, it would help to have an explicit retrieval function $r:T \to \Delta$ mapping token t to $r(t) = \{\sigma' \in \delta \mid token(\sigma') = t\}$.

Another major issue is the space required to store δ, which is potentially quite high since it is a ternary relation. This could be reduced by representing it less exactly. For example, in place of δ, one could use two functions φ and χ (for "flow" and "call" information, respectively) with

$$\varphi \in \sum_{i=0}^{n} (Q_{pi} \to E_{pi}')$$

mapping arcs to data flow information and

$$\chi \in \{p0,\ldots,pn\} \to P(T)$$

mapping each procedure to the set of tokens describing states which call it. Modification of the flow simulation to work with φ and χ is straightforward: containment of a triple $<q,e',t>$ in δ with q in procedure pi would correspond to $\varphi(q) = e'$ and $t \in \chi(pi)$. However, the decoupling of the e' and t components will generally make it impossible to get exact simulation by any choice of T.

8. Conclusion

We have presented a highly flexible approach to flow analysis of programs with recursive data structures and an application of the method to interprocedural flow analysis. We believe that the framework for interprocedural analysis is general enough to encompass virtually all of the previous methods, given appropriate choices of token sets and data approximation lattices. That the method can be extended to handle other language features should be clear from the use of an interpretive semantics.

De Laubenfels and Muchnick are currently working on applying the basic approach to analysis of distributed programs with communications via message queues, and preliminary work promises significantly better flow information than that given by Reif's methods in [Rei79].

Two areas of investigation that appear fruitful are to consider how the computational complexity of the analyzer varies with the choice of token set and the feasibility of building an analyzer with the choice of token set as a parameter.

REFERENCES

Bar78 Barth, Jeffrey, A Practical Interprocedural Data Flow Analysis Algorithm, *Comm. of the ACM*, vol. 21, no. 9, 1978, pp. 724 - 736.

Cou77a Cousot, Patrick & Radhia Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *Conf. Rec. of 4th ACM Symp. on Princ. of Prog. Lang.*, Los Angeles, California, January 1977, pp. 238 - 252.

Cou77b Cousot, Patrick & Radhia Cousot, Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations, *Proc. of ACM Symp. on Artif. Intell. and Prog. Lang.*, *SIGPLAN Notices*, vol. 12, no. 8, August 1977, pp. 1 - 12.

Cou79 Cousot, Patrick & Radhia Cousot, Systematic Design of Program Analysis Frameworks, *Conf. Rec. of 6th ACM Symp. on Princ. of Prog. Lang.*, January 1979, pp. 269 - 282.

Eng75 Engelfriet, Joost, *Tree Automata and Tree Grammars*, DAIMI Report FN-10, Dept. of Computer Science, University of Aarhus, Aarhus, Denmark, April 1975.

Hec77 Hecht, Matthew S., **Flow Analysis of Computer Programs**, Elsevier North-Holland, New York, 1977.

JoM81 Jones, Neil D. & Steven S. Muchnick, Flow Analysis and Optimization of LISP-Like Structures, in Steven S. Muchnick & Neil D. Jones (eds.), **Program Flow Analysis: Theory and Applications**, Prentice-Hall, Englewood Cliffs, New Jersey, 1981, pp. 102 - 131.

Jon80 Jones, Neil D., Flow Analysis of Lambda Expressions, Technical Report DAIMI IR-23, Computer Science Department, University of Aarhus, Aarhus, Denmark, October 1980.

Ken76 Kennedy, Ken, A Comparison of Two Algorithms for Global Data Flow Analysis, *SIAM J. Comput.*, vol. 5, no. 1, 1976, pp. 158 - 180.

Kil73 Kildall, G. A., A Unified Approach to Global Program Optimization, *Conf. Rec. of the ACM Symp. on Princ. of Prog. Lang.*, Boston, Massachusetts, October 1973, pp. 194 - 206.

McC63 McCarthy, John, Towards a Mathematical Science of Computation, **Information Processing 1962**, North-Holland, Amsterdam, 1963, pp. 220 - 226.

Nie80 Nielson, Flemming, Semantic Foundations of Data Flow Analysis, Technical Report DAIMI PB-131, Computer Science Department, Aarhus University, Aarhus, Denmark, February 1981.

Rei79 Reif, John H., Data Flow Analysis of Communicating Processes, *Conf. Rec. of the 6th ACM Symp. on Princ. of Prog. Lang.*, San Antonio, Texas, January 1979, pp. 257 - 268.

Sin72 Sintzoff, Michel, Calculating Properties of Programs by Valuations on Specific Models, *Proc. ACM Conf. on Proving Assertions about Programs*, Las Cruces, New Mexico, January 1972, pp. 203 - 207.

ShP81 Sharir, Micha & Amir Pnueli, Two Approaches to Interprocedural Data Flow Analysis, in Steven S. Muchnick & Neil D. Jones (eds.), **Program Flow Analysis: Theory and Applications**, Prentice-Hall, Englewood Cliffs, New Jersey, 1981, pp. 189 - 234.

Tha73 Thatcher, James W., Tree Automata: An Informal Survey, in Alfred V. Aho (ed.), **Currents in the Theory of Computing**, Prentice-Hall, Englewood Cliffs, New Jersey, 1973, pp. 143 - 172.