Solving Demand Versions of Interprocedural Analysis Problems

Thomas Reps[1]
Datalogisk Institut, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East
Denmark

Abstract

This paper concerns the solution of demand versions of interprocedural analysis problems. In a demand version of a program-analysis problem, some piece of summary information (*e.g.*, the dataflow facts holding at a given point) 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 for which (transient) summary information is computed and/or the amount of information computed at those points. The paper describes how algorithms for demand versions of 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.

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

One of the novel aspects of our work is that 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 [22,3,7]).

---

[1]On sabbatical leave from the University of Wisconsin–Madison.

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).

This paper describes how the above approach can be used to obtain a demand algorithm for the interprocedural "gen-kill problems" (*i.e.*, problems in which the dataflow functions are all of the form $\lambda x . (x - kill) \cup gen$).

There are several reasons why it is desirable to solve the demand versions of interprocedural analysis problems (and, in particular, to solve them using the approach presented in this paper).

- *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.) The use of cached information can further reduce the cost of responding to demands when there is a sequence of demands in between program modifications.
- *Reducing the amount of work spent in preprocessing or other auxiliary phases of a program analysis.* Consider a problem such as flow-insensitive side-effect analysis (*e.g.* MayMod, MayUse, *etc.*), which has a decomposition that includes two subsidiary phases: computing alias information and computing side effects due to reference formal parameters [5,11,10]. In problems that are decomposed into separate phases, not all of the information from subsidiary phases is required in order to answer an "outer-level" query. 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 for program analysis 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.
- *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 interactively ask questions about various aspects of a program [19,26,18,13]. 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.

When interprocedural-analysis problems are encoded in Coral [21] (or some other logic-programming language with a bottom-up evaluation strategy), demand algorithms can be obtained totally automatically. In principle, however, the approach described in the paper is not just restricted to interprocedural-analysis problems encoded in logic-programming languages. That is, the techniques can be carried over—as a hand-applied transformation—to program-analysis implementations written in other languages (such as C).

The remainder of the paper is organized as follows: Section 2 discusses background and assumptions. Section 3 summarizes our methodology for obtaining algorithms that solve the demand versions of interprocedural analysis problems. It also presents two examples that illustrate the capabilities of the magic-sets transformation. Section 4 shows how to obtain demand algorithms for the interprocedural gen-kill problems. Section 5 discusses related work. An excerpt from the transformed program that is the result of applying the magic-sets transformation to the program presented in Section 4 is attached as an Appendix.

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 [1,6,23,10].

In interprocedural analysis problems, not all of the paths in the graph that represents 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 [23,20,8,14,17] (see Definition 4.3). 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 dataflow analysis problems discussed in Section 4 have been simplified in certain ways. In particular, following Sharir and Pnueli [23] we assume that (i) all variables are global variables, (ii) procedures are parameterless, (iii) the programs being analyzed do not contain aliasing, and (iv) the programs being analyzed do not use procedure-valued variables. A few words about each is in order: Simplifications (i) and (ii) prevent the Sharir-Pnueli framework 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 [15]. It is possible to generalize the approach described in Section 4 to implement the more general Knoop-Steffen framework. The interaction between interprocedural dataflow analysis and the computation of aliasing information has already been men-

tioned 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. Finally, 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 [16] and Callahan *et al.* [9]. (Because of space limitations, it is not possible to discuss these issues in more detail.)

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 4, we also make use of a notation from Coral for manipulating relations with set-valued fields; this notation will be explained at the place it is first used.

3. Using the Magic-Sets Transformation to Obtain Demand Algorithms

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 a transformation—known as the Alexander method [22] or the magic-sets transformation [3,7]—that was developed in the logic-programming and deductive-database communities for optimizing the evaluation of recursive queries in deductive databases. In principle, the second step is completely automatic; in practice—at least with the Coral system—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.)

We now present two examples that illustrate the capabilities of the magic-sets transformation. (Readers already familiar with the magic-sets transformation should skip to the next section.)

The magic-sets transformation attempts to combine the advantages of a top-down, goal-directed evaluation strategy with those of a bottom-up evaluation strategy. One disadvantage with top-down, goal-directed search (at least the depth-first one employed in Prolog) is that it is incomplete—it may loop endlessly, failing to find any answer at all, even when answers do exist. Another disadvantage of top-down, goal-directed search is that it may take exponential time on examples that a bottom-up evaluation strategy handles in polynomial time.

A bottom-up strategy starts from the base relations and iteratively applies an "immediate-consequence" operator until a fixed point is reached. One advantage of a bottom-up evaluation strategy is that it is complete. It can be thought of as essentially a dynamic-programming strategy: the values for all smaller subproblems are tabulated, then the answer for the item of interest is selected. However, bottom-up evaluation strategies also have the main drawbacks of dynamic programming, namely that (i) much effort may be expended to solve subproblems that are completely irrelevant to the final answer and (ii) a great deal of space may be used storing solutions to such subproblems.

The magic-sets approach is based on bottom-up evaluation; however, the program evaluated is a *transformed* version of the original program, specialized for answering queries of a given form. In the transformed program, each (transformed) rule has attached to it an additional literal that represents a condition characterizing when the rule is relevant to answering queries of the given form. The additional literal narrows the range of applicability of the rule and hence causes it to "fire" less often.

Example. The gains that can be obtained via the magic-sets transformation can be illustrated by the example of answering reachability queries in directed graphs. Let "edge(v, w)" be a given base relation that represents the edges of a directed graph.

A dynamic-programming algorithm for the reachability problem computes the transitive closure of the entire graph—this information answers *all* possible reachability queries—then selects out the edges in the transitively closed graph that emanate from the point of interest. In a logic-programming system that uses a bottom-up evaluation strategy, the dynamic-programming algorithm can be specified by writing the following program for computing transitive closure:

```
tc(V, W) :- tc(V, X),  edge(X, W).
tc(V, W) :- edge(V, W).
```

In the Coral system, which supports the magic-sets transformation, the additional declaration

```
export tc(bf).
```

directs the system to transform the program to a form that is specialized for answering queries in which the first argument is bound and the second is free (*i.e.*, queries of the form "?tc(a, W)"). The transformed program that results is

```
tc_bf(V,W) :- magic_tc_bf(V),  tc_bf(V,X),  edge(X,W).
tc_bf(V,W) :- magic_tc_bf(V),  edge(V,W).
```

Given a query "?tc(a, W)", the additional fact "magic_tc_bf(a)" is adjoined to the above set of transformed rules. These are then evaluated bottom up to produce (as answers to the query) the tuples of the relation tc_bf.

A magic fact, such as "magic_tc_bf(a)", should be read as an assertion that "The problem of finding tuples of the form tc(a, _) arises in answering the query". In this example there are no rules of the form

```
magic_tc_bf(X) :- . . .
```

Consequently, during evaluation no additional facts are ever added to the magic_tc_bf relation; that is, the only "magic fact" ever generated is the initial one, magic_tc_bf(a). (Our next example will illustrate the more general situation.) Because all of the rules in the transformed program are guarded by a literal "magic_tc_bf(V)", the bottom-up evaluation of the transformed program only visits vertices that are reachable from vertex a. In effect, the original "dynamic-programming" algorithm—perform transitive closure on the entire graph, then select out the tuples of interest—has been transformed into a reachability algorithm that searches only vertices reachable from vertex a.

End of Example.

Example. Suppose we have a base relation that records parenthood relationships (*e.g.*, a tuple "parent(x, x1)" means that x1 is a parent of x), and we would like to be able to find all cousins of a given person who are of the same generation. (In this example, a person is considered to be a "same-generation cousin" of himself.)

In a logic-programming system that uses a bottom-up evaluation strategy, a dynamic-programming algorithm can be specified by writing the following program:

```
same_generation(X, X).
same_generation(X, Y) :- parent(X, X1),
                         same_generation(X1, Y1),
                         parent(Y, Y1).
```

The directive

```
export same_generation(bf).
```

directs the Coral system to transform the program to a form specialized for answering queries of the form "?same_generation(a, Y)", which causes the program to be transformed into:

```
magic_same_generation_bf(U) :- magic_same_generation_bf(V),
                               parent(V, U).
same_generation_bf(X, X) :- magic_same_generation_bf(X).
same_generation_bf(X, Y) :- magic_same_generation_bf(X),
                            parent(X, X1),
                            same_generation_bf(X1, Y1),
                            parent(Y, Y1).
```

Given a query "?same_generation(a, Y)", the additional fact "magic_same_generation_bf(a)." is adjoined to the above set of rules, which are then evaluated bottom up.

Unlike the previous example, the transformed program produced in this example *does* have a rule with "magic_same_generation_bf(U)" in the head.

```
magic_same_generation_bf(U) :- magic_same_generation_bf(V),
                               parent(V, U).
```

The presence of this rule will cause "magic facts" other than the original one to be generated during evaluation. Note that the members of relation magic_same_generation_bf will be exactly the ancestors of a (the so-called "cone of a" [3]). During bottom-up evaluation of the transformed rules, the effect of the magic_same_generation_bf predicate is that attention is restricted to just same-generation cousins of ancestors of a.
End of Example.

Note that in a bottom-up evaluation, the transformed program (the demand algorithm) will never perform more work than the untransformed program (the exhaustive algorithm) would—modulo a small amount of overhead for computing magic facts, which are reported to be only a small fraction of the generated facts [4]. In practice, the demand algorithm usually performs far less work than the exhaustive algorithm.

Beeri and Ramakrishnan have shown that the bottom-up evaluation of the magic-sets-transformed version of a logic program is optimal with respect to a given "sideways-information-passing strategy (sip)"—a strategy for deciding how information gained about tuples in some of a rule's literals is to be used in evaluating other literals in the rule. For a given sip, any evaluator that uses the same sip must generate at least as many facts as are generated during a bottom-up evaluation of the magic-sets-transformed version [7].

In our context, this result relates to the question of minimizing the number of program points for which "transient" dataflow-analysis information is computed and/or the amount of information computed at those points when a given demand is placed for dataflow information. Unfortunately, the Beeri-Ramakrishnan result is only a "relative-optimality" result—it only compares top-down and bottom-up evaluations that use the *same* sip. Consequently, the amount of "transient" summary information that a demand program-analysis algorithm computes will depend on the sip that is employed. Throughout the paper, we follow Coral and assume that the sip involves working left-to-right in a rule, exploiting at a given literal all information gained from evaluating the literals to its left. (This is also the same sip that Prolog's top-down evaluator uses.) Thus, the amount of "transient" summary information computed by the demand program-analysis algorithms we obtain depends on the order in which the literals appear in the rules.

In the subsequent sections of the paper, we do not actually discuss the programs that result from the magic-sets transformation—the transformed programs are quite complicated and presenting them would not aid the reader's understanding. (To convince the reader that this is the case, the Appendix presents an excerpt from the transformed program produced by the Coral system from the program discussed in Section 4.)

4. 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 [23].

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 $F \subseteq L \rightarrow L$ and that F contains the identity function.

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

Definition 4.1. (Sharir and Pnueli [23]). Define $G = \cup \{ G_p \mid p$ is a procedure in the program $\}$, where, for each p, $G_p = (N_p, E_p, r_p)$. Vertex r_p is the entry block of p; N_p is the set of basic blocks in p; $E_p = E_p^0 \cup E_p^1$ is the set of edges of G_p. An edge $(m, n) \in E_p^0$ is an ordinary control-flow edge; it represents a direct transfer of control from one block to another via a goto or an if statement. An edge $(m, n) \in E_p^1$ iff m is a call block and n is the return-site block in p for that call. Observe that vertex n is *within* p as well; it is important to understand that 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 block in any G_p graph has exactly one incoming edge: the E_p^1 edge from the corresponding call block; (ii) the entry block r_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 4.2. (Sharir and Pnueli [23]). Define $G^* = (N^*, E^*, r_{main})$, 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 block and n is the entry block of the called procedure; edge $(m, n) \in E^2$ is a return edge iff m is an exit block of some procedure p and n is a return-site block for a call on p. A call edge (m, r_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 potentially valid execution paths:

Definition 4.3. (Sharir and Pnueli [23]). For each $n \in N$, we define IVP(r_{main}, n) as the set of all ***interprocedurally valid paths*** in G^* that lead from r_{main} to n. A path

$q \in \text{path}_{G^*}(r_{main}, n)$ is in $\text{IVP}(r_{main}, n)$ 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:

(i) A sequence q_2 that contains no return edges is proper.

(ii) 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 4.4. (Sharir and Pnueli [23]). If q is a path in G^*, let f_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:

$$\Phi_n = \bigsqcap_{q \in \text{IVP}(r_{main}, n)} f_q \qquad \text{for each } n \in N^*$$
$$y_n = \Phi_n(\bot) \qquad \text{for each } n \in N^*$$

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 $\phi_{(r_p, n)}(x)$ that map a set of dataflow facts at r_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):

$$\phi_{(r_p, r_p)} = \lambda x . x \qquad \text{for each procedure } p$$
$$\phi_{(r_p, n)} = \bigsqcap_{(m, n) \in E_p} (f_{(m, n)} \circ \phi_{(r_p, m)}) \qquad \text{for each } n \in N_p \text{ not representing a return-site block}$$
$$\phi_{(r_p, n)} = \phi_{(r_q, e_q)} \circ \phi_{(r_p, m)} \qquad \text{for each } n \in N_p \text{ representing a return-site block, where } (m, n) \in E_p^1 \text{ and } m \text{ calls procedure } q$$

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:

$$x_{r_{main}} = \bot$$
$$x_{r_p} = \bigsqcap \{ \phi_{(r_q, c)}(x_{r_q}) \mid c \text{ is a call to } p \text{ in procedure } q \} \qquad \text{for each procedure } p$$
$$x_n = \phi_{(r_p, n)}(x_{r_p}) \qquad \text{for each procedure } p \text{ and } n \in (N_p - \{ r_p \})$$

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, $x_n = y_n$) [23].

4.1. 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, we use *nkill* sets instead of *kill* sets; 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:

$$(nkill_2, gen_2) \circ (nkill_1, gen_1) = (nkill_1 \cap nkill_2, (gen_1 \cap nkill_2) \cup gen_2) \quad (\dagger)$$
$$(nkill_2, gen_2) \sqcap (nkill_1, gen_1) = (nkill_1 \cup nkill_2, gen_1 \cup gen_2) \quad (\ddagger)$$

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 e1 edge are in p; an e1 edge 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.

4.2. 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 $\phi_{(r_p, n)}$, 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 $\phi_{(r_p, n)}$.

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 $\phi_{(r_p, n)}$.

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 $\phi_{(r_p, r_p)}$ 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

$$\phi_{(r_p, n)} = \bigsqcap_{(m,n) \in E_p} (f_{(m,n)} \circ \phi_{(r_p, m)}) \qquad \text{for each } n \in N_p \text{ not representing a return-site block}$$

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

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

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 ∈ NK_set to the *nkill* component of $\phi_{(r_P,N)}$ only if X is in the *nkill* component of $\phi_{(r_P,M)}$.

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

Interprocedural Summary Functions

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

$$\phi_{(r_p, n)} = \phi_{(r_q, e_q)} \circ \phi_{(r_p, m)} \qquad \text{for each } n \in N_p \text{ representing a return-site block, where } (m, n) \in E_p^1 \text{ and } m \text{ calls procedure } q$$

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

$$(nkill_{\phi_{(r_q,e_q)}}, gen_{\phi_{(r_q,e_q)}}) \circ (nkill_{\phi_{(r_p,m)}}, gen_{\phi_{(r_p,m)}}) = (\ nkill_{\phi_{(r_p,m)}} \cap nkill_{\phi_{(r_q,e_q)}},\ (gen_{\phi_{(r_p,m)}} \cap nkill_{\phi_{(r_q,e_q)}}) \cup gen_{\phi_{(r_q,e_q)}}\ ).$$

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 $\phi_{(r_P, n)}$ consists of the X's such that X is in both the *nkill* component of $\phi_{(r_P, M)}$ and the *nkill* component of $\phi_{(r_Q, e_Q)}$.

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

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 ϕ 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:

$$< kill_2, ngen_2 > \sqcap < kill_1, ngen_1 > = < (kill_1 \cap ngen_1) \cup (kill_2 \cap ngen_2), ngen_1 \cup ngen_2 >.$$

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

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

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).

End of Remark.

4.3. 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.

4.4. 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*.

5. Related Work

Previous work on demand-driven dataflow analysis has dealt only with the *intra*procedural case [2,27]. 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 [12]. 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 data flow analysis has dealt only with the exhaustive case [23,15]. This paper has described how to obtain algorithms for solving demand versions of interprocedural analysis problems from their exhaustive counterparts, essentially for free. Section 4 describes how to use Horn clauses to specify an algorithm for the interprocedural gen-kill dataflow-analysis problems. Recently, M. Sagiv, S. Horwitz, and the author have devised a way to extend the techniques described in the paper to a much larger class of dataflow problems—in particular, those in which the dataflow functions are drawn from the collection of distributive functions in $2^D \rightarrow 2^D$, where D is any finite set.

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 [24] 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 an implementation of a demand algorithm for the interprocedural gen-kill dataflow-analysis problems would be to use the program from Section 4 in conjunction with the SUNY-Stony Brook XSB system [25].

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.

Appendix: The Demand Interprocedural Dataflow Analysis Algorithm

The following is an excerpt from the transformed program produced by Coral from the interprocedural dataflow-analysis program presented in Section 4:

```
sup_1_1(Q,P,C) :-
   m_df_fact_bbf(P,start_vertex),
   call_site(Q,P,C).
m_df_fact_bbf(Q,start_vertex) :-
   sup_1_1(Q,P,C).
sup_1_2(Q,X,P,C) :-
   sup_1_1(Q,P,C),
   df_fact_bbf(Q,start_vertex,X).
m_phi_nk_bbb(Q,C,X) :-
   sup_1_2(Q,X,P,C).
df_fact_bbf(P,start_vertex,X) :-
   sup_1_2(Q,X,P,C),
   phi_nk_bbb(Q,C,X).
sup_2_1(Q,P,C) :-
   m_df_fact_bbf(P,start_vertex),
   call_site(Q,P,C).
m_phi_g_bbf(Q,C) :-
   sup_2_1(Q,P,C).

... and so on for 128 more lines
```

References

1. Allen, F.E., "Interprocedural data flow analysis," pp. 398-408 in *Information Processing 74: Proceedings of the IFIP Congress 74*, ed. J.L. Rosenfield, North-Holland, Amsterdam (1974).
2. Babich, W.A. and Jazayeri, M., "The method of attributes for data flow analysis: Part II. Demand analysis," *Acta Informatica* **10**(3) pp. 265-272 (October 1978).
3. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., "Magic sets and other strange ways to implement logic programs," in *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, (1986).
4. Bancilhon, F. and Ramakrishnan, R., "Performance evaluation of data intensive logic programs," pp. 439-517 in *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker, Morgan-Kaufmann (1988).
5. Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY (1979).

6. Barth, J.M., "A practical interprocedural data flow analysis algorithm," *Commun. of the ACM* **21**(9) pp. 724-736 (September 1978).
7. Beeri, C. and Ramakrishnan, R., "On the power of magic," pp. 269-293 in *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, (San Diego, CA, March 1987), (1987).
8. Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 47-56 (July 1988).
9. Callahan, D., Carle, A., Hall, M.W., and Kennedy, K., "Constructing the procedure call multigraph," *IEEE Transactions on Software Engineering* **SE-16**(4) pp. 483-487 (April 1990).
10. Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 57-66 (July 1988).
11. Cooper, K.D. and Kennedy, K., "Fast interprocedural alias analysis," pp. 49-59 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
12. Duesterwald, E., Gupta, R., and Soffa, M.L., "Demand-driven program analysis," Technical Report TR-93-15, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA (October 1993).
13. Horwitz, S. and Teitelbaum, T., "Generating editing environments based on relations and attributes," *ACM Trans. Program. Lang. Syst.* **8**(4) pp. 577-608 (October 1986).
14. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
15. Knoop, J. and Steffen, B., "The interprocedural coincidence theorem," pp. 125-140 in *Proceedings of the Fourth International Conference on Compiler Construction*, (Paderborn, FRG, October 5-7, 1992), *Lecture Notes in Computer Science*, Vol. 641, ed. U. Kastens and P. Pfahler, Springer-Verlag, New York, NY (1992).
16. Lakhotia, A., "Constructing call multigraphs using dependence graphs," pp. 273-284 in *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*,(Charleston, SC, Jan. 11-13, 1993), ACM, New York, NY (1993).
17. Landi, W. and Ryder, B.G., "Pointer-induced aliasing: A problem classification," pp. 93-103 in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, (Orlando, FL, January 1991), ACM, New York, NY (1991).
18. Linton, M.A., "Implementing relational views of programs," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 132-140 (May 1984).
19. Masinter, L.M., "Global program analysis in an interactive environment," Tech. Rep. SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, CA (January 1980).
20. Myers, E., "A precise inter-procedural data flow algorithm," pp. 219-230 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).
21. Ramakrishnan, R., Seshadri, P., Srivastava, D., and Sudarshan, S., "Coral pre-Release 1.0," Software system, Computer Sciences Department, University of Wisconsin, Madison, WI (1993). (Available via ftp from ftp.cs.wisc.edu.)
22. Rohmer, R., Lescoeur, R., and Kersit, J.-M., "The Alexander method, a technique for the processing of recursive axioms in deductive databases," *New Generation Computing* **4**(3) pp. 273-285 (1986).
23. Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones, Prentice-Hall, Englewood Cliffs, NJ (1981).
24. Warren, D.S., "Memoing for logic programs," *Commun. of the ACM* **35**(3) pp. 93-111 (March 1992).
25. Warren, D.S., "XSB Logic Programming System," Software system, Computer Science Department, State University of New York, Stony Brook, NY (1993). (Available via ftp from sbcs.sunysb.edu.)
26. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).
27. Zadeck, F.K., "Incremental data flow analysis in a structured program editor," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, (Montreal, Can., June 20-22, 1984), *ACM SIGPLAN Notices* **19**(6) pp. 132-143 (June 1984).