# 1

# DEMAND INTERPROCEDURAL PROGRAM ANALYSIS USING LOGIC DATABASES

**Thomas W. Reps**

*Computer Sciences Department, University of Wisconsin-Madison*
*1210 W. Dayton Street, Madison, WI 53706*
*USA*
*reps@cs.wisc.edu*

## ABSTRACT

This paper describes how algorithms for demand versions of inerprocedural program-analysis problems can be obtained from their exhaustive counterparts essentially for free, by applying the so-called magic-sets transformation that was developed in the logic-programming and deductive-database communities. Applications to interprocedural dataflow analysis and interprocedural program slicing are described.[1]

## 1 INTRODUCTION

Interprocedural analysis concerns the static examination of a program that consists of multiple procedures. Its purpose is to determine certain kinds of summary information associated with the elements of a program (such as reaching definitions, available expressions, live variables, *etc.*). Most treatments of interprocedural analysis address the *exhaustive* version of the problem: summary information is to be reported for *all* elements of the program. This paper concerns the solution of *demand* versions of interprocedural analysis problems: summary information is to be reported only for a single program element of interest (or a small number of elements of interest). Because the summary information at one program point typically depends on summary information from other points, an important issue is to minimize the number of *other* points

[1] An abbreviated version of this paper, discussing only interprocedural dataflow analysis, appeared in the Proceedings of the Fifth International Conference on Compiler Construction [32].

for which (transient) summary information is computed and/or the amount of information computed at those points.

There are several reasons why it is desirable to solve the demand versions of interprocedural analysis problems:

**Narrowing the focus to specific points of interest.** In program optimization, most of the gains are obtained from making improvements at a program's "hot spots"—in particular, its innermost loops. Although the optimization phases during which transformations are applied can be organized to concentrate on hot spots, there is typically an earlier phase to determine dataflow facts during which an exhaustive algorithm for interprocedural dataflow analysis is used. A demand algorithm can greatly reduce the amount of extraneous information that is computed.

With the approach presented in this paper, answers and intermediate results computed in the course of answering one query can be cached—that is, accumulated and used to compute the answers to later queries. This can go on until such time as the program is modified, whereupon previous results, which may no longer be safe, must be discarded.

**Narrowing the focus to specific dataflow facts of interest.** Even when dataflow information is desired for every flow-graph vertex $n$, the full set of dataflow facts at $n$ may not be required. For example, the possibly-uninitialized variables problem is to determine, for each vertex $n$, the set of program variables that may be uninitialized just before execution reaches $n$. A variable $x$ is possibly uninitialized at $n$ either if there is an $x$-definition-free valid path to $n$ or if there is a valid path to $n$ on which the last definition of $x$ uses some variable $y$ that itself is possibly uninitialized. The solution to the possibly-uninitialized variables problem is usually used to report the places in a program where a variable is used that may not have been initialized.

Although only certain variables are used at each program point, the possibly-uninitialized variables problem associates the full set of uninitialized variables with each program point. Therefore it may be advantageous to use a demand algorithm for dataflow analysis (without changing the definition of the problem to be solved) and for each program point $n$ and variable $v$ used at $n$ issue the query "Is $v$ in the set of possibly-uninitialized variables at $n$?"

**Reducing the amount of work spent in preprocessing or other auxiliary phases of a program analysis.** In problems that can be decomposed into separate phases, not all of the information from subsidiary

phases may be required in order to answer an "outer-level" query. For example, the interprocedural MayMod and MayUse problems are to determine, for each call site $c$, which variables may have their values modified by $c$'s execution and which variables may have their values used by $c$'s execution, respectively [6, 11]. The flow-insensitive versions of these side-effect-analysis problems can be decomposed into two subsidiary phases: computing alias information and computing side effects due to reference formal parameters [6, 12, 11]. Given a demand at the outermost level (*e.g.*, "What is the MayMod set for a given call site $c$ on procedure $p$?"), a demand algorithm has the potential to reduce drastically the amount of work spent in preprocessing or other auxiliary phases by propagating only appropriate demands into earlier phases (*e.g.*, "What are the alias pairs that can hold on entry to $p$?").

With the approach presented in this paper, this capability is obtained for free, as a by-product of the way the composition of two computations is treated by the magic-sets transformation [35, 5, 8].

**Sidestepping incremental-updating problems.** An optimizing transformation performed at one point in the program can invalidate previously computed dataflow information at other points in the program. In some cases, the old information at such points is not a "safe" summary of the possible execution states that can arise there; the dataflow information needs to be updated before it is possible to perform optimizing transformations at such points. However, no good incremental algorithms for interprocedural dataflow analysis are currently known.

An alternative is to use an algorithm for the demand version of the dataflow problem and have the optimizer place appropriate demands. With each demand, the algorithm would be invoked on the *current* program. (As indicated above, any information cached from previous queries would be discarded whenever the program is modified.)

**Demand analysis as a user-level operation.** It is desirable to have program-development tools in which the user can ask questions interactively about various aspects of the program [25, 42, 23, 17]. Such tools are particularly useful when debugging, when trying to understand complicated code, or when trying to transform a program to execute efficiently on a parallel machine.

One of the novel aspects of our work is that it establishes a connection between the ideas and concerns from two different research areas. This connection can be summarized as follows:

> Methods for solving demand versions of interprocedural analysis problems—and in particular interprocedural analysis problems of interest to the community that studies *imperative* programs—can be obtained from their exhaustive counterparts essentially for free, by applying a transformation that was developed in the *logic-programming* and *deductive-database* communities for optimizing the evaluation of recursive queries in deductive databases (the so-called magic-sets transformation).

Our methodology for obtaining algorithms that solve the demand versions of interprocedural analysis problems has two phases: (1) encode the algorithm for the exhaustive version of the problem as a logic program; (2) convert the algorithm for the exhaustive version to a demand algorithm by applying the magic-sets transformation. In principle, the second step is completely automatic. In practice—at least with the Coral system [29, 28, 30]—to obtain the most efficient program, the user may need to rewrite certain recursive rules and reorder literals in some rules. Such concerns are outside the scope of this paper. (In the subsequent sections of the paper, we do not actually discuss the programs that result from the transformation—the transformed programs are quite complicated and presenting them would not aid the reader's understanding.)

This paper describes how the above approach can be used to obtain a demand algorithm for the interprocedural "gen-kill problems". (Gen-kill problems are ones in which the dataflow functions are all of the form $\lambda x.(x - kill) \cup gen$. If $\lambda x.(x - kill_n) \cup gen_n$ is the dataflow function associated with program point $n$, $gen_n$ represents dataflow facts "created" at $n$ and $kill_n$ represents dataflow facts "stopped" by $n$.) The paper also makes use of the approach to obtain a demand algorithm for the interprocedural-slicing problem [42, 15, 18, 34, 33].

The remainder of the paper is organized as follows: Section 2 discusses background and assumptions. Section 3 shows how to obtain a demand algorithm for the interprocedural gen-kill problems. Section 4 discusses how to obtain a demand algorithm for the interprocedural-slicing problem. Section 5 presents some experimental results. Section 6 discusses related work.

## 2 BACKGROUND AND ASSUMPTIONS

Interprocedural analysis is typically carried out using a graph data structure to represent the program: the graph used represents both *intra*procedural

information—information about the individual procedures of the program—and *inter*procedural information—*e.g.*, the call/return linkages, the binding changes associated with entering a new scope in the called procedure, *etc.* For example, in interprocedural dataflow analysis, analysis is carried out on a structure that consists of a control-flow graph for each procedure, plus some additional procedure-linkage information [1, 7, 36, 11].

In interprocedural analysis problems, not all of the paths in the graphs that represent the program correspond to possible execution paths. In general, the question of whether a given path is a possible execution path is undecidable, but certain paths can be identified as being infeasible because they would correspond to execution paths with infeasible call/return linkages. For example, if procedure *Main* calls $P$ twice—say at $c_1$ and $c_2$—one infeasible path would start at the entry point of *Main*, travel through the graph for *Main* to $c_1$, enter $P$, travel through the graph for $P$ to the return point, and return to *Main* at $c_2$ (rather than $c_1$). Such paths fail to account correctly for the calling context (*e.g.*, $c_1$ in *Main*) of a called procedure (*e.g.*, $P$). Thus, in many interprocedural analysis problems an important issue is to carry out the analysis so that only *interprocedurally valid paths* are considered [36, 26, 9, 16, 22] (see Definitions 3.3 and 4.5). With the approach taken in this paper, if the exhaustive algorithm considers only interprocedurally valid paths, then the demand algorithm obtained will also consider only interprocedurally valid paths.

To streamline the presentation, the analysis problems discussed in Sections 3 and 4 have been simplified in certain ways. In particular, following Sharir and Pnueli [36] and Horwitz, Reps, and Binkley [16] we assume that (i) the programs being analyzed do not contain aliasing and (ii) the programs being analyzed do not use procedure-valued variables. We also make some simplifying assumptions about global variables. In Section 3, we follow Sharir and Pnueli and assume that (iii) all variables are global variables and (iv) procedures are parameterless. In Section 4, we make the opposite assumptions: we follow Horwitz, Reps, and Binkley and assume that (v) programs use no global variables, but (vi) procedures may have value-result parameters.

A few words about each are in order: (i) The interaction between interprocedural dataflow analysis and the computation of aliasing information has already been mentioned in the Introduction: with the approach presented in this paper only appropriate demands for aliasing information would be generated, which might greatly reduce the amount of work required for alias analysis. (ii) G. Rosay and the author have been able to develop a method for constructing call multigraphs in the presence of procedure-valued variables that is compatible with the dataflow-analysis method described in the paper; this work combines

and extends the methods described by Lakhotia [21] and Callahan *et al.* [10]. Simplifications (iii) and (iv) prevent the Sharir-Pnueli framework for interprocedural dataflow analysis from being able to handle local variables and formal parameters of procedures in the presence of recursion; however, Knoop and Steffen have presented a generalization of the Sharir-Pnueli framework that lifts this restriction [20]. It is possible to generalize the approach described in Section 3 to implement the more general Knoop-Steffen framework. (v) In programs that use value-result parameter passing, global variables can be converted into additional parameters to each procedure. (vi) Extensions needed to slice programs that use call-by-reference parameter passing are discussed in [16].

In the logic programs given in the paper, we follow the standard naming convention used in Prolog: identifiers that begin with lower-case letters denote ground atoms; those that begin with upper-case letters denote variables. In Section 3, we also make use of a notation from Coral for manipulating relations with set-valued fields. This notation will be explained in Section 3.2, where it is first used.

## 3 INTERPROCEDURAL DATAFLOW ANALYSIS PROBLEMS

This section describes how we can obtain demand algorithms for the interprocedural gen-kill problems by encoding an exhaustive dataflow-analysis algorithm as a logic program and applying the magic-sets transformation. The basis for the exhaustive algorithm is Sharir and Pnueli's "functional approach" to interprocedural dataflow analysis, which, for distributive dataflow functions, yields the *meet-over-all-valid-paths* solution to certain classes of flow-sensitive interprocedural dataflow analysis problems [36].

We assume that $(L, \sqcap)$ is a meet semilattice of dataflow facts with a smallest element $\bot$ and a largest element $\top$. We also assume that dataflow functions are members of a space of monotonic (or distributive) functions $F \subseteq L \to L$ and that $F$ contains the identity function.

Sharir and Pnueli make use of two different graph representations of programs, which are defined below.

**Definition 3.1 (Sharir and Pnueli [36])** Let $G_0, G_1, ..., G_k$ be a collection of **flow graphs**, where each $G_p$ is a directed graph corresponding to a procedure of the program, and $G_p = (N_p, E_p, s_p, e_p)$. The **vertex sets** $N_p$, $0 \leq p \leq k$, are pairwise disjoint, as are the **edge sets** $E_p$. Vertex $s_p$ is the unique **start vertex** of $G_p$; vertex $e_p$ is the unique **exit vertex** of $G_p$. Every procedure call contributes two vertices: a **call** vertex and a **return-site** vertex.

$G_p$'s edges are divided into two disjoint subsets: $E_p = E_p^0 \cup E_p^1$. An edge $(m, n) \in E_p^0$ is an ordinary control-flow edge; it represents a direct transfer of control from one vertex to another. An edge $(m, n)$ is in $E_p^1$ iff $m$ is a call vertex and $n$ is the corresponding return-site vertex. (Observe that vertex $n$ is *within* $p$ as well; an edge in $E_p^1$ does *not* run from $p$ to the called procedure, or vice versa.) Without loss of generality, we assume that (i) a return-site vertex in any $G_p$ graph has exactly one incoming edge: the $E_p^1$ edge from the corresponding call vertex; (ii) the entry vertex $s_p$ in any $G_p$ graph is never the target of an $E_p^0$ edge. □

This first representation, in which the flow graphs of individual procedures are kept separate from each other, is the one used by the exhaustive and demand interprocedural dataflow analysis algorithms. The second graph representation, in which the flow graphs of the different procedures are connected together, is used to define the notion of interprocedurally valid paths.

**Definition 3.2 (Sharir and Pnueli [36])** Define $G^* = (N^*, E^*, s_{main})$, where $N^* = \bigcup_p N_p$ and $E^* = E^0 \cup E^2$, where $E^0 = \bigcup_p E_p^0$ is the collection of all ordinary control-flow edges, and an edge $(m, n) \in E^2$ represents either a **call** or **return** edge. Edge $(m, n) \in E^2$ is a call edge iff $m$ is a call vertex and $n$ is the entry vertex of the called procedure; edge $(m, n) \in E^2$ is a return edge iff $m$ is an exit vertex of some procedure $p$ and $n$ is a return-site vertex for a call on $p$. A call edge $(m, s_p)$ and return edge $(e_q, n)$ **correspond** to each other if $p = q$ and $(m, n) \in E_s^1$ for some procedure $s$. □

The notion of interprocedurally valid paths captures the idea that not all paths through $G^*$ represent potential execution paths:

**Definition 3.3 (Sharir and Pnueli [36])** For each $n \in N$, we define $\mathrm{IVP}(s_{main}, n)$ as the set of all **interprocedurally valid paths** in $G^*$ that lead from $s_{main}$ to $n$. A path $q \in \mathrm{path}_{G^*}(s_{main}, n)$ is in $\mathrm{IVP}(s_{main}, n)$ iff the sequence of all edges in $q$ that are in $E^2$, which we will denote by $q_2$, is **proper** in the following recursive sense:

1. A sequence $q_2$ that contains no return edges is proper.
2. If $q_2$ contains return edges, and $i$ is the smallest index in $q_2$ such that $q_2(i)$ is a return edge, then $q_2$ is proper if $i > 1$ and $q_2(i-1)$ is a call edge corresponding to the return edge $q_2(i)$, and after deleting those two components from $q_2$, the remaining sequence is also proper. □

**Definition 3.4 (Sharir and Pnueli [36])** If $q$ is a path in $G^*$, let $pf_q$ denote the (path) function obtained by composing the functions associated with $q$'s edges (in the order that they appear in path $q$). The **meet-over-all-valid-paths** solution to the dataflow problem consists of the collection of values $y_n$ defined by the following set of equations:

$$\begin{array}{ll} \Phi_n = \sqcap_{q \in \mathrm{IVP}(s_{main}, n)} pf_q & \text{for each } n \in N^* \\ y_n = \Phi_n(\bot) & \text{for each } n \in N^* \end{array}$$

□

The solution to the dataflow analysis problem is not actually obtained from these equations, but from two other systems of equations, which are solved in two phases. In Phase I, the equations deal with *summary dataflow functions*, which are defined in terms of dataflow functions and other summary dataflow functions. In Phase II, the equations deal with actual dataflow *values*.

Phase I of the analysis computes summary functions $\phi_{(s_p,n)}(x)$ that map a set of dataflow facts at $s_p$—the entry point of procedure $p$—to the set of dataflow facts at point $n$ within $p$. These functions are defined as the greatest solution to the following set of equations (computed over a (bounded) meet semilattice of functions):

$$\begin{array}{ll} \phi_{(s_p,s_p)} = \lambda x.x & \text{for each procedure } p \\ \phi_{(s_p,n)} = \sqcap_{(m,n) \in E_p} (f_{(m,n)} \circ \phi_{(s_p,m)}) & \text{for each } n \in N_p \text{ not representing a return-site vertex} \\ \phi_{(s_p,n)} = \phi_{(s_q,e_q)} \circ \phi_{(s_p,m)} & \text{for each } n \in N_p \text{ representing a return-site vertex, where } (m,n) \in E_p^1 \text{ and } m \text{ calls procedure } q \end{array}$$

Phase II of the analysis uses the summary functions from Phase I to obtain a solution to the dataflow analysis problem. This solution is obtained from the

greatest solution to the following set of equations:

$$
\begin{array}{ll}
x_{s_{main}} = \bot & \\
x_{s_p} = \sqcap\{\phi_{(s_q,c)}(x_{s_q}) \mid c \text{ is a call to } p \text{ in procedure } q\} & \text{for each procedure } p \\
x_n = \phi_{(s_p,n)}(x_{s_p}) & \text{for each procedure } p \\
 & \text{and } n \in (N_p - \{s_p\})
\end{array}
$$

Sharir and Pnueli showed that if the edge functions are distributive, the greatest solution to the above set of equations is equal to the meet-over-all-valid-paths solution (*i.e.*, for all $n$, $x_n = y_n$) [36].

## 3.1 Representing an Interprocedural Dataflow Analysis Problem

To make use of the Sharir-Pnueli formulation for our purposes, it is necessary to find an appropriate way to use Horn clauses to express (i) the dataflow functions on the edges of the control-flow graph, (ii) the application of a function to an argument, (iii) the composition of two functions, and (iv) the meet of two functions.

In this paper, we restrict our attention to the class of problems that can be posed in terms of functions of the form $\lambda x.(x - kill) \cup gen$, with $\cup$ as the meet operator. (Examples of such problems are reaching definitions and live variables.) To encode the dataflow functions, instead of *kill* sets we use $\overline{kill}$ sets (which we will denote by *nkill*). That is, each dataflow function is rewritten in the form $\lambda x.(x \cap nkill) \cup gen$. Such a function can be represented as a pair $(nkill, gen)$. Given this representation of edge functions, it is easy to verify that the rules for performing the composition and meet of two functions are as follows:

$$
\begin{array}{ll}
(nkill_2, gen_2) \circ (nkill_1, gen_1) = (nkill_1 \cap nkill_2, (gen_1 \cap nkill_2) \cup gen_2) & (\dagger) \\
(nkill_2, gen_2) \ \sqcap \ (nkill_1, gen_1) = (nkill_1 \cup nkill_2, gen_1 \cup gen_2) & (\ddagger)
\end{array}
$$

An instance of a dataflow analysis problem is represented in terms of five base relations, which represent the following pieces of information:

`e0(p,m,n)` Edge $(m, n) \in E_p^0$; that is $(m, n)$ is an ordinary (intraprocedural) control-flow edge in procedure $p$.

`e1(p,m,n)` Edge $(m,n) \in E_p^1$. (Bear in mind that both endpoints of an edge in $E_p^1$ are in $p$; edge $(m,n)$ does *not* run from $p$ to the called procedure, or vice versa.)

`f(p,m,n,nk_set,g_set)` A tuple `f(p,m,n,nk_set,g_set)` represents the function on edge $(m,n) \in E_p^0$. (The set-valued fields `nk_set` and `g_set` represent the *nkill* set and the *gen* set, respectively.)

`call_site(p,q,m)` Vertex $m$ in procedure $p$ represents a call on procedure $q$.

`universe(u)` $u$ is a set-valued field that consists of the universe of dataflow facts.

## 3.2 The Encoding of Phase I

We now show how to encode Phase I of the Sharir-Pnueli functional approach to dataflow analysis. There are two derived relations, `phi_nk(p,n,x)` and `phi_g(p,n,x)`, which together represent $\phi_{(s_p,n)}$, the summary function for vertex $n$ of procedure $p$. (Recall that each dataflow function corresponds to a pair $(nkill, gen)$.)

`phi_nk(p,n,x)` A tuple `phi_nk(p,n,x)` represents the fact that $x$ is a member of the *nkill* component of $\phi_{(s_p,n)}$.

`phi_g(p,n,x)` A tuple `phi_g(p,n,x)` represents the fact that $x$ is a member of the *gen* component of $\phi_{(s_p,n)}$.

The rules that encode Phase I perform compositions and meets of (representations of) dataflow functions according to equations (†) and (‡) (although the way in which this is accomplished is somewhat disguised).

### *Initialization Rule*

```
phi_nk(P,start_vertex,X) :- universe(U), member(U,X).
```

In Coral, a literal of the form `member(S,X)`, where `S` is a set, causes `X` to be bound successively to each of the different members of `S`. (If `X` is already bound, then `X` is checked for membership in `S`.) Thus, for each procedure $p$ the *nkill* component of the function $\phi_{(s_p,s_p)}$ consists of the universe of dataflow facts (*i.e.*, nothing is killed along the 0-length path from the start vertex to itself).

## *Intraprocedural Summary Functions*

The rules for *intra*procedural summary functions correspond to the equation

$\phi_{(s_p,n)} = \sqcap_{(m,n)\in E_p}(f_{(m,n)} \circ \phi_{(s_p,m)})$ for each $n \in N_p$ not representing a return-site vertex

Note from equation (†) that the composition $f_{(m,n)} \circ \phi_{(s_p,m)}$ is implemented as

$$(nkill_{f_{(m,n)}}, gen_{f_{(m,n)}}) \circ (nkill_{\phi_{(s_p,m)}}, gen_{\phi_{(s_p,m)}})$$
$$= (nkill_{\phi_{(s_p,m)}} \cap nkill_{f_{(m,n)}}, (gen_{\phi_{(s_p,m)}} \cap nkill_{f_{(m,n)}}) \cup gen_{f_{(m,n)}}).$$

The three rules given below create the intraprocedural summary functions.

```
phi_nk(P,N,X) :- e0(P,M,N),
                  f(P,M,N,NK_set,_),
                  member(NK_set,X),
                  phi_nk(P,M,X).
phi_g(P,N,X) :- e0(P,M,N),
                 f(P,M,N,NK_set,_),
                 member(NK_set,X),
                 phi_g(P,M,X).
phi_g(P,N,X) :- e0(P,M,N),
                 f(P,M,N,_,G_set),
                 member(G_set,X).
```

For example, the first rule specifies the following:

> Given an intraprocedural edge in procedure `P` from `M` to `N`, where edge-function $f$'s *nkill* component is `NK_set`, add `X` $\in$ `NK_set` to the *nkill* component of $\phi_{(s_P,N)}$ only if `X` is in the *nkill* component of $\phi_{(s_P,M)}$.

This is another way of saying "Take the intersection of the *nkill* components of $f_{(M,N)}$ and $\phi_{(s_P,M)}$", which is exactly what is required for the *nkill* component of the composition $f_{(M,N)} \circ \phi_{(s_P,M)}$.

## *Interprocedural Summary Functions*

The rules for *inter*procedural summary functions correspond to the equation

$\phi_{(s_p,n)} = \phi_{(s_q,e_q)} \circ \phi_{(s_p,m)}$ for each $n \in N_p$ representing a return-site vertex, where $(m,n) \in E_p^1$ and $m$ calls procedure $q$

From equation (†), the composition $\phi_{(s_q,e_q)} \circ \phi_{(s_p,m)}$ is implemented as

$$\begin{aligned}(&nkill_{\phi_{(s_q,e_q)}}, gen_{\phi_{(s_q,e_q)}}) \circ (nkill_{\phi_{(s_p,m)}}, gen_{\phi_{(s_p,m)}}) \\ &= (nkill_{\phi_{(s_p,m)}} \cap nkill_{\phi_{(s_q,e_q)}}, (gen_{\phi_{(s_p,m)}} \cap nkill_{\phi_{(s_q,e_q)}}) \cup gen_{\phi_{(s_q,e_q)}}).\end{aligned}$$

Thus, the following additional rules account for the propagation of summary information between procedures:

```
phi_nk(P,N,X) :- e1(P,M,N),
                  call_site(P,Q,M),
                  phi_nk(P,M,X),
                  phi_nk(Q,return_vertex,X).
phi_g(P,N,X) :- e1(P,M,N),
                call_site(P,Q,M),
                phi_g(P,M,X),
                phi_nk(Q,return_vertex,X).
phi_g(P,N,X) :- e1(P,M,N),
                call_site(P,Q,M),
                phi_g(Q,return_vertex,X).
```

For instance, the first rule specifies the following:

> Given an edge from `M` to `N` in `P`, where `M` is a call site on procedure `Q`, the *nkill* component of $\phi_{(s_p,n)}$ consists of the `X`'s such that `X` is in both the *nkill* component of $\phi_{(s_P,M)}$ and the *nkill* component of $\phi_{(s_Q,e_Q)}$.

Again, this takes the intersection of the two *nkill* components, which is exactly what is required for the *nkill* component of the composition $\phi_{(s_Q,e_Q)} \circ \phi_{(s_P,M)}$.

Note that there are two rules—one intraprocedural, one interprocedural—whose head is `phi_nk(P,N,X)`, each of which can contribute tuples to the relation `phi_nk`. In the flow graph, a given vertex `N` either has one `e1` predecessor or a number of `e0` predecessors. Thus, the rule for `phi_nk` in the intraprocedural group can cause tuples to be added to `phi_nk` because of different predecessors `M` of `N`. This gives the correct implementation because what is required is the meet (pointwise $\cup$) of the $\phi$ functions obtained from all predecessors, and the rule for the meet of two functions involves the union of *nkill* sets (see equation (‡)). Similar reasoning applies in the case of relation `phi_g`, which is defined using four different rules.

It should also be noted that the reason we chose our function representations to be *nkill/gen* pairs rather than *kill/gen* pairs was so that the meet of two functions could be handled by the union implicit in having multiple rules that define **phi_nk** and **phi_g**, as well as the fact that rules have multiple solutions. If *kill* sets had been used instead, then to implement the function-meet operation we would have needed a way to perform an intersection over the *kill* sets generated from all predecessors of **N**. (With the *nkill/gen* representation of functions, the only intersections needed are for implementing the *composition* of functions. For example, the first component on the right-hand side of rule (†) is $nkill_1 \cap nkill_2$. However, we never have to perform an explicit composition of an *arbitrary* number of functions: every composition involves exactly *two* functions, and hence requires taking the intersection of exactly *two nkill* sets. These (binary) intersection operations are implemented in the Coral program by having two literals—either **f** and **phi_nk** or two **phi_nk**'s—in the *same* rule.) If we were to represent functions as *kill/gen* pairs, the (binary) meet of two functions would involve the operation $kill_1 \cap kill_2$. This would cause a problem because we need to perform meets over functions created from an *arbitrary* number of predecessors. That is, it would be necessary to perform the $k$-ary operation $\bigcap_{i=1}^{k} kill_i$; however, this cannot be captured statically in a single rule.

**Remark.** For similar reasons, a certain amount of finessing is required to handle interprocedural dataflow problems for which the meet operation in the semilattice of dataflow facts is $\cap$. (An example of such a problem is the available-expressions problem.) Such problems are dual to the problems for which meet is $\cup$ in the following sense: if the edge functions are of the form $\lambda x.(x \cap nkill) \cup gen$ they can be put into the form $\lambda x.(x \cup gen) \cap (nkill \cup gen)$. If we define the pair $[a, b]$ to represent the function $\lambda x.(x \cup a) \cap b$, then all flow functions are of the form $[gen, nkill \cup gen]$. Composition and meet of $[\bullet, \bullet]$ pairs for an $\cap$-semilattice of dataflow facts are dual to composition and meet of $(\bullet, \bullet)$ pairs for a $\cup$-semilattice of dataflow facts. However, the meet (pointwise $\cap$) of $k$ functions represented as $[\bullet, \bullet]$ pairs would require performing the $k$-ary operation $\bigcap_{i=1}^{k} gen_i$. Again, the problem is that this cannot be captured statically in a single rule.

To sidestep this difficulty, for intersection problems we would represent functions with *kill* and *ngen* sets: a pair $< kill, ngen >$ would represent the function $\lambda x.(x \cap \overline{kill}) \cup \overline{ngen}$. It can be shown that the meet (pointwise $\cap$) of two functions represented as $< \bullet, \bullet >$ pairs has the following implementation:

$$
\begin{aligned}
&< kill_2, ngen_2 > \sqcap < kill_1, ngen_1 > \\
&\quad =< (kill_1 \cap ngen_1) \cup (kill_2 \cap ngen_2), ngen_1 \cup ngen_2 > .
\end{aligned}
$$

Consequently, the meet of $k$ such functions represented as $< \bullet, \bullet >$ pairs is

$$\prod_{i=1}^{k} < kill_i, ngen_i > = < \bigcup_{i=1}^{k} (kill_i \cap ngen_i), \bigcup_{i=1}^{k} ngen_i > .$$

This avoids the need to perform an intersection of a collection of sets generated from a vertex's predecessors. The only intersections that need to be performed involve information that is generated along an *individual* edge (*i.e.*, $kill_i \cap ngen_i$); such binary intersections can be captured statically in a single rule. Combining the information from the set of *all* incoming edges involves only unions, and this can be handled using multiple rules (with multiple solutions). □

## 3.3 The Encoding of Phase II

In Phase II, (the representations of) dataflow functions are applied to dataflow facts. Given a set of dataflow facts $x$ and a dataflow function represented as a pair $(nkill, gen)$, we need to create the set $(x \cap nkill) \cup gen$.

Phase II involves one derived relation, `df_fact(p,n,x)`, which represents the fact that $x$ is a member of the dataflow-fact set for vertex $n$ of procedure $p$.

```
df_fact(P,start_vertex,X) :- call_site(Q,P,C),
                             df_fact(Q,start_vertex,X),
                             phi_nk(Q,C,X).
df_fact(P,start_vertex,X) :- call_site(Q,P,C),
                             phi_g(Q,C,X).
df_fact(P,N,X) :- N <> start_vertex,
                  df_fact(P,start_vertex,X),
                  phi_nk(P,N,X).
df_fact(P,N,X) :- N <> start_vertex,
                  phi_g(P,N,X).
```

The first and second rules propagate facts *inter*procedurally—from the start vertex of one procedure (`Q`) to the start vertex of a called procedure (`P`). The first rule specifies that

> `X` is a fact at the start vertex of `P` if (i) `P` is called by `Q` at `C`, (ii) `X` is a fact at the start vertex of `Q`, and (iii) `X` is not killed along the path in `Q` from the start vertex to `C`.

The second rule specifies that

> `X` is a fact at the start vertex of `P` if (i) `P` is called by `Q` at `C` and (ii) `X` is generated along the path in `Q` from the start vertex of `Q` to `C`.

As in Phase I, the meet ($\cup$) over all predecessors is handled by the disjunction implicit in having multiple rules that define `df_fact(P,start_vertex,X)`, as well as the fact that rules have multiple solutions.

Rules three and four are similar to rules one and two, but propagate facts *intra*procedurally, *i.e.*, from the start vertex of `P` to other vertices of `P`.

## 3.4 Creating the Demand Version

The directive

```
export df_fact(bbf).
```

directs the Coral system to apply the magic-sets transformation to transform the program to a form that is specialized for answering queries of the form "`?df_fact(p,n,X)`". The transformed program (when evaluated bottom up) is an algorithm for the demand version of the interprocedural dataflow analysis problem: the set of dataflow facts for vertex $n$ of procedure $p$ is the collection of all bindings returned for `X`. During the evaluation of a query "`?df_fact(p, n, X)`", the algorithm computes `phi_nk` and `phi_g` tuples for all vertices on valid paths to vertex $n$, `df_fact` tuples for all start vertices that occur on valid paths to $n$, and `df_fact` tuples for vertex $n$ itself; finally, it selects the bindings for `X` from the `df_fact` tuples for $n$.

# 4 INTERPROCEDURAL SLICING

In this section, we consider the problem of interprocedural slicing. The *slice* of a program with respect to program point $p$ and variable $x$ consists of all statements and predicates of the program that might affect the value of $x$ at point $p$. This concept, originally discussed by Mark Weiser in [42], allows one to isolate individual computation threads within a program. Slicing can be used for such

diverse activities as helping a programmer understand complicated code, aiding debugging [24], automatically parallelizing programs [41, 4], and automatically combining program variants [14]. (See [37] for an extensive survey of work on program slicing.)

The problem of *inter*procedural slicing concerns how to determine a slice of an entire program, where the slice crosses the boundaries of procedure calls. One algorithm for interprocedural slicing was presented by Weiser [42]. However, as pointed out independently by Horwitz, Reps, and Binkley [15] and Hwang, Du, and Chou [18], Weiser's algorithm is *imprecise* in the sense that it can report "effects" that are transmitted (only) through paths in a graph representation of the program that do not represent "feasible execution paths". The algorithms of Horwitz, Reps, and Binkley and Hwang, Du, and Chou improve on Weiser's algorithm by only considering effects transmitted along interprocedurally valid paths.

Previous work on interprocedural slicing has the following drawbacks:

- The interprocedural slicing algorithm of Weiser [42] is imprecise in the sense that it considers paths that are not interprocedurally valid paths.
- The algorithm of Hwang, Du, and Chou can, in the worst case, take exponential time [33].
- As shown in [16], the algorithm of Horwitz, Reps, and Binkley is a polynomial-time algorithm; however, even though the slicing problem asks "What statements and predicates might affect the value of $x$ at $p$?", which is a demand query, the Horwitz-Reps-Binkley algorithm has a phase in which certain auxiliary information is computed by a preliminary exhaustive algorithm. In other words, although slicing is a demand *problem*, the Horwitz-Reps-Binkley algorithm is not a true demand *algorithm*.

In contrast, using the approach described in the paper we have explored a number of true demand algorithms for the interprocedural-slicing problem. For example, although the version of the Horwitz-Reps-Binkley algorithm reported in [16] is not a true demand algorithm, by using the approach described in the paper—encoding the problem as a logic program and applying the magic-sets transformation—we were able to obtain a true demand version of it. To simplify the presentation in this section, however, we will present not the Horwitz-Reps-Binkley algorithm, but a more straightforward algorithm (which we call the *valid-paths algorithm*).

## 4.1 Background

In Weiser's terminology, a *slicing criterion* is a pair $< p, V >$, where $p$ is a program point and $V$ is a subset of the program's variables. In his work, a slice consists of all statements and predicates of the program that might affect the values of variables in $V$ at point $p$. This is a more general kind of slice than is often needed: Rather than a slice taken with respect to program point $p$ and an *arbitrary* variable, one is often interested in a slice taken with respect to a variable $x$ that is *defined* or *used* at $p$. The value of a variable $x$ defined at $p$ is directly affected by the values of the variables used at $p$ and by the loops and conditionals that enclose $p$. The value of a variable $y$ *used* at $p$ is directly affected by assignments to $y$ that reach $p$ and by the loops and conditionals that enclose $p$.

In *intra*procedural slicing—the problem of slicing a program that consists of just a single monolithic procedure—a slice can be determined from the closure of the directly-affects relation. Ottenstein and Ottenstein pointed out how well-suited *program dependence graphs* are for this kind of slicing [27]. Once a program is represented by its program dependence graph, the slicing problem is simply a vertex-reachability problem, and thus slices may be computed in time linear in the size of the dependence graph. (As we shall see in Section 4.2, for *inter*procedural slicing, treating the problem as a simple vertex-reachability problem leads to imprecise slices.)

To understand the results reported in this paper, it is not really important to understand the details of how program dependence graphs are constructed or even the exact nature of the various kinds of edges that occur in them, and for this reason such material is not included in the paper. For the purposes of this paper, it is really only necessary to understand the way in which dependence graphs for different procedures are linked together at call sites to reflect value-result parameter passing. This is explained in Section 4.2.

## 4.2 The Calling-Context Problem and Valid Paths

Weiser describes his method for interprocedural slicing as follows [42]:

> For each criterion $C$ for a procedure $P$, there is a set of criteria $\mathrm{UP}_0(C)$ which are those needed to slice callers of $P$, and a set of

> criteria $\text{DOWN}_0(C)$ which are those needed to slice procedures called by $P$. ... $\text{UP}_0(C)$ and $\text{DOWN}_0(C)$ can be extended to functions UP and DOWN which map sets of criteria into sets of criteria. Let $CC$ be any set of criteria. Then
>
> $$\begin{aligned} \text{UP}(CC) &= \bigcup_{C \in CC} \text{UP}_0(C) \\ \text{DOWN}(CC) &= \bigcup_{C \in CC} \text{DOWN}_0(C) \end{aligned}$$
>
> The union and transitive closure of UP and DOWN are defined in the usual way for relations. $(\text{UP} \cup \text{DOWN})^*$ will map any set of criteria into all those criteria necessary to complete the corresponding slices through all calling and called routines. The complete interprocedural slice for a criterion $C$ is then just the union of the intraprocedural slices for each criterion in $(\text{UP} \cup \text{DOWN})^*(C)$.

However, this method does not produce as precise a slice as possible because the transitive-closure operation fails to account for the calling context of a called procedure. For example, the relation $(\text{UP} \cup \text{DOWN})^*(< p, V >)$ includes the relation $\text{UP}(\text{DOWN}(< p, V >))$. $\text{UP}(\text{DOWN}(< p, V >))$ includes all call sites that call procedures containing the program points in $\text{DOWN}(< p, V >)$, not just the procedure that contains $p$. This fails to account for the calling context, namely the procedure that contains $p$.

**Example**. To illustrate this problem, and the shortcomings of Weiser's algorithm, consider the following example program, which sums the integers from 1 to 10. (Recall that parameters are passed by value-result.)

```
program Main            procedure A(x, y)         procedure Increment(z)
  sum := 0;               call Add(x, y);           call Add(z, 1)
  i := 1;                 call Increment(y)       return
  while i < 11 do       return
    call A(sum, i)
  od;                   procedure Add(a, b)
  output(sum);            a := a + b
  output(i)             return
end
```

Using Weiser's algorithm to slice this program with respect to variable $z$ and the **return** statement of procedure *Increment*, we obtain everything from the original program except for the two output statements in procedure *Main*. However, a closer inspection reveals that computations involving the variable *sum* do not contribute to the value of $z$ at the end of procedure *Increment*;

in particular, neither the initialization of *sum*, nor the first actual parameter of the call on procedure *A* in *Main*, nor the call on *Add* in *A* (which adds the current value of *i* to *sum*) should be included in the slice. The reason these components are included in the slice computed by Weiser's algorithm is as follows: The initial slicing criterion “<end of procedure *Increment*, $\{z\}$>”, is mapped by the DOWN relation to a slicing criterion “<end of procedure *Add*, $\{a\}$>”. The latter criterion is then mapped by the UP relation to *two* slicing criteria—corresponding to *all* sites that call *Add*—the criterion “<call on *Add* in *Increment*, $\{z\}$>” and the (irrelevant) criterion “<call on *Add* in *A*, $\{x, y\}$>”. Weiser's algorithm does not produce as precise a slice as possible because transitive closure fails to account for the calling context (*Increment*) of a called procedure (*Add*), and thus generates a spurious criterion (<call on *Add* in *A*, $\{x, y\}$>).

A more precise slice consists of the following elements:

```
program Main          procedure A(y)          procedure Increment(z)
  i := 1;               call Increment(y)       call Add(z, 1)
  while i < 11 do     return                  return
    call A(i)
  od                  procedure Add(a, b)
end                     a := a + b
                      return
```

This set of program elements is computed by the slicing algorithms of Horwitz, Reps, and Binkley, and Hwang, Du, and Chou. □

The lesson to be learned from this example is that treating the interprocedural-slicing problem as a simple vertex-reachability problem leads to imprecise slices—in particular, it can reports effects that are “transmitted” only through “invalid paths”.

In the interprocedural-slicing problem, the notion of a “valid path” (see Definition 4.5 below) is slightly more general than the one used in the Sharir-Pnueli framework for interprocedural dataflow analysis (Definition 3.3). Valid paths are most easily understood in terms of the “system dependence graph”, a graph used to represent multi-procedure programs (“systems”) and to implement the Horwitz-Reps-Binkley slicing algorithm. The system dependence graph extends previous dependence-graph representations to incorporate collections of procedures (with procedure calls) rather than just monolithic programs. The system dependence graph for the example program discussed above is shown in Figure 1.

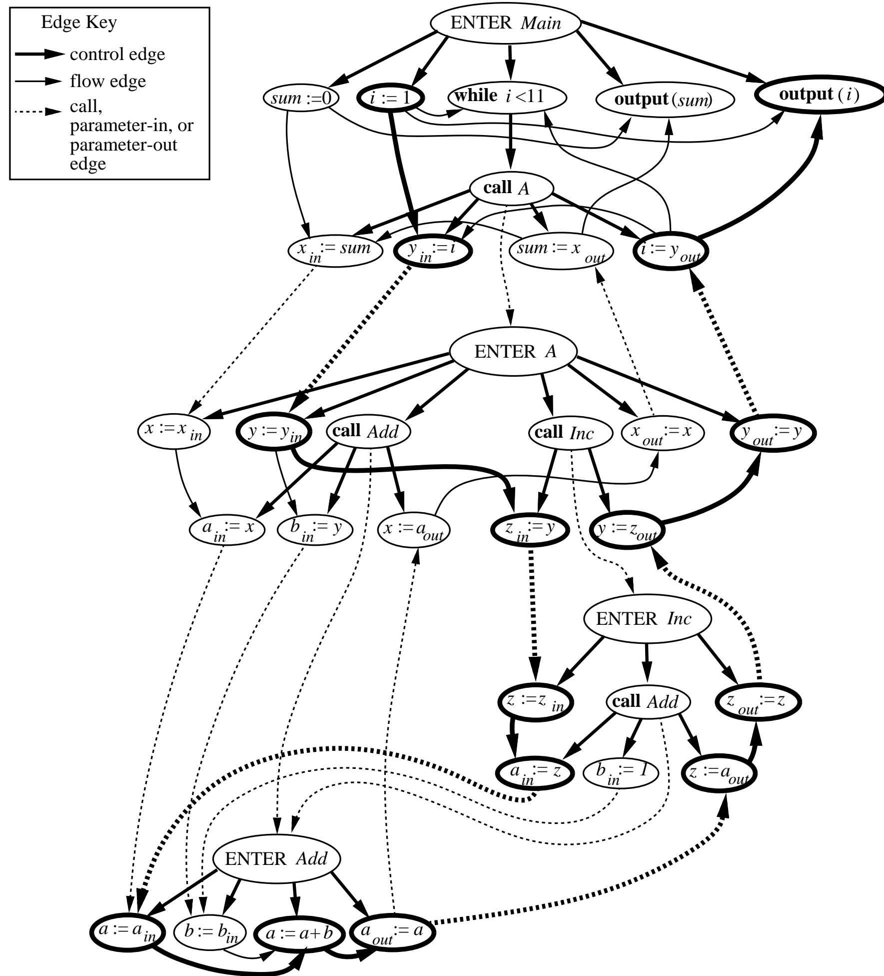


**Figure 1** Example program and corresponding procedure dependence graphs connected with parameter-in, parameter-out, and call edges. The path shown in bold is a same-level valid path from $i := 1$ to the output vertex for variable $i$ in procedure $Main$.

For the purposes of this paper, the salient feature of system dependence graphs is the manner in which dependence graphs for different procedures are linked together at call sites to reflect value-result parameter passing. This involves five kinds of vertices: A call site is represented by a *call-site vertex*; information transfer is represented by four kinds of *parameter vertices*:

- An *actual-in vertex* represents the action taken by the calling procedure to pass in the actual parameter.
- A *formal-in vertex* represents the action taken by the called procedure to receive the parameter's value.
- A *formal-out vertex* represents the action taken by the called procedure to pass back the parameter's return value.
- An *actual-out vertex* represents the action taken by the calling procedure to receive the parameter's return value.

Edges from actual-in vertices to formal-in vertices are called *parameter-in edges*; edges from formal-out vertices to actual-out vertices are called *parameter-out edges*; edges from call-site vertices to entry vertices are called *call edges*. These are shown as dashed lines in Figure 1.

As before, the notion of a "valid path" captures the idea that not every path through a system dependence graph represents a potential execution path:

**Definition 4.5** Let each call site in program $P$ be given a unique index in the range [1...CallSites], where CallSites is the total number of call sites in the program. For each call site $i$, label the call-site's parameter-in edges and parameter-out edges by the symbols "$(_i$" and "$)_i$", respectively; label each call edge from call site $i$ to the entry vertex for the called procedure with "$(_i$".

A **same-level valid path** in the system dependence graph for $P$ is a path in the graph such that the sequence of symbols labeling the parameter-in, parameter-out, and call edges in the path is a string in $L(matched)$, where $L(matched)$ is the language of balanced parentheses generated from nonterminal *matched* by the following context-free grammar:

$$\begin{array}{lcl} matched & \rightarrow & matched \;\; (_i \;\; matched \;\; )_i \quad \text{for } 1 \leq i \leq \text{CallSites} \\ & | & \epsilon \end{array}$$

A **valid path** is a path in the system dependence graph for $P$ such that the sequence of symbols labeling the parameter-in, parameter-out, and call edges is a string in $L(valid)$, where $L(valid)$ is the language generated from nonterminal *valid* in the following grammar (where *matched* is as defined above):[2]

$$
\begin{array}{lcll}
\textit{unbal_right} & \rightarrow & \textit{unbal_right}\ )_i\ \textit{matched} & \text{for } 1 \leq i \leq \text{CallSites} \\
 & | & \textit{matched} & \\
\textit{unbal_left} & \rightarrow & \textit{unbal_left}\ (_i\ \textit{matched} & \text{for } 1 \leq i \leq \text{CallSites} \\
 & | & \textit{matched} & \\
\textit{valid} & \rightarrow & \textit{unbal_right}\ \ \textit{unbal_left} &
\end{array}
$$

□

A same-level valid path from $v$ to $w$ represents the transmission of an effect from $v$ to $w$, where $v$ and $w$ are in the same procedure, via a sequence of execution steps during which the call stack may temporarily grow deeper—because of calls—but never shallower than its original depth before eventually returning to its original depth. An example of a same-level valid path is shown in bold Figure 1. The "*unbal_right*" part of a valid path represents an execution sequence that may leave the call stack shallower than it was originally; the "*unbal_left*" part represents an execution sequence that may leave the call stack deeper than it was originally.

## 4.3 The Encoding of the Valid-Paths Algorithm

Interprocedural slicing is carried out on a collection of procedure dependence graphs together with interprocedural linkage information. This structure is represented in the database in terms of seven base relations, which represent the following items:

`pdg(p,v,w)` Each tuple `pdg(p,v,w)` represents a dependence edge from vertex $v$ to vertex $w$ in the procedure dependence graph for procedure $p$.

`actual_in(p,v)` Vertex $v$ in procedure $p$ is an actual-in vertex.

`formal_in(q,w)` Vertex $w$ in procedure $q$ is a formal-in vertex.

[2] The notion of *unbal_left* in the definition of valid path is equivalent to that of a "proper sequence" used in Definition 3.3.

`formal_out(q,x)` Vertex $x$ in procedure $q$ is a formal-out vertex.

`actual_out(p,y)` Vertex $y$ in procedure $p$ is an actual-out vertex.

`pv(p,v,q,a,b)` Vertex $v$ in procedure $p$ represents the $b^{th}$ actual-in or actual-out vertex of the $a^{th}$ site of a call on $q$ in $p$. (Tuples in which $p = q$ and $a = 1$ actually represent something slightly different, as follows: vertex $v$ in procedure $p$ represents the $b^{th}$ formal-in or formal-out vertex of $p$.)

`call_vertex(p,q,w)` Vertex $w$ in procedure $p$ represents a call on procedure $q$.

The encoding of the valid-paths algorithm involves five derived relations:

- `rpdg(p,v,w)`,
- `matched_path(p,z,u)`,
- `unbalanced_left_path(r,z,p,u)`,
- `unbalanced_right_path(r,z,p,u)`, and
- `valid_path(r,z,p,u)`.

The `rpdg` relation represents the reflexive transitive closure of the `pdg` relation:

```
rpdg(P,V,W) :- rpdg(P,X,W), pdg(P,V,X).
rpdg(P,V,V).
```

The `matched_path` relation captures same-level valid paths (which correspond to paths in the language $L(matched)$ of Definition 4.5). A same-level slice with respect to vertex $z$ in procedure $p$ reports which other vertices within $p$ can affect vertex $z$ (via same-level valid paths that may pass through other procedures). The `matched_path` relation is defined by the following two rules:

```
matched_path(P,Z,U) :- rpdg(P,U,Z).
matched_path(P,Z,U) :- matched_path(P,Z,Y),
                       actual_out(P,Y),
                       pv(P,Y,Q,A,C),
                       pv(Q,X,Q,1,C),
```

```
formal_out(Q,X),
matched_path(Q,X,W),
formal_in(Q,W),
pv(Q,W,Q,1,B),
pv(P,V,Q,A,B),
actual_in(P,V),
rpdg(P,U,V).
```

The second rule specifies that

> `U` is in the same-level slice of `P` with respect to `Z` if (i) actual-out vertex `Y` is in the same-level slice of `P` with respect to `Z`, (ii) formal-out vertex `X` in procedure `Q` is the formal-out vertex that corresponds to `Y`, (iii) formal-in vertex `W` is in the same-level slice of `Q` with respect to `X`, (iv) `V` is the actual-in vertex that corresponds to `W` *at the* **same** *call site as actual-out vertex* `Y` (note the use of logical variable `A` in lines 3 and 9), and (v) there is a path in `P`'s PDG from `U` to `V`.

Observe that the aspect of this rule that constrains the `matched_path` relation to take into account only same-level valid paths is the presence of the "matching groups" of literals that represent a matching call/return linkage:

- the literals `actual_out(P,Y)`, `pv(P,Y,Q,A,C)`, `pv(Q,X,Q,1,C)`, and `formal_out(Q,X)`, which link actual-out vertices to formal-out vertices, represent the transfer to the called procedure;
- the literals `formal_in(Q,W)`, `pv(Q,W,Q,1,B)`, `pv(P,V,Q,A,B)`, and `actual_in(P,V)`, which link formal-in vertices to actual-in vertices, represent the transfer back to the calling procedure *at the* **same** *call site (*`A`*) as actual-out vertex* `Y`.

The `unbalanced_left_path` relation captures paths in the language $L(unbal_left)$ of Definition 4.5. (Recall that an "unbalanced-left" valid path represents a sequence of execution steps during which the call stack may temporarily grow deeper because of calls—but never shallower than its original depth—and where the call stack may be left deeper than it was originally.) The `unbalanced_left_path` relation is defined by the following three rules:

```
unbalanced_left_path(R,Z,P,U) :- P = R, matched_path(R,Z,U).
unbalanced_left_path(R,Z,P,U) :- unbalanced_left_path(R,Z,Q,Y),
                                 formal_in(Q,Y),
                                 pv(Q,Y,Q,1,B),
                                 pv(P,W,Q,A,B),
                                 actual_in(P,W),
                                 matched_path(P,W,U).
unbalanced_left_path(R,Z,P,U) :- unbalanced_left_path(R,Z,Q,Y),
                                 enter_vertex(Q,Y),
                                 call_vertex(P,Q,W),
                                 matched_path(P,W,U).
```

The `unbalanced_right_path` relation captures paths in the language $L(\mathit{unbal_right})$ of Definition 4.5. (Recall that an "unbalanced-right" valid path represents a sequence of execution steps during which the call stack may temporarily grow deeper—because of calls—but where the call stack may be left shallower than it was originally.) The `unbalanced_right_path` relation is defined by the following two rules:

```
unbalanced_right_path(R,Z,P,U) :- P = R, matched_path(R,Z,U).
unbalanced_right_path(R,Z,P,U) :- unbalanced_right_path(R,Z,Q,Y),
                                  actual_out(Q,Y),
                                  pv(Q,Y,P,A,B),
                                  pv(P,W,P,1,B),
                                  formal_out(P,W),
                                  matched_path(P,W,U).
```

Finally, the `valid_path` relation captures paths in the language $L(\mathit{valid})$ of Definition 4.5. The `valid_path` relation is defined by the following rule:

```
valid_path(R,Z,P,U) :- unbalanced_left_path(R,Z,Q,W),
                       unbalanced_right_path(Q,W,P,U).
```

The directive

```
export valid_path(bbff).
```

directs the Coral system to create a transformed program that is specialized for answering queries of the form "`?valid_path(r,z,P,U)`". During the eval-

uation of a query "`?valid_path(r,z,P,U)`" (by bottom-up evaluation of the transformed program) `valid_path` tuples are computed only for the vertices on valid paths to `z`.

# 5 PRELIMINARY EXPERIMENTAL RESULTS

This section reports some performance results obtained on a Sun 10 with pre-Release 1.0 of the Coral deductive database system [28]. Although the demand algorithm for the interprocedural reaching-definitions problem has been tested only on examples of trivial size, the two demand algorithms for the interprocedural-slicing problem (the valid-paths algorithm and the Horwitz-Reps-Binkley algorithm) have been tested on several programs of modest size, the largest being a 757-line program for text formatting taken from Kernighan and Plauger's book on software tools [19]. The following table shows how the execution times of the Coral-obtained demand versions of the two interprocedural-slicing algorithms vary depending on the size of the answer:

| Performance of Demand Algorithms for Interprocedural Slicing (Coral) | | | | | | |
|---|---|---|---|---|---|---|
| Number of vertices in slice | Valid-Paths Alg. | | | Horwitz-Reps-Binkley Alg. | | |
| | User time (secs.) | Syst. time (secs.) | $\frac{\text{tot. time}}{\text{vertex}}$ (secs.) | User time (secs.) | Syst. time (secs.) | $\frac{\text{tot. time}}{\text{vertex}}$ (secs.) |
| 1 | .39 | .14 | .53 | .46 | .40 | .86 |
| 99 | 1.13 | .09 | .012 | 2.05 | .24 | .023 |
| 709 | 67.68 | 1.54 | .098 | 12.75 | .64 | .019 |
| 1800 | 307.51 | 14.99 | .18 | 57.46 | 2.33 | .033 |

We were also able to measure the size of the penalty imposed by implementing an algorithm for interprocedural analysis in Coral (rather than in C). We already had a C implementation of the Horwitz-Reps-Binkley algorithm that ran on a Sun 10 [31]. As mentioned in Section 4, the Horwitz-Reps-Binkley algorithm has a phase in which certain auxiliary information is computed by a preliminary exhaustive algorithm. We mimicked this feature by modifying the Coral implementation of the Horwitz-Reps-Binkley algorithm: the program was partitioned into two separate modules and the magic-sets transformation was only applied to one of the two modules; the module that computed the auxiliary

information was left untransformed, and therefore computed that information in an exhaustive fashion.

| Performance of the (Partially Exhaustive) Horwitz-Reps-Binkley Alg. | | | | | | |
|---|---|---|---|---|---|---|
| Number of vertices in slice | Coral Implementation | | | C Implementation | | |
| | User time (secs.) | Syst. time (secs.) | $\frac{\text{tot. time}}{\text{vertex}}$ (secs.) | User time (secs.) | Syst. time (secs.) | $\frac{\text{tot. time}}{\text{vertex}}$ (secs.) |
| 99 | 149.69 | 2.27 | 1.5 | 37.24 | .18 | .378 |
| 709 | 161.75 | 3.25 | .23 | 37.34 | .27 | .053 |
| 1800 | 215.33 | 5.53 | .12 | 37.50 | .27 | .021 |

These performance figures indicate that at this point the penalty imposed by implementing an algorithm for interprocedural analysis in Coral (rather than in C) is substantial. (The above figures are for a partially exhaustive interprocedural analysis algorithm; presumably the penalty for a fully demand algorithm is similar.)

# 6 RELATED WORK

After the work reported in this paper was completed, the work by D.S. Warren and others concerning the use of tabulation techniques in top-down evaluation of logic programs [39] was brought to my attention. These techniques provide an alternative method for obtaining demand algorithms for program-analysis problems. Rather than applying the magic-sets transformation to a Horn-clause encoding of the (exhaustive) dataflow-analysis algorithm and then using a bottom-up evaluator, the original (untransformed) Horn-clause encoding can simply be evaluated by an OLDT (top-down, tabulating) evaluator. Thus, another way to obtain implementations of demand algorithms for the interprocedural dataflow-analysis and interprocedural-slicing problems would be to use the programs from Sections 3 and 4 in conjunction with the SUNY-Stony Brook XSB system [40].

## 6.1 Dataflow Analysis

Previous work on demand-driven dataflow analysis has dealt only with the *intra*procedural case [3, 43]. The work that has been reported in the present paper

complements previous work on the intraprocedural case in the sense that our approach to obtaining algorithms for demand-driven dataflow analysis problems applies equally well to intraprocedural dataflow analysis. However, in intraprocedural dataflow analysis all paths in the control-flow graph are (statically) valid paths; for this reason, previous work on demand-driven *intra*procedural dataflow analysis does not extend well to the *inter*procedural case, where the notion of *valid paths* is important.

A recent paper by Duesterwald, Gupta, and Soffa discusses a very different approach to obtaining demand versions of (intraprocedural) dataflow analysis algorithms [13]. For each query of the form "Is fact $f$ in the solution set at vertex $v$?", a set of dataflow equations are set up on the flow graph (but as if all edges were reversed). The flow functions on the reverse graph are the (approximate) inverses of the original forward functions. (A special function—derived from the query—is used for the reversed flow function of vertex $v$.) These equations are then solved using a demand-driven fixed-point finding procedure to obtain a value for the entry vertex. The answer to the query (true or false) is determined from the value so obtained. Some of the differences between their work and ours are as follows:

- Their method can only answer *ground queries* of the form "`?df_fact(p,n,x)`". With the approach used in this paper *any combination of bound and free arguments* in a query are possible (*e.g.*, "`?df_fact(p,n,X)`", "`?df_fact(p,N,X)`", "`?df_fact(P,N,x)`", *etc.*).
- Their method does not appear to permit information to be accumulated over successive queries. The equations for a given query are tailored to that particular query and are slightly different from the equations for all other queries. Consequently, answers (and intermediate values) previously computed for other queries cannot be reused.
- It is not clear from the extensions they outline for interprocedural dataflow analysis whether the algorithm obtained will properly account for valid paths.

Previous work on *inter*procedural dataflow analysis has dealt only with the exhaustive case [36, 20]. This paper has described how to obtain algorithms for solving demand versions of interprocedural gen-kill dataflow-analysis problems from their exhaustive counterparts.

There has been some previous work in which intraprocedural dataflow-analysis problems have been expressed using Horn clauses. For instance, one of the

examples in Ullman's book shows how a logic database can be used to solve the intraprocedural reaching-definitions problem [38, pp. 984–987]. U. Assmann has examined a variety of other intraprocedural program-analysis problems [2]. Although Assmann expresses these problems using a certain kind of graph grammar, he points out that this formalism is equivalent to Datalog.

## 6.2 Interprocedural Slicing

Hwang, Du, and Chou did not give an analysis of the running time of their interprocedural-slicing algorithm; however, I was able to show that the Hwang-Du-Chou algorithm could, in the worst case, use time exponential in the size of the program [33]. After discovering this, I began to consider strategies that could be used to overcome this drawback, which led to the approach followed in this paper.

The use of the magic-sets technique immediately yielded something new: a true demand algorithm for interprocedural slicing (that was both precise up to valid paths and ran in polynomial-time). The only precise polynomial-time algorithm that was known previously has a phase in which certain auxiliary information is computed by a preliminary exhaustive algorithm. The example of the magic-sets-generated demand algorithm motivated a reexamination of algorithms for interprocedural slicing. This has led to two advances: (1) the development of better exhaustive algorithms for interprocedural slicing, and (2) the development of several other true demand algorithms for interprocedural slicing (including ones that have straightforward implementations in imperative languages, such as C). (See below.)

## 6.3 Recent Developments

Recently, M. Sagiv, S. Horwitz, and the author have shown a new connection between the interprocedural-slicing problem and a large class of interprocedural dataflow-analysis problems [34, 33]. In particular, they have shown that all of these problems can be reduced to the valid-path-reachability problem. Although the Reps-Sagiv-Horwitz paper does not make use of logic-programming terminology, their work can be used to improve the Horn-clause formulations of both slicing and dataflow analysis that were presented in Sections 3 and 4. In the case of interprocedural dataflow analysis, for instance, their approach extends the work described in Section 3 to a much larger class of dataflow problems than the gen/kill problems—the only restrictions are that the set of

dataflow facts be a finite set and that the dataflow functions distribute over the confluence operator (either union or intersection).

Reps, Sagiv, and Horwitz give new algorithms that are asymptotically better than the best previously known algorithms for the valid-path-reachability problem. They also give algorithms for the demand version of the problem. Unlike the demand algorithms of the present paper, the demand algorithms given by Reps, Sagiv, and Horwitz have straightforward implementations in imperative programming languages, such as C. (This addresses a concern that has been raised by several people who are interested in the idea of demand algorithms for program analysis, but are leery of the overheads imposed by using logic databases for this purpose.)

## Acknowledgements

Alan Demers, Fritz Henglein, Susan Horwitz, Neil Jones, Bernard Lang, Raghu Ramakrishnan, Genevieve Rosay, Mooly Sagiv, Marvin Solomon, Divesh Srivastava, and Tim Teitelbaum provided comments and helpful suggestions about the work. Genevieve Rosay furnished the performance figures for the C implementation of the Horwitz-Reps-Binkley algorithm. Raghu Ramakrishnan, Divesh Srivastava, Praveen Seshadri, and S. Sudarshan helped me out with Coral, patiently answered my questions about the system and quickly fixed a few bugs that came to light.

Some of the work reported in the paper was performed while the author was on sabbatical leave at the Datalogisk Institut, University of Copenhagen, Denmark.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant CCR-9100424, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

## REFERENCES

[1] F.E. Allen. Interprocedural data flow analysis. In *Information Processing*

*74: Proceedings of the IFIP Congress 74*, pages 398–408, 1974.

[2] U. Assmann. On edge addition rewrite systems and their relevance to program analysis. Unpublished report, GMD Forschungsstelle Karlsruhe, Karlsruhe, Germany, 1993.

[3] W.A. Babich and M. Jazayeri. The method of attributes for data flow analysis: Part II. Demand analysis. *Acta Informatica*, 10(3):265–272, October 1978.

[4] L. Badger and M. Weiser. Minimizing communication for synchronizing parallel dataflow programs. In *Proceedings of the 1988 International Conference on Parallel Processing*, St. Charles, IL, August 1988.

[5] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, 1986.

[6] J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pages 29–41, San Antonio, TX, January 1979.

[7] J.M. Barth. A practical interprocedural data flow analysis algorithm. *Commun. of the ACM*, 21(9):724–736, September 1978.

[8] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 269–293, San Diego, CA, March 1987.

[9] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 47–56, Atlanta, GA, June 1988. Appeared in *SIGPLAN Notices 23*, 7 (July 1988).

[10] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, SE-16(4):483–487, April 1990.

[11] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 57–66, Atlanta, GA, June 1988. Appeared in *SIGPLAN Notices 23*, 7 (July 1988).

[12] K.D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, TX, January 1989.

[13] E. Duesterwald, R. Gupta, and M.L. Soffa. Demand-driven program analysis. Technical Report TR-93-15, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, October 1993.

[14] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 35–46, Atlanta, GA, June 1988. Appeared in *SIGPLAN Notices 23*, 7 (July 1988).

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[17] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[18] J.C. Hwang, M.W. Du, and C.R. Chou. Finding program slices for recursive procedures. In *Proceedings of IEEE COMPSAC 88*, Chicago, IL, October 1988.

[19] B. Kernighan and P. Plauger. *Software Tools in Pascal.* Addison-Wesley, Reading, Massachusetts, 1981.

[20] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proceedings of the Fourth International Conference on Compiler Construction*, pages 125–140, Paderborn, FRG, October 1992. Appeared as *Lecture Notes in Computer Science, Vol. 641*, U. Kastens and P. Pfahler (eds.), Springer-Verlag, New York, NY, 1992.

[21] A. Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pages 273–284, Charleston, SC, January 1993.

[22] W. Landi and B.G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, January 1991.

[23] M.A. Linton. Implementing relational views of programs. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132–140, Pittsburgh, Pennsylvania, April 1984. Appeared in *SIGPLAN Notices 19*, 5 (May 1984).

[24] J. Lyle and M. Weiser. Experiments on slicing-based debugging tools. In *Proceedings of the First Conference on Empirical Studies of Programming*, June 1986.

[25] L.M. Masinter. Global program analysis in an interactive environment. Technical Report SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, CA, January 1980.

[26] E. Myers. A precise inter-procedural data flow algorithm. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 219–230, Williamsburg, VA, January 1981.

[27] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, Pittsburgh, PA, April 1984. Appeared in *SIGPLAN Notices 19*, 5 (May 1984).

[28] R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan. Coral pre-Release 1.0 Software System. Available via ftp from ftp.cs.wisc.edu, 1993.

[29] R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan. The Coral user manual: A tutorial introduction to Coral. Unpublished report, Computer Sciences Department, University of Wisconsin, Madison, WI, 1993.

[30] R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan. Implementation of the Coral deductive database system. In *Proceedings of the ACM SIGMOD 93 Conference*, pages 167–176, 1993.

[31] T. Reps. The Wisconsin program-integration system reference manual: Release 2.0. Unpublished report, Computer Sciences Department, University of Wisconsin, Madison, WI, July 1993.

[32] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of the Fifth International Conference on Compiler Construction*, pages 389–403, Edinburgh, Scotland, April 1994. Appeared as *Lecture Notes in Computer Science, Vol. 786*, P. Fritzson (ed.), Springer-Verlag, New York, NY, 1994.

[33] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. Technical Report TOPPS D-216, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark, July 1994.

[34] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report TR 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark, April 1994.

[35] R. Rohmer, R. Lescoeur, and J.-M. Kersit. The Alexander method, a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4(3):273–285, 1986.

[36] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[37] F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, July 1994.

[38] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies.* Computer Science Press, Rockville, MD, 1989.

[39] D.S. Warren. Memoing for logic programs. *Commun. of the ACM*, 35(3):93–111, March 1992.

[40] D.S. Warren. XSB Logic Programming System. Available via ftp from sbcs.sunysb.edu, 1993.

[41] M. Weiser. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters*, 17:129–135, October 1983.

[42] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[43] F.K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, pages 132–143, Montreal, Can., June 1984. Appeared in *SIGPLAN Notices 19*, 6 (June 1984).