Demand-driven Computation of Interprocedural Data Flow *

Evelyn Duesterwald Rajiv Gupta Mary Lou Soffa

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{duester,gupta,soffa}@cs.pitt.edu

Abstract

This paper presents a general framework for deriving demand-driven algorithms for interprocedural data flow analysis of imperative programs. The goal of demand-driven analysis is to reduce the time and/or space overhead of conventional exhaustive analysis by avoiding the collection of information that is not needed. In our framework, a demand for data flow information is modeled as a set of data flow queries. The derived demand-driven algorithms find responses to these queries through a partial reversal of the respective data flow analysis. Depending on whether minimizing time or space is of primary concern, result caching may be incorporated in the derived algorithm. Our framework is applicable to interprocedural data flow problems with a finite domain set. If the problem's flow functions are distributive, the derived demand algorithms provide as precise information as the corresponding exhaustive analysis. For problems with monotone but non-distributive flow functions the provided data flow solutions are only approximate. We demonstrate our approach using the example of interprocedural copy constant propagation.

1 Introduction

Phrased in the traditional data flow framework [KU77], the solution to a data flow problem is expressed as the fixed point of a system of equations. Each equation expresses the solution at one program point in terms of the solution at immediately preceding (or succeeding) points. This formulation results in an inherently exhaustive solution; that is, to find the solution at one program point, the solution at all points must be computed.

This paper presents an alternative approach to program analysis that avoids the costly computation of exhaustive solutions through the demand-driven retrieval of data flow information. We describe a general framework for deriving demand-driven algorithms that is aimed at reducing the time and/or space consumption of conventional exhaustive analyzers.

Demand-driven analysis reduces the analysis cost by preventing the *over-analysis* of a program that occurs if parts of the analysis effort are spent on the collection of superfluous information. Optimizing and parallelizing compilers that exhaustively analyze a program with respect to each data flow problem of interest are likely to over-analyze the program. Typically, code transformations are applied only selectively over the program and therefore require only a subset of the exhaustive data flow solution. For example, some optimizations are applicable to only certain structures in a program, such as loop optimizations. Even if optimizations are applicable everywhere in the program, one may want to reduce the overall optimization overhead by restricting their application to only the most frequently executed regions of the program (e.g., frequently called procedures or inner loops).

One strategy for reducing the analysis cost in these applications is to simply limit the exhaustive analysis to only selected code regions. However, this strategy may prevent the application of otherwise safe optimizations due to the worst case assumptions that would have to be made at the entry and exit points of a selected code region. For example, data flow information that enters a selected code region from outside the region is vital in determining the side effects of procedure calls contained in that region. Similarly, data flow from outside a loop may be needed to simplify and/or determine the loop bounds or array subscripts in the loop. These applications favor a demand-driven approach that allows the reduction of the analysis cost while still providing all necessary data flow information.

Another advantage of demand-driven analysis is its suitability for servicing on-line data flow requests in software tools. Interactive software tools that aid in debugging and understanding of complex code require information to be gathered about various aspects of a program. Typically, the information requested by a user is not exhaustive but selective, i.e., data flow for only a selected area of the program is needed. Moreover, the data flow problems to be solved are not fixed before the software tool executes but can vary depending on the user's requests. For example, during debugging a user may want to know where a certain value is defined in the program, as well as other data flow information that would help locate bugs. A demand-driven analysis approach naturally provides the capabilities to service requests whose nature and extent may vary depending on the user and the program.

The utility of demand-driven analysis has previously been

*Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh

To appear in: Symposium on Principles of Programming Languages (POPL'95), January 1995, San Francisco, CA.

demonstrated for a number of specific analysis problems [CCF92, CHK92, CG93, SY93, SMHY93, Mas94]. Unlike these applications, the objective of our approach is to address demand-based analysis in a general way. We present a lattice based framework for the derivation of demand-driven algorithms for interprocedural data flow analysis. In this framework, a demand for a specific subset of the exhaustive solution is formulated as a set of *queries*. Queries may be generated automatically (e.g., by the compiler) or manually by the user (e.g., in a software tool). A query

$$q =< y, n >$$

raises the question as to whether a specific set of facts y is part of the exhaustive solution at program point n. A response (*true* or *false*) to the query q is determined by propagating q from point n in reverse direction of the original analysis until all points have been encountered that contribute to the response for q. This query propagation is modeled as a partial reversal of the original data flow analysis. Specifically, by reversing the information flow associated with program points, we derive a system of query propagation rules. The response to a query is found after a finite number of applications of these rules. We present a generic demand algorithm that implements the query propagation and discuss two optimizations of the algorithm: (i) early-termination to reduce the response time for a single query and (ii) result caching to optimize the performance over a sequence of queries. In the worst case, in which the amount of information demanded is equal to the exhaustive solution, the asymptotic complexity of the demand algorithm is no worse than the complexity of a standard iterative exhaustive algorithm.

The derivation of demand algorithms is based on a conventional exhaustive interprocedural analysis framework. Several formal frameworks for (exhaustive) interprocedural analysis have been described [CC77, Ros79, JM82, SP81, KS92]. We use the framework by Sharir and Pnueli [SP81] as the basis for our approach. We first follow the assumptions of the Sharir-Pnueli framework and consider programs with parameterless (recursive) procedures and with a single global address space. We then consider extensions to our framework to allow non-procedure valued reference parameters and local variables. These extension are discussed for the example of demand-driven copy constant propagation.

Our approach is applicable to monotone interprocedural data flow problems with a finite domain set (finite set of facts) and yields precise data flow solutions if all flow function are distributive. This finiteness restriction does not apply if the program under analysis consists of only a single procedure (the *intra*procedural case). The distributivity of the flow functions is needed to ensure that the derived demand algorithms are as precise as their exhaustive counterparts. Conceptually, our approach may also be applied on problems with monotone but non-distributive flow functions at the cost of reduced precision. We discuss the loss of information that is caused by non-distributive flow functions and show how our derived demand algorithms can still be used to provide approximate but safe query responses for non-distributive problems.

The class of distributive and finite data flow problems that can be handled precisely includes, among others, the interprocedural versions of the classical bitvector problems, such as live variables and available expressions, as well as common interprocedural problems, such as procedure side-effect analysis [CK88]. We have chosen the example of interprocedural copy constant propagation for illustrating the demand-driven framework in this paper.

Section 2 reviews Sharir and Pnueli's interprocedural framework. In Section 3 we derive a system of query propagation rules from which we establish a generic demand algorithm. We discuss optimizations of the generic algorithm which include early termination and result caching in Section 4. Section 5 demonstrates the demand algorithm using the example of interprocedural copy constant propagation and presents the extensions to include reference parameters and local variables. We discuss related work in Section 6 and conclusions are given in Section 7.

2 Background

A program consisting of a set of (recursive) procedures is represented as an *interprocedural flow graph (IFG)* $G = \{G_1, \ldots, G_k\}$ where $G_p = (N_p, E_p)$ is a directed flow graph representing procedure p. Nodes in N_p represent the statements in p and edges in E_p represent the transfer of control among statements. Two distinguished nodes r_p and e_p represent the unique entry and exit nodes of p. The set $E = \cup\{E_i | 1 \leq i \leq k\}$ denotes the set of all edges in G, $N = \cup\{N_i | 1 \leq i \leq k\}$ denotes the set of all nodes in G and $pred(n) = \{m | (m, n) \in E_i\}$ and $succ(n) = \{m | (n, m) \in E_i\}$ denote the set of immediate predecessors and successors of node n, respectively. We assume that $|E| = O(|N|)$. Finally, $N_{call} \subseteq N$ denotes set of nodes representing procedure calls (call sites) and for each node $n \in N_{call}$, $call(n)$ denotes the procedure called from n. An example of an IFG is shown in Figure 1.

During the analysis, only *valid* interprocedural execution paths should be considered. An execution path π is valid if π returns after a procedure exit node e_p to the matching call site that most recently occurred in π prior to e_p. For a node n in an interprocedural flow graph G, $IP(r_{main}, n)$ denotes the set of valid execution paths from the program entry node r_{main} to node n. For example in Figure 1, the path $1, 2, 3, 4, 6, 7, 10, 11, 4, 5$ is a valid execution path, while the path $1, 2, 3, 4, 6, 7, 10, 11, 9, 11, 4, 5$ is not valid. Note that interprocedural execution paths are not directly represented in an IFG since it does not contain explicit edges between call and procedure entry nodes or between procedure exit nodes and return nodes.

Throughout this paper we assume that G is an interprocedural flow graph representing some program P.

2.1 Interprocedural Analysis Framework

Data flow problems are solved by globally gathering information from a domain set of program facts. This information gathering process is modeled by a pair (L, F), where:

- L is a complete lattice with a partial order $\sqsubseteq$, a least element $\bot$ (bottom), a greatest element $\top$ (top), and a meet operator $\sqcap$ (greatest lower bound) and a dual join operator $\sqcup$ (least upper bound). L has the decreasing chain property (i.e., any decreasing chain $x_1 \sqsupseteq x_2 \sqsupseteq \ldots$ is finite). The bottom element $\bot$ denotes "null information" and we assume that the top element $\top$ denotes "undefined information" [1].

[1] If L does not already contain a top element with the meaning "undefined information", it can always be extended to include an additional new top element.

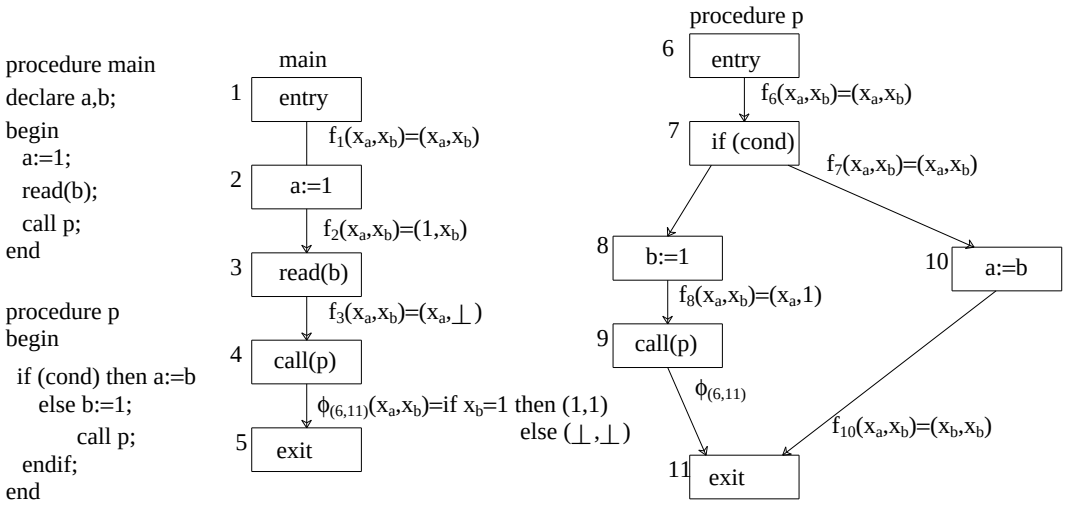


Figure 1: An IFG with the local flow functions for copy constant propagation.

- $F \subseteq \{f : L \mapsto L\}$ is a set of monotone functions over L (i.e, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$).

A *local flow function* $f_n \in F$ is mapped to each node $n \in N - N_{call}$ to model the local data flow effect of executing node n. A data flow analysis is called $\sqcap$-*distributive* if all local flow functions are $\sqcap$-distributive (i.e, $f(x \sqcap y) = f(x) \sqcap f(y)$).

The exhaustive solution to an (interprocedural) data flow problem is the *meet-over-all-valid-paths* solution $mop : N \mapsto L$:

$$mop(n) = \bigsqcap_{p=n_1 \ldots n_k \in IP(r_{main},n)} f_{n_{k-1}} \cdot \ldots \cdot f_{n_1}(\bot)$$

If the analysis is $\sqcap$-distributive then the mop solution can be computed as the greatest fixed point of a system of data flow equations [SP81]. Data flow equations are evaluated in a two-phase approach. During the first phase the data flow effect of each procedure is analyzed independent of its calling context. The results of this phase are procedure *summary functions* as defined in equation system 1. The summary function $\phi_{(r_p,e_p)} : L \mapsto L$ for procedure p maps data flow facts from the entry node r_p to the corresponding set of facts that holds upon procedure exit.

Equation System 1

$$\phi_{(r_p,r_p)}(x) = x$$

$$\phi_{(r_p,n)}(x) = \bigsqcap_{m \in pred(n)} \begin{cases} f_m \cdot \phi_{(r_p,m)}(x) & \text{if } m \notin N_{call} \\ \phi_{(r_q,e_q)} \cdot \phi_{(r_p,m)}(x) & \text{if } m \in N_{call}, \\ & call(m)=q \end{cases}$$

The actual calling context of called procedures is propagated during the second phase based on the summary functions. The solution to the data flow problem is defined as the greatest fixed point of equation system 2, where $x(n)$ describes the data flow solution on entry of node n.

For finite lattices, Sharir and Pnueli propose an iterative worklist-driven tabulation algorithm to solve the equation systems 1 and 2. Their algorithm requires $O(|L| \times |N|)$ space to tabulate equations from 1 and 2 and the time for $O(C \times height(L) \times |L| \times |N|)$ meet operations and/or applications of local flow functions, where C is the maximal number of call sites calling a single procedure and $height(L)$ is the height of lattice L (i.e., the length of the longest chain in L).

Equation System 2

$$x(r_{main}) = \bot$$

For each procedure p:

$$x(r_p) = \bigsqcap_{m \in N_{call} \text{ and } call(m)=p} x(m)$$

For each procedure p and node $n \neq r_p$*:*

$$x(n) = \bigsqcap_{m \in pred(n)} \begin{cases} f_m(x(m)) & \text{if } m \notin N_{call} \\ \phi_{(r_q,e_q)}(x(m)) & \text{if } m \in N_{call}, \\ & call(m)=q \end{cases}$$

2.2 Example: Copy Constant Propagation

We illustrate our approach throughout this paper using the example of copy constant propagation (CCP). CCP is a distributive version of the (non-distributive) constant propagation analysis with expression evaluation [Kil73]. A variable is a *copy constant* if it is either assigned a constant value or it is assigned a copy of another variable that is a copy constant. Since no expressions are evaluated, CCP is less expensive but may discover fewer constants than constant propagation with expression evaluation. Recent studies on interprocedural constant propagation [GT93] indicate that the discovery of constants based on copies may be as effective in practice for the interprocedural propagation as the more costly discovery of constants based on symbolic evaluation.

A demand-driven algorithm for interprocedural copy constant propagation limits the analysis effort to the discovery of the *interesting* interprocedural constants. For example in parallelization, the interesting interprocedural constants in-

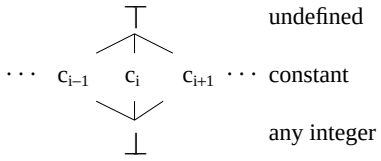


Figure 2: The component lattice for copy constant propagation for a single program variable.

clude the values of variables or formal parameters that occur in array subscript expressions or in loop bounds.

Determining that a variable v is a copy constant requires the simultaneous analysis of all programs variables. CCP is not a *partitionable* [Zad84] analysis that would permit the separate analysis of each variable as, for example, is possible in live variable analysis. The CCP lattice for a program with k variables is the product L^k, where the component lattice L is defined as shown in Figure 2. Thus, each lattice element $x \in L^k$ is a k-tuple with one component $(x)_v \in L$ for each variable v. The meet operator $\sqcap$ and the dual join operator $\sqcup$ are defined pointwise according to the partial order depicted in Figure 2. Of particular interest are the *base elements* in L^k. A base element is a tuple x indicating that a single variable v has some constant value c and that all other variables are not constant: $(x)_v = c$ and $(x)_w = \bot$ for $w \neq v$. We use the simplified notation $[v{=}c] \in L^k$ for such a base element x. For readability, we also write $[v{=}c, w{=}c]$ for the join of base elements:$[v{=}c] \sqcup [w{=}c]$. Furthermore, we also use $\top$ and $\bot$ for the top and bottom element of the product lattice L^k.

We define the distributive flow functions f in CCP pointwise for each component corresponding to one of k variables, i.e., $f(x_1, \ldots, x_k) = (f(x_1, \ldots, x_k)_1, \ldots, f(x_1, \ldots, x_k)_k)$. The component $f(x_1, \ldots, x_k)_w$ of a local flow function f with respect to variable w is defined in Table 1 for various types of assignments. The local flow function for a conditional expression is simply the identity function.

node n	local flow function component $f_n(x)_w$, where $x = (x_1, \ldots, x_k)$
$v := c$	$f_n(x)_w = \begin{cases} c & \text{if } v = w \\ x_w & \text{otherwise} \end{cases}$
$v := u$	$f_n(x)_w = \begin{cases} x_u & \text{if } v = w \\ x_w & \text{otherwise} \end{cases}$
$v := t$ or $read(v)$	$f_n(x)_w = \begin{cases} \bot & \text{if } v = w \\ x_w & \text{otherwise} \end{cases}$

Table 1: The local flow functions in CCP, where u, v and w are variables, c is a constant and t is an expression.

In Figure 1, the local flow function associated with each node is shown below the node. Each lattice element is a pair (x_a, x_b), where the first and second component denote lattice values for variables a and b, respectively. The execution effect of procedure calls (nodes 4 and 9) is modeled by the summary function $\phi_{(6,11)}$ as defined in equation system 1. The full definition of $\phi_{(6,11)}$ is shown only at node 4.

3 Propagating Data Flow Queries

A data flow query q raises the question as to whether a specific set of facts $y \in L$ is a *safe approximation* of the exhaustive solution at a selected program node n. A lattice element y is a safe approximation of the solution $x(n)$ if y is lower in the lattice than $x(n)$.

Definition 1 (Data flow query) *Let $y \in L$ and $n \in N$. A data flow query q is of the form $q =< y, n >$ and denotes the truth value of the term: $y \sqsubseteq x(n)$.*

For the program in Figure 1, consider the question as to whether variable a is a copy constant after returning from the call to procedure p from *main*, i.e., on entry of node 5. The least lattice element that expresses that a has some arbitrary but fixed constant value c is the element $(a{=}c,\ b{=}\bot) = [a{=}c]$ (i.e., variable b may assume any value). Thus, the question corresponds to the query $q =<[a{=}c], 5>$.

We now consider the problem of determining the answer (*true* or *false*) for a query q without exhaustively evaluating the equation systems 1 and 2. Informally, the answer to $q{=}<y, n>$ is obtained by propagating q from node n in reverse direction of the original analysis until all nodes have been encountered that contribute to the answer for q. We model this propagation process as a *partial reversal* of the original data flow analysis.

To illustrate the reversal of the analysis we examine the following cases in the propagation of $q =< y, n >$.

Case (i) (Node $n = r_{main}$): no further propagation of q is possible and q evaluates to *true* if and only if $y = \bot$.

Case (ii) (Node $n = r_p$ for some procedure p): q raises the question as to whether y holds on entry of every invocation of p. It follows that q can be translated into the boolean conjunction of queries $< y, m >$ for every call site m calling procedure p.

Case (iii) (Node n is some arbitrary non-entry node): For simplicity, assume for now that n has a single predecessor m. Equation system 2 shows that $y \sqsubseteq x(n)$ if and only if $y \sqsubseteq h(x(m))$, where h is either a local flow function or a summary function. By the monotonicity of h, the term $y \sqsubseteq h(x(m))$ directly evaluates to *true* if $y \sqsubseteq h(\bot)$ and to *false* if $y \not\sqsubseteq h(\top)$. Otherwise q translates into a new query $q' =< z, m >$ for node m. The lattice element z to be queried on entry of node m should be the least element (i.e., smallest set of facts) such that $z \sqsubseteq x(m)$ implies $y \sqsubseteq h(x(m))$. To find the appropriate query element z for the new query q' we apply the *reverse function* h^r [HL92].

Definition 2 (Reverse function) *Given a complete lattice L and a monotone function $h : L \mapsto L$, the reverse function $h^r : L \mapsto L$ is defined as:*

$$h^r(y) = \bigsqcap \{x \in L : \ y \sqsubseteq h(x)\}$$

The reverse function h^r maps y to the smallest element x such that $y \sqsubseteq h(x)$. Note, that if no such element exists $h^r(y) = \top$ (undefined).

If the function h is $\sqcap$-distributive then the following relationship holds between function h and its reverse h^r [Cou81, HL92]:

$$y \sqsubseteq h(x) \iff h^r(y) \sqsubseteq x \qquad \text{(GC)}$$

The above relationship uniquely determines the reverse function and defines a *Galois connection* [Bir84] between h and its reverse h^r. Note, that the $\sqcap$-distributivity is necessary for establishing this relationship. For the remainder of this section we consider only distributive flow function and show how the resulting relationship between flow functions and

their reverse functions can be exploited during the query propagation.

First consider the following properties of the function reversal. It can easily be shown that the $\sqcap$-distributivity of h implies the $\sqcup$-distributivity of the reverse function h^r: $h^r(x \sqcup y) = h^r(x) \sqcup h^r(y)$. Lemma 1 states relevant properties with respect to the composition, the meet and the join of functions.

Lemma 1 *Let g and h be two $\sqcap$-distributive functions.*
(i) $(g \cdot h)^r = h^r \cdot g^r$
(ii) $(g \sqcap h)^r = g^r \sqcup h^r$

Proof: straightforward and omitted for brevity (see also [HL92]).

Table 2 shows the definition of the reverse flow functions in CCP. For all flow functions $f_n^r(\top) = \top$ and $f_n^r(\bot) = \bot$. By the $\sqcup$-distributivity of the reverse functions, it is sufficient to define f_n^r for the base elements in the lattice L^k.

node n	**reverse flow function $f_n^r([w{=}c])$**
$v{:=}c'$	$f_n^r([w{=}c]) \begin{cases} \bot & \textit{if } w{=}v,\ c{=}c' \\ \top & \textit{if } w{=}v,\ c{\neq}c' \\ [w{=}c] & \textit{otherwise} \end{cases}$
$v{:=}u$	$f_n^r([w{=}c]) \begin{cases} [u{=}c] & \textit{if } w{=}v \\ [w{=}c] & \textit{otherwise} \end{cases}$
v:=t, read(v)	$f_n^r([w{=}c]) \begin{cases} \top & \textit{if } w{=}v \\ [w{=}c] & \textit{otherwise} \end{cases}$

Table 2: The reverse local flow functions in CCP.

The reverse function value $f_n^r([w{=}c])$ denotes the least lattice element, if any exists, that must hold on entry of node n in order for variable w to have the constant value c on exit of n. If $f_n^r([v{=}c]) = \bot$, the trivial value $\bot$ is sufficient on entry of node n (i.e., v always has value c on exit). The value $f_n^r([v{=}c]) = \top$ indicates that there exists no entry value which would cause v to have the value c on exit.

The following propagation rules result immediately from the definition of the exhaustive equation system 2 in a $\sqcap$-distributive data flow framework and the properties of the reverse functions. The operator $\wedge$ denotes the boolean AND operator.

Query Propagation (distributive functions)

$$< y, r_{main} > \iff \begin{cases} \textit{true} & \text{if } y = \bot \\ \textit{false} & \text{otherwise} \end{cases}$$

$$< y, r_p > \iff \bigwedge_{m \in N_{call},\ call(m)=p} < y, m >$$

$$< y, n > \iff \bigwedge_{m \in pred(n)} \begin{cases} \textit{false} & \text{if } {h_m}^r(y) = \top \\ \textit{true} & \text{if } {h_m}^r(y) = \bot \\ < {h_m}^r(y), m > & \text{otherwise} \end{cases}$$

$$\text{where } h_m = \begin{cases} f_m & \text{if } m \notin N_{call} \\ \phi_{(r_p, e_p)} & \text{if } m \in N_{call},\ call(m) = p \end{cases}$$

The propagation rules require the application of reverse functions. If node m is not a call site the reverse function f_m^r can be determined by locally inspecting the (distributive) flow function f_m. Otherwise, if node m calls a procedure p we need to determine the reverse summary function $\phi^r_{(r_p, e_p)}$. We first assume in the next section that all necessary reverse summary functions are available and then discuss their determination in detail in the following section.

3.1 Query Algorithm

The demand algorithm *Query* that implements the query propagation rules is shown in Figure 3. *Query* takes as input a query q and returns the answer for q after a finite number of applications of the propagation rules. *Query* uses a worklist initialized with the input query q. The answer to q is equivalent to the boolean conjunction of the answers to the queries currently in the worklist. During each step a query is removed from the worklist and translated according to the appropriate propagation rule associated with the node under inspection. The query resulting from this translation is merged with the previous query at the node and added to the worklist unless the previous query has not changed (lines 7-8 and 15-16). The algorithm terminates with the answer *true* if the worklist is exhausted and all queries have evaluated to *true*. Otherwise, *false* is returned.

```
Query(y, n)
1.  for each m ∈ N do query[m] ← ⊥
2.  query[n] ← y; worklist← {n};
3.  while worklist ≠ ∅ do
4.     remove a node m from worklist;
5.     case m = r_q for some procedure q:
6.        for each m' ∈ N_call s.t. call(m') = q do
7.           query[m'] ← query[m'] ⊔ query[m];
8.           if query[m'] changed then add m' to worklist;
9.        endfor;
10.    otherwise:
11.       for each m' ∈ pred(m) do
12.          new ← { f^r_{m'}(query[m])          if m' ∉ N_call
                   { φ^r_{(r_q,e_q)}(query[m])   if m'∈N_call, call(m')=q
13.          if (new = ⊤) then return( false )
14.          else if (new ⊐ ⊥) then
15.             query[m'] ← query[m'] ⊔ new;
16.             if query[m'] changed then add m' to worklist;
17.          endif;
18.       endfor;
19. endwhile;
20. return(true);
```

Figure 3: Generic demand algorithm *Query*. *Query*(y, n) returns the response *true* or *false* to the query $q =< y, n >$.

To determine the complexity of the query algorithm we count the number of times a join operation or a reverse function application is performed. A join/reverse function application is performed at a node n in lines 7, 12 and 15 only if the query at a successor of n was changed (or at the entry node of a procedure p if n is a call site of p) which can happen only $O(height(L))$ times. Hence, algorithm *Query* requires in the worst case $O(height(L) \times |N|)$ join operations and/or reverse function applications.

If the program under analysis consists of only a single procedure (the *intra*procedural case), *Query* provides a complete procedure for demand-driven data flow analysis. For the *inter*procedural case, we require an efficient procedure to compute the reverse summary functions, as discussed next.

```
Computeφ^r(p, y)
1.  if M[e_p, y] = y then /* result previously computed */
2.     return(M[r_p, y]);
3   worklist ← {(e_p, y)}; M[e_p, y] = y;
4.  while worklist ≠ ∅ do
5.     remove a pair (n, x) from worklist and let z ← M[n, x];
6.     case n ∈ N_call and call(n) = q:
7.        if M[e_q, z] = z then
8.           for each m ∈ pred(n) do
9.              Propagate(m, x, M[r_q, z]);
10.       else /* trigger computation of φ^r_(r_q,e_q)(z) */
11.          M[e_q, z] ← z and add (e_q, z) to worklist;
12.    case n = r_q for some proc. q:
                 /* Propagate z to call sites if needed */
13.       for each m ∈ N_call such that
                    call(m) = q and M[m, x'] = x for some x' do
14.          for each m' ∈ pred(m) do Propagate(m', x', z);
15.    otherwise:
             /* n is not a call site and not an entry node */
16.          for each m ∈ pred(n) do Propagate(m, x, f^r_n(z));
17. endwhile;
18. return(M[r_p, y]);

Propagate(n, y, new) /* propagate new to M[n, y] */
1.  M[n, y] ← M[n, y] ⊔ new;
2.  if M[n, y] changed then add (n, y) to worklist; endif;
```

Figure 4: $Compute\phi^r(p, y)$ returns the reverse summary function value $\phi^r_{(r_p,e_p)}(y)$ for a procedure p and $y \in L$.

3.2 Reverse Summary Functions

This section discusses an algorithm to compute individual reverse summary function values in order to extend algorithm *Query* to the interprocedural case. An obvious and inefficient way to compute reverse summary functions is to first determine all original summary functions by evaluating equation system 1 and then reverse each function. We describe in this section a more efficient method to directly compute the reverse functions. Our algorithm mirrors the operations performed in Sharir and Pnueli's worklist algorithm for evaluating equation system 1, except that we compute reverse summary function values and the direction in which table entries are computed is reversed. Assuming that the asymptotic cost of meet and local flow function application is the same for join and reverse flow function application, our algorithm has the same worst case complexity as Sharir and Pnueli's algorithm for the original summary functions. As in Sharir and Pnueli's algorithm the tabulation strategy requires the lattice L to be finite.

We first derive an inductive definition of the reverse summary functions from equation system 1. By reversing the order in which summary function are constructed and by applying Lemma 1 we obtain the following definition of the reverse summary function $\phi^r_{(r_p,e_p)}$ for each procedure p:

Equation System 3

$$\phi^r_{(e_p,e_p)}(y) = y$$

$$\phi^r_{(n,e_p)}(y) = \bigsqcup_{m \in succ(n)} \begin{cases} f^r_m \cdot \phi^r_{(m,e_p)}(y) & \text{if } m \notin N_{call} \\ \phi^r_{(r_q,e_q)} \cdot \phi^r_{(m,e_p)}(y) & \text{if } m \in N_{call},\ call(m)=q \end{cases}$$

Figure 4 shows an iterative worklist algorithm $Compute\phi^r$ that, if invoked with a pair (p, y), returns the value $\phi^r_{(r_p,e_p)}(y)$ after a partial evaluation of the equation system 3. Individual function values are stored in a table $M : N \times L \mapsto L$ such that $M[n, y] = \phi^r_{(n,e_p)}(y)$. The table is initialized with $\bot$ and its contents are assumed to persist between subsequent calls to procedure $Compute\phi^r$. Thus, results of previous calls are reused and the table is incrementally computed during a sequence of calls. After calling $Compute\phi^r$ with a pair (p, y) a worklist is initialized with the pair (e_p, y). The contents of the worklist indicate the table entries whose values have changed but the new values have not yet been propagated. During each step a pair is removed from the worklist, its new value is determined and all other entries whose values might have changed as a result are added to the worklist.

We next analyze the cost of k calls to $Compute\phi^r$. Storing the table M requires space for $|N| \times |L|$ lattice elements. To analyze the time complexity we count the number of join operations (in procedure *Propagate*) and reverse flow function applications (at the call to *Propagate* in line 16). The loop in lines 4-17 is executed $O(height(L) \times |L| \times |N|)$ times, which is the maximal number of times the lattice value of a table entry can be raised, i.e., the maximal number of additions to the worklist. In the worst case, the currently inspected node n is a procedure entry node. Processing a procedure entry node r_q results in calls to *Propagate* for each predecessor of a call site of procedure q. Thus, the k calls to $Compute\phi^r$ require in the worst case $O(C \times height(L) \times |L| \times |N|)$ join and/or reverse function applications, where C is the maximal number of call sites calling a single procedure.

Assuming that each access to a reverse summary function in procedure *Query* is replaced by an appropriate call to $Compute\phi^r$, the total cost of algorithm *Query* is $O(C \times height(L) \times |L| \times |N|)$ join and reverse local flow function applications plus the space to store $O(|N| \times |L|)$ lattice elements.

3.3 Queries in non-distributive Frameworks

The distributivity of the original analysis framework is necessary to ensure that queries are decidable, that is, to ensure that the query propagation rules from the previous section yield as precise information as the original exhaustive analysis does. We consider in this section the kind of approximation that is obtainable if the query propagation rules are applied to evaluate queries in the presence of non-distributive flow functions.

If all flow functions in a data flow framework are $\sqcap$-distributive then a data flow query $< y, n >$ evaluates to *true* if and only if element y is part of the solution at node n, i.e., if and only if $y \sqsubseteq x[n]$. If the original analysis framework is monotone but not $\sqcap$-distributive then information may be lost during the query propagating process. Specifically, if a flow function h is monotone but not distributive, then the relation between h and its reverse h^r is weaker than in the distributive case; only the following implication holds (see in contrast the Galois connection (GC)):

$$y \sqsubseteq h(x) \implies h^r(y) \sqsubseteq x$$

As a result of this weaker relationship queries are only semi-decidable in the presence of non-distributive flow functions. Conceptually, the derived query algorithm may also be applied to data flow problems with non-distributive functions. In the presence of non-distributivity, the above implication ensures that if a query $q =< y, n >$ evaluates to *false* then $y \not\sqsubseteq x[n]$. However, nothing can be said if q evaluates to *true*. If appropriate worst assumptions are made for *true*

responses, the query algorithm still provides approximate information in the presence of non-distributive flow functions.

4 Optimized Query Evaluation

This section discusses two ways to improve the performance of the query evaluation. The choice of optimization depends on whether a fast response to (i) a single query or to (ii) a sequence of queries is of primary interest.

To optimize the response time for a single query, the query evaluation includes *early termination*. Recall that the answer to the input query is the boolean conjunction of the answers to the queries currently in the worklist. Thus, the evaluation can directly terminate as soon as one query evaluates to *false*, independent of the remaining contents of the worklist. Early termination is included in algorithm *Query* in Figure 3.

To process a sequence of k queries requires k invocations of *Query*, which may result in the repeated evaluation of the same intermediate queries. Repeated query evaluation can be avoided by maintaining a *result cache*. We outline a simple extension of algorithm *Query* to include result caching. A global cache is maintained that contains for each node n and lattice element y an entry $cache[n, y]$ denoting the previous result, if any, of evaluating the query $< y, n >$. Before a newly generated query q is added to the worklist, the cache is first consulted. The query q is added to the worklist only if the answer for q is not found in the cache. Entries are added to the cache after each terminated query evaluation. Recall that a *false* answer at some node n implies a *false* answer for all previously generated queries at nodes that are reachable from n along some valid execution path. Thus, the cache can be updated in a single pass over the nodes that have been visited during the current invocation of *Query*. First, each visited node n that is reachable from a node that evaluated to *false* is processed and the cache is updated to $cache[n, query[n]] = false$. Note, that at this point the entries $cache[n, z]$ for $query[n] \sqsubseteq z$ could also be set to *false*. For the remaining visited nodes n, where $< query[n], n >$ evaluated to *true*, the cache is updated to $cache[n, query[n]] = true$. Again, the entries $cache[n, z]$ for $z \sqsubseteq query[n]$ could also be set to *true*.

The inclusion of result caching has the effect of incrementally building the complete (exhaustive) data flow solution during a sequence of calls to *Query*. Result caching does not increase the asymptotic time or space complexity of algorithm *Query*. Storing the cache requires $O(|N| \times |L|)$ space and updating the cache requires less than doubling the amount of work performed during the query evaluation. Moreover, the asymptotic worst case complexity of k invocations of *Query* with result caching is the same for any number k, where $1 \leq k \leq |L| \times |N|$ and $|L| \times |N|$ is the number of distinct queries.

5 Copy Constant Propagation

This section illustrates our approach for the problem of interprocedural copy constant propagation. Using this example we show how the query algorithm can be extended to handle programs with formal reference parameters and local variables. The difficulty introduced with formal reference parameters is the potential of aliasing. Ignoring aliasing during the analysis may lead to unsafe (i.e., invalid) query responses. We show how alias information can be incorporated to ensure safe answers to queries.

node n	refined local flow function $f_n(x)$, where $x = (x_1, \ldots, x_k)$
$v := c$	$f_n(x)_w = \begin{cases} c & \text{if } w = v \\ x_w \sqcap c & \text{if } w \in alias(v,n), \\ & w \neq v \\ x_w & \text{otherwise} \end{cases}$
$v := u$	$f_n(x)_w = \begin{cases} x_u & \text{if } w = v \\ x_w \sqcap x_u & \text{if } w \in alias(v,n), \\ & w \neq v \\ x_w & \text{otherwise} \end{cases}$
read(v), *v:=t*	$f_n(x)_w = \begin{cases} \bot & \text{if } w \in alias(v,n) \\ x_w & \text{otherwise} \end{cases}$

Table 3: Local flow functions in CCP refined based on may-alias information.

node n	**refined reverse local flow function** f_n^r
$v := c'$	$f_n^r([w{=}c]) = \begin{cases} \bot & \text{if } w = v \\ & \text{and } c = c' \\ \top & \text{if } w \in alias(v,n) \\ & \text{and } c \neq c' \\ [w{=}c] & \text{otherwise} \end{cases}$
$v := u$	$f_n^r([w{=}c]) = \begin{cases} [u{=}c] & \text{if } w = v \\ [w{=}c,\ u{=}c] & \text{if } w \in alias(v,n), \\ & w \neq v \\ [w{=}c] & \text{otherwise} \end{cases}$
read(v), *v:=t*	$f_n^r([w{=}c]) = \begin{cases} \top & \text{if } w \in alias(v,n) \\ [w{=}c] & \text{otherwise} \end{cases}$

Table 4: Reverse local flow functions in CCP refined based on may-alias information.

For simplicity, we assume programs with flat scoping. The address space *Addr(p)* of a procedure p is defined as: $Addr(p) = Global \cup Local(p) \cup Formal(p)$, where *Global* is the set of global variables in the program, $Local(p)$ is the set of variables local to p and $Formal(p)$ is the set of formal parameters of p.

Two variables x and y are *aliases* in a procedure p if x and y may refer to the same location during some invocation of p. Reference parameters may introduce aliases through the binding mechanism between actual and formal parameters. The determination of precise alias information is *NP*-complete [Mye81]. Therefore, we assume that approximate alias information is provided for each procedure p in form of a summary relation $alias(p)$ as described in [Coo85]. A pair $(x, y) \in alias(p)$ if x is aliased to y in some invocation of p (*may*-aliases). We use $alias(x, p) = \{y \mid (x, y) \in alias(p)\}$ to denote the set of *may* aliases of x in p and $alias(x, n) = alias(x, p)$ if node n is contained in p.

The computation of the $alias(p)$ sets can be expressed as a distributive data flow problem with a finite domain set over a program's call graph [Coo85]. Thus, we can employ the demand-driven analysis concepts from the previous section in order to compute only the relevant alias pairs in procedures that are actually analyzed. However, to avoid confusion in discussing the refinements in this section we make no assumption on how the alias information is computed but assume that sets $alias(p)$ are available to the CCP

algorithm for each procedure p. Using these alias sets we refine the local flow functions from Section 2 to safely account for the potential of aliasing. More precise refinements are possible if, in addition to *may* alias information, *must* alias information is available. The refined local flow functions and the corresponding refined reverse flow function are shown in Table 3 and Table 4, respectively.

The equation system 3 for the determination of reverse summary functions is refined to express the name binding mechanisms of reference parameters at call sites. We define a binding function b_s for each call site s that maps a lattice element x from the calling procedure to the corresponding element $b_s(x)$ in the called procedure according to the parameter passing at s. We will also need to consider the reverse binding b_s^{-1} to translate a lattice element from a called procedure to the corresponding element in the calling procedure. Let s be a call site in a procedure p that passes the actual parameters $(ap_1, \ldots ap_j)$ to the formal parameters $(fp_1, \ldots fp_j)$ in the called procedure q. Furthermore, let $v \in Addr(p)$ and $w \in Addr(q)$:

$$b_s([v{=}c]) = \bigsqcup_{u \in U} \{[u{=}c]\},$$

$$where\ U = (\{v\} \cap Global) \cup \{fp_i \mid ap_i = v\}$$

$$b_s^{-1}([w{=}c]) = \begin{cases} [w{=}c] & if\ w{\in}Global \\ [ap_i{=}c] & if\ w{=}fp_i \\ \bot & otherwise \end{cases}$$

The functions b_s and b_s^{-1} are $\sqcup$-distributive and $b_s(\top) = b_s^{-1}(\top) = \top$.

Equation system 4 shows the refined definition of reverse summary functions. The equations in 4 are defined for base elements $[v{=}c]$, where $v \in Addr(p)$. If $v \notin Addr(p)$ then $\phi^r_{(r_p,e_p)}([v{=}c]) = [v{=}c]$.

Equation System 4

$$\phi^r_{(e_p,e_p)}([v{=}c]) = [v{=}c] \qquad \text{for each procedure } p$$

$$\phi^r_{(n,e_p)}([v{=}c]) = \bigsqcup_{m \in succ(n)} \begin{cases} f^r_m \cdot \phi^r_{(m,e_p)}([v{=}c]) \\ \qquad if\ m \notin N_{call} \\ (b_m^{-1} \cdot \phi^r_{(r_q,e_q)} \cdot b_m)([v{=}c])) \\ \qquad if\ m{\in}N_{call}, call(m) = q \end{cases}$$

Finally, we refine the query propagation rules using the binding functions. The refinement is shown for queries with respect to the base elements in a procedure p, where $n \in N_p\text{-}N_{call}$ and $v{\in}\ Addr(p)$. For an arbitrary set of base elements B, the query $< \bigsqcap_{b \in B} b, n >$ is equivalent to the conjunction $\bigwedge_{b \in B} < b, n >$.

Refined query propagation in CCP

$$< [v = c], r_{main} > \iff false$$

$$< [v{=}c], r_p > \iff \bigwedge_{\substack{m \in N_{call}, \\ call(m)=p}} < [b_m^{-1}(v){=}c], m >$$

$$< [v{=}c], n > \iff$$

$$\bigwedge_{\substack{m \in pred(n), \\ m \notin N_{call}}} \begin{cases} false & if\ f^r_m([v{=}c]){=}\top \\ true & if\ f^r_m([v{=}c]){=}\bot \\ < f^r_m([v{=}c]), m > & otherwise \end{cases}$$

$$\wedge \bigwedge_{\substack{m \in pred(n), \\ m \in N_{call}, \\ call(m) = p}} \begin{cases} false & if\ h_m([v{=}c]){=}\top \\ true & if\ h_m([v{=}c]){=}\bot \\ < h_m([v{=}c]), m > & otherwise \end{cases}$$

where $h_m([v{=}c]) = (b_m^{-1} \cdot \phi^r_{(r_q,e_q)} \cdot b_m)(\bigsqcup_{u \in alias(v,m)} [u{=}c])$

5.1 CCP Query Algorithm

Figure 5 shows algorithm *IsCCP* to respond to a query of the form "Is variable v a copy constant at node n?". If v is a copy constant at n then *IsCCP* returns the constant value c of v. Otherwise *IsCCP* returns *false*. Note, that this query format is more general than previously discussed. More specific queries of the form "Is v a copy constant at n with value c?" can always be answered using the response of *IsCCP*.

Algorithm *IsCCP* is an instance of the generic algorithm *Query* from Section 3, except that *IsCCP* also includes the refinements for reference parameters. To handle the more general query format, the specification of a constant value c is simply dropped; a base element $[v{=}c]$ simplifies to $[v]$ and a query $< [v], n >$ raises the question as to whether variable v has some arbitrary but fixed constant value at node n. A query $< [v], n >$ evaluates to *true* at a node n if node n assigns *any* constant c to variable v, in which case the value c is remembered. If all generated queries evaluate to *true* the join over the remembered constant values is examined. If this join yields a constant c, then c is returned. Otherwise the response is *false*.

The corresponding instance of the generic procedure *Compute*ϕ^r, shown in Figure 6, partially evaluates the equation system 4. By the distributivity of the reverse summary functions, it is sufficient to maintain table entries only for base elements resulting in $MaxAddr \times N$ entries, where *MaxAddr* is the size of the maximal address space in any procedure. Each entry may contain a set of base elements and is therefore of size *MaxAddr*. To keep track of the actual constant values encountered, the table M includes an extra field $M[p, v].val$ for each procedure p and each variable v.

We now consider the cost of procedures *IsCCP* and *Compute*ϕ^r not including the cost of computing the alias information. During an invocation of *IsCCP* a total of $O(MaxAddr \times |N|)$ queries may be generated resulting in $O(MaxAddr \times |N|)$ join and reverse function applications in procedure *IsCCP*. The fixed point computation of table entries in procedure *Compute*ϕ^r requires in the worst case $O(MaxAddr^2 \times |N|)$ table entry updates. As in the general case, each table update may trigger up to C join and/or reverse function applications, where C is the maximal number of call sites calling a single procedure. Assuming join and reverse function applications are performed pointwise, each join or function application requires $O(MaxAddr)$ time resulting in the total time of $O(C \times MaxAddr^3 \times |N|)$.

```
IsCCP(v, n)
1.  for each m ∈ N do query[m] ← ∅;
2.  query[n] ← [v] ; worklist ← {n}; val = ⊥;
3.  while worklist ≠ ∅ do
4.     remove a node m from worklist;
5.     let p be such that m ∈ N_p;
6.     case m = r_q for some proc. q:
7.        for each m' ∈ N_call s.t. call(m') = q do
8.           query[m'] ← query[m'] ⊔ b_m^{-1}(query[m]);
9.           if query[m'] changed then add m' to worklist;
10.       endfor;
11.    otherwise:
12.       for each m' ∈ pred(m) do
                                  ⎧ f^r_{m'}(query[m])                          if m' ∉ N_call
13.    new ←                      ⎨ b^{-1}_{m'}(Computeφ^r(q, b_{m'}(x), val),  if m' ∈ N_call,
                                  ⎩ where x = alias(query[m], p)                call(m') = q
14.          if new = ⊥ and m' ∉ N_call then val ← val ⊔ c,
                where c is the constant assigned at m';
15.          if new = ⊤ or val = ⊤ then return( false )
16.          else if new ⊐ ⊥ then
17.             /* query still unresolved */
18.             query[m'] ← query[m'] ⊔ new;
19.             if query[m'] changed then add m' to worklist;
20.          endif;
21.       endfor;
22. endwhile;
23. if val < ⊤ then return(val) else return(false);
```

Figure 5: Demand algorithm for CCP. $IsCCP(v, n)$ returns the constant c if variable v is a copy constant at node n with value c, otherwise it returns *false*.

5.2 CCP Example

We illustrate algorithm *IsCCP* for copy constant propagation using the program example in Figure 7. The program consists of three procedures *main*, p and q, where $Global = \{x\}$, $Local(main) = \{y\}$, $Formal(p) = \{f\}$ and $Formal(q) = \{g, h\}$. The *may*-alias relations are: $alias(p) = \{(x, f)\}$ and $alias(q) = \{(x, h), (x, g), (y, g), (g, h)\}$. The reverse local flow functions are shown next to each node in the IFG. The table in Figure 7 shows the reverse summary functions. In addition to the reverse summary function entries for the base elements in each procedure (rows for $\phi^r_{(7,9)}$ and $\phi^r_{(10,12)}$) the table also shows intermediate entries (last two rows) collected during the computation of the summary functions. We consider the evaluation of the following query examples:

Query 1: $?IsCCP(h, 10)$ ("Is the formal h of procedure q a copy constant on entry of each invocation of q"?):
Initially, worklist = {10} and $query[10] = [h]$. $Query[10]$ is propagated to queries for the corresponding actual parameters at call sites resulting in: $query[5] = [x]$ and $query[8] = [f]$. Processing $query[8]$ causes the propagation of $[f]$ to node 7 and in turn to actual parameters at the call site at node 4, i.e., $query[4] = [x]$. Assume $query[5]$ is processed next. The global x is passed to formal f of procedure p at node 4. Therefore, the reverse summary function $\phi^r_{(7,9)}$ is inspected for the two arguments [x=c] and [f=c]. The new query element resulting for node 4 is determined as $query[4] = b_4^{-1}(\phi^r_{(7,9)}([x{=}c]) \sqcup \phi^r_{(7,9)}([f{=}c])) = b_4^{-1}([x{=}c, f{=}c]) = [x{=}c]$, which is simplified to $[x]$. Applying the reverse function yields $f_4^r([x]) = \bot$, i.e., the query at node 4 evaluates to *true* and 1 is remembered as the actual constant assigned.

```
Computeφ^r(p, y, val)
1.  worklist ← ∅; res ← ⊥;
2.  let y = [v_1, ..., v_k], where v_i ∈ Addr(p);
3.  for each v_i, where 1 ≤ i ≤ k do
4.     if M[e_p, v_i] = ⊥ then add (e_p, v_i) to worklist;
5.        M[e_p, v_i] = [v_i]; endif;
6.  while worklist ≠ ∅ do
7.     remove a pair (n, w) from worklist;
8.     let p' be the proc. containing n;
9.     let [w_1, ..., w_j] = M[n, w];
10.    case n ∈ N_call and call(n) = q:
11.       for each w_i, where 1 ≤ i ≤ j do
12.          if w_i ∈ Global or w_i is an actual param. at n then
13.             for each z ∈ b_n([w_i]) do
14.                if M[e_q, z] = [z] then
15.                   for each m ∈ pred(n) do
16.                      Propagate(m, w, b_n^{-1}(M[r_q, z]));
17.                   if M[r_q, z] = ⊥ then
18.                      M[p', w].val ← M[p', w].val ⊔ M[q, z].val;
19.                else M[e_q, z] ← [z]; add (e_q, z) to worklist; endif;
20.             endfor;
21.          else /* skip call site if u not passed */
22.             for each m ∈ pred(n) do Propagate(m, w, [w_i]);
23.       endfor;
24.    case n = r_q for some procedure q:
25.       for each m ∈ N_call such that
26.          call(m) = q and b_m^{-1}([w]) ∈ M[m, z] for some z do
             /* if entry M[r_q, w] was requested earlier */
27.          let p'' be the proc. containing m;
28.          for each m' ∈ pred(m) do
29.             Propagate(m', z, b_m^{-1}(M[n, w]));
30.          if M[r_q, w] = ⊥ then
31.             M[p'', z].val ← M[p'', z].val ⊔ M[q, w].val;
32.    otherwise:
33.       for each m ∈ pred(n) do
34.          Propagate(m, w, f^r_m(M[n, w]));
35.       if f^r_n(M[n, w]) = ⊥ then
36.          M[p', w].val ← M[p', w].val ⊔ c, where
37.             c is the const. assigned at m;
38. endwhile;
39. for each v_i, where 1 ≤ i ≤ k do
40.    res ← res ⊔ M[r_p, v_i]; val ← val ⊔ M[p, v_i].val; endfor;
41. return(res);

Propagate(n, v, new) /* propagate new to M[n, v] */
1.  M[n, v] ← M[n, v] ⊔ new;
2.  if M[n, v] changed then add (n, v) to worklist; endif;
```

Figure 6: Algorithm $Computeφ^r(p, y, val)$ in CCP returning the value $\phi^r_{(r_p,e_p)}(y)$ for a procedure p and $y \in L$.

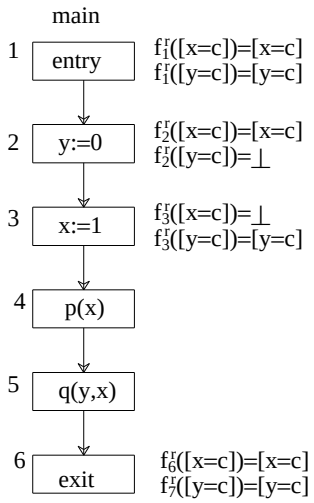
Since the worklist is exhausted, the overall response is 1, indicating that the formal h of procedure q always has the value 1 on entry of procedure q.

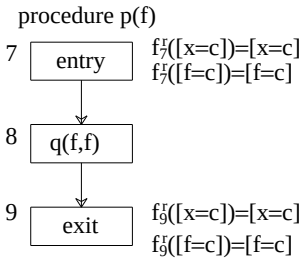
Query 2: $?IsCCP(x, 6)$ ("Is variable x a copy constant after the call to procedure q at node 6"?):
Initially, worklist = {6} and $query[6] = [x]$. Processing $query[6]$ results in: $query[5] = b_5^{-1}(\phi^r_{(10,12)}([x{=}c] \sqcup [h{=}c])) = b_5^{-1}([x{=}c, g{=}c]) = [x{=}c, y{=}c]$ which is simplified to $[x, y]$. We consider the propagation of the query elements $[x]$ and $[y]$ separately. Since y is local to *main*, $[y]$ directly propagates through nodes 4 and 3. At node 2: $f_2^r([y]) = \bot$ and the value 0 is remembered as the actual constant assigned. As for Query 1, propagating the query element $[x]$ from $query[5]$ will eventually lead to the evaluation *true* at node 4 with 1 being the actual constant assigned. Since

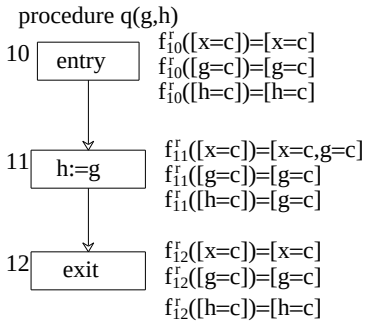
```
procedure main
declare local y; global x;
begin
    read(y);
    x:=1;
    call p(x);
    call q(y,x);
end
```

```
procedure p(f)
begin
    call q(f,f);
end
```

```
procedure q(g,h)
begin
    h:=g;
end
```







Reverse summary function values

Proc. p	argument		Proc. q	argument		
function	$[x=c]$	$[f=c]$	function	$[x=c]$	$[g=c]$	$[h=c]$
$\phi^r_{(7,9)}$	$[x=c,f=c]$	$[f=c]$	$\phi^r_{(10,12)}$	$[x=c,g=c]$	$[g=c]$	$[g=c]$
$\phi^r_{(8,9)}$	$[x=c,f=c]$	$[f=c]$	$\phi^r_{(11,12)}$	$[x=c,g=c]$	$[g=c]$	$[g=c]$
$\phi^r_{(9,9)}$	$[x=c]$	$[f=c]$	$\phi^r_{(12,12)}$	$[x=c]$	$[g=c]$	$[h=c]$

Figure 7: Example for CCP

the two constant values encountered differ, the response of *IsCCP* is *false*.

Note, the response *false* is safe but imprecise. A closer inspection of the code reveals that x is actually a copy constant at node 6 with value 0. The imprecision is due to the loss of information in the alias summary sets. The alias set for a procedure is obtained by merging together the alias configurations for all invocations. Avoiding this kind of inaccuracies would require a separate analysis (i.e., separate reverse summary function) for each different alias configuration that may hold at a procedure. However, the cost of this approach is prohibitive, as it may require an exponential number of different reverse summary functions.

6 Related Work

A number of variations of the notion of demand-driven analysis and the notion of a partial backward propagation of information have separately appeared in the literature.

The concepts of deriving data flow information by backward propagation of assertions was described using operational semantics by Cousot [Cou81] and later developed and implemented in a debugging system for higher-order functions [Bou93]. The analysis for discovering linked conditions in programs described in [SMHY93] is also based on backward propagation of assertions starting from test sites in conditionals.

An important component of our demand-driven approach is the tabulation algorithm *Compute*ϕ^r that implements the lazy evaluation of only relevant (i.e., needed) equation values. The algorithm is essentially a reversed version of Sharir and Pnueli's tabulation algorithm [SP81] to compute the original (unreversed) summary functions. A similar lazy fixed point computation of only the relevant equations was also described in the chaotic iteration algorithms [CC77] and the minimal function graphs for applicative programs [JM73].

Reverse flow functions, which we apply in the query propagation rules, have previously been discussed in [HL92] to demonstrate that an abstract interpretation may be performed in either a forward or a backward direction. The relationship between forward and backward directions of an analysis was also discussed by Cousot [Cou81].

Recently, two approaches to demand-driven *inter*procedural analysis were presented by Reps [Rep94] and Reps et al. [RSH94]. In the first approach [Rep94], a limited class of data flow problems, the *locally separable problems*, are encoded as logic programs. Demand algorithms are then obtained by utilizing fast logic program evaluation techniques developed in the logic-programming and deductive-database communities. In a more recent work by Reps et al. [RSH94] the first approach is generalized to a larger class of problems. In this second approach a data flow problem is transformed into a specialized graph-reachability problem. The graph for the reachability problem, the *exploded super-graph*, is obtained as an expansion of a program's control flow graph by including an explicit graphical representation of each node's flow function. As in our approach, Reps et al. base their approach on a variant of Sharir and Pnueli's interprocedural analysis framework. Similar to our algorithm *Compute*ϕ^r, a caching tabulation algorithm is used to compute the solu-

tion over the graph that. However, the graph-reachability algorithm imposes more restrictions on the problems that can be handled than our approach. Specifically, the graph-reachability approach is applicable to problems where the lattice is a powerset over a finite domain set and where all flow functions are distributive. Although we require distributive functions to ensure precise data flow solutions, our algorithms still provide approximate information in the presence of non-distributive functions. Our approach is less restrictive on the lattice structure in that it is applicable to any finite lattice. The restriction to a finite lattice does not even apply if our approach is used for *intra*procedural analysis.

The utility of demand-driven analysis algorithm has also been demonstrated in a number of demand-driven algorithms developed for specific analysis problems, including the following problems. Babich et al. [BJ78] presented a demand algorithm for *intra*procedural live variable analysis based on attribute grammars. Strom and Yellin [SY93] presented a demand based analysis for typestate checking. The authors experimentally demonstrate that their goal-oriented (and demand-driven) backward analysis is more efficient than the original forward analysis for typestate checking that eagerly collects all available information that may or may not be of relevance. Question propagation, a phase in the algorithm for global value numbering [RWZ88] performs a demand-based backward search in order to locate redundant expressions. This backward search, like our query algorithm, performs the analysis from the points of interest (i.e., the points where an expression is suspected to be redundant) and it also uses early termination to end the search. In procedure cloning [CHK92], procedure clones are created during the analysis on demand whenever it is found that an additional clone will lead to more accurate information. Cytron and Gershbein [CG93] described an algorithm for the incremental incorporation of alias information into SSA form. The actual optimization problem to be performed on the SSA form triggers the expansion of the SSA form to include only the necessary alias-information. Similar ideas have also been implemented in the demand-based expansion algorithm of factored def-def chains [CCF92].

Other related work addresses the goal of reducing the cost of data flow analysis by avoiding the computation of irrelevant intermediate results. Several sparse analysis techniques have been presented to reduce the number of data flow equations by either manipulating the underlying graphical program representation, such as the analyses based on the global value graph [RT82], static single assignment form [RWZ88, WZ85], the sparse evaluation graph [CCF90], the dependence graph [JP93] or by direct manipulation of the equation system through partitioning algorithms [DGS94]. Slotwise analysis [DRZ92] also falls into this class of sparse technique but is limited to bitvector data flow problems. Sparse analysis approaches to reduce the amount of data flow evaluations are complementary to our demand-driven analysis algorithms in that the evaluation of data flow equations is avoided independent of the specific information demanded; that is, the evaluation of those equations is avoided that are irrelevant with respect to even the exhaustive solutions. An interesting combination of the two approaches would be to use, for example, a reduced equation system according to [DGS94] or a sparse evaluation graph as in [CCF90], as the basis for propagating data flow queries.

Incremental data flow analysis [Ros81, Zad84, RP88, PS89] has also addressed the avoidance of exhaustive solution recomputations. However, unlike demand-driven analysis, incremental analysis assumes that an exhaustive solution has previously been computed and is concerned with avoiding exhaustive *re*-computations in response to program changes.

7 Conclusions

This paper described a new demand-driven approach to interprocedural data flow analysis that avoids the costly computation of exhaustive data flow solutions. A general framework for the derivation of demand algorithms covering the class of interprocedural data flow problems with a finite domain set was presented. In a data flow problem with distributive flow functions, the derived algorithms provide as precise information as the corresponding exhaustive algorithm. In the presence of non-distributive flow functions, the derived algorithms still provide approximate solutions. We are currently exploring the utility of approximate query responses for non-distributive problems in practice. Particularly efficient demand algorithms result for the classical bitvector problems. For example in live variable analysis, queries about individual variables are resolved in linear time and space if the analysis is intraprocedural or if the analysis is interprocedural and the programs are alias-free.

The goal of demand-driven program analysis is the reduction of both the time and space cost of solving data flow problems. When discussing result caching we implicitly assumed that reducing the analysis time is of primary interest. However, as programs grow larger, space may become as valuable a resource as analysis time, in particular, if results for a large number of different data flow problems are to be collected. Clearly, result caching should be avoided if the space consumption is the primary concern. Since result caching is optional in our query algorithm, it can be adjusted easily to the needs of the current application. The option of result caching permits the computation of the complete solution with an asymptotic worst case complexity that is no worse than the cost of a standard exhaustive algorithm. To fully evaluate the benefits of the achievable time and space reductions we are currently developing a prototype implementation of a demand-based query algorithm for interprocedural copy constant propagation.

Acknowledgements

We thank Tom Reps and the referees for their comments on an earlier draft of this paper.

References

[Bir84] G. Birkhoff. *Lattice theory*, volume 25. American Mathematical Society, Colloquium Publication, Washington, DC, 3rd edition, '84.

[BJ78] W.A. Babich and M. Jazayeri. The method of attributes for data flow analysis: Part II . Demand analysis. *Acta Informatica*, 10(3), Oct. '78.

[Bou93] F. Bourdoncle. Abstract debugging of high-order imperative languages. In *SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 36–45, Albuquerque, NM, Jun. '93.

[CC77] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Hollan Pub. Co., '77.

[CCF90] J.D. Choi, R.K. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *18th ACM Symp. on Principles of Programming Languages*, pages 55–66, Orlando, FL, Jan. '90.

[CCF92] J.D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Trans. on Software Engineering*, 20(2):105–114, Feb. '92.

[CG93] R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in SSA form. In *SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 36–45, Albuquerque, NM, Jun. '93.

[CHK92] K. Cooper, M. Hall, and K. Kennedy. Procedure cloning. In *IEEE 1992 Int. Conf. on Computer Languages*, pages 96–105, San Francisco, CA, April 1992.

[CK88] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN '88 Symp. on Compiler Construction, published in SIGPLAN Notices*, 23(7):57–66, Jun. '88.

[Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *12th ACM Symp. on Principles of Programming Languages*, pages 281–290, '85.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 303–342. Prentice-Hall, '81.

[DGS94] E. Duesterwald, R. Gupta, and M.L. Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *5th Int. Conf. on Compiler Construction*, pages 357–373, Edinburgh, U.K., Apr. '94. Springer Verlag, LNCS 786.

[DRZ92] D.M. Dhamdhere, B.K. Rosen, and F.K. Zadeck. How to analyze large programs efficiently and informatively. In *SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 212–223, San Francisco, CA, Jun. '92.

[GT93] D. Grove and L. Torczon. Interprocedural constant propagation: a study of jump function implementations. In *SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 90–99, Albuquerque, NM, Jun. '93.

[HL92] J. Hughes and J. Launchbury. Reversing abstract interpretations. In *4th European Symp. on Programming*, pages 269–286, Rennes, France, Feb. '92. Springer Verlag, LNCS 582.

[JM73] N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *13th Symp. on Principles of Programming Languages*, pages 194–206, Florida, 1973.

[JM82] N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th Symp. on Principles of Programming Languages*, pages 66–74, Albuquerque, New Mexico, 1982.

[JP93] R. Johnson and K. Pingali. Dependence-based program analysis. In *SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 78–89, Albuquerque, NM, Jun. '93.

[Kil73] G. Kildall. A unified approach to global program optimization. In *1st ACM Symp. on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, Jan. '73.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *4th Int. Conf. on Compiler Construction*, pages 125–140, Paderborn, Germany, Oct. '92. Springer Verlag, LNCS 641.

[KU77] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, Jul. '77.

[Mas94] V. Maslov. Lazy array data-flow dependence analysis. In *ACM Symp. on Principles of Programming Languages*, pages 1–15, Jan. '94.

[Mye81] E.W. Myers. A precise inter-procedural data flow algorithm. In *8th ACM Symp. on Principles of Programming Languages*, pages 219–230, Williamsburg, Virginia, Jan. '81.

[PS89] L. Pollock and M.L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. on Software Engineering*, 15(12):1537–1549, Dec. '89.

[Rep94] T. Reps. Solving demand versions of interprocedural analysis problems. In *5th Int. Conf. on Compiler Construction*, pages 389–403, Edinburgh, U.K., Apr. '94. Springer Verlag, LNCS 786.

[Ros79] B. Rosen. Data flow analysis for procedural languages. *Journal of the ACM*, 26(2):322–344, '79.

[Ros81] B. Rosen. Linear cost is sometimes quadratic. In *8th ACM Symp. on Principles of Programming Languages*, pages 117–124, Jun. '81.

[RP88] B.G. Ryder and M.C. Paull. Incremental data flow analysis algorithms. *ACM Trans. Programming Languages and Systems*, 10(1):1–50, '88.

[RSH94] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark, '94.

[RT82] J. Reif and R.E. Tarjan. Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, 11(1):81–93, Feb. '82.

[RWZ88] B. Rosen, M. Wegman, and F.K. Zadeck. Global value numbers and redundant computations. In *15th ACM Symp. on Principles of Programming Languages*, pages 12–27, San Diego, CA, Jan. '88.

[SMHY93] A.D. Stoyenko, T.J. Marlowe, W.A. Halang, and M. Younis. Enabling efficient schedulability analysis through conditional linking and program transformations. *Control Engineering Practice*, 1(1):85–105, '93.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, '81.

[SY93] R.E. Strom and D.M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. on Software Engineering*, 19(5):478–485, May '93.

[WZ85] M. Wegman and F.K. Zadeck. Constant propagation with conditional branches. In *12th ACM Symp. on Principles of Programming Languages*, pages 291–299, New Orleans, Jan. '85.

[Zad84] F.K. Zadeck. Incremental data flow analysis in a structured program editor. In *SIGPLAN Symp. on Compiler Construction*, pages 132–143, Jun. '84.