ELSEVIER

Theoretical Computer Science 167 (1996) 131–170

Theoretical
Computer Science

Precise interprocedural dataflow analysis with applications to constant propagation [1]

Mooly Sagiv *,[2], Thomas Reps, Susan Horwitz

Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706 USA

Abstract

This paper concerns interprocedural dataflow-analysis problems in which the dataflow information at a program point is represented by an environment (i.e., a mapping from symbols to values), and the effect of a program operation is represented by a distributive environment transformer. We present two efficient algorithms that produce precise solutions: an exhaustive algorithm that finds values for all symbols at all program points, and a demand algorithm that finds the value for an individual symbol at a particular program point.

Two interesting problems that can be handled by our algorithms are (decidable) variants of the interprocedural constant-propagation problem: *copy-constant propagation* and *linear-constant propagation*. The former interprets program statements of the form $x := 7$ and $x := y$. The latter also interprets statements of the form $x := 5 * y + 17$.

Experimental results on C programs have shown that

- Although solving constant-propagation problems precisely (i.e., finding the meet-over-all-*valid*-paths solution, rather than the meet-over-*all*-paths solution) resulted in a slowdown by a factor ranging from 2.2 to 4.5, the precise algorithm found additional constants in 7 of 38 test programs.
- In contrast to previous results for numeric Fortran programs, linear-constant propagation found more constants than copy-constant propagation in 6 of 38 test programs.
- The demand algorithm, when used to demand values for all uses of scalar integer variables, was faster than the exhaustive algorithm by a factor ranging from 1.14 to about 6.

* Corresponding author.

[1] This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants CCR-8958530 and CCR-9100424, 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), and by a grant from Xerox Corporate Research.
Part of this work was done while the authors were visiting the University of Copenhagen.
A preliminary version of this paper appeared in *Proc. of FASE 95: Colloquium on Formal Approaches in Software Engineering*, (Aarhus, Denmark, May 22-26, 1995) [27].

[2] Current address: Department of Computer Science, The University of Chicago, 1100 East 58th Street, Chicago, IL 60637, USA.

PII S0304-3975(96)00072-2

1. Introduction

This paper concerns how to find precise solutions to a large class of interprocedural dataflow-analysis problems in polynomial time. Of the problems to which our techniques apply, several variants of the *interprocedural constant-propagation problem* stand out as being of particular importance.

In contrast to *intra*procedural dataflow analysis, where "precise" means "meet-over-all-paths" [16], a precise *inter*procedural dataflow-analysis algorithm must provide the "meet-over-all-*valid*-paths" solution. (A path is *valid* if it respects the fact that when a procedure finishes it returns to the site of the most recent call [28, 4, 18, 17, 22, 25, 24, 9, 13]). In this paper, we show how to find the meet-over-all-valid-paths solution for a certain class of dataflow problems in which the dataflow facts are maps ("environments") from some finite set of symbols D to some (possibly infinite) set of values L (i.e., the dataflow facts are members of $Env(D,L)$), and the dataflow functions ("environment transformers" in $Env(D,L) \xrightarrow{d} Env(D,L)$) distribute over the meet operator of $Env(D,L)$. We call this set of dataflow problems the *I*nterprocedural *D*istributive *E*nvironment problems (or IDE problems, for short).

The contributions of this paper can be summarized as follows:

- We introduce a *compact graph representation of distributive environment transformers.*
- We present *an algorithm for finding meet-over-all-valid-paths solutions.* For general IDE problems the algorithm will not necessarily terminate. However, we identify a subset of IDE problems for which the algorithm does terminate and runs in time $O(ED^3)$, where E is the number of edges in the program's control-flow graph and D is the number of symbols in an environment.
- We study two natural variants of the constant-propagation problem: copy-constant propagation [10] and linear-constant propagation, which extends copy-constant propagation by interpreting statements of the form $x := a * y + b$, where a and b are literals or user-defined constants. The IDE problems that correspond to both of these variants fall into the above-mentioned subset; consequently, our techniques *solve all instances of these constant-propagation problems in time* $O(E\,\text{MaxVisible}^3)$, where "MaxVisible" is the maximum number of variables visible in any procedure of the program. The algorithms obtained in this way improve on the well-known constant-propagation work from Rice [5, 12] in two ways:

  1. The Rice algorithm is not precise for recursive programs. (In fact, it may fall into an infinite loop when applied to recursive programs.)

  2. Because of limitations in the way "return jump functions" are generated, the Rice algorithm does not even yield precise answers for all non-recursive programs. In contrast, our algorithm yields *precise results, for both recursive and non-recursive programs.*
- In Section 6 we present a demand dataflow-analysis algorithm for the class of IDE problems. This demand algorithm is more general than both the demand algorithm

of Duesterwald et al. [9] and the demand algorithm of Horwitz et al. [13]. For example, it can handle linear-constant-propagation problems, which neither of the above algorithms can handle.

- Our dataflow-analysis algorithms have been implemented and used to analyze C programs. Our experimental results have shown that:
  - Although solving constant-propagation problems precisely resulted in a slowdown by a factor ranging from 2.2 to 4.5, the precise algorithm found additional constants in 7 of 38 test programs.
  - In contrast to previous results for numeric Fortran programs [12], linear-constant propagation found more constants than copy-constant propagation in 6 of 38 test programs.
  - The demand algorithm, when used to demand values for all uses of scalar integer variables, was faster than the exhaustive algorithm by a factor ranging from 1.14 to about 6.

The remainder of the paper is organized as follows: In Section 2 we introduce the copy-constant-propagation and linear-constant-propagation problems. Linear-constant propagation is used in subsequent sections to illustrate our ideas. In Section 3 we define the class of IDE problems. In Section 4, we define a compact graph representation of distributive environment transformers and show how to use these graphs to find the meet-over-all-valid-paths solution to a dataflow problem. Section 5 presents our algorithm for solving IDE problems. In Section 5.4, we discuss the application of our approach to copy-constant propagation and linear-constant propagation. In Section 6 we extend our algorithm to perform demand-driven dataflow analysis. Experiments in which our algorithm has been applied to perform copy and linear-constant propagation on C programs are reported in Section 7. Section 8 discusses related work.

2. Distributive constant-propagation problems

There are (at least) two important variants of the constant-propagation problem that fit into the framework presented in this paper: copy-constant propagation and linear-constant propagation. In copy-constant propagation, a variable x is discovered to be constant either if it is assigned a constant value (e.g., $x := 3$) or if it is assigned the value of another variable that is itself constant (e.g., $y := 3$; $x := y$). All other forms of assignment (e.g., $x := y + 1$) are (conservatively) assumed to make x non-constant.

Linear-constant propagation identifies a superset of the instances of constant variables found by copy-constant propagation. Variable x is discovered to be constant either if it is assigned a constant value (e.g., $x := 3$) or if it is assigned a value that is a linear function of one variable that is itself constant (e.g., $y := 3$; $x := 2 * y + 5$). All other forms of assignment are assumed to make x non-constant.

Constant propagation is of importance in optimizing compilers for two reasons: (i) programs run faster when constants are substituted at compile time for constant variables; and (ii) the results of constant propagation enable other optimizing transformations, which in turn permits more efficient code to be produced.

3. The IDE framework

3.1. Program representation

A program is represented using a directed graph $G^* = (N^*, E^*)$ called a *supergraph*. G^* consists of a collection of flowgraphs $G_1, G_2, \ldots$ (one for each procedure), one of which, G_{main}, represents the program's main procedure. Each flowgraph G_p has a unique *start* node s_p, and a unique *exit* node e_p. The other nodes of the flowgraph represent statements and predicates of the program in the usual way,[3] except that a procedure call is represented by two nodes, a *call* node and a *return-site* node.

In addition to the ordinary intraprocedural edges that connect the nodes of the individual flowgraphs, for each procedure call, represented by call-node c and return-site node r, G^* has three edges:

- An intraprocedural *call-to-return-site* edge from c to r
- An interprocedural *call-to-start* edge from c to the start node of the called procedure
- An interprocedural *exit-to-return-site* edge from the exit node of the called procedure to r.

The call-to-return-site edges are included so that we can handle programs with local variables and parameters; the dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables that holds at the call site to be combined with the information about global variables that holds at the end of the called procedure.

Example 3.1. Fig. 1 shows an example program and its supergraph. For the moment ignore the edge labels. This program will be used in the rest of the paper as a running example.

3.2. Interprocedural paths

Definition 3.2. A *path* of length j from node m to node n is a (possibly empty) sequence of j edges, which will be denoted by $[e_1, e_2, \ldots, e_j]$, such that the source of e_1 is m, the target of e_j is n, and for all i, $1 \leqslant i \leqslant j-1$, the target of edge e_i is the source of edge e_{i+1}. Path concatenation is denoted by $||$.

[3] The nodes of a flowgraph can represent individual statements and predicates; alternatively, they can represent basic blocks. In our examples and experiments, nodes represent individual statements and predicates.

```
declare x: integer
program main
begin
        call P(7)
        print (x) /* x is a constant here */
end

procedure P (value a : integer)
begin /* a is not a constant here */
        if a > 0 then
          a := a − 2
          call P (a)
          a := a + 2
        fi
        x := −2 * a + 5
        /* x is not a constant here */
end
```

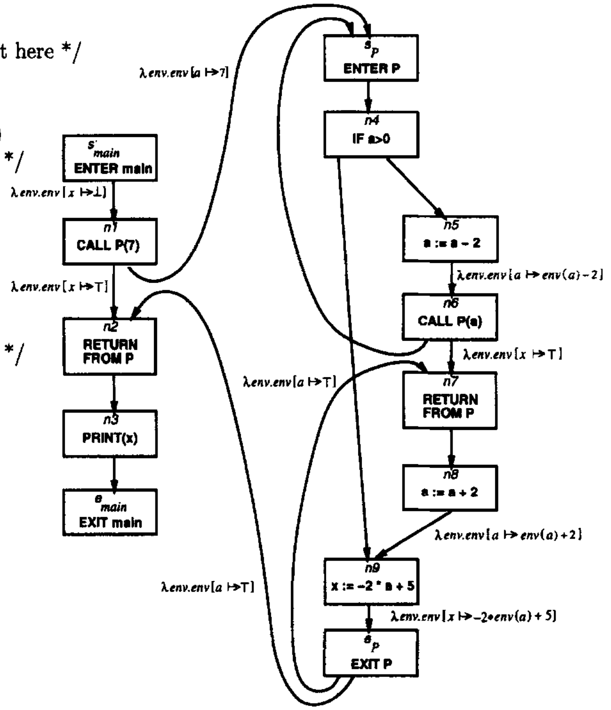


Fig. 1. An example program and its labeled supergraph G^*. The environment transformer for all unlabeled edges is $\lambda env.env$.

The notion of an (*interprocedurally*) *valid path* is necessary to capture the idea that not all paths in G^* represent potential execution paths. A valid path is one that respects the fact that a procedure always returns to the site of the most recent call. To understand the algorithm of Section 5, it is useful to distinguish further between a *same-level valid path* – a path in G^* that starts and ends in the same procedure, and in which every call has a corresponding return (and vice versa) – and a *valid path* – a path that may include one or more unmatched calls.

Definition 3.3. The sets of *same-level valid paths* and *valid paths* in G^* are defined inductively as follows:

- The empty path is a *same-level valid path* (and therefore a *valid path*).
- Path $p \parallel [e]$ is a *valid path* if either e is not an exit-to-return-site edge and p is valid, or e is an exit-to-return-site edge and $p = p_h \parallel [e_c] \parallel p_t$ where p_t is a same-level valid path, p_h is a valid path, and the source node of e_c is the call node that matches the return-site node at the target of e. Such a path is a *same-level valid path* if p_h is also a same-level valid path.

We denote the set of valid paths from node m to node n by $VP(m,n)$.

Example 3.4. In the supergraph shown in Fig. 1, the path

$$s_{main} \rightarrow n1 \rightarrow s_p \rightarrow n4 \rightarrow n9 \rightarrow e_p \rightarrow n2$$

is a (same-level) valid path; the path

$$s_{main} \rightarrow n1 \rightarrow s_p \rightarrow n4 \rightarrow n9$$

is a (non-same-level) valid path because the call-to-start edge $n1 \rightarrow s_p$ has no matching exit-to-return-site edge; the path

$$s_{main} \rightarrow n1 \rightarrow s_p \rightarrow n4 \rightarrow n9 \rightarrow e_p \rightarrow n7$$

is not a valid path because the exit-to-return-site edge $e_p \rightarrow n7$ does not correspond to the preceding call-to-start edge $n1 \rightarrow s_p$.

3.3. Environments and environment transformers

Definition 3.5. Let D be a finite set of symbols. Let L be a finite-height meet semilattice with a top element $\top$.[4] We denote the meet operator by $\sqcap$. The set $Env(D,L)$ of *environments* is the set of functions from D to L. The following operations are defined on $Env(D,L)$:

- The meet operator on $Env(D,L)$, denoted by $env_1 \sqcap env_2$, is $\lambda d.(env_1(d) \sqcap env_2(d))$.
- The top element in $Env(D,L)$, denoted by Ω, is $\lambda d.\top$.
- For an environment $env \in Env(D,L)$, $d \in D$, and $l \in L$, the expression $env[d \mapsto l]$ denotes the environment in which d is mapped to l and any other symbol $d' \neq d$ is mapped to the value $env(d')$.

Example 3.6. In the case of integer constant propagation:

- D is the set of integer program variables.
- $L = Z_\perp^\top$ where $x \sqsubseteq y$ iff $y = \top$, $x = \perp$, or $x = y$. Thus, the height of $Z_\perp^\top$ is 3.

In a constant-propagation problem, $Env(D,L)$ is used as follows: If $env(d) \in Z$ then the variable d has a known constant value in the environment env; the value $\perp$ denotes non-constant and $\top$ denotes an unknown value.

Definition 3.7. An environment transformer $t: Env(D,L) \rightarrow Env(D,L)$ is *distributive* (denoted by $t: Env(D,L) \xrightarrow{d} Env(D,L)$) iff for every $env_1, env_2, \ldots \in Env(D,L)$, and $d \in D$, $(t(\bigsqcap_i env_i))(d) = \bigsqcap_i (t(env_i))(d)$. Note that this equality must also hold for infinite sets of environments.

3.4. The meet-over-all-valid-paths solution

A dataflow problem is specified by annotating each edge e of G^* with an environment transformer that captures the effect of the program operation at the source of e.

[4] Hence, L is also complete and has a least element, denoted by $\perp$.

Definition 3.8. An *instance* of an *interprocedural distributive environment problem* (or *IDE problem* for short) is a four-tuple, $IP = (G^*, D, L, M)$, where

- G^* is a supergraph.
- D and L are as defined in Definition 3.5.
- $M : E^* \to (Env(D,L) \overset{d}{\to} Env(D,L))$ is an assignment of distributive environment transformers to the edges of G^*.

Definition 3.9. Let $IP = (G^*, D, L, M)$ be an IDE problem instance. The *meet-over-all-valid-paths* solution of IP for a given node $n \in N^*$, denoted by MVP_n, is defined as follows:

$$MVP_n \overset{\text{def}}{=} \bigsqcap_{q \in VP(s_{main}, n)} M(q)(\Omega),$$

where M is extended to paths by composition, i.e.,

$$M([\,]) = \lambda env.env$$

and

$$M([e_1, e_2, \ldots, e_j]) \overset{\text{def}}{=} M(e_j) \, o \, M(e_{j-1}) \, o \, \cdots \, o \, M(e_2) \, o \, M(e_1).$$

In an IDE problem, the environment transformer associated with an intraprocedural edge e represents a safe approximation to the actual semantics of the code at the source of e. Functions on call-to-return-site edges extract (from the dataflow information valid immediately before the call) dataflow information about local variables that must be re-established after the return from the call. Functions on exit-to-return-site edges extract dataflow information that is both valid at the exit site of the called procedure and relevant to the calling procedure.

Note that call-to-return-site edges introduce some additional paths in the supergraph that do not correspond to standard program-execution paths. The intuition behind the IDE framework is that the interprocedurally valid paths of Definition 3.3 correspond to "paths of action" for particular *subsets* of the runtime entities (e.g., global variables). The path function along a particular path contributes only *part* of the dataflow information that reflects what happens during the corresponding run-time execution. The facts for other subsets of the runtime entities (e.g., local variables) are handled by different "trajectories", for example, paths that take "short-cuts" via call-to-return-site edges.

In the case of linear-constant propagation, the interesting environment transformers are those associated with edges whose sources are start nodes, call nodes, exit nodes, or nodes that represent assignment statements.

Linear-constant propagation handles assignments of the form $x := c$ and $x := c_1 * y + c_2$, where c, c_1, and c_2 are literals or user-defined constants. The environment transformers associated with these assignment statements are of the form $\lambda env.env[x \mapsto c]$ and $\lambda env.env[x \mapsto c_1 * env(y) + c_2]$, respectively. For example, the transformer associated with edge $n9 \to e_P$ in the supergraph of Fig. 1 is $\lambda env.env[x \mapsto -2 * env(a) + 5]$.

For other assignment statements, for example, $x := y+z$, the associated environment transformer is $\lambda env.env[x \mapsto \bot]$. This transformer is a safe approximation to the actual semantics of the assignment; the transformer that exactly corresponds to the semantics, $\lambda env.env[x \mapsto env(y)+env(z)]$, cannot be used in the IDE framework because it is not distributive.

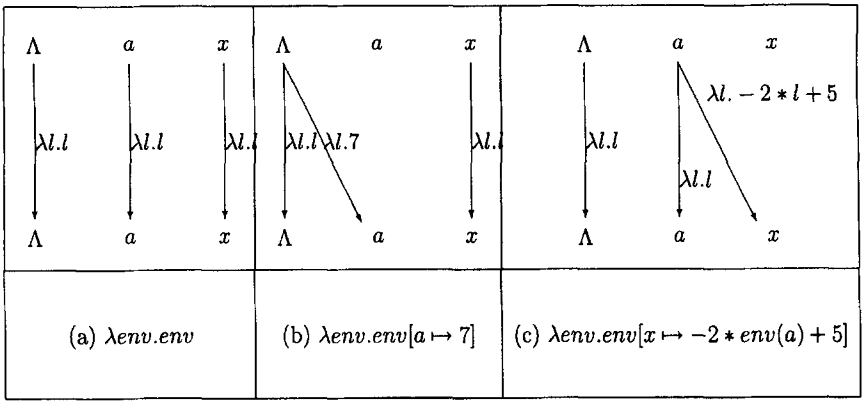
Whether edges out of start nodes have non-identity environment transformers depends on the semantics of the programming language. For example, these edges' environment transformers may reflect the fact that a procedure's local variables are uninitialized at the start of the procedure; that is, the transformers would be: $\lambda env.env[x_1 \mapsto \bot][x_2 \mapsto \bot]\ldots[x_n \mapsto \bot]$ for all local variables x_i. The environment transformers for the edges out of the start node for the program's main procedure may also reflect the fact that global variables are uninitialized when the program is started. For instance, in our running example we make the assumption that globals are uninitialized when execution begins, and thus the environment transformer associated with edge $s_{main} \rightarrow n1$ in the supergraph of Fig. 1 is $\lambda env.env[x \mapsto \bot]$.

The environment transformers associated with call-to-start edges reflect the assignments of actual parameters to formal parameters. For call-by-value-result parameters, the environment transformers associated with exit-to-return-site edges reflect the assignments of formals back to actuals. For example, the transformer associated with edge $n1 \rightarrow s_P$ in the supergraph of Fig. 1 is $\lambda env.env[a \mapsto 7]$. The transformer associated with edge $e_P \rightarrow n7$ in the supergraph of Fig. 1 is $\lambda env.env[a \mapsto \top]$, since the value of the local variable a of P at e_P has no impact on the value of the local variable a at $n7$. Instead, the value of a at $n7$ is equal to the value of a at $n6$, obtained via the environment transformer $\lambda env.env[x \mapsto \top]$, which is associated with edge $n6 \rightarrow n7$. In contrast, the value of the global variable x at $n7$ is equal to the value of x at e_P, obtained via the environment transformer $\lambda env.env[a \mapsto \top]$, which is associated with edge $e_P \rightarrow n7$.

Aliasing (e.g., due to pointers or reference parameters) can be handled conservatively. For example, if x and y might be aliased before the statement $x := 5$, then the corresponding environment transformer would be $\lambda env.env[x \mapsto 5][y \mapsto (5 \sqcap env(y))]$.

4. From supergraphs to "exploded" supergraphs

In this section, we show that the meet-over-all-valid-paths solution in G^* can be found by finding the "meet-over-all-realizable-paths" solution of a related problem in an "exploded" supergraph $G^\sharp$. $G^\sharp$ is obtained by pasting together graphs that represent the "pointwise" behavior of G^*'s environment-transformer functions. Representing these functions at a finer level of granularity leads to efficient dataflow-analysis algorithms because operations such as meets and compositions of functions can often be carried out by trivial, unit-cost operations on the pointwise representation.



(a) $\lambda env.env$

(b) $\lambda env.env[a \mapsto 7]$

(c) $\lambda env.env[x \mapsto -2 * env(a) + 5]$

Fig. 2. The pointwise representations for three of the environment transformers that occur in the running example program.

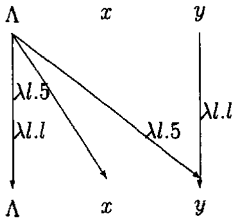


Fig. 3. The pointwise representation of the environment transformer $\lambda env.env[x \mapsto 5][y \mapsto (5 \sqcap env(y))]$.

4.1. A pointwise representation of environment transformers

One of the keys to the efficiency of our dataflow-analysis algorithm is the use of a *pointwise* representation of environment transformers. In this section, we show that every distributive environment transformer $t: Env(D,L) \xrightarrow{d} Env(D,L)$ can be represented using a directed graph whose edges are labeled by functions from L to L. For example, Fig. 2 illustrates pointwise representations for three of the environment transformers from the example program shown in Fig. 1. Fig. 3 illustrates the pointwise representation of an environment transformer that approximates aliases as described in Section 3.4.

In general, the pointwise representation of a distributive environment transformer $t: Env(D,L) \xrightarrow{d} Env(D,L)$ is a labeled graph with $2(D+1)$ nodes and at most $(D+1)^2$ edges. Each edge $d' \rightarrow d$ is annotated with a function $f_{d',d}$ from L to L. Function $f_{d',d}$ captures the effect that the value of symbol d' in the argument environment has on the value of symbol d in the result environment. Function $f_{\Lambda,d}$ is used to represent the effects on symbol d that are independent of the argument environment. For any symbol d, the value of $t(env)(d)$[5] can be determined by taking the meet of the values

[5] We assume that application associates to the left; that is, $t(env)(d)$ equals $(t(env))(d)$, not $t((env)(d))$.

of $D + 1$ individual function applications:

$$t(env)(d) = f_{\Lambda,d}(\top) \sqcap \prod_{d' \in D} f_{d',d}(env(d'))$$

Informally, we say that the "macro-function" t is represented by the "micro-functions" $f_{d',d}$.

If an edge from d' to d is labeled with the function $\lambda env.\top$, it can be omitted from the graph (and we say that d does not depend on d').

These ideas are formalized in the following definition.

Definition 4.1. Let $t: Env(D,L) \xrightarrow{d} Env(D,L)$ be an environment transformer and let Λ be a symbol not in D. The *pointwise representation* of t, denoted by $R_t: (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \rightarrow (L \xrightarrow{d} L)$, is defined by

$$R_t(d',d) \overset{\text{def}}{=} \begin{cases} \lambda l.l & d' = d = \Lambda & \text{(i)} \\ \lambda l.t(\Omega)(d) & d' = \Lambda, d \in D & \text{(ii)} \\ \lambda l.\top & d' \in D, d = \Lambda & \text{(iii)} \\ \lambda l.\top & d', d \in D \wedge \forall l.t(\Omega[d' \mapsto l])(d) = t(\Omega)(d) & \text{(iv)} \\ \lambda l.l & d', d \in D \wedge \forall l.t(\Omega[d' \mapsto l])(d) = t(\Omega)(d) \sqcap l & \text{(v)} \\ \lambda l. \left\{ \begin{matrix} \top & l = \top \\ t(\Omega[d' \mapsto l])(d) & \text{o.w.} \end{matrix} \right\} & \text{otherwise} & \text{(vi)} \end{cases}$$

Also, for a given pointwise representation $R_t: (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\}) \rightarrow (L \xrightarrow{d} L)$, the *interpretation* of R_t, $[\![R_t]\!]: Env(D,L) \xrightarrow{d} Env(D,L)$, is the distributive environment transformer defined by

$$[\![R_t]\!](env)(d) \overset{\text{def}}{=} R_t(\Lambda,d)(\top) \sqcap \prod_{d' \in D} R_t(d',d)(env(d')) \qquad (1)$$

It is easy to verify that $R_t(d',d)$ is always a distributive function.

The intuition behind the definition of R_t is that macro-function t is broken down into micro-functions as discussed above. The general case is case (vi): micro-function $R_t(d',d)$, where neither d' nor d is Λ, is the function

$$\lambda l. \begin{cases} \top & l = \top \\ t(\Omega[d' \mapsto l])(d) & \text{otherwise} \end{cases} \qquad (2)$$

This function captures the effect that the value of d' in t's argument environment has on the value of d in the result environment.

The remainder of the cases can be explained as follows:

Case (i): Case (i) is included for technical reasons so that the compositions of the micro-functions in the pointwise representations of functions t_1 and t_2 correspond to the macro-function composition $t_2 \, o \, t_1$.

Case (ii): Case (ii) captures an upper bound on the value of d that will result from an application of t. The micro-functions $R_t(\Lambda,d)$ capture the effects on d that are independent of the argument environment. These micro-functions play a role similar to the "gen" sets of gen-kill problems. (Note that these are the only non-co-strict micro-functions in Definition 4.1.)

Cases (iii) *and* (iv): The function $\lambda l.\top$ is used whenever possible:

- In case (iii), functions of the form $R_t(d', \Lambda)$ do not appear on the right-hand side of Eq. (1) (and thus their values are irrelevant).
- In case (iv), $\lambda l.\top$ is used in place of (2) when, for all l, $t(\Omega[d' \mapsto l])(d)$ is equal to $t(\Omega)(d)$ (i.e., $R_t(d', \Lambda)(\top)$), in which case $t(\Omega[d' \mapsto l])(d)$ does not contribute anything new to the right-hand side of Eq. (1).

Case (v): The identity function is used whenever possible, i.e., $\lambda l.l$ is used in place of (2) when, for all l, the right-hand side of Eq. (1) will have the same value when l is substituted for $t(\Omega[d' \mapsto l])(d)$.

Example 4.2. The general-case micro-function (case (vi) of Definition 4.1) is illustrated by the micro-function on the edge $a \rightarrow x$ in Fig. 2(c), where $t = \lambda env.env[x \mapsto -2 * env(a) + 5]$. In particular,

$$
\begin{aligned}
R_t(a,x) &= \lambda l.\begin{cases} \top & l = \top \\ t(\Omega[a \mapsto l])(x) & \text{otherwise} \end{cases} \\
&= \lambda l.\begin{cases} \top & l = \top \\ (\Omega[a \mapsto l][x \mapsto -2 * (\Omega[a \mapsto l])(a) + 5])(x) & \text{otherwise} \end{cases} \\
&= \lambda l.\begin{cases} \top & l = \top \\ (\Omega[a \mapsto l][x \mapsto -2 * l + 5])(x) & \text{otherwise} \end{cases} \\
&= \lambda l.\begin{cases} \top & l = \top \\ -2 * l + 5 & \text{otherwise} \end{cases} \\
&= \lambda l. -2 * l + 5.
\end{aligned}
$$

The last step assumes that operations $*$ and $+$ are the natural extensions of multiplication and addition that also operate on $\top$ and $\bot$.

Cases (i)–(v) are illustrated in Fig. 3, where $t = \lambda env.env[x \mapsto 5][y \mapsto (5 \sqcap env(y))]$:

- By Definition 4.1(i), $R_t(\Lambda, \Lambda) = \lambda l.l$.
- Since $t(\Omega)(x) = \Omega[x \mapsto 5][y \mapsto (5 \sqcap \Omega(y))](x) = 5$, $R_t(\Lambda, x) = \lambda l.5$ (case (ii)).
- Since $t(\Omega)(y) = \Omega[x \mapsto 5][y \mapsto (5 \sqcap \Omega(y))](y) = 5 \sqcap \Omega(y) = 5 \sqcap \top = 5$, $R_t(\Lambda, y) = \lambda l.5$ (case (ii)).
- By Definition 4.1(iii), both $R_t(x, \Lambda)$ and $R_t(y, \Lambda)$ are $\lambda l.\top$.
- Since for all l, $t(\Omega[x \mapsto l])(x) = t(\Omega)(x) = 5$, $R_t(x,x) = \lambda l.\top$ (case (iv)).
  Similarly, since for all l, $t(\Omega[y \mapsto l])(x) = t(\Omega)(x) = 5$, $R_t(y,x) = \lambda l.\top$.
  Finally, since for all l, $t(\Omega[x \mapsto l])(y) = t(\Omega)(y) = 5$, $R_t(x,y) = \lambda l.\top$.
- Since for all l, $t(\Omega[y \mapsto l])(y) = \Omega[x \mapsto 5][y \mapsto (5 \sqcap \Omega[y \mapsto l](y))](y) = 5 \sqcap l = t(\Omega)(y) \sqcap l$, $R_t(y,y) = \lambda l.l$ (case (v)).

Theorem 4.3. *For every* $t \colon Env(D,L) \xrightarrow{d} Env(D,L)$, $t = [\![R_t]\!]$.

Proof. By Definition 4.1, we have to show that for every $env \in Env(D,L)$, and $d \in D$,

$$t(env)(d) = [\![R_t]\!](env)(d) \tag{3}$$

$$= R_t(\Lambda,d)(\top) \sqcap \prod_{d' \in D} R_t(d',d)(env(d')) \tag{4}$$

First, we claim that

$$R_t(\Lambda,d)(\top) \sqcap \prod_{d' \in D} R_t(d',d)(env(d')) = t(\Omega)(d) \sqcap \prod_{d' \in D} t(\Omega[d' \mapsto env(d')])(d) \tag{5}$$

To show (5), we first show that $\sqsupseteq$ holds in (5). By Definition 4.1(ii), $R_t(\Lambda,d)(\top) = t(\Omega)(d)$, and by Definition 4.1(iv)–(vi), for every $d' \in D$,

$$R_t(d',d)(env(d')) \sqsupseteq t(\Omega[d' \mapsto env(d')])(d).$$

Therefore,

$$R_t(\Lambda,d)(\top) \sqcap \prod_{d' \in D} R_t(d',d)(env(d')) \sqsupseteq t(\Omega)(d) \sqcap \prod_{d' \in D} t(\Omega[d' \mapsto env(d')])(d)$$

We now show that $\sqsubseteq$ holds in (5). By Definition 4.1(ii), $R_t(\Lambda,d)(\top) = t(\Omega)(d)$, and by Definition 4.1(iv)–(vi), for every $d' \in D$,

$$R_t(\Lambda,d)(\top) \sqcap R_t(d',d)(env(d')) \sqsubseteq t(\Omega[d' \mapsto env(d')])(d).$$

Therefore,

$$\begin{aligned} & R_t(\Lambda,d)(\top) \sqcap \prod_{d' \in D} R_t(d',d)(env(d')) \\ & = R_t(\Lambda,d)(\top) \sqcap \prod_{d' \in D} (R_t(\Lambda,d)(\top) \sqcap R_t(d',d)(env(d'))) \\ & \sqsubseteq t(\Omega)(d) \sqcap \prod_{d' \in D} t(\Omega[d' \mapsto env(d')])(d) \end{aligned}$$

To complete the proof it is sufficient to show that

$$t(env)(d) = t(\Omega)(d) \sqcap \prod_{d' \in D} t(\Omega[d' \mapsto env(d')])(d) \tag{6}$$

This is shown by induction on k, the number of symbols in env that are not mapped to $\top$.

Basis: For $k = 0$, $env = \Omega$ and therefore all the terms of the form $\Omega[d' \mapsto env(d')]$ on the right-hand side of (6) are equal to Ω. Hence, (6) trivially holds.

Induction hypothesis: Let d be an arbitrary element of D and assume that for every $env \in Env(D,L)$ with exactly k symbols not mapped to $\top$, (6) holds for t, d, and env.

Induction step: Let $env \in Env(D,L)$ be an arbitrary environment with $k+1$ symbols not mapped to $\top$ and let us show that (6) holds for t, d, and env.

Let $d_0 \in D$, such that, $env(d_0) \neq \top$ and let $env' \stackrel{\text{def}}{=} env[d_0 \mapsto \top]$. By definition, $env = env' \sqcap \Omega[d_0 \mapsto env(d_0)]$ and therefore, since t is distributive,

$$t(env)(d) = t(env')(d) \sqcap t(\Omega[d_0 \mapsto env(d_0)])(d). \tag{7}$$

Since in *env'*, *k* symbols are not mapped to $\top$, the induction hypothesis implies that

$$\begin{aligned} t(env')(d) &= t(\Omega)(d) \sqcap \bigsqcap_{d' \in D} t(\Omega[d' \mapsto env'(d')])(d) \\ &= t(\Omega)(d) \sqcap t(\Omega[d_0 \mapsto env'(d_0)])(d) \sqcap \bigsqcap_{d' \in D - \{d_0\}} t(\Omega[d' \mapsto env(d')])(d). \end{aligned}$$

By the definition of *env'*, $env'(d_0) = \top$ and therefore $\Omega[d_0 \mapsto env'(d_0)] = \Omega$. Hence, we get

$$t(env')(d) = t(\Omega)(d) \sqcap \bigsqcap_{d' \in D - \{d_0\}} t(\Omega[d' \mapsto env(d')])(d). \tag{8}$$

The proof is completed by substituting the right-hand side of (8) for $t(env')(d)$ in (7). □

4.2. *The labeled exploded supergraph*

Definition 4.4. Let $IP = (G^*, D, L, M)$ be an IDE problem instance. The *labeled exploded supergraph* of *IP* is a directed graph $G^\sharp = (N^\sharp, E^\sharp)$ where

$$N^\sharp \stackrel{\text{def}}{=} N^* \times (D \cup \{\Lambda\})$$

and

$$E^\sharp \stackrel{\text{def}}{=} \{\langle m, d' \rangle \to \langle n, d \rangle \mid m \to n \in E^*, R_{M(m \to n)}(d', d) \neq \lambda l.\top\}.$$

Edge labels are given by a function $EdgeFn: E^\sharp \to (L \xrightarrow{d} L)$ defined to be

$$EdgeFn(\langle m, d' \rangle \to \langle n, d \rangle) \stackrel{\text{def}}{=} R_{M(m \to n)}(d', d).$$

A path *p* in $G^\sharp$ is a *realizable path* if the corresponding path in G^* is a valid path. We denote the set of realizable paths from an exploded-graph node $m^\sharp$ to an exploded-graph node $n^\sharp$ by $RP(m^\sharp, n^\sharp)$. *Same-level realizable paths*, denoted by $SLRP(m^\sharp, n^\sharp)$, are defined similarly.

Also, for all paths $p \in VP(s_{main}, n)$ and $d \in D \cup \{\Lambda\}$, we use $r(p, d)$ to denote the set of realizable paths from $\langle s_{main}, \Lambda \rangle$ to $\langle n, d \rangle$ that correspond to *p*.

Example 4.5. Fig. 4 contains the exploded supergraph for the running example program labeled with the non-identity *EdgeFn* functions.

Definition 4.6. Let $IP = (G^*, D, L, M)$ be an IDE problem instance. The *meet-over-all-realizable-paths* solution of *IP* for a given exploded node $n^\sharp \in N^\sharp$, denoted by $MRP_{n^\sharp}$, is defined as follows:

$$MRP_{n^\sharp} \stackrel{\text{def}}{=} \bigsqcap_{q \in RP(\langle s_{main}, \Lambda \rangle, n^\sharp)} PathFn(q)(\top)$$

where *PathFn* is *EdgeFn* extended to paths by composition.

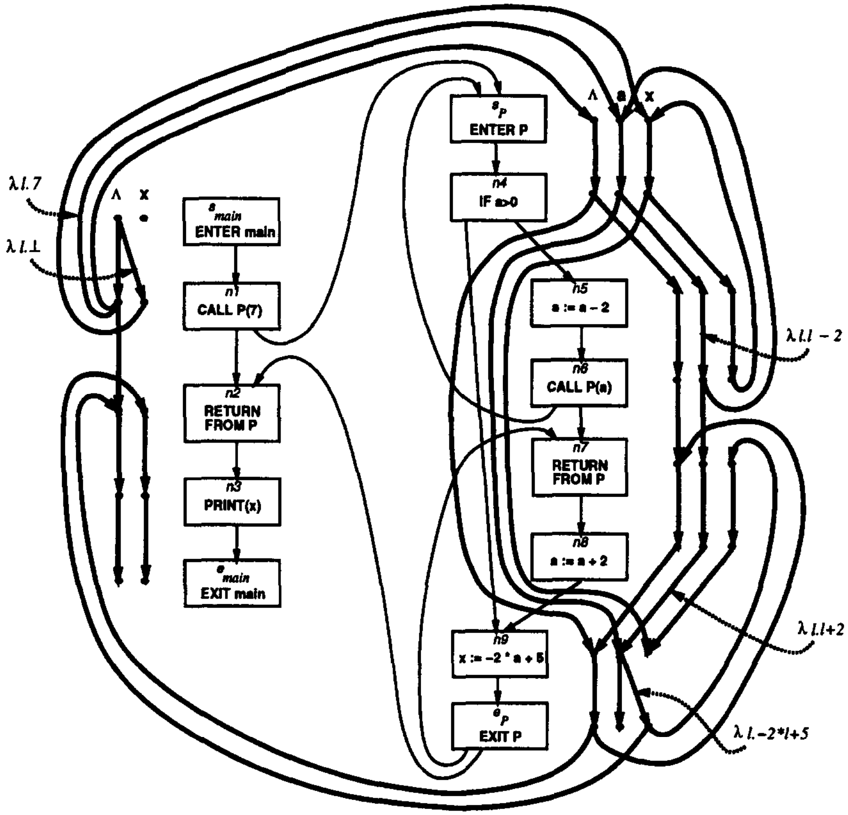


Fig. 4. The labeled exploded supergraph for the running example program for the linear-constant-propagation problem. The edge functions are all $\lambda l.l$ except where indicated.

We will show that the meet-over-all-valid-paths solution to an IDE problem can be obtained by finding the meet-over-all-realizable-paths solution in $G^\sharp$. A key step in this argument is to show that compositions of the macro-functions along paths in G^* are emulated by compositions of the micro-functions along paths in $G^\sharp$. This is captured by the following lemma.

Lemma 4.7. *For every $n \in N^*$, $d \in D$, and path $p \in VP(s_{main}, n)$,*

$$M(p)(\Omega)(d) = \bigsqcap_{r \in r(p,d)} PathFn(r)(\top) \tag{9}$$

Proof. By induction on the length of p.

Basis: For a length-0 path p, $r(p,d) = \phi$ and therefore both sides of (9) have the value $\top$.

Induction hypothesis: Assume that for a path $p = [e_1, e_2, \ldots e_j] \in VP(s_{main}, n)$ and for every $d \in D$, the lemma holds.

Induction step: Let $p' = [e_1, e_2, \ldots, e_j, e_{j+1}] \in VP(s_{main}, n)$ and let $d \in D$. We have

$$\begin{aligned}
M(p')(\Omega)(d) &= (M(e_{j+1})\, o\, M(p))(\Omega)(d) \\
&= (M(e_{j+1})(M(p)(\Omega)))(d) && \text{Definition of } o \\
&= ([\![R_{M(e_{j+1})}]\!](M(p)(\Omega)))(d) && \text{Theorem 4.3} \\
&= R_{M(e_{j+1})}(\Lambda, d)(\top) \sqcap \bigsqcap_{d' \in D} R_{M(e_{j+1})}(d', d)(M(p)(\Omega)(d')) && \text{Definition 4.1} \\
&= \begin{array}{l} R_{M(e_{j+1})}(\Lambda, d)(\top) \sqcap \\ \bigsqcap_{d' \in D} R_{M(e_{j+1})}(d', d)\left(\bigsqcap_{r \in r(p,d')} PathFn(r)(\top)\right) \end{array} && \text{Induction hypothesis} \\
&= \begin{array}{l} R_{M(e_{j+1})}(\Lambda, d)(\top) \sqcap \\ \bigsqcap_{d' \in D} \bigsqcap_{r \in r(p,d')} R_{M(e_{j+1})}(d', d)(PathFn(r)(\top)) \end{array} && \begin{array}{l}\text{Distributivity of} \\ R_{M(e_{j+1})}(d', d)\end{array} \\
&= \begin{array}{l} R_{M(e_{j+1})}(\Lambda, d)(\top) \sqcap \\ \bigsqcap_{d' \in D, R_{M(e_{j+1})}(d', d) \neq \lambda l.\top} \bigsqcap_{r \in r(p,d')} \\ \qquad R_{M(e_{j+1})}(d', d)(PathFn(r)(\top)) \end{array} \\
&= \bigsqcap_{r \in r(p',d)} PathFn(r)(\top) \qquad \square
\end{aligned}$$

We now state the theorem that is the basis for our algorithm for solving IDE problems.

Theorem 4.8. *For every* $n \in N^*$ *and* $d \in D$, $MVP_n(d) = MRP_{\langle n,d \rangle}$.

Proof. Let $p \in VP(s_{main}, n)$. Then, using Lemma 4.7 and the fact that $r(p,d) \subseteq RP(\langle s_{main}, \Lambda \rangle, \langle n, d \rangle)$,

$$\begin{aligned}
M(p)(\Omega)(d) &= \bigsqcap_{r \in r(p,d)} PathFn(r)(\top) \sqsupseteq \bigsqcap_{r \in RP(\langle s_{main}, \Lambda \rangle, \langle n,d \rangle)} PathFn(r)(\top) \\
&= MRP_{\langle n,d \rangle}
\end{aligned}$$

and therefore

$$MVP_n(d) = \bigsqcap_{p \in VP(s_{main}, n)} M(p)(\Omega)(d) \sqsupseteq MRP_{\langle n,d \rangle}$$

Now let $r_0 \in RP(\langle s_{main}, \Lambda \rangle, \langle n, d \rangle)$ and let p be the corresponding path in G^*. Then, by Lemma 4.7,

$$M(p)(\Omega)(d) = \bigsqcap_{r \in r(p,d)} PathFn(r)(\top) \sqsubseteq PathFn(r_0)(\top)$$

and therefore $MVP_n(d) \sqsubseteq MRP_{\langle n,d \rangle}$. □

The consequence of this theorem is that we can solve an IDE problem by solving the meet-over-all-realizable-paths problem on the labeled exploded supergraph.

5. An algorithm for solving IDE problems

In this section, we present an algorithm to compute the meet-over-all-valid-paths solution to a given dataflow problem instance *IP*. The input to the algorithm is the labeled exploded supergraph $G^\#$. The algorithm computes a value $val(n^\#) \in L$ for each exploded graph node $n^\# \in N^\#$. When the algorithm terminates, for all $n^\# \in N^\#$, $val(n^\#) = MRP_{n^\#}$.

The algorithm operates in two phases, which are shown in Figs. 5 and 7. In Phase I, the algorithm builds up *jump functions* (recorded in *JumpFn*) and *summary functions* (recorded in *SummaryFn*). Jump functions and summary functions are defined in terms of *edge functions* (*EdgeFn*), and other jump functions and summary functions. In Phase II, the jump functions are used to determine the actual *values* associated with nodes of the exploded graph.

5.1. Phase I

Phase I is performed by procedure ForwardComputeJumpFunctionsSLRPs, shown in Fig. 5. ForwardComputeJumpFunctionsSLRPs is a dynamic-programming algorithm that progressively computes jump functions, which are functions from L to L, for longer and longer same-level-realizable paths in $G^\#$. The jump functions to $\langle n, d\rangle$ summarize the effects of same-level realizable paths from the start node of n's procedure p to $\langle n, d\rangle$. There may be a jump function from $\langle s_p, d'\rangle$ to $\langle n, d\rangle$ for all $d' \in D \cup \{\Lambda\}$. ForwardComputeJumpFunctionsSLRPs also computes summary functions, which summarize the effects of same-level realizable paths from nodes of the form $\langle c, d'\rangle$, where c is a call node, to $\langle r, d\rangle$, where r is the corresponding return-site node.

ForwardComputeJumpFunctionsSLRPs is a worklist algorithm that computes successively better approximations to the jump and summary functions. It starts by initializing jump and summary functions to $\lambda l.\top$ (lines [1]–[4]). The worklist is initialized to $\{\langle s_{main}, \Lambda\rangle \to \langle s_{main}, \Lambda\rangle\}$, since we know that there is a length-0 path from $\langle s_{main}, \Lambda\rangle$ to $\langle s_{main}, \Lambda\rangle$ (line [5]), and $JumpFn(\langle s_{main}, \Lambda\rangle \to \langle s_{main}, \Lambda\rangle)$ is initialized to the identity function, *id* (line [6]). Fig. 6 depicts the configurations that are used by ForwardComputeJumpFunctionsSLRPs to progressively compute better approximations to jump and summary functions for longer and longer same-level realizable paths.

To reduce the amount of work performed, ForwardComputeJumpFunctionsSLRPs uses an idea similar to the "minimal-function-graph" approach [14]: Only after a jump function for a path from a node of the form $\langle s_p, d_1\rangle$ to a node of the form $\langle c, d_2\rangle$ has been processed, where c is a call on procedure q, will a path from $\langle s_q, d_3\rangle$ to $\langle s_q, d_3\rangle$ be put on the worklist — and then only if edge $\langle c, d_2\rangle \to \langle s_q, d_3\rangle$ is in $E^\#$ (lines [12]–[13]).

5.2. Phase II

Phase II is performed by procedure ComputeValues, shown in Fig. 7. In this phase, the jump functions are used to determine the actual values associated with nodes of

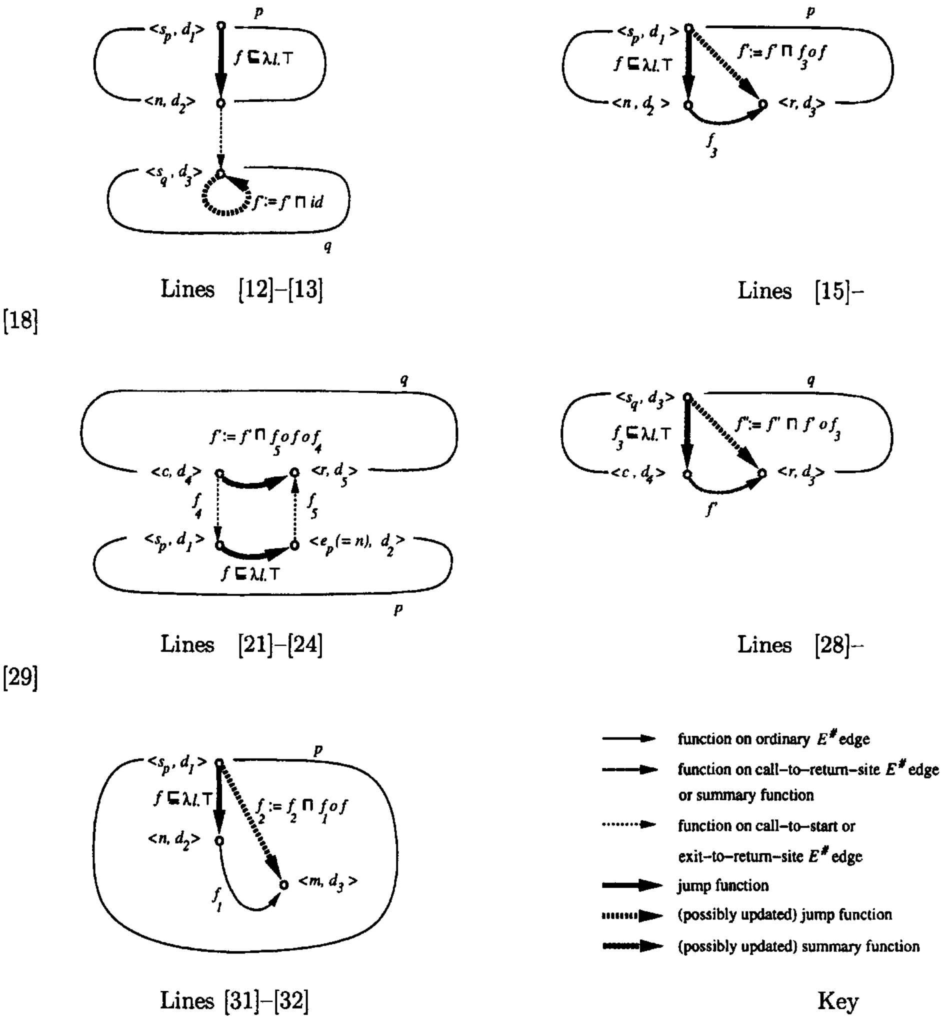
```
procedure ForwardComputeJumpFunctionsSLRPs()
     begin
[1]    for all $\langle s_p, d'\rangle$, $\langle m, d\rangle$ such that $m$ occurs in procedure $p$ and $d', d \in D \cup \{\Lambda\}$ do
[2]      $JumpFn(\langle s_p, d'\rangle \to \langle m, d\rangle) = \lambda l.\top$ od
[3]    for all corresponding call-return pairs $(c, r)$ and $d', d \in D \cup \{\Lambda\}$ do
[4]      $SummaryFn(\langle c, d'\rangle \to \langle r, d\rangle) = \lambda l.\top$ od
[5]    $PathWorkList := \{\langle s_{main}, \Lambda\rangle \to \langle s_{main}, \Lambda\rangle\}$
[6]    $JumpFn(\langle s_{main}, \Lambda\rangle \to \langle s_{main}, \Lambda\rangle) := id$
[7]    while $PathWorkList \neq \emptyset$ do
[8]       Select and remove an item $\langle s_p, d_1\rangle \to \langle n, d_2\rangle$ from $PathWorkList$
[9]       let $f = JumpFn(\langle s_p, d_1\rangle \to \langle n, d_2\rangle)$
[10]      switch($n$)
[11]         case $n$ is a call node in $p$, calling a procedure $q$:
[12]           for each $d_3$ such that $\langle n, d_2\rangle \to \langle s_q, d_3\rangle \in E^\sharp$ do
[13]             Propagate $(\langle s_q, d_3\rangle \to \langle s_q, d_3\rangle, id)$ od
[14]           let $r$ be the return-site node that corresponds to $n$
[15]           for each $d_3$ such that $e = \langle n, d_2\rangle \to \langle r, d_3\rangle \in E^\sharp$ do
[16]             Propagate$(\langle s_p, d_1\rangle \to \langle r, d_3\rangle, EdgeFn(e)\ o\ f)$ od
[17]           for each $d_3$ such that $f_3 = SummaryFn(\langle n, d_2\rangle \to \langle r, d_3\rangle) \neq \lambda l.\top$ do
[18]             Propagate$(\langle s_p, d_1\rangle \to \langle r, d_3\rangle, f_3\ o\ f)$ od endcase
[19]         case $n$ is the exit node of $p$:
[20]           for each call node $c$ that calls $p$ with corresponding return-site node $r$ do
[21]             for each $d_4, d_5$ such that $\langle c, d_4\rangle \to \langle s_p, d_1\rangle \in E^\sharp$ and $\langle e_p, d_2\rangle \to \langle r, d_5\rangle \in E^\sharp$ do
[22]               let $f_4 = EdgeFn(\langle c, d_4\rangle \to \langle s_p, d_1\rangle)$ and
[23]                 $f_5 = EdgeFn(\langle e_p, d_2\rangle \to \langle r, d_5\rangle)$ and
[24]                 $f' = (f_5\ o\ f\ o\ f_4) \sqcap SummaryFn(\langle c, d_4\rangle \to \langle r, d_5\rangle)$
[25]               if $f' \neq SummaryFn(\langle c, d_4\rangle \to \langle r, d_5\rangle)$ then
[26]                 $SummaryFn(\langle c, d_4\rangle \to \langle r, d_5\rangle) := f'$
[27]                 let $s_q$ be the start node of $c$'s procedure
[28]                 for each $d_3$ such that $f_3 = JumpFn(\langle s_q, d_3\rangle \to \langle c, d_4\rangle) \neq \lambda l.\top$ do
[29]                   Propagate$(\langle s_q, d_3\rangle \to \langle r, d_5\rangle, f'\ o\ f_3)$ od fi od od endcase
[30]         default:
[31]           for each $\langle m, d_3\rangle$ such that $\langle n, d_2\rangle \to \langle m, d_3\rangle \in E^\sharp$ do
[32]             Propagate$(\langle s_p, d_1\rangle \to \langle m, d_3\rangle, EdgeFn(\langle n, d_2\rangle \to \langle m, d_3\rangle)\ o\ f)$ od endcase
[33]      end switch od
     end

procedure Propagate($e, f$)
     begin
[34]   let $f' = f \sqcap JumpFn(e)$
[35]   if $f' \neq JumpFn(e)$ then
[36]     $JumpFn(e) := f'$
[37]     Insert $e$ into $PathWorkList$ fi
     end
```

Fig. 5. The algorithm for Phase I.

the exploded graph. Phase II consists of two subphases:

(i) An iterative algorithm is used to propagate values from start nodes to call nodes and from call nodes to start nodes. To compute a new approximation to the value at call node $\langle c, d'\rangle$ in procedure p, $JumpFn(\langle s_p, d\rangle \to \langle c, d'\rangle)$ is applied to the current approximation at node $\langle s_p, d\rangle$ (lines [7]–[10]). To compute a new approximation to


p
<s_p, d_1>
$f \sqsubseteq \lambda l.\top$
<n, d_2>
<s_q, d_3>
$f' := f \sqcap id$
q
p
<s_p, d_1>
$f \sqsubseteq \lambda l.\top$
$f' := f' \sqcap f_3 \circ f$
<n, d_2>
<r, d_3>
f_3
q
$f' := f' \sqcap f_5 \circ f \circ f_4$
<c, d_4>
<r, d_5>
f_4
f_5
<s_p, d_1>
<e_p(= n), d_2>
$f \sqsubseteq \lambda l.\top$
p
q
<s_q, d_3>
$f_3 \sqsubseteq \lambda l.\top$
$f' := f' \sqcap f \circ f_3$
<c, d_4>
<r, d_5>
f
p
<s_p, d_1>
$f \sqsubseteq \lambda l.\top$
$f_2 := f_2 \sqcap f_1 \circ f$
<n, d_2>
<m, d_3>
f_1
function on ordinary $E^\#$ edge
function on call-to-return-site $E^\#$ edge or summary function
function on call-to-start or exit-to-return-site $E^\#$ edge
jump function
(possibly updated) jump function
(possibly updated) summary function

Lines [12]–[13]

Lines [15]–

[18]

Lines [21]–[24]

Lines [28]–

[29]

Lines [31]–[32]

Key

Fig. 6. The above five diagrams show the situations handled in the indicated lines of ForwardComputeJumpFunctionsSLRPs.

the value at start node $\langle s_q, d' \rangle$, $EdgeFn(\langle n,d \rangle \to \langle s_q, d' \rangle)$ is applied to the current approximation at all nodes $\langle n,d \rangle$, where n is a call on q (lines [11]–[13]). At the end of this subphase, for all procedures p and all d, $val(\langle s_p, d \rangle) = MRP_{\langle s_p, d \rangle}$.

(ii) Values are computed for all nodes $\langle n,d \rangle$ that are neither start nor call nodes. This is done by applying $JumpFn(\langle s_p, d' \rangle \to \langle n,d \rangle)$ to $val(\langle s_p, d' \rangle)$ for all d' (where p is the procedure that contains n), and taking the meet of the resulting values (lines [15]–[17]).

Note that $val(\langle s_{main}, \Lambda \rangle)$ is initialized to $\perp$ in Phase II(i). In fact, the initial value could be anything other than $\top$; $\top$ cannot be used because then the test in PropagateValue would fail, and the algorithm would not visit all nodes of the form $\langle n, \Lambda \rangle$. The

```
procedure ComputeValues()
  begin
    /* Phase II(i) */
    for each $n^\sharp \in N^\sharp$ do $val(n^\sharp) := \top$ od
    $val(\langle s_{main}, \Lambda \rangle) := \bot$
    $NodeWorkList := \{\langle s_{main}, \Lambda \rangle\}$
    while $NodeWorkList \neq \emptyset$ do
      Select and remove an exploded-graph node $\langle n, d \rangle$ from $NodeWorkList$
      switch($n$)
        case $n$ is the start node of $p$:
          for each $c$ that is a call node inside $p$ do
            for each $d'$ such that $f' = JumpFn(\langle n, d \rangle \rightarrow \langle c, d' \rangle) \neq \lambda l.\top$ do
              PropagateValue($\langle c, d' \rangle$, $f'(val(\langle s_p, d \rangle))$) od od endcase
        case $n$ is a call node in $p$, calling a procedure $q$:
          for each $d'$ such that $\langle n, d \rangle \rightarrow \langle s_q, d' \rangle \in E^\sharp$ do
            PropagateValue($\langle s_q, d' \rangle$, $EdgeFn(\langle n, d \rangle \rightarrow \langle s_q, d' \rangle)(val(\langle n, d \rangle))$) od endcase
      end switch od
    /* Phase II(ii) */
    for each node $n$, in a procedure $p$, that is not a call or a start node do
      for each $d', d$ such that $f' = JumpFn(\langle s_p, d' \rangle \rightarrow \langle n, d \rangle) \neq \lambda l.\top$ do
        $val(\langle n, d \rangle) := val(\langle n, d \rangle) \sqcap f'(val(\langle s_p, d' \rangle))$ od od
  end

procedure PropagateValue($n^\sharp$, $v$)
  begin
    let $v' = v \sqcap val(n^\sharp)$
    if $v' \neq val(n^\sharp)$ then
      $val(n^\sharp) := v'$
      Insert $n^\sharp$ into $NodeWorkList$ fi
  end
```

Fig. 7. The algorithm for Phase II.

particular non-$\top$ value is irrelevant: that value is propagated to all nodes of the form $\langle n, \Lambda \rangle$, but because the function on an edge from one of these nodes to a non-Λ node $m^\sharp$ is always a constant function (see Definition 4.1), the value at $\langle n, \Lambda \rangle$ cannot affect the value at $m^\sharp$.

Example 5.1. When applied to the exploded graph of Fig. 4, our algorithm discovers that x has the constant value -9 at node $n3$ (the print statement in the main procedure), and that a does *not* have a constant value at node s_P (the start node of procedure P). During Phase I, the algorithm computes the following relevant jump and summary functions:

$$JumpFn(\langle s_P, a \rangle \rightarrow \langle n6, a \rangle) = \lambda l. l - 2$$

$$JumpFn(\langle s_P, a \rangle \rightarrow \langle e_P, x \rangle) = \lambda l. -2 * l + 5$$

$$SummaryFn(\langle n1, \Lambda \rangle \rightarrow \langle n2, x \rangle) = \lambda l. -9$$

$$JumpFn(\langle s_{main}, \Lambda \rangle \rightarrow \langle n2, x \rangle) = \lambda l. -9$$

$$JumpFn(\langle s_{main}, \Lambda \rangle \rightarrow \langle n3, x \rangle) = \lambda l. -9$$

During Phase II(i), values are propagated as follows to discover that a is not constant at node s_P:

$$val(\langle s_{main}, \Lambda\rangle) = \bot$$
$$val(\langle n1, \Lambda\rangle) = \bot$$
$$val(\langle s_P, a\rangle) = 7$$
$$val(\langle n6, a\rangle) = 5$$
$$val(\langle s_P, a\rangle) = 5 \sqcap 7 = \bot$$

During Phase II(ii), $JumpFn(\langle s_{main}, \Lambda\rangle \rightarrow \langle n3, x\rangle)$ is applied to $val(\langle s_{main}, \Lambda\rangle)$, producing

$$val(\langle n3, x\rangle) = -9$$

5.3. Termination and cost issues

It is possible to prove the partial correctness of the algorithm given in Sections 5.1 and 5.2 (i.e., if the algorithm finishes, then for every exploded-graph node $n^{\sharp} \in N^{\sharp}$, $val(n^{\sharp}) = MRP_{n^{\sharp}}$).

Theorem 5.2 (Partial Correctness of the Algorithm). *If ComputeValues terminates, then for every node $m \in N^*$ and $d \in D$, $val(\langle m, d\rangle) = MRP_{\langle m,d\rangle}$.*

The algorithm does not terminate for all IDE problems; however, it does terminate for all copy-constant-propagation problems, all linear-constant-propagation problems, and, in general, for all problems for which the space F of micro-functions contains no infinite decreasing chains. (Note that it is possible to construct infinite decreasing chains even in certain distributive variants of constant propagation [28, p. 206].)

The cost of the algorithm is dominated by the cost of Phase I. This phase can be carried out particularly efficiently if there exists a way of representing the micro-functions such that certain operations on micro-functions can be computed in unit time.

These termination and cost issues motivate the following definition.

Definition 5.3. A class of micro-functions $F \subseteq L \xrightarrow{d} L$ has an *efficient representation* if

- $id \in F$ and F is closed under functional meet and composition.
- F has a finite height (under the pointwise ordering).
- There is a representation scheme for F with the following properties:
  Apply: Given a representation for a function $f \in F$, for every $l \in L$, $f(l)$ can be computed in constant time.[6]

[6] We assume a uniform-cost measure, rather than a logarithmic-cost measure; e.g., operations on integers can be performed in constant time.

Composition: Given the representations for any two functions $f_1, f_2 \in F$, a representation for the function $f_1 \, o \, f_2 \in F$ can be computed in constant time.

Meet: Given the representations for any two functions $f_1, f_2 \in F$, a representation for the function $f_1 \sqcap f_2 \in F$ can be computed in constant time.

EQU: Given the representations for any two functions $f_1, f_2 \in F$, it is possible to test in constant time whether $f_1 = f_2$.

Storage: There is a constant bound on the storage needed for the representation of any function $f \in F$.

An IDE problem instance $IP = (G^*, D, L, M)$ is *efficiently representable* if for every $e \in E^*$, and $d', d \in D$, $R_{M(e)}(d', d) \in F$ for some class of functions F that has an efficient representation.

Note that in the above definition we do not impose any restrictions on $R_{M(e)}(d', d)$ when either d' or d is Λ. This is based on the assumption that the constant functions and the identity function can always be represented in an efficient manner. (Similarly, we assume that $\lambda l.\top$ can always be represented in an efficient manner.)

In describing the cost of the algorithm it is convenient to introduce the notions of *jump edge* and *summary edge*. A jump edge is a pair of exploded-graph nodes whose jump function is not equal to $\lambda l.\top$; likewise, a summary edge is a pair of exploded-graph nodes whose summary function is not equal to $\lambda l.\top$.

The source of a jump edge is a node of the form $\langle s_p, d \rangle$, where s_p is the start node of some procedure p; thus, there can be at most $D + 1$ jump-edge sources in each procedure. Each iteration of Phase I extends a known jump edge by composing it with (the function of) either an $E^\#$ edge or a summary edge. There are at most $O(ED^2)$ such edges. Because each edge e can be used in the operation "extend a jump edge along edge e" once for every jump-edge source, there are at most $O(ED^3)$ such composition steps.

For each jump edge and summary edge from an exploded node $\langle n, \Lambda \rangle$, the jump-function value can change at most height-of-L times. Similarly, jump edges and summary edges emanating from other exploded nodes $\langle n, d \rangle$, $d \in D$, can change at most height of F times. Consequently, the total cost of Phase I, and thus of the entire algorithm, is bounded by $O(ED^3)$ (where the constant of proportionality depends on the heights of L and F).

In the case of both copy and linear-constant propagation, the size of D is bounded by MaxVisible (the maximum number of variables visible in any procedure of the program), and the height of L is 3. For copy-constant propagation, the height of F is 1; for linear-constant propagation, the height of F is 4 (see Section 5.4 below). Consequently, our techniques solve all instances of these constant-propagation problems in time $O(E \, \text{MaxVisible}^3)$.

5.4. Some efficiently representable IDE problems

5.4.1. Finite distributive subset problems

The IDE framework generalizes a class of interprocedural dataflow-analysis problems that we have treated in previous work. We call these problems the *interprocedural, finite, distributive, subset problems*, or *IFDS problems*, for short. In IFDS problems, the dataflow facts form a finite set U, and the dataflow functions (which are of type $2^U \to 2^U$) distribute over the meet operator (either union or intersection) [25, 24, 13]. The IFDS problems include all *locally separable* problems – the interprocedural versions of classical "bit-vector" or "gen-kill" problems (e.g., reaching definitions, available expressions, and live variables) – as well as non-locally-separable problems such as truly-live variables [11], copy-constant propagation [10, p. 660], and possibly-uninitialized variables [25, 24, 13].

Every IFDS problem can be treated as an IDE problem by representing the set of dataflow facts as an environment that corresponds to the set's characteristic function: Suppose U is the finite set of dataflow facts, and suppose the meet operation is $\cup$.[7] The meet semi-lattice 2^U can be represented as $Env(U, \{\bot, \top\})$ where $\bot \sqsubseteq \top$. If $env \in Env(U, \{\bot, \top\})$ represents set $S \in 2^U$, then $env(u) = \bot$ iff $u \in S$. For example, the maximum environment $\lambda u.\top$ represents the set $\emptyset$, the environment $\lambda u.\top[x \mapsto \bot]$ represents the set $\{x\}$, and the minimum environment $\lambda u.\bot$ represents the set U.

When IFDS problems are treated as IDE problems, the only micro-functions that arise are id and $\lambda l.\bot$. All of the occurrences of micro-function $\lambda l.\bot$ are associated with edges of the form $\langle m, \Lambda\rangle \to \langle n, d\rangle$. The only functions on "non-Λ" edges are identity functions. Since $id \circ id = id$ and $id \sqcap id = id$, the class $I = \{id\}$ is trivially a class of functions that has an efficient representation.

5.4.2. Copy-constant propagation

In copy-constant propagation, the micro-functions that arise are either id or of the form $\lambda l.c$, where c is either a manifest constant that appears somewhere in the program or $\bot$.[8] However, all of the constant functions $\lambda l.c$ are associated with edges of the form $\langle m, \Lambda\rangle \to \langle n, d\rangle$. Thus, the only functions on "non-Λ" edges are identity functions, so again we are dealing with the class $I = \{id\}$, which is trivially a class of functions that has an efficient representation.

5.4.3. Linear-constant propagation

Linear-constant propagation can be handled using the set of functions $F_{lc} = \{\lambda l.(a * l + b) \sqcap c \mid a \in Z - \{0\}, b \in Z, \text{and } c \in Z_\bot^\top\}$. (The functions where $a = 0$ are the constant functions, and, as in copy-constant propagation, these are all associated with

[7] IFDS problems in which the meet operator is intersection can be handled by transforming them to a complementary union problem.

[8] Although copy-constant propagation can be handled as an IFDS problem — and hence encoded as an IDE problem with only the functions id and $\lambda l.\bot$ as described in Section 5.4.1 — it is far more efficient to treat it directly as an IDE problem. (See the discussion in Section 8.1.)

"Λ" edges.) Every function $f \in F_{lc}$ can be represented by a triple (a, b, c), where $a \in Z - \{0\}$, $b \in Z$, $c \in Z_{\perp}^{\top}$, and

$$f = \lambda l. \begin{cases} \top & l = \top \\ (a * l + b) \sqcap c & \text{otherwise.} \end{cases}$$

The third component c is needed so that the meet of two functions can be represented. For example, consider the code fragment

```
    if ··· then
a :     $y := 5 * x - 7$
    else
b :     $y := 3 * x + 1$
    fi
c : ···
```

Variable y is only constant at c when the initial value of x is 4, and in this case y's value is 13. Micro-function $R_{M(a \to c)}(x, y)$, the micro-function into $\langle c, y \rangle$ from the then-branch, is $\lambda l.5 * l - 7$, which is represented by $(5, -7, \top)$. Micro-function $R_{M(b \to c)}(x, y)$, the micro-function into $\langle c, y \rangle$ from the else-branch, is $\lambda l.3 * l + 1$, which is represented by $(3, 1, \top)$. Therefore, $R_{M(a \to c)}(x, y) \sqcap R_{M(b \to c)}(x, y)$ is equal to the function

$$\lambda l. \begin{cases} 13 & l = 4 \\ \perp & \text{otherwise} \end{cases}$$

which is also equal to the function $\lambda l.(5 * l - 7) \sqcap 13$. It is the latter way of expressing the function that corresponds to a triple, namely $(5, -7, 13)$.

F_{lc} has an efficient representation because:

- $id \in F_{lc}$ $(a = 1,\ b = 0,\ c = \top)$
- Longest chains in F_{lc} have the form: $\lambda l.\top \sqsupset \lambda l.(a * l + b) \sqsupset \lambda l.(a * l + b) \sqcap c \sqsupset \lambda l.\perp$, for some $a, b, c \in Z$.
- The four representation requirements are met:

  Apply: Trivial.

  Meet:

$$(a_1, b_1, c_1) \sqcap (a_2, b_2, c_2) = \begin{cases} (a_1, b_1, c_1 \sqcap c_2) & a_1 = a_2, b_1 = b_2 \\ (a_1, b_1, c) & c = (a_1 * l_0 + b_1) \sqcap c_1 \sqcap c_2, \\ & \text{where } l_0 = (b_1 - b_2)/(a_2 - a_1) \in Z \\ (1, 0, \perp) & \text{otherwise} \end{cases}$$

  Composition: $(a_1, b_1, c_1) o (a_2, b_2, c_2) = ((a_1 * a_2), (a_1 * b_2 + b_1), ((a_1 * c_2 + b_1) \sqcap c_1))$. Here it is assumed that $x * \top = \top * x = x + \top = \top + x = \top$ for $x \in Z_{\perp}^{\top}$ and that $x * \perp = \perp * x = x + \perp = \perp + x = \perp$ for $x \in Z_{\perp}$.

  EQU: All representations except that of $\lambda l.\perp$ are unique. Any two triples in which $c = \perp$ represent $\lambda l.\perp$. However, equality can still be tested in unit time.

Linear-constant propagation can be also performed on real numbers $R_{\perp}^{\top}$. In this case, the meet operation is slightly simpler because there is no need to test whether $a_2 - a_1$ divides $b_1 - b_2$ evenly – only that $a_2 \neq a_1$ if $b_2 \neq b_1$.

6. A demand dataflow-analysis algorithm

In this section, we give a demand algorithm for the IDE framework. The demand algorithm finds the value for a given symbol $\overline{d} \in D$ at a given supergraph node $\overline{n} \in N^*$. The demand algorithm is similar to the exhaustive algorithm of Section 5. However, in the demand algorithm, the traversals of $G^\sharp$ used to compute jump and summary functions are backwards (i.e., edges are traversed from target to source). Furthermore, whereas in the exhaustive algorithm all jump edges have sources of the form $\langle s_p, d\rangle$, in the demand algorithm there are two different kinds of jump edges:

- In procedure BackwardComputeJumpFunctions, the target of every jump edge generated is the demand node $\langle \overline{n}, \overline{d}\rangle$. These jump edges, which are recorded in the table *JumpFnToQuery*, summarize how the dataflow value at a given exploded node affects the value at $\langle \overline{n}, \overline{d}\rangle$.
- In procedure BackwardComputeJumpFunctionsSLRPs, all the jump edges generated have targets of the form $\langle e_p, d\rangle$. These jump edges, which are recorded in the table *JumpFn*, summarize the effects of same-level realizable paths from a node $\langle n, d'\rangle$ to $\langle e_p, d\rangle$, where p is the procedure containing n.

Given a demand for the dataflow value at exploded node $\langle \overline{n}, \overline{d}\rangle$, $MRP_{\langle \overline{n}, \overline{d}\rangle}$ is computed by procedure ComputeExplodedNodeValue, shown in Fig. 8, which has two phases:

(i) The jump functions in *JumpFnToQuery* are computed by the procedure BackwardComputeJumpFunctions, shown in Fig. 9, during a backwards traversal of $G^\sharp$.

(ii) The meet-over-all-realizable-paths values are computed by the procedure ComputeValuesForVisitedNodes, shown in Fig. 11. In particular, at the end of this procedure, $val(\langle \overline{n}, \overline{d}\rangle) = MRP_{\langle \overline{n}, \overline{d}\rangle}$.

The demand algorithm is a caching algorithm, i.e., the values of *JumpFn*, SummaryFn, *val*, and *NodesWithKnownValues* are accumulated across different calls to ComputeExplodedNodeValue. We maintain the invariant that for exploded nodes in the *NodesWithKnownValues* set, the meet-over-all-realizable-paths value is already stored in *val*.

The procedure BackwardComputeJumpFunctions, shown in Fig. 9, is a dynamic-programming algorithm that computes jump functions from exploded nodes to the demand node $\langle \overline{n}, \overline{d}\rangle$, for increasingly longer paths. On every iteration of the while loop in lines [5]–[25], a node $\langle n, d\rangle$ is removed from the worklist, and procedure Visit is invoked to process some predecessor $n^\sharp$ of $\langle n, d\rangle$. If the meet-over-all-realizable-paths value of $n^\sharp$ is known (i.e., $n^\sharp$ is in *NodesWithKnownValues*), then $n^\sharp$ is inserted into the set *SourceNodesRelevantToQuery*. (In phase (ii), procedure ComputeValuesForVisitedNodes starts from nodes in *SourceNodesRelevantToQuery* and goes forward, computing values for successors.) If the meet-over-all-realizable-paths value of $n^\sharp$ is not yet known, a better approximation to *JumpFnToQuery*$(n^\sharp)$ is computed (lines [31]–[34]). If *JumpFnToQuery*$(n^\sharp)$ changes, then $n^\sharp$ is placed into *NodeWorkList* to be processed later in the main loop of BackwardComputeJumpFunctions. The node set *VisitedNodes* accumulates the exploded nodes that have been processed.

```
declare
  $G^* = (N^*, E^*)$: global exploded supergraph
  JumpFn: global table of jump functions /* Preserved across calls */
          initialization: for all $\langle s_p, d' \rangle$, $\langle m, d \rangle$ such that $m$ occurs in procedure $p$ and $d', d \in D \cup \{$
                                $JumpFn(\langle s_p, d' \rangle \to \langle m, d \rangle) = \lambda l.\top$
                              od
  SummaryFn: global table of jump functions /* Preserved across calls */
          initialization: for all corresponding call-return pairs $(c, r)$ and $d', d \in D \cup \{\Lambda\}$ do
                                $SummaryFn(\langle c, d' \rangle \to \langle r, d \rangle) = \lambda l.\top$
                              od
  val: global table of node values /* Preserved across calls */
          initialization: for all $n \in N^*$ do
                                $val(\langle n, \Lambda \rangle) := \bot$
                                for all $d \in D$ do
                                  $val(\langle n, d \rangle) := \top$
                              od od
  NodesWithKnownValues: global node set /* Preserved across calls */
          initialization: $NodesWithKnownValues := \{\langle n, \Lambda \rangle \mid n \in N^*\}$
  JumpFnToQuery: global table of jump functions
          initialization: for all $n^\sharp \in N^\sharp$ do
                                $JumpFnToQuery(n^\sharp) := \lambda l.\top$
                              od
  PathWorkList: global set, initially empty
  NodeWorkList, SourceNodesRelevantToQuery, VisitedNodes: global node set, initially empty

procedure ComputeExplodedNodeValue($\langle \overline{n}, \overline{d} \rangle$)
begin
  BackwardComputeJumpFunctions($\langle \overline{n}, \overline{d} \rangle$)
  ComputeValuesForVisitedNodes()
  for all $n^\sharp \in VisitedNodes$ do
    $JumpFnToQuery(n^\sharp) := \lambda l.\top$
  od
end
```

Fig. 8. The demand algorithm.

The procedure BackwardComputeJumpFunctions employs the procedure BackwardComputeJumpFunctionsSLRPs to compute summary edges on demand. BackwardComputeJumpFunctionsSLRPs is the “dual” of ForwardComputeJumpFunctionsSLRPs, which appears in Fig. 5. BackwardComputeJumpFunctionsSLRPs starts from the exit node of a procedure and progressively computes jump functions for longer and longer same-level realizable paths leading to the exit node.

Unlike ForwardComputeJumpFunctionsSLRPs, BackwardComputeJumpFunctionsSLRPs is able to make use of a technique for “short-circuiting” the computation of summary functions: Because $\lambda l.\bot$ is the least element of the domain of micro-functions, if BackwardComputeJumpFunctionsSLRPs ever discovers that function on a jump edge whose source is of the form $\langle n, \Lambda \rangle \to \langle e_p, d_1 \rangle$, there is no need to process any more jump edges to node $\langle e_p, d_1 \rangle$. Therefore, on discovering such an edge, BackwardComputeJumpFunctionsSLRPs inserts the jump function $\lambda l.\bot$ into $JumpFn(\langle s_p, \Lambda \rangle \to \langle e_p, d_1 \rangle)$ and into the worklist (lines [28]–[31]). Furthermore, when a jump edge

```
procedure BackwardComputeJumpFunctions($\langle \bar{n}, \bar{d} \rangle$)
    begin
[1]     $SourceNodesRelevantToQuery := \emptyset$
[2]     $VisitedNodes := \emptyset$
[3]     $NodeWorkList := \{\langle \bar{n}, \bar{d} \rangle\}$
[4]     $JumpFnToQuery(\langle \bar{n}, \bar{d} \rangle) := id$
[5]     while $NodeWorkList \neq \emptyset$ do
[6]         Select and remove an exploded-graph node $\langle n, d \rangle$ from $NodeWorkList$
[7]         let $f = JumpFnToQuery(\langle n, d \rangle)$
[8]         switch($n$)
[9]             case $n$ is a return-site node of a call node $c$ in $p$, calling a procedure $q$:
[10]                $PathWorkList := \emptyset$
[11]                for each $d'$ such that $\langle e_q, d' \rangle \rightarrow \langle n, d \rangle \in E^\sharp$ do
[12]                    Propagate($\langle e_q, d' \rangle \rightarrow \langle e_q, d' \rangle$, $id$) od
[13]                BackwardComputeJumpFunctionsSLRPs()
[14]                for each $d'$ such that $e = \langle c, d' \rangle \rightarrow \langle n, d \rangle \in E^\sharp$ do
[15]                    Visit($\langle c, d' \rangle$, $\langle \bar{n}, \bar{d} \rangle$, $f \circ EdgeFn(e)$) od
[16]                for each $d'$ such that $f' = SummaryFn(\langle c, d' \rangle \rightarrow \langle n, d \rangle) \neq \lambda l.\top$ do
[17]                    Visit($\langle c, d' \rangle$, $\langle \bar{n}, \bar{d} \rangle$, $f \circ f'$)) od endcase
[18]            case $n$ is the start node of $p$:
[19]                for each call node $c$ that calls $p$ do
[20]                    for each $d'$ such that $e = \langle c, d' \rangle \rightarrow \langle n, d \rangle \in E^\sharp$ do
[21]                    Visit($\langle c, d' \rangle$, $\langle \bar{n}, \bar{d} \rangle$, $f \circ EdgeFn(e)$) od endcase
[22]            default:
[23]                for each $e = \langle m, d' \rangle$ such that $\langle m, d' \rangle \rightarrow \langle n, d \rangle \in E^\sharp$ do
[24]                    Visit($\langle m, d' \rangle$, $\langle \bar{n}, \bar{d} \rangle$, $f \circ EdgeFn(e)$) od endcase
[25]        end switch od
    end

procedure Visit($n^\sharp$, $\langle \bar{n}, \bar{d} \rangle$, $f$)
    begin
[26]    if $n^\sharp \in NodesWithKnownValues$ then
[27]      Insert $n^\sharp$ into $SourceNodesRelevantToQuery$
[28]    else
[29]      if $n^\sharp \notin VisitedNodes$ then
[30]        Insert $n^\sharp$ into $VisitedNodes$ fi
[31]      let $f' = f \sqcap JumpFnToQuery(n^\sharp)$
[32]      if $f' \neq JumpFnToQuery(n^\sharp)$ then
[33]        $JumpFnToQuery(n^\sharp) := f'$
[34]        Insert $n^\sharp$ into $NodeWorkList$ fi fi
    end
```

Fig. 9. Phase I of the demand algorithm. (Auxiliary procedure Propagate is given in Fig. 10.)

$\langle n, d_2 \rangle \rightarrow \langle e_p, d_1 \rangle$ is taken out of the worklist (line [4]), it is processed only if it is itself of the form $\langle s_p, \Lambda \rangle \rightarrow \langle e_p, d_1 \rangle$, or if $JumpFn(\langle s_p, \Lambda \rangle \rightarrow \langle e_p, d_1 \rangle) \neq \lambda l.\bot$.

Procedure ComputeValuesForVisitedNodes, shown in Fig. 11, computes meet-over-all-realizable-paths values in a manner similar to procedure ComputeValues of Fig. 7. However, there are a number of differences:

- ComputeValuesForVisitedNodes starts from the set of nodes *SourceNodesRelevant ToQuery*, rather than from the single exploded node $\langle s_{main}, \Lambda \rangle$.

```
procedure BackwardComputeJumpFunctionsSLRPs()
    begin
      while $PathWorkList \neq \emptyset$ do
         Select and remove an item $\langle n, d_2\rangle \to \langle e_p, d_1\rangle$ from $PathWorkList$
         let $f = JumpFn(\langle n, d_2\rangle \to \langle e_p, d_1\rangle)$
         if $JumpFn(\langle s_p, \Lambda\rangle \to \langle e_p, d_1\rangle) \neq \lambda l.\bot$ or $n = s_p$ and $d_2 = \Lambda$ then
           switch($n$)
              case $n$ is a return-site node of a call $c$ in $p$, calling a procedure $q$:
                 for each $d_3$ such that $\langle e_q, d_3\rangle \to \langle n, d_2\rangle \in E^\sharp$ do
                    Propagate($\langle e_q, d_3\rangle \to \langle e_q, d_3\rangle, id$) od
                 for each $d_3$ such that $e = \langle c, d_3\rangle \to \langle n, d_2\rangle \in E^\sharp$ do
                    Propagate($\langle c, d_3\rangle \to \langle e_p, d_1\rangle, f \circ EdgeFn(e)$) od
                 for each $d_3$ such that $f_3 = SummaryFn(\langle c, d_3\rangle \to \langle n, d_2\rangle) \neq \lambda l.\top$ do
                    Propagate($\langle c, d_3\rangle \to \langle e_p, d_1\rangle, f \circ f_3$) od endcase
              case $n$ is the start node of $p$:
                 for each call node $c$ in $q$ that calls $p$ with corresponding return-site node $r$ do
                    for each $d_4, d_5$ such that $\langle c, d_5\rangle \to \langle n, d_2\rangle \in E^\sharp$ and $\langle e_p, d_1\rangle \to \langle r, d_4\rangle \in E^\sharp$ do
                       let $f_5 = EdgeFn(\langle c, d_5\rangle \to \langle n, d_2\rangle)$ and
                           $f_4 = EdgeFn(\langle e_p, d_1\rangle \to \langle r, d_4\rangle)$ and
                           $f' = (f_4 \circ f \circ f_5) \sqcap SummaryFn(\langle c, d_5\rangle \to \langle r, d_4\rangle)$
                       if $f' \neq SummaryFn(\langle c, d_5\rangle \to \langle r, d_4\rangle)$ then
                         $SummaryFn(\langle c, d_5\rangle \to \langle r, d_4\rangle) := f'$
                         for each $d_3$ such that $f_3 = JumpFn(\langle r, d_4\rangle \to \langle e_q, d_3\rangle) \neq \lambda l.\top$ do
                            Propagate($\langle c, d_5\rangle \to \langle e_q, d_3\rangle, f_3 \circ f'$) od fi od od endcase
              default:
                 for each $e = \langle m, d_3\rangle$ such that $\langle m, d_3\rangle \to \langle n, d_2\rangle \in E^\sharp$ do
                    Propagate($\langle m, d_3\rangle \to \langle e_p, d_1\rangle, f \circ EdgeFn(e)$) od endcase
           end switch od fi
    end
procedure Propagate($\langle n, d_2\rangle \to \langle e_p, d_1\rangle, f$)
    begin
      let $f' = f \sqcap JumpFn(\langle n, d_2\rangle \to \langle e_p, d_1\rangle)$
      if $f' = \lambda l.\bot$ and $d_2 = \Lambda$ then $n := s_p$ fi
      if $f' \neq JumpFn(\langle n, d_2\rangle \to \langle e_p, d_1\rangle)$ then
        $JumpFn(\langle n, d_2\rangle \to \langle e_p, d_1\rangle) := f'$
        Insert $\langle n, d_2\rangle \to \langle e_p, d_1\rangle$ into $PathWorkList$ fi
    end
```

Fig. 10. The algorithm to compute jump functions for same-level realizable paths on demand.

- ComputeValuesForVisitedNodes only computes values for the nodes in *Visited Nodes*. This is done in order to decrease the running time for processing a single demand.
- ComputeValuesForVisitedNodes involves only one phase. In contrast, ComputeValues has two phases: in the first phase it computes meet-over-all-realizable-paths values for all call and start nodes; in the second phase it computes meet-over-all-realizable-paths values for all other nodes.

Example 6.1. Consider the call ComputeExplodedNodeValue($\langle n3, x\rangle$) for the exploded graph shown in Fig. 4. The following jump and summary functions are computed by

```
procedure ComputeValuesForVisitedNodes()
    begin
        $NodeWorkList := SourceNodesRelevantToQuery$
        while $NodeWorkList \neq \emptyset$ do
            Select and remove an exploded-graph node $\langle n, d\rangle$ from $NodeWorkList$
            let $v = val(\langle n, d\rangle)$
            switch($n$)
                case $n$ is a call on $q$ in $p$ with a corresponding return-site node $r$:
                    for each $d'$ such that $e = \langle n, d\rangle \rightarrow \langle s_q, d'\rangle \in E^\sharp \wedge \langle s_q, d'\rangle \in VisitedNodes$ do
                        PropagateValue $(\langle s_q, d'\rangle, EdgeFn(e)(v))$ od
                    for each $d'$ such that $e = \langle n, d\rangle \rightarrow \langle r, d'\rangle \in E^\sharp \wedge \langle r, d'\rangle \in VisitedNodes$ do
                        PropagateValue$(\langle r, d'\rangle, EdgeFn(e)(v))$ od
                    for each $d'$ such that $f = SummaryFn(\langle n, d\rangle \rightarrow \langle r, d'\rangle) \neq \lambda l.\top \wedge \langle r, d'\rangle \in VisitedNode$
                        PropagateValue$(\langle r, d'\rangle, f(v))$ od endcase
                case $n = e_p$: skip endcase
                default:
                    for each $\langle m, d'\rangle$ such that $e = \langle n, d\rangle \rightarrow \langle m, d'\rangle \in E^\sharp \wedge \langle m, d'\rangle \in VisitedNodes$ do
                        PropagateValue$(\langle m, d'\rangle, EdgeFn(e)(v))$ od endcase
            end switch od
        $NodesWithKnownValues := NodesWithKnownValues \cup VisitedNodes$
    end

procedure PropagateValue($n^\sharp, v$)
    begin
        let $v' = v \sqcap val(n^\sharp)$
        if $v' \neq val(n^\sharp)$ then
            $val(n^\sharp) := v'$
            Insert $n^\sharp$ into $NodeWorkList$ fi
    end
```

Fig. 11. Phase II of the demand algorithm.

BackwardComputePathFunctions:

$$JumpFnToQuery(\langle n3, x\rangle) = \lambda l.l$$

$$JumpFnToQuery(\langle n2, x\rangle) = \lambda l.l$$

$$JumpFn(\langle e_p, x\rangle \rightarrow \langle e_p, x\rangle) = \lambda l.l$$

$$JumpFn(\langle n9, a\rangle \rightarrow \langle e_p, x\rangle) = \lambda l.-2*l+5$$

$$JumpFn(\langle n8, a\rangle \rightarrow \langle e_p, x\rangle) = \lambda l.-2*(l+2)+5 = \lambda l.-2*l+1$$

$$JumpFn(\langle n7, a\rangle \rightarrow \langle e_p, x\rangle) = \lambda l.-2*l+1$$

$$JumpFn(\langle n6, a\rangle \rightarrow \langle e_p, x\rangle) = \lambda l.-2*l+1$$

$$JumpFn(\langle n5, a\rangle \rightarrow \langle e_p, x\rangle) = \lambda l.-2*l+5$$

$$JumpFn(\langle n4, a\rangle \rightarrow \langle e_p, x\rangle) = \lambda l.-2*l+5$$

$$JumpFn(\langle s_p, a\rangle \rightarrow \langle e_p, x\rangle) = \lambda l.-2*l+5$$

$$SummaryFn(\langle n1, \Lambda\rangle \rightarrow \langle n2, x\rangle) = \lambda l.-9$$

$$JumpFnToQuery(\langle n1, \Lambda\rangle) = \lambda l.-9$$

The following values are computed by ComputeValuesForVisitedNodes

$$val(n2,x) = -9$$
$$val(n3,x) = -9$$

The reader may wonder why ComputeValuesForVisitedNodes is called to compute values for *all* visited nodes, when $MRP_{\langle \bar{n},\bar{d}\rangle}$ can simply be computed as

$$\prod_{n^{\#} \in SourceNodesRelevantToQuery} JumpFnToQuery(n^{\#})(val(n^{\#})) \qquad (10)$$

at the end of BackwardComputeJumpFunctions. This simpler computation can be performed if the goal is an algorithm tailored to the task of answering a *single* demand. The algorithm as presented is tailored for better performance on a *sequence* of demands: Procedure ComputeValuesForVisitedNodes is invoked to make sure that the meet-over-all-realizable-paths value is known for all nodes visited during the call on BackwardComputeJumpFunctions. Consequently, on subsequent calls to BackwardComputeJumpFunctions – to satisfy later demands – these nodes need not be re-visited.

Our demand algorithm is designed so that it has the same worst-case asymptotic complexity as the exhaustive algorithm of Section 5 when the sequence of demands consists of all $N^{\#}$ nodes: In particular, the time is bounded by $O(ED^3)$ for efficiently representable IDE instances.

Because a dataflow value at one point might depend on all other values at all other points, theoretically, the worst-case asymptotic complexity of the demand algorithm is $O(ED^3)$, even for a single demand. (This is true even if $MRP_{\langle \bar{n},\bar{d}\rangle}$ is computed immediately at the end of BackwardComputeJumpFunctions via (10).) However, in the experiments discussed in Section 7, the demand algorithm, used to demand values for all uses of scalar integer variables, was faster than the exhaustive algorithm in all cases.

7. Experiments

We have carried out several experiments to determine the feasibility of our proposed algorithms. Three dataflow-analysis algorithms were used in the experiments:

Precise Exhaustive: The exhaustive algorithm of Section 5, which considers only realizable paths in $G^{\#}$.

Precise Demand: The demand algorithm of Section 6, which also considers only realizable paths in $G^{\#}$.

Naive Exhaustive: An exhaustive algorithm that considers *all* paths rather than just the realizable paths. This algorithm is safe, but may be less accurate than the precise algorithms. For example, for the program in Fig. 1, the Naive Exhaustive algorithm would not identify variable x as a constant at the print statement in procedure *main*.

The three algorithms were implemented in C and used with a front end that analyzes a C program and generates the corresponding exploded supergraphs for copy-constant propagation and linear-constant propagation (for scalar integer variables).

In the experiments, pointers were handled conservatively: Every call via a procedure-valued pointer was considered to be a possible call to every procedure of an appropriate type that was passed as a parameter or whose value was assigned to a variable somewhere in the program. Every assignment through a pointer was considered to conditionally kill all variables to which the "&" operator was applied somewhere in the program; all uses through pointers were considered to be non-constant.

Temporary variables were introduced as part of normalizing statements containing operations with side effects (e.g., pre- and post-increment). Without some care, this transformation could have distorted the relative performance of the exhaustive and demand algorithms: An exhaustive algorithm could spend considerable effort propagating dataflow information for temporaries beyond the sites at which they are used. This could have skewed the results artificially in favor of the demand algorithm. In our experiments, the effect was negligible because temporaries were reused: The total number of temporary variables was very small, and thus the cost of propagating information about temporaries was a small fraction of the total work performed by the exhaustive algorithms.

The study used 38 C programs; some came from the SPEC integer benchmark suite [29] and some were standard UNIX utilities. Fig. 12 gives information about the characteristics of the test programs. The second column indicates the code size (lines of C source code after the C preprocessor has been applied and blank lines removed). The third column gives the number of uses of scalar integer variables.

Tests were carried out on a Sun SPARCstation 20 Model 71 with 64 MB of RAM. We used each of the three algorithms to perform copy and linear-constant propagation on each of the 38 programs, recording running times and the number of uses of scalar integer variables that were detected as constants. These data are presented in Fig. 13. The number of constants detected by each algorithm, reported in columns 2, 4, 6, and 8, respectively, indicates the number of places found by each algorithm where constants could be substituted for variables to improve the code. In all our reported results, running times reflect the trimmed mean of five data points (i.e., all experiments were run five times, and the average running times were computed by discarding the high and low values). All running times are the sum of "user cpu-time" and "system cpu-time" (in seconds) for the algorithms once the exploded supergraph is constructed. Boldface is used to emphasize the cases in which the algorithms did not all detect the same number of constants. (The Precise Exhaustive and Precise Demand algorithms always detect the same constants; therefore, we have not repeated that data under "Precise Demand".)

These data allowed us to make the following comparisons:

- The running times and accuracies of the Naive Exhaustive algorithm versus those of the Precise Exhaustive algorithm.

Example	Lines	# of Uses of Scalar Integer Variables
diff.diffh	303	137
genetic	336	150
allroots	427	70
ul	451	168
compress	657	288
stanford	665	570
clinpack	695	402
travel	725	200
lex315	747	197
sim	748	1357
mway	806	647
pokerd	1099	475
ansitape	1222	293
loader	1255	251
gcc.main	1285	363
voronoi	1394	150
ratfor	1531	515
livc	1674	833
struct.beauty	1701	338
diff.diff	1761	663
xmodem	1809	519
compiler	1908	594
learn.learn	1954	199
gnugo	1963	952
triangle	1968	2154
football	2075	1724
dixie	2439	310
eqntott	2470	939
twig	2555	356
cdecl	2577	244
lex	2645	1402
patch	2746	899
assembler	2994	355
unzip	3261	920
tbl	3462	1500
gcc.cpp	4061	927
simulator	4239	928
li	6054	431

Fig. 12. Information about the 38 test programs.

- The running times and accuracies of copy-constant propagation versus linear-constant propagation.
- The running times of the Precise Demand algorithm versus those of the Precise Exhaustive algorithm.

Example	Naive Exhaustive				Precise Exhaustive				Precise Demand	
	Copy		Linear		Copy		Linear		Copy	Linear
	consts	time	consts	time	consts	time	consts	time	time	time
diff.diffh	1	0.19	1	0.19	1	0.53	1	0.60	0.17	0.17
genetic	0	0.09	0	0.10	0	0.27	0	0.31	0.12	0.12
allroots	10	0.19	10	0.19	10	0.52	10	0.58	0.10	0.10
ul	2	0.26	2	0.26	2	0.63	2	0.72	0.37	0.40
compress	18	1.04	18	1.06	18	2.59	18	2.91	0.97	0.98
stanford	15	0.49	15	0.53	15	1.35	15	1.54	0.31	0.32
clinpack	**123**	0.61	**129**	0.63	**131**	1.57	**137**	1.79	0.38	0.37
travel	**27**	0.39	**31**	0.38	**36**	1.03	**39**	1.13	0.24	0.25
lex315	3	0.55	3	0.57	3	1.56	3	1.75	0.33	0.34
sim	4	0.86	4	0.92	4	2.41	4	2.71	1.68	1.49
mway	7	1.67	7	1.66	7	4.56	7	5.38	1.71	1.82
pokerd	0	0.95	0	0.96	0	2.47	0	2.77	0.90	0.96
ansitape	5	2.01	5	2.03	5	5.30	5	5.96	2.18	2.22
loader	10	1.34	10	1.36	10	3.36	10	3.74	0.67	0.66
gcc.main	12	1.50	12	1.56	12	4.14	12	4.43	1.50	1.56
voronoi	0	0.94	0	0.96	0	2.58	0	2.73	0.73	0.77
ratfor	4	1.09	4	1.11	4	2.93	4	3.12	2.29	2.46
livc	11	2.09	11	2.11	11	5.30	11	5.98	0.88	0.90
struct.beauty	7	1.52	7	1.57	7	4.18	7	4.77	2.43	2.50
diff.diff	8	4.13	8	4.40	8	10.66	8	12.44	2.35	2.41
xmodem	**6**	2.83	**10**	2.85	**13**	7.25	**17**	8.29	2.51	2.59
compiler	6	1.57	6	1.58	6	5.13	6	5.75	3.37	3.89
learn.learn	2	2.06	2	2.09	2	4.73	2	5.31	2.18	2.29
gnugo	**6**	1.17	**6**	1.27	**10**	2.83	**10**	3.23	1.25	1.33
triangle	0	1.71	0	1.74	0	4.22	0	4.79	0.99	1.02
football	0	3.94	0	4.20	0	9.54	0	10.53	4.84	4.98
dixie	7	1.88	7	1.92	7	5.42	7	5.90	1.73	1.76
eqntott	9	2.13	9	2.34	9	4.90	9	5.49	1.86	1.94
twig	3	3.49	3	3.56	3	8.38	3	9.39	2.70	2.92
cdecl	13	1.48	13	1.45	13	3.75	13	4.01	3.28	3.07
lex	**4**	4.71	**4**	5.18	**6**	12.73	**7**	13.02	9.18	9.59
patch	4	5.47	4	5.71	4	13.76	4	16.32	6.35	6.78
assembler	9	4.89	9	4.95	9	12.41	9	14.14	2.58	2.65
unsip	**9**	4.32	**9**	4.38	**12**	12.36	**13**	13.50	4.65	4.83
tbl	0	4.57	0	4.56	0	10.14	0	11.47	5.30	5.91
gcc.cpp	**15**	9.17	**15**	9.37	**15**	23.85	**21**	27.75	9.39	9.71
simulator	7	4.67	7	4.70	7	12.17	7	13.09	2.50	2.54
li	1	12.32	1	12.09	1	50.27	1	55.78	20.60	21.12

Fig. 13. Running times and number of constants detected.

7.1. Comparison 1: Naive Exhaustive vs. Precise Exhaustive

Fig. 14 summarizes the relative times of the Naive Exhaustive algorithm versus the Precise Exhaustive algorithm for both copy and linear-constant propagation. Recall that the asymptotic running time of the Precise Exhaustive algorithm is bounded by $O(E\,\text{MaxVisible}^3)$, whereas the asymptotic running time of the Naive Exhaustive algorithm is bounded by $O(E\,\text{MaxVisible})$. In our test sample, we found that solving constant-propagation problems precisely resulted in a slowdown by a factor ranging

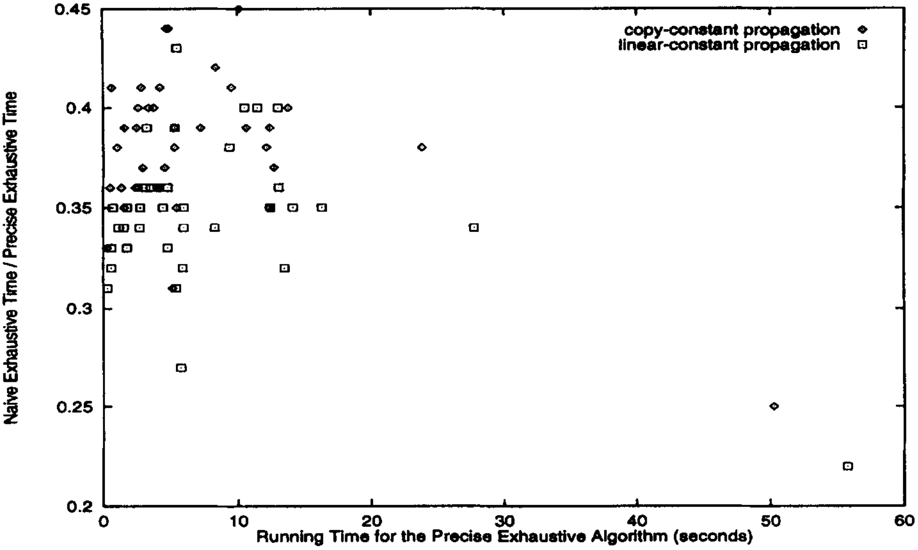


Fig. 14. The relative times of the Naive Exhaustive algorithm versus the Precise Exhaustive algorithm for both copy and linear-constant propagation.

from 2.2 to about 4.5. However, the Precise Exhaustive algorithm found additional constants in 7 of the 38 test programs (see Fig. 13).

7.2. Comparison 2: Copy-Constant Propagation vs. Linear-Constant Propagation

Fig. 15 summarizes the relative times for copy-constant propagation versus linear-constant propagation (for both the Precise Exhaustive algorithm and the Naive Exhaustive algorithm). These results indicate that the overhead for performing linear-constant propagation is relatively minor. At best, copy-constant propagation is about 9% faster for the Naive Exhaustive algorithm, and about 16% faster for the Precise Exhaustive algorithm.

We also compared the accuracies of copy and linear-constant propagation. In our study, linear-constant propagation found more constants than copy-constant propagation in 6 out of the 38 test programs for the Precise Exhaustive algorithm and in 3 out of the 38 test programs for the Naive Exhaustive algorithm. Furthermore, in 7 out of the 38 test programs, linear-constant propagation via the Precise Exhaustive algorithm found more constants than copy-constant propagation via the Naive Exhaustive algorithm. These results are in contrast to previous results reported by Grove and Torczon for numeric Fortran programs [12], in which no differences in accuracy were found between "pass-through parameter" constant propagation (which

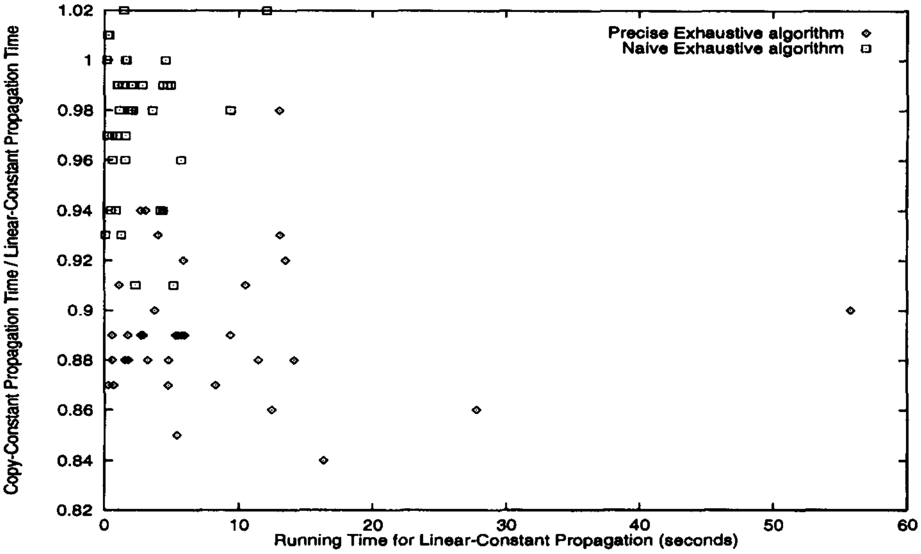


Fig. 15. The relative times for copy-constant propagation versus linear-constant propagation (for both the Precise Exhaustive algorithm and the Naive Exhaustive algorithm).

is even weaker than copy-constant propagation) and "polynomial parameter" constant propagation (which is stronger than linear-constant propagation).[9]

7.3. *Comparison 3: Precise Demand vs. Precise Exhaustive*

Fig. 16 summarizes the relative times of the Precise Demand algorithm versus the Precise Exhaustive algorithm for both copy and linear-constant propagation. For the Precise Demand algorithm the times given in columns 10 and 11 of the table in Fig. 13 are the total times for a sequence of demands. However, a demand was *not* placed at every node of the exploded supergraph; instead, a demand was placed for every use of a scalar integer variable, since this information is sufficient to determine all opportunities for replacing variables by constants. (Thus, column three of the table in Fig. 12 gives the number of demands issued for each test program.)

The Precise Demand algorithm was faster than the Precise Exhaustive algorithm on all test programs; the speedup observed ranged from 1.14 to about 6.

8. Related work

This paper concerns interprocedural dataflow-analysis problems in which the dataflow information at a program point is represented by an environment, and the effect of a

[9] The algorithm used by Grove and Torczon in their study did not necessarily determine precise interprocedural information because of limitations in the way the algorithm handled "return jump functions". This may have distorted their results.

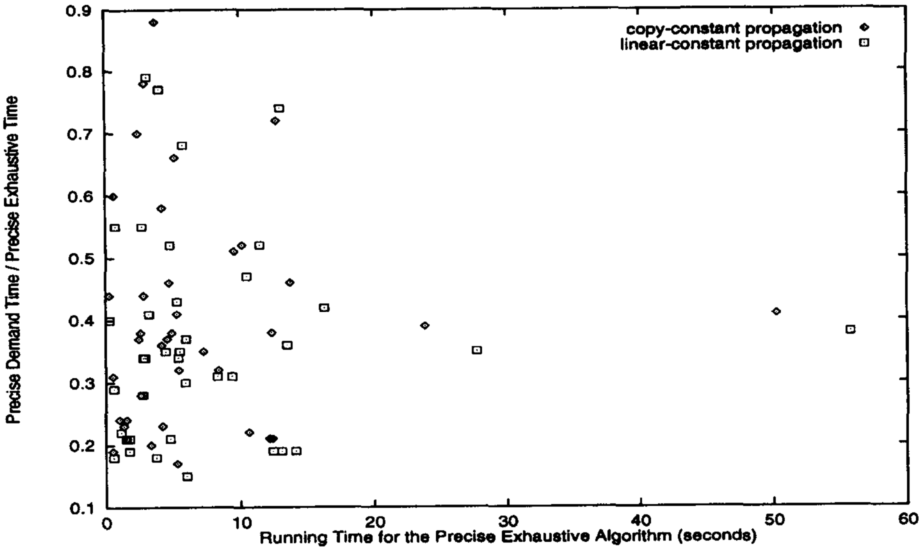


Fig. 16. The relative times of the Precise Demand algorithm versus the Precise Exhaustive algorithm for both copy and linear-constant propagation.

program operation is represented by a distributive environment transformer. We have described an algorithm to solve such problems precisely in polynomial time. In this section, we explain how our ideas and results relate to previous work.

8.1. The IDE framework

The IDE framework is based on earlier interprocedural dataflow-analysis frameworks defined by Sharir and Pnueli [28] and Knoop and Steffen [17], as well as the IFDS framework that we proposed earlier [25, 24, 13]. The IDE framework is basically the Sharir-Pnueli framework with three modifications:

(i) The dataflow domain is restricted to be a domain of environments.

(ii) The dataflow functions are restricted to be distributive environment transformers.

(iii) The edge from a call node to the corresponding return-site node can have an associated dataflow function.

Conditions (i) and (ii) are restrictions that make the IDE framework less general than the full Sharir–Pnueli framework. Condition (iii), however, generalizes the Sharir–Pnueli framework and permits it to cover programming languages in which recursive procedures have local variables and parameters (which the Sharir–Pnueli framework does not). A different generalization to handle recursive procedures with local variables and parameters was proposed by Knoop and Steffen [17].

As discussed in Section 5.4.1, the IDE framework is a strict generalization of the IFDS framework. In IFDS problems, the set of dataflow facts D is a finite set and the dataflow functions (which are in $2^D \rightarrow 2^D$) distribute over the meet operator (either union or intersection, depending on the problem). All IFDS problems can be encoded as IDE problems. On the other hand, only some IDE problems can be encoded as IFDS problems. For example, an IDE problem in which L is infinite – such as the linear-constant-propagation problem – cannot be translated into an IFDS problem. Consequently, this paper strictly extends the class of interprocedural dataflow-analysis problems known to be solvable in polynomial time.

In addition, even when L is finite, the algorithm presented in this paper will perform much better than the algorithm for IFDS problems for many kinds of problems. For example, consider the problem of copy-constant propagation: In any given problem instance, the size of L is no larger than the number of literals in the program; the IDE version of copy-constant propagation involves environments of size D, where D is the set of program variables; by contrast, the set of dataflow facts for the IFDS version is $D \times L$. This has a substantial impact in practice: For some C programs of about 1300 lines that we tested, the IFDS version ran out of virtual memory, whereas the IDE version finished in a few seconds. (To date, we have run the IDE algorithm – for the more general linear-constant-propagation problem – on programs as large as 6000 lines.)

In our previous papers, we showed how IFDS problems could be solved precisely in polynomial time by transforming them into a particular kind of *graph-reachability* problem – not an ordinary reachability problem, but reachability along realizable paths. This transformation yields an efficient interprocedural dataflow-analysis algorithm because the realizable-path reachability problem can be solved by an efficient dynamic-programming algorithm. In the present paper, we show how to generalize these techniques from IFDS problems to IDE problems. In making this generalization, the following new issues arise:

- Although the transformation we apply to IDE problems is similar to the one used for IFDS problem, the transformed problem that results is a realizable-path *summary* problem, not a realizable-path *reachability* problem. That is, in the transformed graph we are no longer concerned with a pure reachability problem, but with values obtained by applying functions along (realizable) paths. (The relationship between transformed IFDS problems and transformed IDE problems is similar to the relationship between ordinary graph-reachability problems and generalized problems that compute summaries over paths, such as shortest-path problems, closed-semiring path problems, etc. [1, 7].)
- The algorithm's efficiency depends on the use of compact representations of the functions that label edges in (the transformed) IDE problems. For example, in Section 5.4.3 we showed how the functions that arise in the linear-constant-propagation problem can be represented very simply using triples of integers.

The IDE (and IFDS) problems can be solved by a number of previous algorithms, including the "elimination", "iterative", and "call-strings" algorithms given by Sharir and Pnueli and the algorithm of Cousot and Cousot [8]. However, for general IFDS and IDE problems, both the iterative and call-strings algorithms can take exponential time in the worst case. Knoop and Steffen give an algorithm similar to Sharir and Pnueli's "elimination" algorithm [17]. The efficiencies of the Sharir–Pnueli and Knoop–Steffen elimination algorithms depend, among other things, on the way functions are represented. No representations are discussed in [28, 17]; however, even if the techniques of the present paper are used, because the Sharir–Pnueli and Knoop–Steffen algorithms manipulate functions as a whole, rather than pointwise, they are not as efficient as the algorithm presented here.

Recently, Ramalingam has shown how a framework very similar to the IDE framework can be used to develop a theory of "dataflow frequency analysis" in which information is obtained about how often and with what probability a dataflow fact holds true during program execution [20].

8.2. Constant-Propagation Algorithms

Our algorithms for solving IDE problems can be used to find precise (i.e., meet-over-all-valid-paths) solutions for both copy and linear-constant propagation problems in polynomial time. For both copy-constant propagation and linear-constant propagation, there are several antecedents. A version of interprocedural copy-constant propagation was developed at Rice and has been in use for many years. The algorithm is described in [5], and studies of how the algorithm performs in practice on Fortran programs were carried out by Grove and Torczon [12]. The Rice algorithm has two potential drawbacks that our algorithms do not have.

- The Rice algorithm is not precise for recursive programs. (In fact, it may fall into an infinite loop when applied to recursive programs.)
- The precise function that captures how procedure p transforms an input environment is

$$\lambda env. \bigsqcap_{r \in SLRP(s_p, e_p)} M(r)(env). \tag{11}$$

  However, the Rice algorithm uses only an approximation to (11) (the so-called "return jump function"). Because of this approximation, the Rice algorithm does not even yield precise answers for non-recursive programs.

In contrast, the solutions to copy and linear-constant propagation problems obtained with our algorithms are precise for both non-recursive and recursive programs. Our algorithms generate *precise* "return jump functions": In particular, the collection of micro-functions of the form $JumpFn(\langle s_p, d' \rangle \to \langle e_p, d \rangle)$ represents (11).

An algorithm for precise copy-constant propagation (for both recursive and non-recursive programs) was given using the IFDS framework by Reps et al. [25, 13]. However, as discussed in Section 8.1, there is a significant drawback to formulating copy-constant propagation as an IFDS problem: The running time and the space used both depend on the quantity "number of literals in the program".

We have also shown in this paper how to solve linear-constant-propagation problems, which in general find a superset of the instances of constant variables found by copy-constant propagation. Several others have also examined classes of constant-propagation problems more general than copy-constant propagation [15, 30, 12, 19, 6].

- Karr used linear algebra to define a safe algorithm for (intraprocedural) affine problems (i.e., problems in which relationships of the form $x := a_1 y_1 + \cdots + a_k y_k + c$ are tracked) [15].
- Steffen and Knoop address the more general problem of determining whether a *subexpression* (rather than a variable) has a constant value [30]. They define a decidable version of the problem and give an algorithm for the intraprocedural setting. In the case of loop-free code, the algorithm is optimal.
- Grove and Torczon defined a class of polynomial jump functions [12], which are more general than the linear jump functions used in our work; however, because of limitations in the way they define "return jump functions", their algorithm does not necessarily find precise interprocedural information.
- An algorithm given by Metzger and Stroud can handle statements of the form $x := ay + bz + c$ [19], which is a more general form than can be handled by the IDE framework. (The environment transformer that corresponds to such a statement, $\lambda env.env[x \rightarrow a * env(y) + b * env(z) + c]$ is not distributive.) However, their algorithm is imprecise; it does not find the "meet-over-all-valid-paths" solution.
- Carini and Hind defined an algorithm for interprocedural constant propagation (extending the work of Wegman and Zadeck [33]) that can handle non-distributive dataflow functions (and thus is more general than our algorithm) [6]. However, since they do not propagate values from called functions back to calling functions, their results are even less precise than our Naive Exhaustive algorithm.

Wegman and Zadeck [33], building on earlier work by Wegbreit [32], examined the interaction between constant propagation and dead-code elimination. This issue is not addressed in our work.

8.3. Demand Dataflow-Analysis Algorithms

Section 6 presented a demand algorithm for solving IDE problems, and the experiments reported in Section 7 indicate that for constant-propagation problems in C programs the demand algorithm is superior to the exhaustive algorithm (at least in programs of up to 6000 lines). The relationship between the demand algorithm of Section 6 and the exhaustive algorithm of Section 5 is similar to the relationship that holds for IFDS problems between the demand algorithm of [25, 13] and the exhaustive algorithm of [25, 24].

One approach to obtaining demand algorithms for interprocedural dataflow-analysis problems was described by Reps [23, 21]. Reps presented a way in which algorithms that solve demand versions of interprocedural analysis problems can be obtained automatically from their exhaustive counterparts (expressed as logic programs) by making use of the "magic-sets transformation", a general transformation developed in the logic-programming and deductive-database communities for creating efficient demand versions of (bottom-up) logic programs [26, 2, 3, 31].

Reps illustrated this approach by showing how to obtain a demand algorithm for the interprocedural locally separable problems. Subsequent work by Reps et al. extended the logic-programming approach to the class of IFDS problems [25, 24]. (The latter papers do not make use of logic-programming terminology; however, the exhaustive algorithms described in the papers have straightforward implementations as logic programs. Demand algorithms can then be obtained by applying the magic-sets transformation.)

A different approach to obtaining demand versions of interprocedural dataflow-analysis algorithms has been investigated by Duesterwald et al. [9]. In their approach, a set of dataflow equations is 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 forward flow functions. These equations are then solved using a demand-driven fixed-point-finding procedure.

The demand algorithm of Section 6 has the following advantages over the algorithm given by Duesterwald et al.:

(1) Their algorithm only applies when L has a *finite number of elements*, whereas we require only that L and F be of *finite height*. For example, linear-constant propagation, where L has an infinite number of elements, is outside the class of problems handled by their algorithm.

(2) Instead of computing the value of d at n, their algorithm answers queries of the form "Is the value of d at $n \sqsupseteq l$?" for a given value $l \in L$. In linear-constant propagation, there is no way to use queries of this form to find the constant value of a given variable.

(3) When restricted to IFDS problems, the worst-case cost of the Duesterwald–Gupta–Soffa technique is $O(E\,D\,2^D)$. In contrast, the worst-case cost of our demand algorithm is $O(E\,D^3)$.

Duesterwald et al. also give a specialized copy-constant-propagation algorithm that remedies problems (2) and (3) for copy-constant propagation.

References

[1] A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, Magic sets and other strange ways to implement logic programs, in: *Proc. 5th ACM Symp. on Principles of Database Systems*, 1986.

[3] C. Beeri and R. Ramakrishnan, On the power of magic, in: *Proc. 6th ACM Symp. on Principles of Database Systems*, San Diego, CA (March 1987) 269–293.

[4] D. Callahan, The program summary graph and flow-sensitive interprocedural data flow analysis, in: *SIGPLAN Conf. on Programming Languages Design and Implementation* (1988) 47–56.

[5] D. Callahan, K. Cooper, K. Kennedy and L. Torczon, Interprocedural constant propagation, in: *SIGPLAN Symp. on Compiler Construction* (1986) 152–161.

[6] P. Carini and M. Hind, Flow-sensitive interprocedural constant propagation, in: *SIGPLAN Conference on Programming Languages Design and Implementation* (1995) 23–31.

[7] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms* (MIT Press, New York, 1990).

[8] P. Cousot and R. Cousot, Static determination of dynamic properties of recursive procedures, in: E. Neuhold, ed., *Formal Descriptions of Programming Concepts* (IFIP WG 2.2, St. Andrews, Canada, August 1977) (North-Holland, Amsterdam, 1978) 237–277.

[9] E. Duesterwald, R. Gupta and M. Soffa, Demand-driven computation of interprocedural data flow, in: *ACM Symp. on Principles of Programming Languages* (1995) 37–48.

[10] C. Fischer and R. LeBlanc, *Crafting a Compiler* (Benjamin/Cummings, Menlo Park, CA, 1988).

[11] R. Giegerich, U. Moncke and R. Wilhelm, Invariance of approximative semantics with respect to program transformation, in: *GI 81 11th GI Annual Conf. Informatik-Fachberichte 50* (Springer, New York, 1981) 1–10.

[12] D. Grove and L. Torczon, A study of jump function implementations, in: *SIGPLAN Conf. on Programming Languages Design and Implementation* (1993) 90–99.

[13] S. Horwitz, T. Reps and M. Sagiv, Demand interprocedural dataflow analysis, in: *Proc. 3rd ACM SIGSOFT Symp. on the Foundations of Software Engineering* (1995) 104–115. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/fse95.ps).

[14] N. Jones and A. Mycroft, Data flow analysis of applicative programs using minimal function graphs, in: *ACM Symp. on Principles of Programming Languages* (1986) 296–306.

[15] M. Karr, Affine relationship among variables of a program, *Acta Inform.* **6** (1976) 133–151.

[16] G. Kildall, A unified approach to global program optimization, in: *ACM Symp. on Principles of Programming Languages* (1973) 194–206.

[17] J. Knoop and B. Steffen, The interprocedural coincidence theorem, in: *Proc. Internat. Conf. on Compiler Construction* (1992) 125–140.

[18] W. Landi and B. Ryder, Pointer induced aliasing: a problem classification, in: *ACM Symp. on Principles of Programming Languages* (1991) 93–103.

[19] R. Metzger and S. Stroud, Interprocedural constant propagation: an empirical study, *ACM Lett. on Programming Languages Systems* **2** (1993).

[20] G. Ramalingam, Data flow frequency analysis, in: *SIGPLAN Conf. on Programming Languages Design and Implementation,* May 1996.

[21] T. Reps, Demand interprocedural program analysis using logic databases, in: R. Ramakrishnan, ed., *Applications of Logic Databases* (Kluwer Academic Publishers, Dordrecht, 1994).

[22] T. Reps, Solving demand versions of interprocedural analysis problems, in: *Proc. Internat. Conf. on Compiler Construction* (1994) 389–403.

[23] T. Reps, Solving demand versions of interprocedural analysis problems, in: *Proc. 5th Internat. Conf. on Compiler Construction,* Edinburgh, Scotland (1994) 389–403. (Appeared in: P. Fritzson (ed.) Lecture Notes in Computer Science, Vol. 786 (Springer, New York, 1994.)

[24] T. Reps, S. Horwitz and M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: *ACM Symp. on Principles of Programming Languages* (1995) 49–61. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/popl95.ps.)

[25] T. Reps, M. Sagiv and S. Horwitz, Interprocedural dataflow analysis via graph reachability, Tech. Report TR 94-14, Datalogisk Institut, University of Copenhagen, 1994. (Available on the WWW from URL http://www.cs.wisc.edu/wpis/papers/diku-tr94-14.ps.)

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

[27] M. Sagiv, T. Reps and S. Horwitz, Precise interprocedural dataflow analysis with applications to constant propagation, in: P. Mosses, M. Nielsen and M. Schwartzbach, eds., *Proc. FASE'95: Coll. on Formal Approaches in Software Engineering,* Lecture Notes in Computer Science, Vol. 915, Aarhus, Denmark (Springer, Berlin, 1995) 651–665.

[28] M. Sharir and A. Pnueli, Two approaches to interprocedural data flow analysis, in: S. Muchnick and N. Jones, eds., *Program Flow Analysis: Theory and Applications,* Ch. 7 (Prentice-Hall, Englewood Cliffs, NJ, 1981) 189–234.

[29] SPEC Component CPU Integer Release 2/1992 (Cint92), Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1992.

[30] B. Steffen and J. Knoop, Finite constants: characterizations of a new decidable set of constants, *Theoret. Comput. Sci.* **80** (1991) 303–318.

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

[32] B. Wegbreit, Property extraction in well-founded property sets, *IEEE Trans. Software Eng.* **1** (1975) 270–285.

[33] M. Wegman and F. Zadeck, Constant propagation with conditional branches, in: *ACM Symp. on Principles of Programming Languages,* 1985.