e79]. It emerges from these comparisons that this algorithm is very fast applied separately to each procedure. For interprocedural application, nly added task is the solution of systems in the data flow analysis phase, his is known to be fast. Although this extended algorithm has not yet implemented, it can reasonably be assumed to be efficient.

6-6. CONCLUSION

An algorithm for interprocedural optimization has been presented. It is d to a set of procedures compiled together and which call each other. ased on the algorithm presented in [More79] for elimination of partial dancies in a procedure. The interprocedural application is performed wo-pass mechanism. The information on each procedure for the appli- of the basic algorithm is computed by a preliminary data flow analysis This phase, which requires a particular processing order on the set of lures, gives for each procedure P information which represents the t of a call of P on the environment at the calling point. The second suppresses partial redundancies by treating the procedures individually rse calling order, reflecting in the treatment of a called procedure the ation gathered during the treatment of its callers.

n some cases, optimization can be improved by iterating part or all of ocess. The algorithm can be used for recursive procedures, but in this ome approximations are made in the data flow analysis phase in order id unpredictable costs in the algorithm. As presented here, the algo- uns in time linear in the size of the program. Implementation of the lgorithm has been shown to be efficient and well within the state of the actual compilers. The same claims seem to be applicable to its inter- ural extension.

Chapter 7

Two Approaches to Interprocedural Data Flow Analysis†

Micha Sharir
Amir Pnueli

7-1. INTRODUCTION

Under the general heading of program analysis we can find today two disciplines which, even though they have similar aims, differ in the means and tools they apply to the task of analysis. The first is the discipline of *program verification*. This is usually presented as the process of finding invariants of the program, or in other words fully characterizing the behavior of the program, discovering all the properties of all possible executions [Mann74, Cous77e]. As such, it is extremely ambitious and hence a priori doomed to failure on theoretical grounds for all but the most restricted program models.

The second discipline falling under the name of *program analysis* is the more pragmatically oriented data flow analysis. Associated with optimizing compilers, this methodology is very much concerned with questions of effectiveness and efficiency, in particular the trade-off between effort invested and the increment in the quality of produced code gained. Quite understandably,

†The work of the first author was partially supported by the National Science Foundation under grant MCS76–00116 and the United States Department of Energy under grant EY-76-C-02-3077.

its objectives are more modest. The reduced ambitiousness is expressed in not trying to extract *all* properties of the program but concentrating on several simple, well-defined properties such as the availability of expressions, the types and attributes of dynamic values, the constancy of variables, etc.

A basic technique used to analyze procedureless programs (or single procedures) is to transform them into flow graphs [Alle69] and assume that all paths in the graphs can represent actual executions of the program. This model does not describe the "true" run-time situation correctly, and in fact most of the graph paths are not feasible, i.e., do not represent possible executions of the program. However, this model is widely adopted for two main reasons:

1. Its relatively simple structure enables us to develop a comprehensive analytic theory, to construct simple algorithms which perform the required program analysis, and to investigate general properties of these algorithms in detail (cf. [Hech77, Aho77] or Chapter 1 for recent surveys of the subject).
2. Isolation of feasible paths from nonfeasible ones is known to be an undecidable problem, closely related to the Turing machine halting problem.

This classical technique faces significant problems in the presence of procedures. These problems reflect the dependence of individual interprocedural branches upon each other during program execution, a dependence which is known at compile time and is essentially independent of any computation performed during execution. Interprocedural branching is thus much easier to analyze than intraprocedural branches, which usually depend on the values assumed by various program variables. It is therefore very tempting to exploit our special knowledge of this branching pattern in program analysis, thereby tracing the program flow in a more accurate manner.

Interprocedural flow cannot be treated as a simple extension of the intraprocedural flow, but calls for a more complicated model whose mathematical properties require special analysis. In addition, many programming languages include features such as procedure variables and parameter passing by reference or by name [Aho77] which complicate the analysis of interprocedural flow.

It is therefore not surprising that interprocedural analysis has been neglected in much research on date flow analysis. Most of the recent literature on this subject virtually ignores any interprocedural aspect of the analysis, or splits the interprocedural analysis into a preliminary analysis phase which gathers overestimated information about the properties of each procedure in a program and which is followed by an intraprocedural analysis of each procedure, suppressing any interprocedural transfer of control and using

instead the previously collected, overestimated information to deduce the effects of procedure calls on the program behavior [Alle74]. These approaches use a relatively simple model of the program at the expense of some information loss, arguing that such a loss is intrinsic anyway even in a purely intraprocedural model.

However, there is a growing feeling among researchers that more importance should be given to interprocedural analysis, especially in deeper analyses with more ambitious goals, where avoidance of flow overestimation is likely to be significant in improving the results of the analysis. This is true in particular for analyses related to program verification, in which area several recent papers, notably [DeBa75, Grei75, Hare76, Gall78, Cous77e] have already addressed this issue.

Recently, however, the interest in more accurate interprocedural data flow analysis has increased considerably, and new approaches to the problem appear in several recent works by Rosen [Rose79], Barth [Bart77a], Lomet [Lome75], and others. All these works attempt to generalize, achieve more accurate information than, or be more pragmatic than the traditional methods mentioned earlier. However, none of these methods achieves complete generality. They are all interested in gathering only *local* effects of procedure calls, are limited to simple bit-vector data flow problems, and do not view interprocedural analysis as an integral part of the global data flow analysis, but rather as a preliminary phase, completely independent from the actual program analysis phase. For example, they all ignore the problem of computing data at procedure entries interprocedurally, and are therefore forced to make worst-case assumptions about these values. However, they all can handle recursion. Rosen's work [Rose79] also handles reference parameters and derives "sharpest" static information, at the cost of a rather complex algorithm.

In this paper we introduce two new techniques for performing interprocedural data flow analysis. These techniques are almost generally applicable; they derive the sharpest static information, they integrate interprocedural analysis with intraprocedural analysis, and they handle recursion properly. These two approaches use two somewhat different graph models for the program being analyzed. The first approach, which we term the *functional approach*, views procedures as collections of structured program blocks and aims to establish input-output relations for each such block. One then interprets procedure calls as "super operations" whose effect on the program status can be computed using those relations. This approach relates rather closely to most of the known techniques dealing with interprocedural flow, such as the "worst-case assumptions," mixed with processing of procedures in "inverse invocation order" [Alle74], Rosen's "indirect arcs" method [Rose79], inline expansion of procedures [Alle77], and most of the known interprocedural techniques for program verification [Grei75, Gall78,

Hare76, Cous77e]. Our version of this first technique has the advantage of being rather simple to define and implement (potentially admitting rather efficient implementations for several important special cases), as well as the other advantages mentioned above.

Our second technique, which we term the *call-strings approach*, is somewhat orthogonal to the first approach. This second technique blends interprocedural flow analysis with the analysis of intraprocedural flow, and in effect turns a whole program into a single flow graph. However, as information is propagated through this graph, it is "tagged" with an encoded history of the procedure calls encountered during propagation. In this way we make interprocedural flow explicit, which enables us to determine, whenever we encounter a procedure return, what part of the information at hand can validly be propagated through this return, and what part has a conflicting call history that bars such propagation.

Surprisingly enough, very few techniques using this kind of logic have been suggested up to now. We may note in this connection that a crude approach, but one using similar logic, would be one in which procedure calls and returns are interpreted as ordinary branch instructions. Even though the possibility of such an approach has been suggested occasionally in the literature, it has never been considered seriously as an alternative interprocedural analysis method. A related approach to program verification has been investigated by de Bakker and Meertens [DeBa75], but, again, this has been quite an isolated attempt and one with rather discouraging results, which we believe to be due mainly to the ambitious nature of the analyses considered. There is some resemblance, though, between this second approach and the inline expansion method [Alle77] (see Section 7-4 for details).

We shall show that an appropriate sophistication of this approach is in fact quite adequate for data flow analysis, and gives results quite comparable with those of the functional approach. This latter approach also has the merit that it can easily be transformed into an approximative approach, in which some details of interprocedural flow are lost, but in which the relevant algorithms become much less expensive.

A problem faced by any interprocedural analysis is the possible presence of recursive procedures. The presence of such procedures causes interprocedural flow to become much more complex than it is in the nonrecursive case, mainly because the length of a sequence of nested calls can be arbitrarily large. Concerning our approaches in this case, we will show that they always converge in the nonrecursive case, but may fail to yield an effective solution of several data flow problems (such as constant propagation) for recursive programs. It will also be seen that much more advanced techniques are needed if we are to cope fully with recursion for such problems.

We note that it is always possible to transform a program with procedures into a procedureless program by converting procedure calls and returns into ordinary branch instructions, monitored by an explicit stack. If we do this and simply subject the resulting program to intraprocedural analysis, then we are in effect ignoring all the delicate properties of the interprocedural flow and thus inevitably overestimating flow. This simple observation shows that the attempt to perform more accurate interprocedural analysis can be viewed as a first (and relatively easy) step toward accurate analysis of more sophisticated properties of programs than are caught by classical global analysis.

This chapter is organized as follows: Section 7-2 contains preliminary notations and terminology. Section 7-3 presents the functional approach, first in abstract, definitional terms, and then shows that it can be effectively implemented for data flow problems which possess a finite semilattice of possible data values and sketches an algorithm for that purpose. We also discuss several cases in which unusually efficient implementation is possible. (These cases include many of those considered in classical data flow analyses.) Section 7-4 presents the call-strings approach in abstract, definitional terms showing that it also yields the solution we desire, though in a manner which is not necessarily effective in the most general case. In Section 7-5 we show that this latter approach can be effectively implemented if the semilattice of relevant data values is finite and investigate some of the efficiency parameters of such an implementation. Section 7-6 presents a variant of the call-strings approach which aims at a relatively simple, but only approximative, implementation of interprocedural data flow analysis. Section 7-7 is a concluding section in which some further directions of research are suggested and discussed.

We would like to express our gratitude to Jacob T. Schwartz for encouragement and many helpful suggestions and comments concerning this research, and to Barry K. Rosen for careful reviewing and helpful comments on this manuscript.

7-2. NOTATIONS AND TERMINOLOGY

In this section we will review various basic notations and terminology used in intraprocedural analysis, which will be referred to and modified subsequently. The literature on data flow analysis is by now quite extensive, and we refer the reader to [Hech77], [Aho77], or Chapter 1, three excellent recent introductory expositions of that subject.

To analyze a program consisting of several subprocedures, each subprocedure p, including the main program, is first divided into *basic blocks*. An (extended) basic block is a maximal single-entry multiexit sequence of code. For convenience, we will assume that each procedure call constitutes

a single-instruction block. We also assume that each subprocedure p has a unique exit block, denoted by e_p, which is also assumed to be a single-instruction block, and that p has a unique entry (root) block, denoted by r_p.

Assume for the moment that p contains no procedure calls. Then the *flow graph* G_p of p is a rooted directed graph whose nodes are the basic blocks of p, whose root is r_p, and which contains an edge (m, n) for each direct transfer of control from the basic block m to (the start of) the basic block n, effected by some branch instruction. The presence of calls in p induces several possible interprocedural extensions of the flow graph, which will be discussed in the next section.

Let G be any rooted directed graph. G is denoted by a triplet (N, E, r), where N is the set of its nodes, E the set of edges, and r its root. A *path* p in G is a sequence of nodes in N $(n_1, n_2, \ldots, n_k)$ such that for each $1 \leq j < k$, $(n_j, n_{j+1}) \in E$. p is said to lead from n_1 (its initial node) to n_k (its terminal node). p can be also represented as the corresponding sequence of edges $((n_1, n_2), \ldots, (n_{k-1}, n_k))$. The *length* of p is defined as the number of edges along p ($k - 1$ in the above notation). For each pair of nodes $m, n \in N$ we define $\text{path}_G(m, n)$ as the set of all paths in G leading from m to n.

We assume that the program to be analyzed is written in a programming language with the following semantic properties: Procedure parameters are transferred by value, rather than by reference or by name (so that we can, and will, ignore the problem of "aliasing" discussed by Rosen [Rose79]), and there are no procedure variables or external procedures. We also assume that the program has been translated into intermediate-level code in which the transfer of values between actual arguments and formal parameters of a procedure is explicit in the code and is accomplished by argument-transmitting assignments, inserted before and after procedure calls. Because of this last assumption, formal parameters can be treated in the same way as other global variables. (For simplicity, we ignore here some aspects of recursive value stacking, which gives these "assignments" extra flavor. For example, a formal parameter of a recursive procedure p will have the same value after the "epilog" of a recursive call in p to p as the value it had before the call. Such considerations can be incorporated into our techniques, but will not be discussed in this paper. The reader may find it helpful to think of our model as allowing only parameterless procedures, in which case the above problems do not exist.) All these assumptions are made in order to simplify our treatment and are rather reasonable. If the first two assumptions are not satisfied, then things become much more complicated, though not beyond control. The third assumption is rather arbitrary but most convenient. (In [Cous77e], e.g., the converse assumption is made, namely that global variables are passed between procedures as parameters, an assumption which we believe to be less favorable technically.)

A *global data flow framework* is defined to be a pair (L, F), where L is a semilattice of data or attribute information and F is a space of functions acting on L (and describing a possible way in which data may propagate along program flow paths). Let $\wedge$ denote the semilattice operation of L (called a *meet*), which is assumed to be idempotent, associative, and commutative. We assume that L contains a smallest element, denoted by 0, usually signifying null (worst-case) information (see below), and also a largest element Ω, corresponding to "undefined" information (see Section 7-3 for more details). F is assumed to be closed under functional composition and meet, to contain an identity map, and to be *monotone*, i.e., to be such that for each $f \in F$, $x, y \in L$, $x \leq y$ implies $f(x) \leq f(y)$. L is also assumed to be *bounded*, i.e., not to contain any infinite decreasing sequence of distinct elements. (L, F) is called a *distributive* framework if, for each $f \in F$ and $x, y \in L$, $f(x \wedge y) = f(x) \wedge f(y)$. We also assume that F contains a constant map f_Ω, which maps each $x \in L$ to Ω. This map corresponds to impossible propagation (see below).

Given a global data flow framework (L, F) and a flow graph G, we associate with each edge (m, n) of G a propagation function $f_{(m,n)} \in F$, which represents the change of relevant data attributes as control passes from the start of m, through m, to the start of n. (Recall that a basic block may have more than one exit, so that $f_{(m,n)}$ must depend on n as well as m.)

Once the set $S = \{f_{(m,n)} : (m, n) \in E\}$ is given, we can define a (graph-dependent) space F of propagation functions as the smallest set of functions acting in L which contains S, f_Ω and the identity map $\mathbf{id}_L$, and which is closed under functional composition and meet. It is clear that this F is monotone iff S is monotone, and that F is distributive iff S is distributive.

Once F is defined, we can formulate the following general set of data propagation equations, where, for each $n \in N$, x_n denotes the data available at the start of n:

$$\begin{aligned} x_r &= 0 \\ x_n &= \bigwedge_{(m,n) \in E} f_{(m,n)}(x_m) \qquad n \in N - \{r\} \end{aligned} \tag{7-1}$$

These equations describe attribute propagation "locally." That is, they show the relation between attributes collected at adjacent basic blocks, starting with null information at the program entry.

The solutions of these equations approximate the following abstractly defined function known as the *meet-over-all-paths* solution to a data flow problem

$$y_n = \bigwedge \{f_p(0) : p \in \text{path}_G(r, n)\} \qquad n \in N \tag{7-2}$$

Here we define $f_p = f_{(n_{k-1}, n_k)} \circ f_{(n_{k-2}, n_{k-1})} \circ \ldots \circ f_{(n_1, n_2)}$ for each path $p = (n_1, n_2, \ldots, n_k)$. If p is null, then f_p is defined to be the identity map on L.

Many algorithms which solve the system of equations (7-1) are known by now. These algorithms fall into two main categories: (1) iterative algo-

rithms, which use only functional applications [Kild73, Hech75, Kam76, Hech77, Tarj76], and (2) elimination algorithms, which use functional compositions and meets [Alle76, Grah76, Tarj75b]. These elimination algorithms require some additional properties of F to allow elimination of program loops, a process which may require a computation of an infinite meet in F, unless such properties are assumed. Most of the algorithms in both categories yield the maximum fixed-point solution to Eqs. (7-1), which does coincide with the solution (7-2) provided that the data flow framework in question is distributive [Kild73], but which may fail to do so if the framework is only monotone [Kam77]. However, even in this latter case we still have $x_n \leq y_n$ for all $n \in N$; i.e., obtain an underestimated solution, which is always a safe one [Hech77]. In what follows, we will assume some basic knowledge of these classical data flow algorithms.

7-3. THE FUNCTIONAL APPROACH TO INTERPROCEDURAL ANALYSIS

In this section we present our first approach to interprocedural analysis. This approach treats each procedure as a structure of blocks and establishes relations between attribute data at its entry and related data at *any* of its nodes. Using these relations, attribute data is propagated directly through each procedure call.

We prepare for our description by giving some definitions and making some observations concerning the interprocedural nature of general programs. Let us first introduce the notion of an *interprocedural flow graph* of a computer program containing several procedures. We can consider two alternative representations of such a graph G. In the first representation, we have $G = \bigcup \{G_p : p \text{ is a procedure in the program}\}$, where, for each p, $G_p = (N_p, E_p, r_p)$, and where r_p is the entry block of p, N_p is the set of all basic blocks within p, and $E_p = E_p^0 \cup E_p^1$ is the set of edges of G_p. An edge $(m, n) \in E_p^0$ iff there can be a direct transfer of control from m to n (via a "go-to" or "if" statement, and $(m, n) \in E_p^1$ iff m is a call block and n is the block immediately following that call.

Thus this representation, which is the one to be used explicitly in our first approach, separates the flow graphs of individual procedures from each other.

A second representation, denoted by G^*, is defined as follows: $G^* = (N^*, E^*, r_1)$, where $N^* = \bigcup_p N_p$, and $E^* = E^0 \cup E^1$, where $E^0 = \bigcup_p E_p^0$ and an edge $(m, n) \in E^1$ iff either m is a call block and n is the entry block of the called procedure [in which case (m, n) is called a *call edge*], or if m is an exit block of some procedure p and n is a block immediately following a call to p [in which case (m, n) is called a *return edge*]. The call edge (m, r_p) and a return edge (e_q, n) are said to *correspond* to each other if $p = q$ and $(m, n) \in E_s^1$, for some procedure s. Here r_1 is the entry block of the main program, sometimes also denoted as r_{main}. Of course, not all paths through G^* are (even statically) feasible, in the sense of representing potentially valid execution paths, since the definition of G^* ignores the special nature of procedure calls and returns. For each $n \in N^*$ we define $\text{IVP}(r_1, n)$ as the set of all interprocedurally valid paths in G^* which lead from r_1 to n. A path $q \in \text{path}_{G^*}(r_1, n)$ is in $\text{IVP}(r_1, n)$ iff the sequence of all edges in q which are in E^1, which we will write as q_1 or $q|_{E^1}$, is *proper* in the following recursive sense:

1. A tuple q_1 which contains no return edges is proper.
2. If q_1 contains return edges, and i is the smallest index in q_1 such that $q_1(i)$ is a return edge, then q_1 is proper if $i > 1$ and $q_1(i-1)$ is a call edge corresponding to the return edge $q_1(i)$, and after deleting those two components from q_1, the remaining tuple is also proper.

Remark: It is interesting to note that the set of all proper tuples over E^1, as well as $\bigcup_n \text{IVP}(r_1, n)$, can be generated by a context-free grammar (but not by a regular grammar), in contrast with the set of all possible paths in G^*, which is regular.

For each procedure p and each $n \in N_p$, we also define $\text{IVP}_0(r_p, n)$ as the set of all interprocedurally valid paths q in G^* from r_p to n such that each procedure call in q is completed by a subsequent corresponding return edge in q. More precisely, a path $q \in \text{path}_{G^*}(r_p, n)$ is in $\text{IVP}_0(r_p, n)$ iff $q_1 = q|_{E^1}$ is *complete*, in the following recursive sense.

1. The null tuple is complete.
2. A tuple q_1 is complete if it is either a concatenation of two complete tuples, or else it starts with a call edge, terminates with the corresponding return edge, and the rest of its components constitute a complete tuple.

Example 1.

```
Main program          Procedure p
read a, b;            if a = 0 then return;
t := a * b;           else
call p;                 a := a - 1;
t := a * b;             call p;
print t;                t := a * b
stop;                 endif;
end                   return;
                      end
```

This set of equations possesses a maximum fixed-point solution defined as follows: Let F be ordered by writing $g_1 \geq g_2$ for $g_1, g_2 \in F$ iff $g_1(x) \geq g_2(x)$ for all $x \in L$.

Start by putting

$$\phi^0_{(r_p, r_p)} = \mathbf{id}_L \qquad \text{for each procedure } p$$
$$\phi^0_{(r_p, n)} = f_\Omega \qquad \text{for each } n \in N_p - \{r_p\}$$

and then apply Eqs. (7-4) iteratively in a round-robin fashion to obtain new approximations to the ϕ's. (This can be done using iterations of either the Gauss-Seidel type or the Jacobi type, though the former is a better approach.) Let $\phi^i_{(r_p,n)}$ denote the ith approximation computed in this manner. Since $\phi^0_{(r_p,n)} \geq \phi^1_{(r_p,n)}$ for all p, n, it follows inductively that $\phi^i_{(r_p,n)} \geq \phi^{i+1}_{(r_p,n)}$ for each p, n and $i \geq 0$.

A problem which arises here is that F need not in general be a bounded semilattice, even if L is bounded. If L is finite, then F must be finite and therefore bounded, but if L is not finite, F need not in general be bounded.

Nevertheless, even if the sequence $\{\phi^j_{(r_p,n)}\}_{j\geq 0}$ is infinite for some p, n, we still can define its limit, denoted by $\phi_{(r_p,n)}$, as follows: For each $x \in L$, the sequence $\{\phi^j_{(r_p,n)}(x)\}_{j\geq 0}$ is decreasing in L, and since L is bounded, it must be finite, and we define $\phi_{(r_p,n)}(x)$ as its limit. [To ensure that $\phi_{(r_p,n)} \in F$ we must impose another condition upon F, namely: for each decreasing sequence $\{g_t\}_{t>0}$ of functions in F, the limit defined as above is also in F. However, since we will assume that L is finite (so that F is bounded) in any practical application of this approach, we introduce this condition only temporarily, for the sake of the following abstract reasoning. Thus, the above process defines a solution also in F.] Thus, the above process defines a solution $\{\phi_{(r_p,n)}\}_{p,n}$ to Eqs. (7-4), though not necessarily effectively. It is easy to check that the limiting functions defined by the iterative process that we have described are indeed a solution, and that in fact they are the maximal fixed-point solution of (7-4).

Having obtained this solution, we can use it to compute a solution to our data flow problem. For each basic block n let $x_n \in L$ denote the information available at the start of n. Then we have the following set of equations:

$$x_{r_{\text{main}}} = 0 \in L \tag{7-5a}$$

$$x_{r_p} = \bigwedge \{\phi_{(r_q,c)}(x_{r_q}) : q \text{ is a procedure and } c \text{ is a call to } p \text{ in } q\} \qquad \text{for each procedure } p \tag{7-5b}$$

$$x_n = \phi_{(r_p,n)}(x_{r_p}) \qquad \text{for each procedure } p, \text{ and } n \in N_p - \{r_p\} \tag{7-5c}$$

These equations can be (effectively) solved by a standard iterative algorithm, which yields the maximal fixed-point solution of (7-5).

We illustrate the above procedure for solution of Eqs. (7-4) and (7-5) by applying it to Example 1 introduced earlier, in which we suppose that available expressions analysis is to be performed. Our interprocedural analysis will show that $a * b$ is available upon exit from the recursive procedure p, so that its second computation in the main program is redundant and can therefore be eliminated. (Some traditional interprocedural methods will fail to detect this fact, since the expression $a * b$ is killed in p.) For simplicity we will only show that part of the analysis which pertains directly to the single expression $a * b$. Assuming this simplification, $L = \{0, 1, \Omega\}$, where 1 indicates that $a * b$ is available and 0 that it is not, and F contains precisely four functions [recall that $f(\Omega) = \Omega$ always]; the "constant" functions $\mathbf{0}$ and $\mathbf{1}$, $\mathbf{id}_L$ and f_Ω. With these notations, Eqs. (7-4) read

$$\phi_{(r_1,r_1)} = \mathbf{id}$$
$$\phi_{(r_1,c_1)} = \mathbf{1} \circ \phi_{(r_1,r_1)}$$
$$\phi_{(r_1,n_1)} = \phi_{(r_2,c_2)} \circ \phi_{(r_1,c_1)}$$
$$\phi_{(r_1,e_1)} = \mathbf{1} \circ \phi_{(r_1,n_1)}$$
$$\phi_{(r_2,r_2)} = \mathbf{id}$$
$$\phi_{(r_2,c_2)} = \mathbf{0} \circ \phi_{(r_2,r_2)}$$
$$\phi_{(r_2,n_2)} = \phi_{(r_2,e_2)} \circ \phi_{(r_2,c_2)}$$
$$\phi_{(r_2,e_2)} = [\mathbf{id} \circ \phi_{(r_2,r_2)}] \wedge [\mathbf{1} \circ \phi_{(r_2,n_2)}]$$

Table 7-1 summarizes the iterative solution of these equations:

Table 7-1

Function	*Initial value*	*After one iteration*	*After two iterations*	*After three iterations*
$\phi_{(r_1,r_1)}$	id	id	id	id
$\phi_{(r_1,c_1)}$	f_Ω	1	1	1
$\phi_{(r_1,n_1)}$	f_Ω	f_Ω	1	1
$\phi_{(r_1,e_1)}$	f_Ω	f_Ω	1	1
$\phi_{(r_2,r_2)}$	id	id	id	id
$\phi_{(r_2,c_2)}$	f_Ω	0	0	0
$\phi_{(r_2,n_2)}$	f_Ω	f_Ω	0	0
$\phi_{(r_2,e_2)}$	f_Ω	id	id	id

Thus, the first stage of our solution stabilizes after three iterations. Next we solve Eqs. (7-5), which read as follows:

$$x_{r_1} = 0$$
$$x_{r_2} = \phi_{(r_1,c_1)}(x_{r_1}) \wedge \phi_{(r_2,c_2)}(x_{r_2})$$
$$= \mathbf{1}(x_{r_1}) \wedge \mathbf{0}(x_{r_2})$$

For these equations we see after two iterations that

$$x_{r_1} = x_{r_2} = 0$$

from which, using (7-5c), we obtain the complete solution

$$x_{r_1} = x_{r_2} = x_{c_2} = x_{n_2} = x_{e_2} = 0$$
$$x_{c_1} = x_{n_1} = x_{e_1} = 1$$

i.e., $a * b$ is available at the start of n_1, which is what we wanted to show.

Next we shall analyze the properties of the solution of Eqs. (7-4) and (7-5) as defined above. As in intraprocedural analysis our main objective is to show that this solution coincides with the meet-over-all-paths solution defined (in the interprocedural case) as follows:

$$\psi_n = \bigwedge \{f_q : q \in \text{IVP}(r_{\text{main}}, n)\} \in F \qquad \text{for each } n \in N^* \tag{7-6}$$

$$y_n = \psi_n(0) \qquad \text{for each } n \in N^* \qquad \text{(this is the meet-over-all-paths solution)} \tag{7-7}$$

Lemma 7-3.2. Let $n \in N_p$ for some procedure p. Then

$$\phi_{(r_p,n)} = \bigwedge \{f_q : q \in \text{IVP}_0(r_p, n)\}$$

Proof. We first prove, by induction on i, that for all $i \geq 0$

$$\phi^i_{(r_p,n)} \geq \bigwedge \{f_q : q \in \text{IVP}_0(r_p, n)\}$$

Indeed, for $i = 0$, if $n = r_p$ then $\phi^0_{(r_p,r_p)} = \mathbf{id}_L = f_{q_0}$, where $q_0 \in \text{IVP}_0(r_p, r_p)$ is the empty path from r_p to r_p, so that $\phi^0_{(r_p,r_p)} \geq \bigwedge \{f_q : q \in \text{IVP}_0(r_p, r_p)\}$. If $n \neq r_p$ then $\phi^0_{(r_p,n)} = f_\Omega \geq f$ for all $f \in F$. Thus the assertion is true for $i = 0$.

Suppose that it is true for some i. For either kind of iterative computation of the functions ϕ^{i+1} using Eqs. (7-4) we have

$$\phi^{i+1}_{(r_p,n)} \geq \bigwedge_{(m,n)\in E_p} (h_{(m,n)} \circ \phi^i_{(r_p,m)})$$
$$\geq \bigwedge_{(m,n)\in E_p} (h_{(m,n)} \circ \bigwedge \{f_q : q \in \text{IVP}_0(r_p, m)\})$$

for each procedure p and $n \in N_p - \{r_p\}$. (Note here that if $n = r_p$, then $\phi^{i+1}_{(r_p,n)} = \phi^i_{(r_p,n)} = \phi^0_{(r_p,n)} \geq \{f_q : q \in \text{IVP}_0(r_p, n)\}$. Our chain of equalities and inequalities then continues.)

$$= \bigwedge_{(m,n)\in E_p^0} (f_{(m,n)} \circ \bigwedge \{f_q : q \in \text{IVP}_0(r_p, m)\}) \wedge$$
$$\bigwedge_{\substack{(m,n)\in E_p^1 \\ m \text{ calls } p'}} (\phi_{(r_{p'},e_{p'})} \circ \bigwedge \{f_q : q \in \text{IVP}_0(r_p, m)\})$$
$$\geq \bigwedge_{(m,n)\in E_p^0} (\bigwedge \{f_{q\|(m,n)} : q \in \text{IVP}_0(r_p, m)\}) \wedge$$
$$\bigwedge_{\substack{(m,n)\in E_p^1 \\ m \text{ calls } p'}} (\bigwedge \{f_{q'} : q' \in \text{IVP}_0(r_{p'}, e_{p'})\} \circ \bigwedge \{f_q : q \in \text{IVP}_0(r_p, m)\})$$
$$= \bigwedge_{(m,n)\in E_p^0} (\bigwedge \{f_{q\|(m,n)} : q \in \text{IVP}_0(r_p, m)\}) \wedge$$
$$\bigwedge_{\substack{(m,n)\in E_p^1 \\ m \text{ calls } p'}} (\bigwedge \{f_{q\|(m,r_{p'})\|q'\|(e_{p'},n)} : q \in \text{IVP}_0(r_p, m),\ q' \in \text{IVP}_0(r_{p'}, e_{p'})\})$$

It is easily checked that for each function f_{q_1} appearing in the last right-hand side, $q_1 \in \text{IVP}_0(r_p, n)$. Hence, this last right-hand side must be

$$\geq \{f_q : q \in \text{IVP}_0(r_p, n)\}$$

The same inequality is then seen to apply to the limit function $\phi_{(r_p,n)}$ as well.

To prove the inequality in the other direction, we will show that for each $q \in \text{IVP}_0(r_p, n)$, $f_q \geq \phi_{(r_p,n)}$. This will be proven by induction on the length of q. If this length is 0, then n must be equal to r_p and $f_q = \phi_{(r_p,r_p)} = \mathbf{id}_L$. Suppose that the assertion is true for all p, n and all $q \in \text{IVP}_0(r_p, n)$ whose length is $\leq k$, and let there be given p, n, q such that the length of q is $k + 1$. Let (m, n) be the last edge in q, so that we can write $q = q_1 \| (m, n)$.

If $(m, n) \in E_p^0$, then $q_1 \in \text{IVP}_0(r_p, m)$ and its length is $\leq k$. Therefore $f_{q_1} \geq \phi_{(r_p,m)}$ and by (7-4) we have

$$f_q = f_{(m,n)} \circ f_{q_1} \geq h_{(m,n)} \circ \phi_{(r_p,m)} \geq \phi_{(r_p,n)}$$

If $(m, n) \in E^1$, then $m = e_{p'}$ for some procedure p'. It is easily seen from the definition of IVP_0, that q can be decomposed as $q_1 \| (m_1, r_{p'}) \| q_2 \| (e_{p'}, n)$, such that $(m_1, n) \in E_p^1$, $q_1 \in \text{IVP}_0(r_p, m_1)$, $q_2 \in \text{IVP}_0(r_{p'}, e_{p'})$. Since $f_{(m_1,r_{p'})} = f_{(e_{p'},n)} = \mathbf{id}_L$ (since m_1 and $e_{p'}$ are single instruction blocks, containing only an interprocedural branch instruction), we have

$$f_q = f_{q_2} \circ f_{q_1}$$

But both q_1 and q_2 have length $\leq k$, so that by Eq. (7-4) and the induction hypothesis, we obtain

$$f_q \geq \phi_{(r_{p'},e_{p'})} \circ \phi_{(r_p,m_1)} = h_{(m_1,n)} \circ \phi_{(r_p,m_1)} \geq \phi_{(r_p,n)}$$

This proves our assertion, from which the lemma follows immediately. ■

Let us now define, for each basic block n,

$$\chi_n = \bigwedge \{\phi_{(r_{p_j},n)} \circ \phi_{(r_{p_{j-1}},c_{j-1})} \circ \cdots \circ \phi_{(r_{p_1},c_1)} : \tag{7-8}$$
$p_1 =$ main program, p_j is the procedure containing n,
and for each $i < j$, c_j is a call to p_{i+1} from $p_i\}$

$$z_n = \chi_n(0) \tag{7-9}$$

Theorem 7-3.3. $\psi_n = \chi_n$ for each $n \in N^*$.

Proof. Let $q \in \text{IVP}(r_{\text{main}}, n)$. By Lemma 7-3.1, q admits a decomposition $q = q_1 \| (c_1, r_{p_2}) \| q_2 \| \cdots \| (c_{j-1}, r_{p_j}) \| q_j$ as in Eq. (7-3); i.e., there exist procedures $p_1 =$ main program, $p_2, \ldots, p_j =$ the procedure containing n, and calls $c_1, \ldots, c_{j-1}$ such that for each $i < j$, c_i is a call to p_{i+1} from p_i, and $q_i \in \text{IVP}_0(r_{p_i}, c_i)$, and also $q_j \in \text{IVP}_0(r_{p_j}, n)$.

Thus, by Lemma 7-3.2, we have

$$f_q = f_{q_j} \circ f_{q_{j-1}} \circ \ldots \circ f_{q_1} \geq \phi_{(r_{p_j}, n)} \circ \phi_{(r_{p_{j-1}}, c_{j-1})} \circ \ldots \circ \phi_{(r_{p_1}, c_1)} \geq \chi_n$$

Hence, $\psi_n \geq \chi_n$.

Conversely, let $p_1, \ldots, p_j, c_1, \ldots, c_{j-1}$ be as in Eq. (7-8). By Lemma 7-3.2 we have

$$\begin{aligned}
&\phi_{(r_{p_j}, n)} \circ \phi_{(r_{p_{j-1}}, c_{j-1})} \circ \ldots \circ \phi_{(r_{p_1}, c_1)} \\
&\quad = \bigwedge \{f_{q_j} \circ f_{q_{j-1}} \circ \ldots \circ f_{q_i} : q_i \in \mathrm{IVP}_0(r_{p_i}, c_i) \\
&\qquad\qquad \text{for each } i < j \text{ and } q_j \in \mathrm{IVP}_0(r_{p_j}, n)\} \\
&\quad = \{f_{q_1 \| (c_1, r_{p_2}) \| q_2 \ldots \| (c_{j-1}, r_{p_j}) \| q_j} : \text{same as above}\}
\end{aligned}$$

By Lemma 7-3.1, each concatenated path in the last set expression belongs to $\mathrm{IVP}(r_{\mathrm{main}}, n)$. Thus, the last expression is

$$\geq \bigwedge \{f_q : q \in \mathrm{IVP}(r_{\mathrm{main}}, n)\} = \psi_n$$

Therefore $\chi_n \geq \psi_n$ so that χ_n and ψ_n are equal for each $n \in N^*$. ■

We can now prove our main result:

Theorem 7-3.4. For each basic block $n \in N^*$, $x_n = y_n = z_n$.

Proof. It is immediate from Theorem 7-3.3 that $y_n = z_n$ for each $n \in N^*$. We claim that $x_{r_p} = z_{r_p}$ for all procedures p in the program. By Eqs. (7-5c), (7-8), and (7-9) this will imply that $x_n = z_n$ for all n.

To prove our claim, we define a new flow graph $G_c = (N_c, E_c, r_1)$, where N_c is the set of all entry blocks and call blocks in the program.

$E_c = E_c^0 \cup E_c^1$ is the set of edges of G_c. An edge $(m, n) \in E_c^0$ iff m is the entry node of some procedure p and n is a call within p. Moreover, $(m, n) \in E_c^1$ iff m is a call to some procedure p and n is the entry of p. As before, r_1 is the entry block of the main program. We now define a data flow problem for G_c by associating a data-propagating map $g_{(m,n)} \in F$ with each $(m, n) \in E_c$, in such a way that

$$g_{(m,n)} = \begin{cases} \phi_{(m,n)} & \text{if } (m, n) \in E_c^0 \\ \mathbf{id}_L & \text{if } (m, n) \in E_c^1 \end{cases}$$

It is clear that Eqs. (7-5a) and (7-5b) are equivalent to the iterative equations for the new data flow problem. On the other hand, Eqs. (7-8) and (7-9) define the meet-over-all paths solution for the same problem, if we substitute only entry blocks or call blocks for n. Since F is assumed to be distributive, it follows by Kildall's Theorem [Kild73], that $x_{r_p} = z_{r_p}$ for each procedure p, and this completes the proof of our theorem. ■

It is now time to discuss the pragmatic problems that will affect attempts to use the functional approach to interprocedural analysis that we have sketched. The main problem is, obviously, how to compute the ϕ's effectively if L is not finite (or if F is not bounded). As examples below will show, in the most general case the functional approach does not and cannot yield an effective algorithm for solving Eqs. (7-4) and (7-5). Moreover, even if the iterative computation of the ϕ's converges, we must still face the problem of space needed to represent these functions. Since the functional method that we have outlined manipulates the ϕ's directly, instead of just applying them to elements of L, it can increase the space required for data flow analysis if L is finite, and may even fail to give finite representation to the ϕ's if L is infinite. We note here that our functional approach belongs to the class of elimination algorithms for solving data flow problems (a class of methods which includes the interval-oriented algorithms of Cocke and Allen [Alle76], and Tarjan's fast elimination algorithms [Tarj75b]), since it uses functional compositions and meets in addition to functional applications. All such elimination algorithms face similar problems, and in practical terms are therefore limited to cases in which the elements of F possess some compact and simple representation, in which meets and compositions of elements of F can be easily calculated, and in which F is a bounded semilattice (or else relevant infinite meets in F are easy to calculate). This family of cases includes the classical "bit-vector" data flow problems (e.g., analysis for available expressions, use-definition chaining, cf. [Hech77]).

It is interesting to ask whether it is possible to modify the functional approach so that it avoids explicit functional compositions and meets, and thus becomes an *iterative* approach. This is possible if L is finite, and an implementation having this property will be sketched below.

The following example will illustrate some of the pragmatic problems noted above, and also some potential advantages of the functional approach over any iterative variant of it. Suppose that we want to perform constant propagation (see, e.g., [Hech77] for a description of the standard framework used in this analysis). Consider the following code:

Example 2.

```
Main program          Procedure p
A := 0;               if cond then
call P;                 A := A + 1;
print A;                call p;
end                     A := A - 1
                      endif;
                      return;
                      end
```

If we do not allow symbolic representation of the ϕ's, then, in any iterative approach, we shall have to compute $\phi_{(r_p, e_p)}(\{(A, 0)\})$, for which we need to compute (for the second level of recursion) $\phi_{(r_p, e_p)}(\{(A, 1)\})$, etc., computing $\phi_{(r_p, e_p)}(\{(A, k)\})$ for all integers $k \geq 0$. Thus, an iterative approach would diverge in this example.

However, if symbolic or some other compact representation of the ϕ's is possible, then it can be advantageous to manipulate these functions directly, without applying them to elements of L till their final values have been obtained. This can give us an overall description of their behavior, allowing them to be calculated in relatively few iterations. For example, in the example shown above, it is easily checked that $\phi_{(r_p, e_p)}$ is found to be $\mathbf{id}_L$ after two iterations.

However, convergence of the purely functional approach is not ensured in general. To see this, consider the following slight modification of the preceding example.

Example 3.

Main program	*Procedure p*
$A := 0$;	**if** cond **then**
call p;	$A := A + 2 + \text{sign}(A - 100)$;
print A;	**call** p;
end	$A := A - 1$;
	endif
	return;
	end

It is fairly easy to check that the purely functional approach (which uses symbolic representation of the ϕ's) will diverge if negative integers are included in the program domain. Intuitively, this is due to the fact that it takes more than $100 + k$ iterations through Eqs. (7-4) to detect that $\phi_{(r_p, e_p)}(\{(A, -k)\}) = \varnothing$ for all $k \geq 0$.

Remark: The data flow framework required for constant propagation is in general not distributive. However, it can be shown that the standard framework for constant propagation becomes distributive if the program contains only one single variable and each propagation between adjacent basic blocks either sets the value of that variable to some constant, or calculates the output value of the variable from its input value in a one-to-one manner, as in the above examples.

These examples indicate that if L is not finite, divergence can actually occur. If L is infinite but F is bounded, then a symbolic functional approach would converge, whereas an iterative approach could still diverge if infinite

space were needed to represent the ϕ's. Moreover, we have at present no simple criterion which guarantees that F is bounded in cases in which L is infinite. For these reasons, we will henceforth assume that L is a finite semilattice. We can then summarize our results up to this point as follows:

Corollary 7-3.5. If (L, F) is a distributive data flow framework and the semilattice L is finite, then the iterative solution of Eqs. (7-4) converges and, together with Eqs. (7-5), yields the meet-over-all-interprocedurally-valid-paths solution (7-7).

Next we shall sketch an algorithm which implements the functional approach for general frameworks with a finite semilattice L. We do not assume that any compact representation for elements of F is available, nor that their compositions and meets are easy to calculate, but instead give a purely iterative representation to the functional approach, which avoids all functional compositions and meets and also computes the ϕ's only for values which reach some relevant procedure entry during propagation.

Our algorithm is workpile-driven. The functions ϕ are represented by a two-dimensional partially defined map PHI: $N^* \times L \rightarrow L$, so that for each $n \in N^*$, $x \in L$, PHI(n, x) represents $\phi_{(r_p, n)}(x)$, where p is the procedure containing n. The substeps of the algorithm are as follows:

1. Initialize WORK $:= \{(r_1, 0)\}$, PHI$(r_1, 0) := 0$. [WORK is a subset of $N^* \times L$, containing pairs (n, x) for which PHI(n, x) has been changed and its new value has not yet been propagated to successor blocks of n.]
2. While WORK $\neq \varnothing$, remove an element (n, x) from WORK, and let $y =$ PHI(n, x).
   (a) If n is a call block in a procedure q, calling a procedure p, then
      (i) If $z =$ PHI(e_p, y) is defined, let m be the unique block such that $(n, m) \in E_q^1$, and *propagate* (x, z) to m. [By this we mean: assign PHI$(m, x) :=$ PHI$(m, x) \wedge z$, where undefined PHI(m, x) is interpreted as Ω; if the value of PHI(m, x) has changed, add (m, x) to WORK.]
      (ii) Otherwise, propagate (y, y) to r_p. This will trigger propagation through p, which will later trigger propagation to the block following n in q (see below).
   (b) If n is the exit block of some procedure p, i.e., $n = e_p$, find all pairs (m, u) such that m is a block following some call c to p, and PHI$(c, u) = x$, and for each such pair propagate (u, y) to m.

(c) If n is any other block in some procedure p, then, for each $m \in E_p^0 - \{n\}$, propagate $(x, f_{(n,m)}(y))$ to m.

3. Repeat step (2) till WORK $= \varnothing$. When this happens, PHI represents the desired ϕ functions, computed only for "relevant" data values, from which the x solution can be readily computed as follows:

$$x_n = \bigwedge_{a \in L} \text{PHI}(n, a) \qquad \text{for each } n \in N^*$$

Step (3) thus implies that in the implementation we have sketched separate analysis to compute the x solution is unnecessary.

We omit analysis of the above algorithm, which in many ways would resemble an analysis of the abstract approach. However, so as not to avoid the issue of the correctness of our algorithm, we outline a proof of its total correctness, details of which can be readily filled in by the reader. The proof consists of several steps:

1. The algorithm terminates if L is finite, since each element (n, x) of $N^* \times L$ (which is a finite set) is added to WORK only a finite number of times, because the values assumed by PHI(n, x) upon successive insertions constitute a strictly decreasing sequence in L, which must of course be finite.
2. We claim that for each $n \in N^*$,

$$x_n \leq \bigwedge_{a \in L} \text{PHI}(n, a) \tag{7-10}$$

To prove this claim, we show, using induction on the sequence of steps executed by the algorithm, that at the end of the ith step, $x_n \leq \bigwedge_{a \in L} \text{PHI}^i(n, a)$, for each $n \in N^*$, $a \in L$, where PHI^i denotes the value of PHI at the end of the ith step. In executing the ith step, we propagate some pair $(a, b) \in L \times L$ to some $n \in N^*$. By examining all possible cases, it is easy to show, using the induction hypothesis, that $x_n \leq b$, from which (7-10) follows immediately.
3. In order to prove the converse inequality, it is sufficient, by Theorem 7-3.4, to show that for each $n \in N^*$ and $q \in \text{IVP}(r_1, n)$, $f_q(0) \geq \bigwedge_{a \in L} \text{PHI}(n, a)$. To do this, we first need the following assertion:

Assertion. Let p be a procedure, $n \in N_p$ and $a \in L$ for which PHI(n, a) has been computed by our algorithm. Then, for each path $q \in \text{IVP}_0(r_p, n)$, $f_{\hat{q}}(a) \geq \text{PHI}(n, a)$.

Proof. We proceed by induction on the length of q. This is trivial if the length $= 0$. Suppose that it is true for all p, n, a, and q with length less than some $k > 0$, and let $q \in \text{IVP}_0(r_p, n)$ be of length k.

Write $q = \hat{q} \,||\, (m, n)$ and observe that either $(m, n) \in E^0$, in which case

$$f_q(a) = f_{(m,n)}(f_{\hat{q}}(a)) \geq f_{(m,n)}(\text{PHI}(m, a)) \geq \text{PHI}(n, a)$$

(the last inequality follows from the structure of our algorithm), or (m, n) is a return edge, in which case q can be written as $\hat{q}_1 \,||\, (c, r_{p'}) \,||\, \hat{q}_2 \,||\, (m, n)$, where $\hat{q}_1 \in \text{IVP}_0(r_p, c)$, $\hat{q}_2 \in \text{IVP}_0(r_{p'}, m)$, and we have

$$f_q(a) = f_{\hat{q}_2}(f_{\hat{q}_1}(a)) \geq f_{\hat{q}_2}(\text{PHI}(c, a)) \geq \text{PHI}(m, \text{PHI}(c, a)) \geq \text{PHI}(n, a)$$

4. Now let q be any path in IVP(r_1, n). Decompose q as in (7-3), $q = q_1 \,||\, (c_1, r_{p_2}) \,||\, \ldots \,||\, (c_j, r_{p_{j+1}}) \,||\, q_{j+1}$. Then, using the monotonicity of F, we have

$$f_{q_1}(0) \geq \text{PHI}(c_1, 0) = a_1$$

$$f_{q_2}(f_{q_1}(0)) \geq f_{q_2}(a_1) \geq \text{PHI}(c_2, a_1) = a_2$$

[This is because our algorithm will propagate (a_1, a_1) to r_{p_2}, so that PHI(c_2, a_1) will eventually have been computed.] Continuing in this manner, we obtain $f_q(0) \geq \text{PHI}(n, a_j)$, which proves (3). This completes the proof of the total correctness of our algorithm. ■

Example 4. Consider Example 1 given above. The steps taken by our iterative algorithm are summarized in Table 7-2 [where, for notational convenience, we represent PHI as a set of triplets, so that it contains (a, b, c) iff PHI$(a, b) = c$]:

Table 7-2

Initially			$(r_1, 0, 0)$	$\{(r_1, 0)\}$
Propagate	*From*	*To*	*Entries added to PHI*	*WORK*
(0, 1)	r_1	c_1	$(c_1, 0, 1)$	$\{(c_1, 0)\}$
(1, 1)	c_1	r_2	$(r_2, 1, 1)$	$\{(r_2, 1)\}$
(1, 0)	r_2	c_2	$(c_2, 1, 0)$	$\{(c_2, 1)\}$
(1, 1)	r_2	e_2	$(e_2, 1, 1)$	$\{(c_2, 1), (e_2, 1)\}$
(0, 0)	c_2	r_2	$(r_2, 0, 0)$	$\{(e_2, 1), (r_2, 0)\}$
(0, 1)	e_2	n_1	$(n_1, 0, 1)$	$\{(r_2, 0), (n_1, 0)\}$
(0, 0)	r_2	c_2	$(c_2, 0, 0)$	$\{(n_1, 0), (c_2, 0)\}$
(0, 0)	r_2	e_2	$(e_2, 0, 0)$	$\{(n_1, 0), (c_2, 0), (e_2, 0)\}$
(0, 1)	n_1	e_1	$(e_1, 0, 1)$	$\{(c_2, 0), (e_2, 0), (e_1, 0)\}$
(0, 0)	c_2	n_2	$(n_2, 0, 0)$	$\{(e_2, 0), (e_1, 0), (n_2, 0)\}$
(1, 0)	e_2	n_2	$(n_2, 1, 0)$	$\{(e_1, 0), (n_2, 0), (n_2, 1)\}$
(0, 0)	e_2	n_2	—	$\{(e_1, 0), (n_2, 0), (n_2, 1)\}$
—	e_1	—	—	$\{(n_2, 0), (n_2, 1)\}$
(0, 1)	n_2	e_2	—	$\{(n_2, 1)\}$
(1, 1)	n_2	e_2	—	$\varnothing$

Finally we compute the x solution of Eqs. (7-4) and (7-5) in step (3) of our iterative algorithm as follows:

$$x_{r_1} = \text{PHI}(r_1, 0) = 0$$
$$x_{c_1} = \text{PHI}(c_1, 0) = 1$$
$$x_{n_1} = \text{PHI}(n_1, 0) = 1$$
$$x_{e_1} = \text{PHI}(e_1, 0) = 1$$
$$x_{r_2} = \text{PHI}(r_2, 0) \wedge \text{PHI}(r_2, 1) = 0$$
$$x_{c_2} = \text{PHI}(c_2, 0) \wedge \text{PHI}(c_2, 1) = 0$$
$$x_{n_2} = \text{PHI}(n_2, 0) \wedge \text{PHI}(n_2, 1) = 0$$
$$x_{e_2} = \text{PHI}(e_2, 0) \wedge \text{PHI}(e_2, 1) = 0$$

Remark: In our treatment of the functional approach, we have deliberately avoided the issue of its efficient and pragmatic implementation for special simple frameworks in which elimination is feasible. For example, the iterative solution of Eqs. (7-4) may not be the best approach and could be replaced, e.g., by interval-based analysis [Alle76]. Also one might benefit from processing procedures in some useful order, as in [Alle74]. In this chapter we have preferred to emphasize the general approach and its analysis and general applicability. Details of an efficient, pragmatic, and interval-based implementation will be discussed in a subsequent paper.

7-4. THE CALL-STRING APPROACH TO INTERPROCEDURAL ANALYSIS

We now describe a second approach to interprocedural analysis. This approach views procedure calls and returns in much the same way as any other transfer of control, but takes care to avoid propagation along interprocedurally invalid paths. This is achieved by tagging propagated data with an encoded history of procedure calls along which that data has propagated. This contrasts with the idea of tagging it by the lattice value attained on entrance to the most recent procedure, as in the functional approach. In our second approach, this "propagation history" is updated whenever a call or a return is encountered during propagation. This makes interprocedural flow explicit and increases the accuracy of propagated information. Moreover, by passing to approximate, but simpler, encodings of the call history, we are able to derive approximate, underestimated information for any data flow analysis, which should nevertheless remain more accurate than that derived by ignoring all interprocedural constraints on the propagation. The fact that this second approach allows us to perform approximate data flow analysis even in cases in which convergence of a full analysis is not ensured or when the space requirements of a full analysis is prohibitive gives this second approach real advantages.

We will first describe our second approach in a somewhat abstract manner. We will then suggest several modifications which yield relatively efficient convergent algorithms for many important cases.

As before, we suppose that we are given an interprocedural flow graph G, but this time we make an explicit use of the second representation $G^* = (N^*, E^*, r_1)$ of G. That is, we blend all procedures in G into one flow graph, but distinguish between intraprocedural and interprocedural edges.

Definition. A *call string* γ is a tuple of call blocks $c_1, c_2, \ldots, c_j$ in N^* for which there exists an execution path $q \in \text{IVP}(r_1, n)$, terminating at some $n \in N^*$, such that the decomposition (7-3) of q has the form $q_1 \| (c_1, r_{p_2}) \| q_2 \| \ldots \| (c_j, r_{p_{j+1}}) \| q_{j+1}$ where $q_i \in \text{IVP}_0(r_{p_i}, c_i)$ for each $i \leq j$ and $q_{j+1} \in \text{IVP}_0(r_{p_{j+1}}, n)$. To show the relation between q and γ we introduce a map CM such that $\text{CM}(q) = \gamma$. By the uniqueness of the decomposition (7-3) (cf. Lemma 7-3.1) this map is single-valued. γ can be thought of as the contents of a stack containing the locations of all call instructions which have not yet been completed.

Example 5. In Example 1 of Section 7-3 the following call strings are possible:

λ—the null call string, (c_1), (c_1c_2), $(c_1c_2c_2)$, etc.

However, for each n in the main program and each $q \in \text{IVP}(r_1, n)$, $\text{CM}(q) = \lambda$; no other call strings can "tag" such paths. All the other call strings "tag" paths leading to nodes in the procedure p, and indicate all possible calling sequences (i.e., contents of a stack of all uncompleted calls at some point of the program's execution) that can materialize as execution advances to p.

Let Γ denote the space of all call strings γ corresponding (in the above sense) to interprocedurally valid paths in G^*. Note that if G^* is nonrecursive, then Γ is finite; otherwise Γ will be infinite, and as we shall soon see, this can cause difficulties for our approach.

Let (L, F) be the data flow framework under consideration. We define a new framework (L^*, F^*), which reflects the interprocedural constraints in G^* in an implicit manner, as follows: $L^* = L^\Gamma$, i.e., L^* is the space of all maps from Γ into L. Since we assume that L contains a largest "undefined" element Ω, we can identify L^* with the space of all partially defined maps from Γ into $L - \{\Omega\}$. If Γ is finite, then the representation of L^* as a space of partially defined maps is certainly more efficient, but for abstract purposes the first representation is more convenient. (In examples below, however, we will use partial map representation for elements of L^*.) If $\xi \in L^*$ and $\gamma \in \Gamma$, then heuristically $\xi(\gamma)$ denotes that part of the propagated data which has been propagated along execution paths in $\text{CM}^{-1}\{\gamma\}$.