[More79]. It emerges from these comparisons that this algorithm is very fast when applied separately to each procedure. For interprocedural application, the only added task is the solution of systems in the data flow analysis phase, and this is known to be fast. Although this extended algorithm has not yet been implemented, it can reasonably be assumed to be efficient.

6-6. CONCLUSION

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

In some cases, optimization can be improved by iterating part or all of this process. The algorithm can be used for recursive procedures, but in this case, some approximations are made in the data flow analysis phase in order to avoid unpredictable costs in the algorithm. As presented here, the algorithm runs in time linear in the size of the program. Implementation of the basic algorithm has been shown to be efficient and well within the state of the art of actual compilers. The same claims seem to be applicable to its interprocedural 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:

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

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 program is transformed into the interprocedural flow graph in Fig. 7-1, which includes both edges of $\bigcup_p E_p$ and of E^* (where solid arrows denote intraprocedural edges, dotted arrows denote edges in $\bigcup_p E_p^1$, and dashed arrows denote interprocedural edges, i.e., edges in E^1):

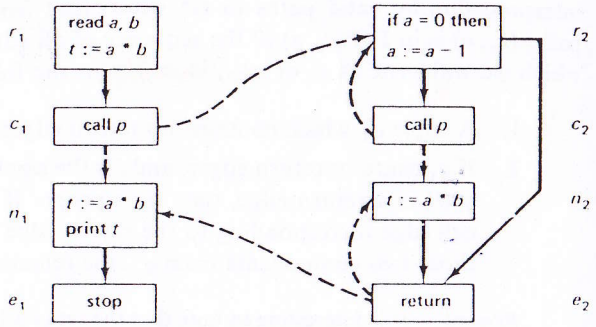

Figure 7-1

The following path q_1 in G^* is an interprocedurally valid path throughout the program [i.e., $q_1 \in \mathrm{IVP}(r_1, e_1)$]: $q_1 = (r_1, c_1, r_2, c_2, r_2, e_2, n_2, e_2, n_1, e_1)$; however, the path $q_2 = (r_1, c_1, r_2, c_2, r_2, e_2, n_1, e_1)$ is not, since $q_2|_{E^1} \equiv ((c_1, r_2), (c_2, r_2), (e_2, n_1))$ is not proper, as can be easily checked. Similarly the path $q_3 = (r_2, c_2, r_2, e_2, n_2)$ is in $\mathrm{IVP}_0(r_2, n_2)$, whereas $q_4 = (r_2, c_2, r_2, c_2, e_2, n_2)$ is not. Heuristically, q_3 reaches n_2 in the same incarnation of p in which it has started, but q_4 reaches n_2 in another incarnation which has been invoked by the initial one.

The notions introduced above appear in the following Path Decomposition Lemma:

Lemma 7-3.1. Let $n \in N^*$ and $q \in \mathrm{IVP}(r_1, n)$. Then there exist procedures $p_1, p_2, \ldots, p_j$, where p_1 is the main program and p_j the procedure containing n, and calls $c_1, \ldots, c_{j-1}$ such that for each $i < j$ c_i is in p_i and calls p_{i+1}, and q can be represented as

$$q = q_1 \| (c_1, r_{p_2}) \| q_2 \| \ldots \| (c_{j-1}, r_{p_j}) \| q_j \tag{7-3}$$

where for each $i < j$, $q_i \in \mathrm{IVP}_0(r_{p_i}, c_i)$ and $q_j \in \mathrm{IVP}_0(r_{p_j}, n)$. Conversely, any path which admits such a decomposition is in $\mathrm{IVP}(r_1, n)$. Moreover, this decomposition is unique.

Proof. Let $q^* = q|_{E^1}$. If q^* is empty, then it is also complete, so that $q \in \mathrm{IVP}_0(r_1, n)$, and we have the trivial decomposition $q = q$ with $j = 1$ (n must belong to the main program in this case).

Otherwise, in view of the definition of a proper E^1-tuple, and by making repeated deletions of adjacent call edges and corresponding return edges, we can reduce q^* to a tuple q^{**} which is either a null tuple or a nonempty tuple containing only call edges. Let $j =$ length of $q^{**} + 1$. If $j = 1$, i.e., if q^{**} is empty, it is readily seen that q^* is complete and that n belongs to the main program, and we have again the trivial decomposition $q = q$.

If $j > 1$, let $c_i = q^{**}(i)(1)$, $i = 1, \ldots, j-1$, and put $p_1 =$ main program, $p_{i+1} =$ the procedure called from c_i, $i = 1, \ldots, j-1$. In view of the way in which q^{**} was obtained from q, it follows that c_i is in p_i for each $i < j$. Let $m_0 = 1$ and m_i be the original index of $q^{**}(i)$ in q, $i = 1, \ldots, j-1$. Then we have the decomposition $q = q_1 \| (c_1, r_{p_2}) \| q_2 \ldots \| (c_{j-1}, r_{p_j}) \| q_j$ where $q_i = q(m_{i-1} + 1 : m_i - 1)$, $i = 1, \ldots, j-1$, and $q_j = q(m_{j-1} + 1 :)$.† It is easily verified that $q_i|_{E^1}$ is complete for each $i \leq j$, and therefore $q_i \in \mathrm{IVP}_0(r_{p_i}, c_i)$ for $i < j$ and $q_j \in \mathrm{IVP}_0(r_{p_j}, n)$.

The proof of the converse assertion is simpler, and follows directly from the definitions of IVP and IVP_0.

The uniqueness of this decomposition is also easy to establish, since $c_1, \ldots, c_{j-1}$ are precisely all the calls along q which are not subsequently completed, and it is fairly obvious from the definitions that these calls and their positions in q are unique, which immediately implies the uniqueness of the whole decomposition. ■

We can now describe our "functional" approach to interprocedural analysis. Let (L, F) be a distributive data flow framework for G. In the first phase of the functional approach we take F as the direct basis for our analysis. More precisely, for each procedure p and each $n \in N_p$, we define an element $\phi_{(r_p, n)} \in F$ which describes the manner in which attributes in L are propagated from the start of r_p to the start of n along paths in $\mathrm{IVP}_0(r_p, n)$. These functions must satisfy the following (nonlinear) equations, whose heuristic meaning should be self-explanatory: For each $(m, n) \in E^0$, let $f_{(m,n)} \in F$ denote the associated propagation effect. Then

$$\begin{aligned} &\phi_{(r_p, r_p)} \leq \mathbf{id}_L && \text{for each procedure } p \\ &\phi_{(r_p, n)} = \bigwedge_{(m,n) \in E_p} (h_{(m,n)} \circ \phi_{(r_p, m)}) && \text{for each } n \in N_p \end{aligned} \tag{7-4}$$

where

$$h_{(m,n)} = \begin{cases} f_{(m,n)} & \text{if } (m, n) \in E_p^0 \\ \phi_{(r_q, c_2)} & \text{if } (m, n) \in E_p^1 \text{ and } m \text{ calls procedure } q \end{cases}$$

†For any tuple or string a, $a(i:j)$ denotes its subpart from the ith component to the jth one, inclusive; $a(i:)$ denotes the subpart of a from the ith component to its 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 $\{g_i\}_{i > 0}$ 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:

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

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

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

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 **0** and **1**, $\mathbf{id}_L$ and f_Ω. With these notations, Eqs. (7-4) read

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

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
$\phi_{(r_1, c_1)}$	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
$\phi_{(r_2, c_2)}$	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

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

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:

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

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

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

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

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

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

Indeed, for $i = 0$, if $n = r_p$ then $\phi^0_{(r_p, r_p)} = \mathbf{id}_L = f_{q_0}$, where $q_0 \in \mathrm{IVP}_0(r_p, r_p)$ is the empty path from r_p to r_p, so that $\phi^0_{(r_p, r_p)} \geq \bigwedge \{f_q : q \in \mathrm{IVP}_0(r_p, r_p)\}$. 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

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

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

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

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

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

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

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

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 $q \in \mathrm{IVP}_0(r_p, n)$, $f_q \geq \phi_{(r_p, n)}$. 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 $q \in \mathrm{IVP}_0(r_p, n)$ 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 $(m, n) \in E^0_p$, then $q_1 \in \mathrm{IVP}_0(r_p, m)$ 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 $m = c_{p'}$ 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 $(m_1, n) \in E^1_p$, $q_1 \in \mathrm{IVP}_0(r_p, m_1)$, $q_2 \in \mathrm{IVP}_0(r_{p'}, e_{p'})$. 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 $q \in \mathrm{IVP}(r_{\mathrm{main}}, n)$. By Lemma 7-3.1, q admits a decomposition $q = q_1 \| (c_1, r_{p_2}) \| q_2 \| \ldots \| (c_{j-1}, r_{p_j}) \| q_j$ 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 $q_i \in \mathrm{IVP}_0(r_{p_i}, c_i)$, and also $q_j \in \mathrm{IVP}_0(r_{p_j}, n)$.

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

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

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

```
A := 0;
call P;
print A;
end
```

Procedure p

```
if cond then
   A := A + 1;
   call p;
   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: $N^* \times L \to L$, 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 $z = \text{PHI}(e_p, y)$ 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:

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

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 $q \in \mathrm{IVP}(r_1, n)$, terminating at some $n \in N^*$, such that the decomposition (7-3) of q has the form $q_1 \| (c_1, r_{p_2}) \| q_2 \| \cdots \| (c_j, r_{p_{j+1}}) \| q_{j+1}$ where $q_i \in \mathrm{IVP}_0(r_{p_i}, c_i)$ for each $i \leq j$ and $q_{j+1} \in \mathrm{IVP}_0(r_{p_{j+1}}, n)$. To show the relation between q and γ we introduce a map CM such that $\mathrm{CM}(q) = \gamma$. 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), $(c_1 c_2)$, $(c_1 c_2 c_2)$, etc.

However, for each n in the main program and each $q \in \mathrm{IVP}(r_1, n)$, $\mathrm{CM}(q) = \lambda$; 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 $\mathrm{CM}^{-1}\{\gamma\}$.

If we define a meet operation in L^* as a pointwise meet on Γ, i.e., if for $\xi_1, \xi_2 \in L^*$, $\gamma \in \Gamma$, we define $(\xi_1 \wedge \xi_2)(\gamma) = \xi_1(\gamma) \wedge \xi_2(\gamma)$, then L^* becomes a semilattice. The smallest element in L^* is 0^*, where $0^*(\gamma) = 0$ for each $\gamma \in \Gamma$. The largest element in L^* is Ω^*, where $\Omega^*(\gamma) = \Omega$ for each $\gamma \in \Gamma$. Note that unless Γ is finite, L^* need not be bounded. However, if $\xi_1 \geq \xi_2 \geq \ldots \geq \xi_n \geq \ldots$ is an infinite decreasing chain in L^*, its limit is well-defined and can be computed as follows: For each $\gamma \in \Gamma$, the chain $\xi_1(\gamma) \geq \xi_2(\gamma) \geq \ldots$ must be finite (since L is bounded). Define $(\lim \xi_n)(\gamma)$ as the final value of that chain. Obviously $\lim \xi_n = \bigwedge_n \xi_n$ and in the same manner it can be shown that $\bigwedge_n \xi_n$ exists for any sequence $\{\xi_i\}_{i \geq 1}$ in L^*.

In order to describe F^*, we first need to define a certain operation in Γ.

Definition. $\circ : \Gamma \times E^* \to \Gamma$ is a partially defined binary operation such that for each $\gamma \in \Gamma$ and $(m, n) \in E^*$ such that $CM^{-1}\{\gamma\} \cap IVP(r_1, m) \neq \varnothing$ we have

$$\gamma \circ (m, n) = \begin{cases} \gamma & \text{if } (m, n) \in E^0 \\ \gamma \,||\, [m] & \text{if } (m, n) \text{ is a call edge in } E^1 \\ & \text{(i.e., if } m \text{ is a call block)} \\ \gamma(1 : \#\gamma - 1) & \text{(i.e., } \gamma \text{ without its last component)} \\ & \text{if } (m, n) \text{ is a return edge in } E^1 \text{ such} \\ & \text{that } \gamma(=\gamma) \text{ is its corresponding call edge} \end{cases}$$

in all other cases, $\gamma \circ (m, n)$ is undefined. Here $\#\gamma$ denotes the length of γ.

The following lemma can be proved in an obvious and straightforward way.

Lemma 7-4.1. Let $\gamma \in \Gamma$, $(m, n) \in E^*$, $q \in IVP(r_1, m)$ such that $CM(q) = \gamma$. Then $\gamma_1 = \gamma \circ (m, n)$ is defined iff $q_1 = q \,||\, (m, n)$ is in $IVP(r_1, n)$, in which case $CM(q_1) = \gamma_1$.

The operation $\circ$ defines the manner in which call strings are updated as data is propagated along an edge of the flow graph. Loosely put, the above lemma states that path incrementation is transformed into $\circ$ by the "homomorphism" CM.

Example 6. In Example 1 of Section 7-3, we have

$$\lambda \circ (c_1, r_2) = (c_1)$$
$$(c_1) \circ (c_2, r_2) = (c_1 c_2)$$
$$(c_1 c_2) \circ (e_2, n_2) = (c_1)$$

and

$$(c_1 c_2) \circ (e_2, n_1)$$

is undefined, indicating that after p had called itself once, the return from p must be to the block following c_2 in p; it is illegal to return then to the main program.

Next, let $(m, n) \in E^*$, and let $f_{(m,n)} \in F$ be the data propagation map associated with (m, n). Note that by our assumptions $f_{(m,n)} = \mathbf{id}_L$ if $(m, n) \in E^1$, since in these cases m is a block containing only a jump which in itself does not affect data attributes. Define $f^*_{(m,n)} : L^* \to L^*$ as follows: For each $\xi \in L^*$, $\gamma \in \Gamma$,

$$f^*_{(m,n)}(\xi)(\gamma) = \begin{cases} f_{(m,n)}(\xi(\gamma_1)) & \text{if there exists (a necessarily unique)} \\ & \gamma_1 \text{ such that } \gamma_1 \circ (m, n) = \gamma \\ \Omega & \text{otherwise} \end{cases}$$

The intuitive interpretation of this formula is as follows: $f^*_{(m,n)}(\xi)$ represents information at the start of n which is obtained by propagation of the information ξ, known at the start of m, along the edge (m, n). For each $\gamma_1 \in \Gamma$ for which $\xi(\gamma_1)$ is defined, we propagate $\xi(\gamma_1)$, the γ_1-selected data available at the start of m, to the start of n in standard intraprocedural fashion (that is, using $f_{(m,n)}$). However, this propagated data is now associated not with γ_1 but with $\gamma_1 \circ (m, n)$, which "tags" the set of paths obtained by concatenating (m, n) to all paths which are "tagged" by γ_1, which lead to m, and along which $\xi(\gamma_1)$ has been propagated. If $\gamma_1 \circ (m, n)$ is undefined, then, by Lemma 7-4.1, $\xi(\gamma_1)$ should not be propagated through (m, n) since no path which leads to m and is tagged by γ_1 can be concatenated with (m, n) in an interprocedurally valid manner. In this case, we simply discard $f_{(m,n)}(\xi(\gamma_1))$, as indicated by the above formula.

Example 7. In Example 1 of Section 7-3, let $\xi_0 = \{(\lambda, 1)\} \in L^*$. Then (for notational convenience, call strings are written without enclosing parentheses): $\xi_1 = f^*_{(c_1, r_2)}(\xi_0) = \{(c_1, 1)\}$, since $\lambda \circ (c_1, r_2) = c_1$ and ξ_0 is defined only at λ. Note that $f_{(c_1, r_2)} = \mathbf{id}$, as is the case for all interprocedural jumps.

$$\xi_2 = f^*_{(r_2, c_2)}(\xi_1) = \{(c_1, f_{(r_2, c_2)}(1))\} = \{(c_1, \upsilon)\}$$

This edge is intraprocedural, so that call strings need not be modified.

$$\xi_3 = f^*_{(c_2, r_2)}(\xi_2) = \{(c_1 c_2, 0)\}$$
$$\xi_4 = f^*_{(r_2, e_2)}(\xi_3) = \{(c_1 c_2, 0)\}$$
$$\xi_5 = f^*_{(e_2, n_2)}(\xi_4) = \{(c_1, 0)\}$$

[But note that, e.g., $f^*_{(e_2, n_1)}(\xi_4) =$ totally undefined map (Ω^*) in L^*, since the only $\gamma_1 \in \Gamma$ for which $\gamma_1 \circ (e_2, n_1)$ is defined is the string c_1, but $\xi_4(c_1)$ is undefined.]

$$\xi_6 = f^*_{(n_2, c_2)}(\xi_5) = \{(c_1, 1)\}$$
$$\xi_7 = f^*_{(c_2, n_1)}(\xi_6) = \{(\lambda, 1)\} \qquad \text{(compare with } f^*_{(e_2, n_1)}(\xi_4)!)$$

To summarize, we have traced one possible interprocedurally valid path from c_1 to n_1, starting with the information that $a * b$ is available at c_1 and obtaining the fact that it is still available at n_1 (considering just this path, of course). An attempt to return to the main program prematurely resulted in completely discarding the information.

F^* is now defined as the smallest subset of maps acting in L^* which contains $\{f^*_{(m,n)} : (m, n) \in E^*\}$ and the identity map in L^* and which is closed under functional composition and meet.

Lemma 7-4.2.

1. If F is monotone in L, then F^* is monotone in L^*.
2. If F is distributive in L, then F^* is distributive in L^*.
3. If F is distributive in L, then for each $(m, n) \in E$, $f^*_{(m,n)}$ is *continuous* in L^*, that is, $f^*_{(m,n)}\left(\bigwedge_k \xi_k\right) = \bigwedge_k f^*_{(m,n)}(\xi_k)$, for each collection $\{\xi_k\}_{k\geq 1} \subseteq L^*$.

Proof. It is easily seen that it is sufficient to prove (1) or (2) for the set $\{f^*_{(m,n)} : (m, n) \in E^*\}$, and this is straightforward from the definitions.

To prove (3), note that for each $\gamma \in \Gamma$ for which there exists $\gamma_1 \in \Gamma$ such that $\gamma_1 \circ (m, n) = \gamma$ we have

$$f^*_{(m,n)}\left(\bigwedge_{k\geq 1} \xi_k\right)(\gamma) = f_{(m,n)}\left(\bigwedge_{k\geq 1} \xi_k(\gamma_1)\right)$$

But since L is bounded, there exists $k_0(\gamma_1)$ such that the last expression equals $f_{(m,n)}\left(\bigwedge_{1\leq k\leq k_0(\gamma_1)} \xi_k(\gamma_1)\right)$, which in turn, by the distributivity of $f_{(m,n)}$, equals

$$\bigwedge_{1\leq k\leq k_0(\gamma_1)} f_{(m,n)}(\xi_k(\gamma_1)) = \bigwedge_{1\leq k\leq k_0(\gamma_1)} f^*_{(m,n)}(\xi_k)(\gamma) \geq \left(\bigwedge_{k\geq 1} f^*_{(m,n)}(\xi_k)\right)(\gamma)$$

Thus $f^*_{(m,n)}\left(\bigwedge_{k\geq 1} \xi_k\right) \geq \bigwedge_{k\geq 1} f^*_{(m,n)}(\xi_k)$. The converse inequality is immediate from the monotonicity of $f^*_{(m,n)}$. ■

Remark: Note that interprocedural, as distinct from intraprocedural, data flow frameworks depend heavily on the flow graph (Γ itself may vary from one flow graph to another). Thus, for example, there is no simple way to obtain F^* directly from F without any reference to the flow graph. This will not create any problems in the sequel, and we argue that even in the intraprocedural case it is a better practice to regard data flow frameworks as graph-dependent.

We can now define a data flow problem for G^*, using the new framework (L^*, F^*), in which we seek the maximal fixed-point solution of the following equations in L^*:

$$\begin{aligned} x^*_{r_1} &= \{(\lambda, 0)\} \qquad \text{where } \lambda \text{ is the null call string} \\ x^*_n &= \bigwedge_{(m,n)\in E^*} f^*_{(m,n)}(x^*_m) \qquad n \in N^* - \{r_1\} \end{aligned} \tag{7-11}$$

We can show the existence of a solution to these equations in the following manner: Let $x^{*(0)}_{r_1} = \{(\lambda, 0)\}$, $x^{*(0)}_n = \Omega^*$ for all $n \in N^* - \{r_1\}$. Then apply Eqs. (7-11) iteratively to obtain new approximations to the x^*'s. Let $x^{*(i)}_n$ denote the ith approximation computed in this manner.

Since $x^{*(0)}_n \geq x^{*(1)}_n$ for all $n \in N^*$, it follows inductively, from the monotonicity of $f^*_{(m,n)}$ for each $(m, n) \in E^*$, that $x^{*(i)}_n \geq x^{*(i+1)}_n$ for all $i \geq 0$, $n \in N^*$. Thus, for each $n \in N^*$, $\{x^{*(i)}_n\}_{i\geq 0}$ is a decreasing chain in L^*, having a limit, and we define $x^*_n = \lim_i x^{*(i)}_n$. It is rather straightforward to show that $\{x^*_n\}_{n\in N^*}$ is indeed a solution to (7-11) and that in fact it is the maximal fixed-point solution of (7-11).

Having defined this solution, we will want to convert its values to values in L, because L^* has been introduced only as an auxiliary semilattice, and our aim is really to obtain data in L for each basic block. Since there is no longer a need to split the data at node n into parts depending on the interprocedural flow leading to n, we can combine these parts together, i.e., take their meet. For each $n \in N^*$, we can then simply define

$$x'_n = \bigwedge_{\gamma\in\Gamma} x^*_n(\gamma) \tag{7-12}$$

A detailed example of applying this technique to our running example (Example 1 of Section 7-3) will be given in the next section.

In justifying the approach that we have just outlined, our first step is to prove that x'_n coincides with the meet-over-all-interprocedurally-valid-paths solution y_n defined in the preceding section. This can be shown as follows:

Definition. Let $\text{path}_{G^*}(r_1, n)$ denote the set of all execution paths (whether interprocedurally valid or not) leading from r_1 to $n \in N^*$. For each $p = (r_1, s_2, \ldots, s_k, n) \in \text{path}_{G^*}(r_1, n)$ define $f^*_p = f^*_{(s_k,n)} \circ f^*_{(s_{k-1},s_k)} \circ \ldots \circ f^*_{(r_1,s_2)}$. For each $n \in N^*$ define $y^*_n = \bigwedge \{f^*_p(x^*_{r_1}) : p \in \text{path}_{G^*}(r_1, n)\}$.

Since $\text{path}_{G^*}(r_1, n)$ is at most countable, this (possibly infinite) meet in L^* is well defined.

Theorem 7-4.3. If (L, F) is a distributive data flow framework, then, for each $n \in N^*$, $x^*_n = y^*_n$.

Proof. The proof follows (it is quite similar to the proof of an analogous theorem of Kildall for a bounded semilattice [Kild73]):

1. Let $n \in N^*$ and $p \equiv (r_1, s_2, \ldots, s_k, n) \in \text{path}_{G^*}(r_1, n)$. By (7-11) we have

$$x^*_{s_2} \leq f^*_{(r_1,s_2)}(x^*_{r_1})$$
$$x^*_{s_3} \leq f^*_{(s_2,s_3)}(x^*_{s_2})$$
$$\vdots$$
$$x^*_n \leq f^*_{(s_k,n)}(x^*_{s_k})$$

Combining all these inequalities, and using the monotonicity of the f^*'s, we obtain $x^*_n \leq f^*_p(x^*_{r_1})$, and therefore $x^*_n \leq y^*_n$.

2. Conversely, we will prove by induction on i that

$$x^{*(i)}_n \geq y^*_n \qquad \text{for all } i \geq 0,\ n \in N^*$$

Indeed, let $i = 0$. If $n \neq r_1$, then $x^{*(0)}_n = \Omega^* \geq y^*_n$. On the other hand, the null execution path $p_0 \in \text{path}_{G^*}(r_1, r_1)$, so that $y^*_{r_1} \leq f^*_{p_0}(x^*_{r_1}) = x^*_{r_1} = x^{*(0)}_{r_1}$. Thus the assertion is true for $i = 0$. Suppose that it is true for some $i \geq 0$. Then $x^{*(i+1)}_{r_1} = x^{*(i)}_{r_1} \geq y^*_{r_1}$, and for each $n \in N^* - \{r_1\}$ we have

$$x^{*(i+1)}_n = \bigwedge_{(m,n)\in E^*} f^*_{(m,n)}(x^{*(i)}_m) \geq \bigwedge_{(m,n)\in E^*} f^*_{(m,n)}(y^*_m)$$

by the induction hypothesis.

We now need the following:

Lemma 7-4.4. For each $(m, n) \in E^*$, $f^*_{(m,n)}(y^*_m) \geq y^*_n$.

Proof. Since $f^*_{(m,n)}$ is distributive and continuous on L^* (Lemma 7-4.2), we have

$$\begin{aligned} f^*_{(m,n)}(y^*_m) &= f^*_{(m,n)}(\wedge\{f^*_p(x^*_{r_1}); p \in \text{path}_{G^*}(r_1, m)\}) \\ &= \wedge\{f^*_{(m,n)}(f^*_p(x^*_{r_1})): p \in \text{path}_{G^*}(r_1, m)\} \\ &\geq \wedge\{f^*_q(x^*_{r_1}): q \in \text{path}_{G^*}(r_1, n)\} = y^*_n \quad \blacksquare \end{aligned}$$

Now returning to Theorem 7-4.3, it follows by Lemma 7-4.4 that $x^{*(i+1)}_n \geq \bigwedge_{(m,n)\in E^*} y^*_n = y^*_n$ (each $n \in N^*$ is assumed to have predecessors). Hence assertion (2) is established, and it follows that for each $n \in N^*$, $x^*_n = \lim_i x^{*(i)}_n = \bigwedge_{i\geq 1} x^{*(i)}_n \geq y^*_n$, so that $x^*_n = y^*_n$. $\blacksquare$

Lemma 7-4.5. Let $n \in N^*$, $p = (r_1, s_1, \ldots, s_k, n) \in \text{path}_{G^*}(r_1, n)$ and $\gamma \in \Gamma$. Then $f^*_p(x^*_{r_1})(\gamma)$ is defined iff $p \in \text{IVP}(r_1, n)$ and $\text{CM}(p) = \gamma$. If this is the case, then $f^*_p(x^*_{r_1})(\gamma) = f_p(0)$.

Proof. The proof is by induction on $l(p)$, the length of p (i.e., the number of edges in p). If p is the null path, then n must be equal to r_1. Moreover, $\text{CM}(p) = \lambda$, $p \in \text{IVP}(r_1, r_1)$ and $f^*_p(x^*_{r_1}) = x^*_{r_1}$ is defined only at λ and equals $0 = f_p(0)$. Thus our assertion is true if $l(p) = 0$.

Suppose that this assertion is true for all $n \in N^*$ and $p \in \text{path}_{G^*}(r_1, n)$such that $l(p) < \lambda$. Let $n \in N^*$ and $p = (r_1, s_2, \ldots, s_k, n)$ be a path of length k in $\text{path}_{G^*}(r_1, n)$. Let $p_1 = (r_1, s_2, \ldots, s_k)$. By definition, for each $\gamma \in \Gamma$ we have

$$f^*_p(x^*_{r_1})(\gamma) = f^*_{(s_k,n)}[f^*_{p_1}(x^*_{r_1})](\gamma)$$
$$= \begin{cases} f_{(s_k,n)}[f^*_{p_1}(x^*_{r_1})(\gamma_1)] & \text{if there exists } \gamma_1 \in \Gamma \text{ such that } \gamma_1 \circ (s_k, n) = \gamma \\ \Omega & \text{otherwise} \end{cases}$$

Thus $f^*_p(x^*_{r_1})(\gamma)$ is defined iff there exists $\gamma_1 \in \Gamma$ such that $\gamma_1 \circ (m, n) = \gamma$ and $f^*_{p_1}(x^*_{r_1})(\gamma_1)$ is defined. By our inductive hypothesis, this is the case iff $p_1 \in \text{IVP}(r_1, s_k)$, $\text{CM}(p_1) = \gamma_1$ and $\gamma_1 \circ (s_k, n) = \gamma$. By Lemma 7-4.1, these last conditions are equivalent to $p \in \text{IVP}(r_1, n)$ and $\text{CM}(p) = \gamma$.

If this is the case, then again, by our inductive hypothesis, $f^*_{p_1}(x^*_{r_1})(\gamma_1) = f_{p_1}(0)$ and so

$$f^*_p(x^*_{r_1})(\gamma) = f_{(s_k,n)}[f_{p_1}(0)] = f_p(0) \quad \blacksquare$$

Now we can prove the main result of this section:

Theorem 7-4.6. For each $n \in N^*$, $x'_n = y_n$.

Proof. Let $\gamma \in \Gamma$. By Theorem 7-4.3,

$$x^*_n(\gamma) = \{f^*_p(x^*_{r_1})(\gamma): p \in \text{path}_{G^*}(r_1, n)\}$$

and by Lemma 7-4.5,

$$= \wedge\{f_p(0): p \in \text{IVP}(r_1, n) \text{ such that } \text{CM}(p) = \gamma\}$$

Thus, by (7-12),

$$x'_n = \bigwedge_{\gamma\in\Gamma} x^*_n(\gamma) = \wedge\{f_p(0): p \in \text{IVP}(r_1, n)\} = y_n \quad \blacksquare$$

Corollary 7-4.7. If the flow graph G^* is nonrecursive, then the iterative solution of Eqs. (7-11) that we have described will converge and yield the desired meet-over-all-interprocedurally-valid-paths solution of these equations.

Proof. Convergence is assured since Γ is finite, and hence L^* is bounded. Thus (L^*, F^*) is a distributive data flow framework, and by standard arguments the iterative solution of (7-11) must converge

[Kild73, Hech77]. Therefore, Theorem 7-4.6 implies that the limiting solution coincides with the meet-over-all-paths solution. ■

The call-strings approach is of questionable feasibility if Γ is infinite, i.e., if G^* contains recursive procedures. Moreover, as for the functional approach, it is rather hopeless to convert it into an effective algorithm for handling the most general cases of certain data flow problems such as constant propagation. However, as we shall see in the following section, a fairly practical variant of the call-strings approach can be devised for data flow frameworks with a finite semilattice L.

Let us also observe the similarity between the call-string approach and the inline expansion method (discussed, e.g., in [Alle77]). Indeed, tagging data by call strings amounts essentially to creating virtual copies of each procedure, one copy for each possible calling sequence reaching that procedure. Indeed let $\gamma = c_1 c_2 \ldots c_j \in \Gamma$. Then, if c_j calls procedure p from procedure p', we can substitute the virtual copy of p corresponding to γ at the place of c_j in the virtual copy of p' corresponding to $\gamma' = c_1 c_2 \ldots c_{j-1}$. Doing so, c_j and the return from p become no-ops, and we get a full inline expansion of procedures.

7-5. DATA FLOW ANALYSES USING A FINITE SEMILATTICE

Let (L, F) be a distributive data flow framework such that L is finite. As we have seen, the functional approach described in Section 7-3 converges for such a framework. We will show in this section that it is also possible to construct a call-strings algorithm which converges for these frameworks. As noted in the previous section, convergence is ensured if Γ is finite. The idea behind our modified approach is to replace Γ by some finite subset Γ_0 and allow propagation only through interprocedurally valid paths which are mapped into elements of Γ_0. Such an approach is not generally feasible because it can lead to an overestimated (and unsafe) solution, since it does not trace information along all possible paths. However, using the finiteness of L, we will show that Γ_0 can be chosen in such a way that no information gets lost and the algorithm produces the same solution as defined in Section 7-4.

We begin to describe our approach without fully specifying Γ_0. Later we will show how Γ_0 should depend on L in order to guarantee the above solution.

Definitions.

1. Let Γ_0 be some finite subset of Γ with the property that if $\gamma \in \Gamma_0$ and γ_1 is an initial subtuple of γ, then $\gamma_1 \in \Gamma_0$ too.

2. For each $n \in N^*$, let $\mathrm{IVP}'(r_1, n)$ denote the set of all $q \in \mathrm{IVP}(r_1, n)$ such that for each initial subpath q_1 of q (including q), $\mathrm{CM}(q_1) \in \Gamma_0$.
3. We also modify the $\circ$ operation so that it acts in Γ_0 rather than in Γ, as follows: If $\gamma \in \Gamma_0$, $(m, n) \in E^*$ such that there exists $q \in \mathrm{IVP}'(r_1, m)$ where $\mathrm{CM}(q) = \gamma$, then

$$\gamma \circ (m, n) = \begin{cases} \gamma & \text{if } (m, n) \in E^0 \\ \gamma \,||\, [m] & \text{if } (m, n) \text{ is a call edge in } E^1 \text{ and } \gamma \,||\, [m] \in \Gamma_0 \\ \gamma(1 : \#\gamma - 1) & \text{if } (m, n) \text{ is a return edge in } E^1 \text{ and } \gamma(\#\gamma) \text{ is the call block preceding } n \\ \text{undefined in all other cases} & \end{cases}$$

The only difference between this definition of $\circ$ and the previous one is that it will not add a call block m to a call string γ unless the resulting string is in Γ_0. When this is not the case, information tagged by γ will be lost when propagating through (m, n), unless it is also tagged by some other call string to which m can be concatenated.

The following lemma is analogous to Lemma 7-4.1:

Lemma 7-5.1. Let $\gamma \in \Gamma_0$, $(m, n) \in E^*$, $q \in \mathrm{IVP}'(r_1, m)$ such that $\mathrm{CM}(q) = \gamma$. Then $\gamma_1 = \gamma \circ (m, n)$ is defined iff $q_1 = q \,||\, (m, n)$ is in $\mathrm{IVP}'(r_1, n)$, in which case $\mathrm{CM}(q_1) = \gamma_1$.

We now define a data flow framework (L^*, F^*) in much the same way as in Section 7-4, but replace Γ by Γ_0. This leads to a bounded semilattice $L^* = L^{\Gamma_0}$ and to a distributive data flow framework (L^*, F^*).

Hence, Eqs. (7-11) come to be effectively solvable by any standard iterative algorithm which yields their maximal fixed-point solution. To this solution we will want to apply the following final calculation, which is a variant of (7-12):

$$x''_n = \bigwedge_{\gamma \in \Gamma_0} x^*_n(\gamma) \tag{7-13}$$

Careful scrutiny of the analysis of the previous section reveals that the only place where the nature of Γ_0 and the operation $\circ$ are referred to is in Lemma 7-4.1, and it is easily seen that if we replace Γ and $\circ$ by Γ_0 and the modified $\circ$, throughout the previous analysis, and also replace $\mathrm{IVP}(r_1, n)$ by $\mathrm{IVP}'(r_1, n)$ for all $n \in N^*$, then by proofs completely analogous to those presented in Section 7-4 (but with one notable difference, i.e., that there is now no need to worry about continuity of F^* or infinite meets in L^*, since L^* is now known to be bounded) we obtain the following:

Theorem 7-5.2. For each $n \in N^*$

$$x_n'' = y_n'' \equiv \bigwedge \{f_p(0) : p \in \text{IVP}'(r_1, n)\}$$

Up to this point, our suggested modifications have been quite general and do not impose any particular requirements upon L or upon Γ_0. On the other hand, Theorem 7-5.2 implies that x_n'' is an overestimated solution, and as such is useless for purposes of our analysis, as it can yield unsafe information (e.g., it may suggest that an expression is available whereas it may actually be unavailable), unless we can show that x_n'' coincides with the meet-over-all-interprocedurally-valid-paths solution of the attribute propagation equations which concern us. As will be shown below, this is indeed the case if L is finite.

Definition. Let $M \geq 0$ be an integer. Define Γ_M as the (finite) set of all call strings whose lengths do not exceed M. Γ_M obviously satisfies the conditions of part (1) of the previous definition.

Lemma 7-5.3. Let (L, F) be a data flow framework with L a finite semilattice, and let $M = K(|L| + 1)^2$, where K is the number of call blocks in the program being analyzed and $|L|$ is the cardinality of L. Let $\Gamma_0 = \Gamma_M$. Then, for each $n \in N^*$ and each execution path $q \in \text{IVP}(r_1, n)$ there exists another path $q' \in \text{IVP}'(r_1, n)$ such that $f_q(0) = f_{q'}(0)$.

Proof. By induction on the length of q. If the length is 0, then $n = r_1$ and q is the null execution path, which belongs to both $\text{IVP}(r_1, r_1)$ and $\text{IVP}'(r_1, r_1)$, so that our assertion is obviously true in this case.

Suppose that the lemma is true for all paths whose length is less than some $k \geq 1$, and let $n \in N^*$, $q \in \text{IVP}(r_1, n)$ be a path of length k. If $q \in \text{IVP}'(r_1, n)$ then there is nothing to prove, so assume that this is not the case, and let q_0 be the shortest initial subpath of q such that $\text{CM}(q_0) \notin \Gamma_0$. Then q_0 can be decomposed according to Eq. (7-3) as follows:

$$q_0 = q_1 \| (c_1, r_{p_2}) \| q_2 \| \ldots \| (c_j, r_{p_{j+1}}) \| q_{j+1}$$

Hence $j > M$. Next, consider the sequence $\{(c_i, \alpha_i, \beta_i)\}_{i=1}^{j}$, where, for each $i \leq j$, $\alpha_i = f_{q_i} \circ f_{q_{i-1}} \circ \ldots \circ f_{q_1}(0)$, and β_i is either Ω if the call at c_i is not completed in q (this call is certainly not completed in q_0), or $f_{\hat{q}_i}(0)$ if the call at c_i is completed in q, and $\hat{q}_i$ is the initial subpath of q ending at the return which completes the call. Thus, for each call, the sequence records the calling block, the value propagated along this path up to the call, and the value propagated up to the corresponding return, if it materializes. The number of distinct elements of such a sequence is at most $K(|L| + 1)^2 = M$ (we do not count Ω as an element of L; if we did, then the bound can be reduced to $K|L|^2$). Since $j > M$, this sequence must contain at least two identical components (c_a, α_a, β_a) and (c_b, α_b, β_b), where $a < b \leq j$.

Now, if $\beta_a = \beta_b = \Omega$, then neither of the calls c_a, c_b is completed in q. If we rewrite q as

$$q = q_1' \| (c_a, r_{p_{a+1}}) \| q_2' \| (c_b, r_{p_{b+1}}) \| q_3'$$

then it is easily seen that the shorter path $\hat{q} = q_1' \| (c_a, r_{p_{a+1}}) \| q_3'$ is also in $\text{IVP}(r_1, n)$. Moreover

$$\alpha_a = f_{q_1'}(0) = \alpha_b = f_{q_2'} \circ f_{q_1'}(0)$$

so that

$$f_q(0) = f_{q_3'} \circ f_{q_2'} \circ f_{q_1'}(0) = f_{q_3'} \circ f_{q_1'}(0) = f_{\hat{q}}(0)$$

By our induction hypothesis there exists $\hat{q}' \in \text{IVP}'(r_1, n)$ such that $f_{\hat{q}'}(0) = f_{\hat{q}}(0) = f_q(0)$, which proves the lemma for q.

On the other hand, if $\beta_a = \beta_b \neq \Omega$, then it follows that both calls c_a and c_b are completed in q, with c_b necessarily completed first. Thus we can write

$$q = q_1' \| (c_a, r_{p_{a+1}}) \| q_2' \| (c_b, r_{p_{b+1}}) \| q_3' \| (e_{p_{b-1}}, n_b) \| q_4' \| (e_{p_{a-1}}, n_a) \| q_5'$$

where $n_a = n_b$ is the block immediately following c_a. Again it follows that $\hat{q} = q_1' \| (c_a, r_{p_{a+1}}) \| q_3' \| (e_{p_{a-1}}) \| q_5'$ is in $\text{IVP}(r_1, n)$. Moreover

$$\alpha_a = f_{q_1'}(0) = \alpha_b = f_{q_2'} \circ f_{q_1'}(0)$$

$$\beta_a = f_{q_4'} \circ f_{q_3'} \circ f_{q_2'} \circ f_{q_1'}(0) = \beta_b = f_{q_3'} \circ f_{q_2'} \circ f_{q_1'}(0)$$

from which one easily obtains $f_q(0) = f_{\hat{q}}(0)$, and the proof can now continue exactly as before. ■

The main result of this section now follows immediately:

Theorem 7-5.4. Let (L, F) be a distributive data flow framework with a finite semilattice L, and let $\Gamma_0 = \Gamma_M$, with M as defined above. Then, for each $n \in N^*$, $x_n'' = y_n$. That is, the modified algorithm described at the beginning of the present section yields a valid interprocedural solution.

Proof. Since $\text{IVP}'(r_1, n) \subset \text{IVP}(r_1, n)$ we have $x_n'' \geq y_n$. On the other hand, let $q \in \text{IVP}(r_1, n)$. By Lemma 7-5.3 there exists $q' \in \text{IVP}'(r_1, n)$ such that $f_q(0) = f_{q'}(0) \geq \bigwedge \{f_p(0) : p \in \text{IVP}'(r_1, n)\} = x_n''$. Hence $y_n = \bigwedge \{f_q(0) : q \in \text{IVP}(r_1, n)\} \geq x_n''$. ■

Remark: Note that in Lemma 7-5.3 and Theorem 7-5.4, K can be replaced by the maximal number K' of distinct calls in any sequence of nested calls in the program being analyzed. In most cases this gives a significant improvement of the bound on M appearing in those two results.

We have now shown that finite data flow frameworks are solvable by a modified call-strings approach. However, the size of Γ_0 can be expected to be large enough to make this approach as impractical as the corresponding functional approach. But in several special cases we can reduce the size of Γ_0 still further. The following definition is taken from [Rose78b], rewritten in our notation:

Definition. A data flow framework (L, F) is called *decomposable* if there exists a finite set A and a collection of data flow frameworks $\{(L_\alpha, F_\alpha)\}_{\alpha \in A}$, such that

1. $L = \prod_{\alpha \in A} L_\alpha$, ordered in a pointwise manner induced by the individual orders in each L_α.
2. $F \subseteq \sum_{\alpha \in A} F_\alpha$. That is, for each $f \in F$ there exists a collection $\{f^\alpha\}_{\alpha \in A}$ where $f^\alpha \in F$ for each $\alpha \in A$, such that for each $x = (x_\alpha)_{\alpha \in A} \in L$ we have

$$f(x) = (f^\alpha(x_\alpha))_{\alpha \in A}$$

In the cases covered by this definition we can split our data flow framework into a finite number of "independent" frameworks, each inducing a separate data flow problem, and obtain the solution to the original problem simply by grouping all the individual solutions together.

For example, the standard framework (L, F) for available expressions analysis is decomposable into subframeworks each of which is a framework for the availability of a single expression. Formally, let A be the set of all program expressions. For each $\alpha \in A$ let $L = \{0, 1\}$ where 1 indicates that α is available and 0 that it is not. Then $\{0, 1\}^A$ is isomorphic with L (which is more conveniently represented as the power set of A). It is easily checked that each $f \in F$ can be decomposed as $\sum_{\alpha \in A} f^\alpha$, where for each $\alpha \in A$, $f^\alpha \in F_\alpha$, and is either the constant **0** if α can be killed by the propagation step described by f, is the constant **1** if α is unconditionally generated by that propagation step, and is the identity map in all other cases. The frameworks used for use-definition chaining and live variables have analogous decompositions.

A straightforward modification of Lemma 7-5.3, applied to each (L_α, F_α) separately yields the following improved result for decomposable frameworks:

Theorem 7-5.5. Let (L, F) be a decomposable distributive data flow framework with a finite semilattice. Define $M = K \cdot \max_{\alpha \in A} (|L_\alpha| + 1)^2$ and let $\Gamma_0 = \Gamma_M$. Then, for each $n \in N^*$, $y''_n = y_n$.

In the special case of available expressions analysis this is certainly an improvement of Theorem 7-5.4, since it reduces the bound on the length of permissible call strings from $K \cdot O(4^{|A|})$ to $9K$. For this analysis we can do even better since available expression analysis has the property appearing in the following definition.

Definition. A decomposable data flow framework (L, F) is called *1-related* if, for each $\alpha \in A$, F_α consists only of constant functions and identity functions.

This property is characteristic of situations in which there exists at most one point along each path which can affect the data being propagated. Indeed, consider a framework having this property; let $\alpha \in A$ and let $p = (s_1, s_2, \ldots, s_k)$ be an execution path. Let $j \leq k$ be the largest index such that $f^\alpha_{(s_{j-1}, s_j)}$ is a constant function. Then clearly $f^\alpha_p = f^\alpha_{(s_{j-1}, s_j)}$ and is therefore also a constant. Hence in this case the effect of propagation in L_α through p is independent of the initial data and is determined by the edge (s_{j-1}, s_j) alone. If no such j exists, then $f_p = \mathbf{id}|_{L_\alpha}$, in which case no point along p affects the final data.

Note also that since each F_α is assumed to be closed under functional meet, it follows that if (L, F) is 1-related then the only constant functions that F_α can contain are **0** (the smallest element in L_α) and **1** (the largest element). Hence we can assume, with no loss of generality, that L_α is the trivial lattice $\{0, 1\}$ for each $\alpha \in A$. All the classical data flow analyses mentioned above have 1-related frameworks. It is therefore easily seen that, under these assumptions, 1-related frameworks are those having a semilattice of effective height 1 [Rose78b, Section 7].

For frameworks having the 1-related property it is easy to replace an execution path q by a shorter subpath $\hat{q}$ such that $f^\alpha_{\hat{q}}(0) = f^\alpha_q(0)$ for some $\alpha \in A$. Indeed, to obtain such a $\hat{q}$ we have only to ensure that $\hat{q}$ is also interprocedurally valid and that the last edge (s, s') in q for which $f_{(s,s')}$ is constant belongs to $\hat{q}$. This observation allows us to restrict the length of permissible call strings still further. The following can then be shown:

Theorem 7-5.6. Let (L, F) be a 1-related distributive data flow framework. Put $\Gamma_0 = \Gamma_{3K}$. Then, for each $n \in N^*$, $x''_n = y_n$.

The analysis developed in this section and the previous one can be modified to deal with nondistributive data flow problems. In the nondistributive case, Theorems 7-4.6 and 7-5.2 guarantee only inequalities of the form

$x'_n \leq y_n$ (resp. $x''_n \leq y''_n$) for all $n \in N^*$. The arguments in this section show that under appropriate conditions $y''_n = y_n$ for each $n \in N^*$, so that assuming these conditions Theorems 7-5.4, 7-5.5, 7-5.6 all yield the inequalities $x''_n \leq y_n$ for each $n \in N^*$. Thus, in the nondistributive case, our approach leads to an underestimated solution, as is the case for intraprocedural iterative algorithms for nondistributive frameworks [Kam77].

Example 8. We return to Example 1 studied in Section 7-3. Since available expressions analysis uses a 1-related framework, and since the flow graph appearing in that example satisfies $K = K' = 2$, we can take $\Gamma_0 = \Gamma_6$, and apply Kildall's iterative algorithm [Kild73] to solve Eqs. (7-11). Table 7-3 summarizes the steps which are then performed (for notational convenience call strings are written without enclosing parentheses):

Table 7-3

Propagate			*Workpile of nodes from which further propagation is required*
From	*To*	*Updated x^* value*	
(initially)		$x^*_{r_1} = \{(\lambda, 0)\}$	$\{r_1\}$
r_1	c_1	$x^*_{c_1} = \{(\lambda, 1)\}$	$\{c_1\}$
c_1	r_2	$x^*_{r_2} = \{(c_1, 1)\}$	$\{r_2\}$
r_2	c_2	$x^*_{c_2} = \{(c_1, 0)\}$	$\{c_2\}$
r_2	e_2	$x^*_{e_2} = \{(c_1, 1)\}$	$\{c_2, e_2\}$
c_2	r_2	$x^*_{r_2} = \{(c_1, 1), (c_1c_2, 0)\}$	$\{e_2, r_2\}$
e_2	n_2	$x^*_{n_2} = \Omega^*$(unchanged)	$\{r_2\}$
e_2	n_1	$x^*_{n_1} = \{(\lambda, 1)\}$	$\{r_2, n_1\}$
r_2	c_2	$x^*_{c_2} = \{(c_1, 0), (c_1c_2, 0)\}$	$\{n_1, c_2\}$
r_2	e_2	$x^*_{e_2} = \{(c_1, 1), (c_1c_2, 0)\}$	$\{n_1, c_2, e_2\}$
n_1	e_1	$x^*_{e_1} = \{(\lambda, 1)\}$	$\{c_2, e_2, e_1\}$
c_2	r_2	$x^*_{r_2} = \{(c_1, 1), (c_1c_2, 0), (c_1c_2c_2, 0)\}$	$\{e_2, e_1, r_2\}$
e_2	—	$x^*_{n_2} = \{(c_1, 0)\}$	$\{e_1, r_2, n_2\}$
e_2	n_1	—	$\{e_1, r_2, n_2\}$
e_1	—	—	$\{r_2, n_2\}$
		...	

The next steps of the algorithm update $x^*_{r_2}$, $x^*_{c_2}$, $x^*_{n_2}$, $x^*_{e_2}$ in similar fashion, adding new entries with increasingly longer call strings, up to a string $c_1c_2c_2c_2c_2c_2$, but none of $x^*_{r_1}$, $x^*_{c_1}$, $x^*_{n_1}$, or $x^*_{e_1}$ is ever modified. Final x^* values for the blocks appearing in our example are:

$$x^*_{r_2} = x^*_{c_2} = \{(c_1, 1), (c_1c_2, 0), (c_1c_2c_2, 0), \ldots, (c_1c_2c_2c_2c_2c_2, 0)\}$$

$$x^*_{c_1} = \{(c_1, 0), (c_1c_2, 0), \ldots, (c_1c_2c_2c_2c_2c_2, 0)\}$$

$$x^*_{n_2} = \{(c_1, 0), (c_1c_2, 0), \ldots, (c_1c_2c_2c_2c_2, 0)\} \quad (\neq x^*_{c_2}, \text{ by the way})$$

An x'' solution can now easily be computed; of course, this is identical to the solutions obtained by previous methods.

Note that in this example there was no need to maintain call strings of length up to 6 (length 2 would have sufficed). However, to derive correct information in the example depicted as Fig. 7-2, we need call strings in which one call appears three times.

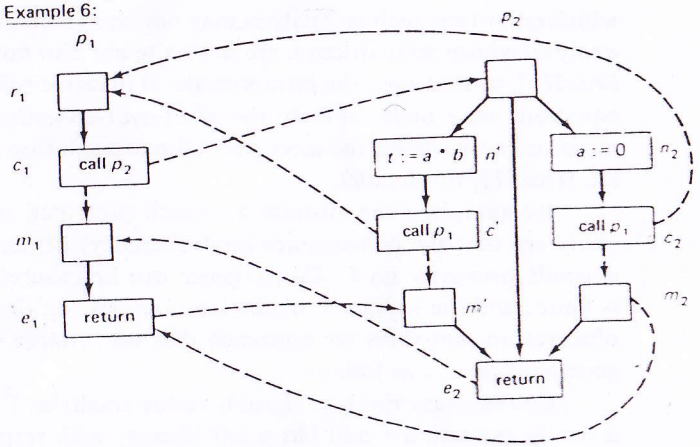

Figure 7-2

The shortest path in Fig. 7-2 showing that $a * b$ is *not* available at m' is $q = (r_1, c_1, r_2, n', c', r_1, c_1, r_2, n_2, c_2, r_1, c_1, r_2, e_2, m_1, e_1, m_2, e_2, m_1, e_1, m')$, in which c_1 appears three times before any of the calls in q is completed.

It is an interesting and challenging problem to find, for a given flow graph, by some preliminary analysis, an optimal set Γ_0 of call strings needed to perform some particular interprocedural data flow analysis without losing information.

7-6. AN APPROXIMATIVE CALL-STRING APPROACH

In this section we present a modification of the call-string approach developed in Section 7-4, which yields a convergent algorithm for any data flow analysis, even though this algorithm may in general fail to produce precisely the desired (meet-over-all-interprocedurally-valid-paths) solution. However, the output of the algorithm to be presented will always be an underestimated (and hence safe) solution. This compromise, which is useful even when L is finite, can make the call-string approach much more efficient.

Moreover, if L is infinite, or if F is not bounded or does not admit compact representation, then this modified approach is one of the very few ways to perform interprocedural analysis that we know of.

Three things should be kept in mind when evaluating any approximative approach to an interprocedural data flow problem: (1) Even in intraprocedural analysis, a meet-over-all-paths solution is itself an underestimation to the "true" run-time situation, since many of the static execution paths which enter into such an analysis may not be executable; (2) many data flow analyses whose semilattices L are not finite are also not distributive [Kam77, Shar78a], so that even the intraprocedural iterative solution of the data flow equations may underestimate the meet-over-all-paths solution; and (3) in nondistributive cases, the meet-over-all-paths solution may not be calculable (cf. [Hech77] for details).

By analyzing the abstract approach presented in Section 7-4, we can easily see that the convergence (and efficiency) of the call-strings approach depends primarily on Γ. Convergence can be ensured in general only if Γ is finite; and the smaller Γ is, the less complex the algorithm becomes. This observation motivates the approach that we propose in this section, whose general outline is as follows.

Choose some finite (preferably rather small) set $\hat{\Gamma}$ which is closed under a binary operation $*$ and has a left identity with respect to this operation. (We suggest that in practice $*$ be associative and noncommutative, but the general description given below will not assume this.) As in Section 7-4, let Γ denote the set of all call strings. Choose an "encoding" map σ which maps each call block to some element of $\hat{\Gamma}$. Using $*$, we can extend σ to Γ by putting $\sigma(\gamma) = \sigma(c_1) * \sigma(c_2) * \ldots * \sigma(c_j)$ (computed left to right) for each $\gamma = (c_1, c_2, \ldots, c_j) \in \Gamma$. We also define $\sigma(\lambda)$ to be w, the left-identity of $\hat{\Gamma}$.

Let (L, F) be any (not necessarily distributive) data flow framework. We will define a modified data flow framework (L^*, F^*) in essentially the same way as we did in Section 7-4, but with some differences reflecting the nature of the approximative approach, as detailed below.

L^* is defined as $L^{\hat{\Gamma}}$. All the observations made in Section 7-4 concerning L^* still apply, only now L^* is bounded since $\hat{\Gamma}$ has been assumed to be finite.

As before, in order to define F^*, we first define an updating operation between encoded call strings and edges in E^*. This updating operation is now more complex than that defined earlier, and need not be one-to-one and single-valued any more. It is therefore best described by assigning to each edge $(m, n) \in E^*$ a relation $R_{(m,n)}$ in $\hat{\Gamma}$. Essentially, $R_{(m,n)}$ is the identity relation for each $(m, n) \in E^0$ and for each call edge (m, n) and its corresponding return edge (m', n') we have $R_{(m',n')} = R^{-1}_{(m,n)}$. Then a path $(n_1, n_2, \ldots, n_k)$ will be considered to be *acceptable* if and only if $R_{(n_1,n_2)} \circ R_{(n_2,n_3)} \circ \ldots \circ R_{(n_{k-1},n_k)} \neq \varnothing$. To make these relations bear some meaningful relationship to the updating map $\circ$ defined in Section 7-4, we first define for each $(m, n) \in E^*$ a relation $I_{(m,n)}$ in Γ, so that for each $\gamma_1, \gamma_2 \in \Gamma$, $\gamma_1 I_{(m,n)} \gamma_2$ iff $\gamma_2 = \gamma_1 \circ (m, n)$, and then require the relation $\sigma \circ R_{(m,n)} \circ \sigma^{-1}$ to contain the relation $I_{(m,n)}$. This condition will guarantee that every interprocedurally valid path is also acceptable by our encoding scheme, but not necessarily vice versa.

To make the above ideas more precise, we suggest the following construction to obtain such suitable relations:

Definition. For each procedure p in the program being analyzed, define $\mathrm{ECS}(p) = \{\sigma(\mathrm{CM}(q)) : q \in \mathrm{IVP}(r_1, r_p)\}$. This is the set of all encoded call strings which result from interprocedurally valid paths reaching the entry of p.

These sets can be calculated by a rather simple preliminary analysis based upon the following set of equations (where *main* denotes the main program, which is assumed to be nonrecursive):

$$\mathrm{ECS}(main) = \{w\}$$
$$\mathrm{ECS}(p) = \{\alpha * \sigma(c) : c \text{ is a call to } p \text{ from some procedure } p' \text{ and } \alpha \in \mathrm{ECS}(p')\} \quad \text{for } p \neq main \tag{7-14}$$

After initializing each $\mathrm{ECS}(p)$ to $\varnothing$, for all $p \neq main$, these equations can be solved iteratively in a fairly standard way. (The iterative solution will converge because $\hat{\Gamma}$ is finite.) It is a simple matter to prove that the iterative solution yields the sets $\mathrm{ECS}(p)$ defined above.

Using the sets ECS, we now define the following objects: for each $n \in N^*$, a set of *interprocedurally acceptable* paths leading from the main entry to n, denoted by $\mathrm{IAP}(r_1, n)$; a modified set-valued map $\widehat{\mathrm{CM}}$ from $\bigcup_{n \in N^*} \mathrm{IAP}(r_1, n)$ to $2^{\hat{\Gamma}}$; and a modified relation-valued map $R: E^* \to 2^{\hat{\Gamma} \times \hat{\Gamma}}$. For each $(m, n) \in E^*$, $R_{(m,n)}$ is a relation in $\hat{\Gamma}$, so that for each $\alpha, \beta \in \hat{\Gamma}$ we have

$$\alpha R_{(m,n)} \beta \text{ iff} \begin{cases} \alpha = \beta \in \mathrm{ECS}(p) & \text{if } (m, n) \in E^0 \text{ for some procedure } p \\ \alpha \in \mathrm{ECS}(p) \text{ and } \beta = \alpha * \sigma(m) & \text{if } (m, n) \text{ is a call edge from procedure } p \\ \beta \in \mathrm{ECS}(p) \text{ and } \alpha = \beta * \sigma(c) & \text{if } (m, n) \text{ is a return edge corresponding to a call edge from a call block } c \text{ in procedure } p \end{cases}$$

Using these relations, we define the map $\widehat{\mathrm{CM}}$, so that for each $n \in N^*$ and each path $q \in \mathrm{path}_{G^*}(r_1, n)$ of the form $(r_1, s_2, s_3, \ldots, s_{k-1}, n)$ we have

$$\widehat{\mathrm{CM}}(q) = R_{(r_1,s_2)} \circ R_{(s_2,s_3)} \circ \ldots \circ R_{(s_{k-1},n)}\{w\}$$

Finally, for each $n \in N^*$ we define

$$\mathrm{IAP}(r_1, n) = \{q \in \mathrm{path}_G\cdot(r_1, n): \widehat{\mathrm{CM}}(q) \neq \varnothing\}$$

The intuitive meaning of these concepts can be explained as follows: Since we have decided to record the actual call string by a homomorphism CM of paths into a finite set $\hat{\Gamma}$, it is inevitable that we will also admit paths which are not in $\mathrm{IVP}(r_1, n)$. Thus $\mathrm{IAP}(r_1, n) \supseteq \mathrm{IVP}(r_1, n)$ and will also contain paths which the encoding CM cannot distinguish from valid IVP paths. In particular, some returns to other than their originating calls will have to be admitted.

Having defined IAP, $\widehat{\mathrm{CM}}$, and R, we next define F^* in essentially the same manner as in Section 7-4. Specifically, for each $(m, n) \in E^*$ we define $f^*_{(m,n)}: L^* \to L^*$ as follows: For each $\xi \in L^*$, $\alpha \in \hat{\Gamma}$

$$f^*_{(m,n)}(\xi)(\alpha) = \bigwedge \{f_{(m,n)}(\xi(\alpha_1)): \alpha_1 R_{(m,n)} \alpha\}$$

where, by definition, an empty meet yields Ω.

F^* is now constructed from the functions $f^*_{(m,n)}$ exactly as before. The heuristic significance of this definition is the same as in Section 7-4, only now the "tag" updating which occurs when propagation takes place along an interprocedural edge involves less extensive and less precise information. The modified updating operation that has just been defined can be both one-to-many and many-to-one, possibilities which are both reflected in the above formula. It is easy to verify that both monotonicity and distributivity are preserved as we pass from (L, F) to (L^*, F^*).

Next we associate with (L^*, F^*) the data flow problem of determining the maximal fixed-point solution of the equations

$$\begin{aligned} x^*_{r_1} &= \{(w, 0)\} \\ x^*_n &= \bigwedge_{(m,n) \in E^*} f^*_{(m,n)}(x^*_m) \qquad n \in N^* - \{r_1\} \end{aligned} \tag{7-15}$$

As previously, a solution of these equations can be obtained by standard iterative techniques. Once this solution has been obtained, we make the following final calculation:

$$\hat{x}_n = \bigwedge_{\alpha \in \hat{\Gamma}} x^*_n(\alpha) \tag{7-16}$$

The techniques of Section 7-4 can now be applied to analyze the procedure just described. Theorem 7-4.3 retains its validity, if restated as follows:

Theorem 7-6.2.

1. If (L, F) is distributive, then, for each $n \in N^*$, $x^*_n = y^*_n \equiv \bigwedge \{f^*_p(x^*_{r_1}): p \in \mathrm{path}_G\cdot(r_1, n)\}$.
2. If (L, F) is monotone, then, for each $n \in N^*$, $x^*_n \leq y^*_n$.

Instead of Lemma 7-4.5, the following variant applies:

Lemma 7-6.3. Let $n \in N^*$, $p \in \mathrm{path}_G\cdot(r_1, n)$ and $\alpha \in \hat{\Gamma}$. Then $f^*_p(x^*_{r_1})(\alpha)$ is defined iff $\alpha \in \widehat{\mathrm{CM}}(p)$, in which case $f^*_p(x^*_{r_1})(\alpha) = f_p(0)$.

Proof. By induction on the length of p. The assertion is obvious if p is the null execution path. Suppose that it is true for all paths with length $< k$ and let $p = (r_1, s_2, \ldots, s_k, n) \in \mathrm{path}_G\cdot(r_1, n)$ be a path of length k. Let $p_1 = (r_1, s_2, \ldots, s_k)$. Then for each $\alpha \in \hat{\Gamma}$ we have

$$\begin{aligned} f^*_p(x^*_{r_1})(\alpha) &= f^*_{(s_k,n)}[f^*_{p_1}(x^*_{r_1})](\alpha) \\ &= \bigwedge \{f_{(s_k,n)}[f^*_{p_1}(x^*_{r_1})(\alpha_1)]: \alpha_1 R_{(s_k,n)} \alpha\} \end{aligned}$$

Thus $f^*_p(x^*_{r_1})(\alpha)$ is defined iff there exists $\alpha_1 \in \hat{\Gamma}$ such that $\alpha_1 R_{(s_k,n)} \alpha$ and $f^*_{p_1}(x_{r_1})(\alpha_1)$ is defined. By inductive hypothesis, this is true iff there exists $\alpha_1 \in \widehat{\mathrm{CM}}(p_1)$ and $\alpha_1 R_{(s_k,n)} \alpha$, and, by the definition of $R_{(s_k,n)}$ and $\widehat{\mathrm{CM}}$, this last assertion is true iff $\alpha \in \widehat{\mathrm{CM}}(p)$. Hence, applying the inductive hypothesis again, $f^*_{p_1}(x^*_{r_1})(\alpha_1) = f_{p_1}(0)$, for all α_1 appearing in the above meet, so that this meet equals $f_{(s_k,n)}[f_{p_1}(0)] = f_p(0)$. ■

Remark: As previously noted, and can be seen, e.g., from the proof of the last lemma, use of an encoding scheme creates chances for propagation through paths which are not interprocedurally valid. However, our lemma shows that even if an execution path q is encoded by more than one element of Γ, all these "tags" are associated with the same information, namely $f_q(0)$. Thus information is propagated correctly along each path, only more paths are now acceptable for that propagation. These observations will be made more precise in what follows.

Lemma 7-6.4. For each $n \in N^*$, $\mathrm{IVP}(r_1, n) \subseteq \mathrm{IAP}(r_1, n)$.

Proof. Let $q \in \mathrm{IVP}(r_1, n)$ for some $n \in N^*$. We will show, by induction on the length of q, that $\sigma(\mathrm{CM}(q)) \in \widehat{\mathrm{CM}}(q)$, so that by Lemma 7-6.1, $q \in \mathrm{IAP}(r_1, n)$.

Our assertion is obvious if q is the null execution path. Suppose it is true for all paths whose length is less than some $k \geq 0$, and let $n \in N^*$, $q \in \mathrm{IVP}(r_1, n)$ whose length is k. Write $q = q_1 \| (m, n)$. By inductive hypothesis, $\sigma(\mathrm{CM}(q_1)) \in \widehat{\mathrm{CM}}(q_1)$. Now, three cases are possible:

1. $(m, n) \in E^0$. In this case $\widehat{\mathrm{CM}}(q) = \widehat{\mathrm{CM}}(q_1)$ and $\mathrm{CM}(q) = \mathrm{CM}(q_1)$ so that $\sigma(\mathrm{CM}(q)) \in \widehat{\mathrm{CM}}(q)$.
2. (m, n) is a call edge. Then, by definition, $\widehat{\mathrm{CM}}(q)$ contains $\sigma(\mathrm{CM}(q_1)) * \sigma(m) = \sigma(\mathrm{CM}(q))$.

3. (m, n) is a return edge. Let (c', r_p) denote the corresponding call edge. Since $q \in \text{IVP}(r_1, n)$, q can be decomposed as $q' || (c', r_p) || q'' || (m, n)$, where $q' \in \text{IVP}(r_1, c')$ and $q'' \in \text{IVP}_0(r_p, m)$. It is evident from the definitions of the quantities involved that that $\text{CM}(q) = \text{CM}(q')$ and that $\text{CM}(q_1) = \text{CM}(q') || (c')$. Hence $\sigma(\text{CM}(q_1)) = \sigma(\text{CM}(q)) * \sigma(c')$. It thus follows that $\sigma(\text{CM}(q))$ is a member of the set $\{\beta \in \text{ECS}(p) \mid \beta * \sigma(c') = \sigma(\text{CM}(q_1))\}$ which, by definition, is a subset of $\widehat{\text{CM}}(q)$. ■

We can now state an analog of Theorem 7-4.6:

Theorem 7-6.5.

1. If (L, F) is a distributive data flow framework, then, for each $n \in N^*$
$$\hat{x}_n = \bigwedge \{f_p(0) : p \in \text{IAP}(r_1, n)\} \leq y_n$$
2. If (L, F) is only monotone, then, for each $n \in N^*$
$$\hat{x}_n \leq \bigwedge \{f_p(0) : p \in \text{IAP}(r_1, n)\} \leq y_n$$

Proof.

1. Let $\alpha \in \hat{\Gamma}$. By Theorem 7-6.2 and Lemmas 7-6.1 and 7-6.3, we have
$$\begin{aligned} x_n^*(\alpha) &= \bigwedge \{f_p^*(x_{r_1}^*)(\alpha) : p \in \text{path}_G(r_1, n)\} \\ &= \bigwedge \{f_p(0) : p \in \text{IAP}(r_1, n), \alpha \in \widehat{\text{CM}}(p)\} \end{aligned}$$
Thus, by Eq. (7-16)
$$\hat{x}_n = \bigwedge_{\alpha \in \hat{\Gamma}} x_n^*(\alpha) = \bigwedge \{f_p(0) : p \in \text{IAP}(r_1, n)\}$$
By Lemma 7-6.4, this is
$$\leq \bigwedge \{f_p(0) : p \in \text{IVP}(r_1, n)\} = y_n$$
proving (1).
2. Can be proved in a manner completely analogous to the proof of (1), using part (2) of Theorem 7-6.2. ■

Thus $\{\hat{x}_n\}_{n \in N^*}$ is an underestimation of the meet-over-all-paths solution $\{y_n\}_{n \in N^*}$. The degree of underestimation depends on the deviation of $\text{IAP}(r_1, n)$ from $\text{IVP}(r_1, n)$, and this deviation is in turn determined by the choice of $\hat{\Gamma}$, $*$, and σ. The most extreme underestimation results if we let $\text{IAP}(r_1, n) = \text{path}_G(r_1, n)$ for all $n \in N^*$, i.e., define $\hat{\Gamma} = \{w\}$, $w * w = w$, and let σ map all calls to w. If we do this, then the resulting problem is essentially equivalent

to a purely intraprocedural analysis, in which procedure calls and returns are interpreted as mere branch instructions.

Another more interesting encoding scheme is as follows. Choose some integer $k > 1$, and let $\hat{\Gamma}$ be the ring of residue classes modulo k. Let $m > 1$ be another integer. For each $\alpha_1, \alpha_2 \in \hat{\Gamma}$, define $\alpha_1 * \alpha_2 = m \cdot \alpha_1 + \alpha_2 (\text{mod } k)$. Let σ be any map which maps call blocks to values between 0 and $m - 1$ (preferably in a one-to-one way). In this scheme, call strings are mapped into a base m representation modulo k of some encoding of their call blocks. Note that if $k = \infty$, i.e., if we operate with integers rather than in modular arithmetic, then $\hat{\Gamma}$ and Γ are isomorphic, with $\mathbf{b}$ corresponding to concatenation. If $k = m^j$, for some $j \geq 1$, and σ is one-to-one and does not map any call block to 0, then the encoding scheme just proposed can roughly be described as follows: Keep only the last j calls within each call string. As long as the length of a call string is less than j, update it as in Section 7-4. However, if q is a call string of length j, then, when appending to it a call edge, discard the first component of q and add the new call block to its end. When appending a return edge, check if it matches the last call in q, and, if it does, delete this call from q and add to its start all possible call blocks which call the procedure containing the first call in q. This approximation may be termed a *call-string suffix approximation*.

At present we do not have available a comprehensive theory of the proper choice of an encoding scheme. Appropriate choice of such a scheme may depend on the program being analyzed, and reflects the trade-off between tolerable complexity of the interprocedural analysis and some desired level of accuracy.

7-7. CONCLUSION

In this chapter we have studied in some detail two basic approaches to interprocedural analysis of rather general data flow problems. We have seen that by requiring the associated semilattice L to be finite, both approaches yield convergent algorithms which produce the "sharpest" interprocedural information, in a natural sense.

The main concern has been to introduce a comprehensive theory of interprocedural data flow analysis for general frameworks. Subsequent research in this area should address itself to more pragmatic issues that arise when trying to implement our approaches. Some of these issues are:

(a) *Pragmatic implementation of the functional approach for bit-vector problems.* These data flow frameworks are amenable to elimination techniques, which are more efficient than iterative techniques. However, our basic way of solving Eqs. (7-4) is iterative in nature and hence is not optimal for these problems. One would mainly like to come up with an algorithm

which incorporates standard intraprocedural elimination techniques, such as interval analysis, in a modular manner, which will enable us to implement the functional approach as an extension of already existing intraprocedural algorithms rather than as a completely different algorithm.

In addition, one might wish to study the efficiency of such an implementation, bearing in mind that recursion is a somewhat rare phenomenon in actual programs, and that co-recursion is much rarer. This issue is closely related to Allen's approach of processing procedures in "inverse invocation order" (see also [Rose79] for a similar observation). However, careful refinement of this method is required to handle recursion. Additional gain might be achieved by processing "offline" parts of the flow graph which are call-free, so that one does not have to repeat all the intraprocedural processing whenever an interprocedural effect is propagated. One possible approach to this problem, which, however, is probably not the best possible one for implementation, is indicated in [Rose79, Section 8].

(b) *Pragmatic implementation of more complex interprocedural data flow problems.* If the relevant framework is not amenable to elimination, then the functional approach may be inadequate for such a problem. Moreover, some commonly occurring complex data flow problems, such as constant propagation [Hech77], type analysis [Tene74b, Jone76], value flow [Schw75b] or range analysis [Harr77a], are usually solved by algorithms which make use of the *use-definition map* [Alle69] in a way which propagates information only to points where it is actually needed. As indicated in [Shar77], interprocedural extension of such algorithms calls for some proper interprocedural extension of the use-definition map itself. It seems that such an extension can be based on the call-strings approach (or the approximative call-strings approach), but exact details have yet to be worked out.

(c) *Extending our approaches to handle reference parameters.* Here the problem of "aliasing" (i.e., temporary equivalence of two program variables during a procedure call) arises, which complicates matters considerably if "sharpest" information is still to be obtained. Major work in this area has been done by Rosen [Rose79].

(d) *Extending the ideas of the call-strings approach methods which take into account more semantic restrictions on the execution flow.* Thus only flow paths which satisfy such restrictions would be traced during analysis. The call-return pattern of interprocedural flow is but one such possible restriction (though a very important one). For example, one might also keep track of the values of boolean flags which control intraprocedural branches. Current research in such directions by Holley and Rosen at IBM seems quite promising. (We are indebted to Barry K. Rosen for some stimulating discussion concerning the above-mentioned research.)

The present chapter has been motivated by the research on the design and implementation of an optimizing compiler for the SETL programming language at Courant Institute, New York University. SETL is a very-high-level language [Schw75d] which fits into our interprocedural model; i.e., parameters are called by value and no procedure variables are allowed. Active research is now under way to implement the approaches suggested in this chapter in the optimizer of our system, as discussed in (a) and (b) above.