The Undecidability of Aliasing

G. RAMALINGAM
IBM T. J. Watson Research Center

Alias analysis is a prerequisite for performing most of the common program analyses such as reaching-definitions analysis or live-variables analysis. Landi [1992] recently established that it is impossible to compute statically precise alias information—either may-alias or must-alias—in languages with if statements, loops, dynamic storage, and recursive data structures: more precisely, he showed that the may-alias relation is not recursive, while the must-alias relation is not even recursively enumerable. This article presents simpler proofs of the same results.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.4.1 [**Mathematical Logic**]: Computability Theory; F.4.3 [**Formal Languages**]: Decision Problems

General Terms: Languages, Theory

Additional Key Words and Phrases: Alias analysis, pointer analysis

1. INTRODUCTION

Compilers and various other programming tools make good use of static program analysis. To solve most program analysis problems, such as the problem of determining live variables, one requires alias information, or information about whether two L-valued expressions may/must have the same value at some program point. Informally, two names or L-valued expressions are said to alias each other at a particular point during program execution if both refer to the same location. In the may-alias problem, one is interested in identifying aliases that can occur during some execution of the program, while in the must-alias problem, one is interested in identifying aliases that occur in all executions of the program. Obviously, such information is relevant to most dataflow analysis problems.

Program analysis is commonly performed under the conservative assumption that all paths in the program are executable, since the problem of deciding if an arbitrary path in a program is executable is undecidable. This simplifying assumption makes it possible to solve a number of program analysis problems. Unfortunately, even this assumption is not sufficient to make the may-alias or must-alias problem decidable.

Author's address: IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598.

Names a and b are said to *may-alias* each other at a program point if there exists a path P from the program entry to that program point, such that a and b refer to the same location after execution along path P. Names a and b are said to *must-alias* each other at a program point if, for all paths P from the program beginning to the program point, a and b both refer to the same location after execution along path P. Landi [1992] recently established that even the simpler intraprocedural versions of the may-alias and must-alias problems are undecidable for languages that permit recursive data structures to be built. Here, we present a simpler proof of the same result. We establish the undecidability of this problem by reducing the *Post's Correspondence Problem*, or PCP, to the may-alias problem. Section 2 of the article presents proofs of the undecidability results, while Section 3 discusses related work.

2. THE UNDECIDABILITY OF ALIASING

The decision version of the may-alias problem is the following: given a program point in a program and two names, decide if the may-alias relation holds between the given names at the given program point. More generally, we are interested in generating the set of all may-alias facts that hold true. This set can be made finite by restricting attention to the names and L-valued expressions that occur in the program.

Definition 2.1. The *Post's Correspondence Problem*, or PCP is the following: Given arbitrary lists A and B of r strings each in $\{0,1\}^+$, say

$$A = w_1, w_2, \ldots, w_r$$
$$B = z_1, z_2, \ldots, z_r$$

does there exist a nonempty sequence of integers $i_1, i_2, \ldots, i_k$ such that

$$w_{i_1} w_{i_2} \cdots w_{i_k} = z_{i_1} z_{i_2} \cdots z_{i_k}.$$

THEOREM 2.2. *The PCP is undecidable* [*Hopcroft and Ullman* 1979].

THEOREM 2.3. *The intraprocedural may-alias problem is undecidable for languages with if statements, loops, dynamic storage, and recursive data structures.*

PROOF. We relate PCP to the may-alias problem as follows. Consider a binary tree with root *root*. A binary string consisting of 0s and 1s can be interpreted as representing a path from the root of the binary tree with 0 representing a left branch and 1 a right branch. Define *branch*(0) to be *left* and *branch*(1) to be *right*. For any binary string $b_0 b_1 \cdots b_n$, define $path(b_0 b_1 \cdots b_n)$ to be $branch(b_0) \rightarrow branch(b_1) \rightarrow \cdots \rightarrow branch(b_n)$. Let α and β be two binary strings. Then, $\alpha = \beta$ iff $root \rightarrow path(\alpha)$ and $root \rightarrow path(\beta)$ refer to the same node in the binary tree. Essentially, this is the idea behind our reduction of PCP to the may-alias problem.

Given an instance of PCP, we construct the program in Figure 1. The program is written in C, but it can be written in any language with if statements, loops, dynamic storage, and recursive data structures. The may-alias relation holds between $*(q \to \mathit{left})$ and *node* at line 39 iff the given instance of PCP has a solution, as explained below. Ignore, for the moment, lines 7 through 19, and assume that *root* points to a binary tree at line 20. There are r different branches inside the loop of lines 22 through 35—number these branches 1 through r. Any path P in the program from line 20 to line 36 that iterates through the loop (lines 22 through 35) t times corresponds to a sequence σ of t integers, where the jth element in the sequence denotes the branch taken during the jth time through the loop. Furthermore, $*p$ and $*q$ will alias each other at the end of path P iff the sequence σ is a solution to the given PCP instance, *provided that root pointed to a "sufficiently large" binary tree at line 20.*

Instead of actually constructing a binary tree, we use the code in lines 9 through 19 to "generate" all possible paths through a binary tree. In doing this, the pointer fields of newly allocated tree nodes are not initialized to a null pointer as might be done usually. Instead, these fields are initialized to point to a special node called *undefined* whose *left* and *right* fields are initialized to point to itself. This ensures that every possible path from the program beginning to line 36 can be executed without raising any errors such as dereferencing a null pointer. Consequently, at line 36, either pointer p has a "proper value" and points to some node allocated in line 12 or 15, or pointer p points to the node *undefined*. The same claim holds true for pointer q. Consequently, the given instance of PCP has a solution iff there exists some execution path to line 36, at the end of which $p = q \neq \&\mathit{undefined}$. Checking for this condition can be converted into checking for a may-alias fact using line 38. Obviously, the may-alias relation holds between $*(q \to \mathit{left})$ and *node* at line 39 iff the given instance of PCP has a solution. □

The may-alias relation, however, is recursively enumerable because we can enumerate all paths in the program, and for any given path, we can determine the aliases that hold after execution along that path.

THEOREM 2.4. *The intraprocedural must-alias problem is undecidable for languages with if statements, loops, dynamic storage, and recursive data structures. The intraprocedural must-alias relation is not even recursively enumerable.*

PROOF. The undecidability of the must-alias problem follows immediately from the undecidability of the may-alias problem. Consider Figure 1. Line 40 shows how must-alias information can be used to compute may-alias information. The must-alias relation holds between *node* and $*(\mathit{node.left})$ in line 41 iff the may-alias relation does *not* hold between *node* and $*(q \to \mathit{left})$ in line 39. The complement of a recursively enumerable but nonrecursive set is not recursively enumerable. It follows that the must-alias relation is not even recursively enumerable. □

3. RELATED WORK

Kam and Ullman [1977] established that the problem of computing the meet-over-all-paths solution to a monotonic dataflow analysis framework, or monotone MOP problem, is undecidable by reducing a modified version of

```
[1]   main() {
[2]       int i;
[3]       struct tree_node {
[4]           int value;
[5]           struct tree_node *left, *right;
[6]       } *root, *p, *q, *t, node, undefined;

[7]       undefined.left = &undefined;  undefined.right = &undefined;
[8]       root = malloc(sizeof(struct tree_node));  root->left = &undefined;  root->right = &undefined;
[9]       t = root;
[10]      while ( ··· ) {
[11]          if ( ··· ) {
[12]              t->right = malloc(sizeof(struct tree_node));
[13]              t = t->right;
[14]          } else {
[15]              t->left = malloc(sizeof(struct tree_node));
[16]              t = t->left;
[17]          }
[18]          t->left = &undefined;  t->right = &undefined;
[19]      }

[20]      p = root;
[21]      q = root;
[22]      do {
[23]          i = ··· ;
[24]          if (i == 1) {
[25]              p = p-> path(w_1); q = q-> path(z_1);
[26]          } else if (i == 2) {
[27]              p = p-> path(w_2); q = q-> path(z_2);
[28]          } else if (i == 3) {
[29]          ···
[30]          } else if (i == r-1) {
[31]              p = p-> path(w_{r-1}); q = q-> path(z_{r-1});
[32]          } else {
[33]              p = p-> path(w_r); q = q-> path(z_r);
[34]          }
[35]      } while ( ··· )
[36]      /* The given PCP instance has an affirmative answer iff ∃ some execution path to this point after
[37]       * which p = q ≠ &undefined. */
[38]      p->left = &node; undefined.left = &undefined;
[39]      /* The given PCP instance has an affirmative answer iff *(q->left) may-alias node at this point. */
[40]      node.left = &node; q->left->left = &undefined;
[41]      /* node must-alias *(node.left) here  iff  not  *(q->left) may-alias node in line 39. */
[42]  }
```

Fig. 1. The program corresponding to an instance $w_1, \ldots, w_r, z_1, \ldots, z_r$ of the PCP problem. Note that *path*(α) is an abbreviation for a sequence of dereferences through the *left* and *right* fields as determined by the binary string α.

PCP to the monotone MOP problem. The proof presented in this article is similar to the proof of Kam and Ullman. However, as Kam and Ullman observe, their result says only that no algorithm that solves *any* monotonic dataflow analysis problem exists. However, they do not rule out the existence of algorithms for a *specific* monotonic dataflow analysis problem, such as the may-alias problem. In other words, the meet-over-all-paths problem for arbitrary monotonic dataflow analysis frameworks is more general than the may-alias problem. Consequently, the undecidability of the latter problem is a stronger result than the undecidability of the former problem.

Larus [1988; 1989] showed that alias analysis was NP-hard in languages with recursive data structures. Landi [1992] presented the first proof that the may-alias problem is not recursive and that the must-alias problem is not even recursively enumerable. He established these results by reducing the halting problem to these problems, and this article presents simpler proofs for the same results.

In the absence of recursively defined data structures, various versions of the aliasing problems become decidable, but remain difficult. Myers [1981] showed that many interprocedural static-analysis problems are NP-complete. Refer to Landi's thesis [1991] for a comprehensive classification of the complexity of various restricted versions of the aliasing problems. Not surprisingly, the problem of computing conservative approximations to the may-alias and must-alias relations in the presence of pointers has attracted and continues to attract much attention. Pfeiffer's thesis [1991] presents a comprehensive overview of this area. Refer to Landi and Ryder [1992] and Choi et al. [1993] for more recent work in this area.

ACKNOWLEDGMENTS

The author thanks the anonymous referees, Charles Fischer, and William Landi for their comments, which helped improve this article and made the proofs more precise.

REFERENCES

CHOI, J.-D., BURKE, M. G. AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (Charleston, S. Carolina). ACM, New York, 232–245.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass.

KAM, J. B. AND ULLMAN, J. D. 1977. Monotone data flow analysis frameworks. In *Acta Informatica 7*, 305–317.

LANDI, W. 1992. Undecidability of static analysis. 1992. *Lett. Program. Lang. Syst. 1*, 4 (Dec.).

LANDI, W. 1991. Interprocedural aliasing in the presence of pointers. Ph.D. thesis, Dept. of Computer Science, Rutgers Univ., New Brunswick, N.J.

LANDI, W. AND RYDER, B. G. 1992. A safe approximate algorithm for pointer-induced aliasing. *SIGPLAN Not. 27*, 7 (July), 235–248.

LARUS, J. R. 1989. Restructuring symbolic programs for concurrent execution on multiprocessors. Ph.D. thesis, Univ. of California, Berkeley, Calif. (May).

LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflicts between structure accesses. *SIGPLAN Not. 23*, 7 (July), 21–34.

MYERS, E. 1981. A precise inter-procedural data flow algorithm. In *Conference Record of the 8th ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 26–28). ACM, New York.

PFEIFFER, P. 1991. Dependence-based representations for programs with reference variables. Ph.D. dissertation and Tech. Rep. TR-1037, Computer Sciences Dept., Univ. of Wisconsin, Madison, Wis. (Aug.).

Received June 1993; revised October 1993 and March 1994; accepted May 1994

A Generalized Theory of Bit Vector Data Flow Analysis

UDAY P. KHEDKER and DHANANJAY M. DHAMDHERE
Indian Institute of Technology

The classical theory of data flow analysis, which has its roots in unidirectional flows, is inadequate to characterize bidirectional data flow problems. We present a generalized theory of bit vector data flow analysis which explains the known results in unidirectional and bidirectional data flows and provides a deeper insight into the process of data flow analysis. Based on the theory, we develop a worklist-based generic algorithm which is uniformly applicable to unidirectional and bidirectional data flow problems. It is simple, versatile, and easy to adapt for a specific problem. We show that the theory and the algorithm are applicable to all bounded monotone data flow problems which possess the property of the separability of solution.

The theory yields valuable information about the complexity of data flow analysis. We show that the complexity of worklist-based iterative analysis is the same for unidirectional and bidirectional problems. We also define a measure of the complexity of round-robin iterative analysis. This measure, called *width*, is uniformly applicable to unidirectional and bidirectional problems and provides a tighter bound for unidirectional problems than the traditional measure of *depth*. Other applications include explanation of isolated results in efficient solution techniques and motivation of new techniques for bidirectional flows. In particular, we discuss edge splitting and edge placement and develop a feasibility criterion for decomposition of a bidirectional flow into a sequence of unidirectional flows.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*complexity of proof procedures*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Bidirectional data flows, data flow analysis, data flow frameworks

1. INTRODUCTION

Data flow analysis is the process of collecting information about the uses and definitions of data items in a program. This information is put to a variety of uses, viz., program design, debugging, optimization, maintenance, and docu-

This work was done when the first author was a research student at the Indian Institute of Technology.

Authors' addresses: U. P. Khedkar, Department of Computer Science, University of Poona, Pune 411 007, India; email: uday@pucsd.ernet.in; D. M. Dhamdhere, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay 400 076, India; email: dmd@cse.iitb.ernet.in.

mentation. Compilers typically use data flow analysis to collect information for the purpose of code optimization.[1]

Data flows used in code optimization involve *unidirectional* dependencies mostly i.e., the data flow information available at a node in the program flow graph is influenced either by its predecessors or by its successors. Such data flows can be readily classified into *forward* and *backward* data flows [Aho et al. 1986]. In *bidirectional* problems, the information available at a node depends on its predecessors as well as its successors. The Morel and Renvoise Algorithm for partial redundancy elimination [Morel and Renvoise 1979] (also called MRA) is a representative bidirectional problem. The advantage of bidirectional problems is that they unify several optimizations reducing both the size and the running time of an optimizer. For example, MRA unifies the traditional optimizations of code movement, common-subexpression elimination, and loop optimization. The Composite Hoisting and Strength Reduction Algorithm [Joshi and Dhamdhere 1982a; 1982b] unifies code movement, strength reduction, and loop optimization.

Though bidirectional data flow problems have been known for over a decade, it has not been possible to explain the intricacies of bidirectional flows using the traditional theory of data flow analysis. Although a fixed-point solution for a bidirectional problem exists, the flow of information and the safety of an assignment cannot be characterized formally. Because of this theoretical lacuna, efficient solutions to bidirectional problems have not been found though some isolated and ad hoc results have been obtained [Dhamdhere 1988a; Dhamdhere and Patil 1993; Dhamdhere et al. 1992].

In this article we present a theory which handles undirectional as well as bidirectional data flow problems uniformly. Apart from explaining the known results in unidirectional and bidirectional flows, it provides deeper insights into the process of data flow analysis. Though the exposition of the theory is based on the *bit vector problems*, the theory is applicable to all bounded monotone data flow problems which possess the property of separability of solution. Several proofs have been omitted from the article for brevity; they can be found in Khedker and Dhamdhere [1992].

Section 2 introduces MRA which is used as a representative example throughout the article. Section 3 reviews the classical theory of data flow analysis. Section 4 defines bit vector problems formally, generalizes the traditional concepts, and provides generic data flow equations which facilitate uniform specification of data flow problems. A worklist-based generic algorithm is developed in Section 5. Arising out of a generalized theory, it is uniformly applicable to unidirectional and bidirectional data flow problems. This section also analyses the performance of the generic algorithm and shows that the complexity of the worklist-based iterative analysis is the same for unidirectional and bidirectional problems.

[1]Data flow analysis can be either *interprocedural* or *intraprocedural*. We restrict ourselves to the latter in this article except that the interprocedural information at the entry/exit of a procedure is considered for analyzing data flows within the procedure.

Section 6 discusses several applications of the generalized theory in the complexity of data flow analysis. A new measure called the *width* (w) of a graph for a data flow framework is defined which is shown to bound the number of iterations of round-robin analysis for unidirectional and bidirectional problems. We show that the width provides a tighter bound for unidirectional problems than the traditional measure of *depth*. This section also explains several known results in the solution of bidirectional data flows, viz., edge splitting and edge placement, and develops a feasibility criterion for decomposition of bidirectional flows into a sequence of unidirectional flows. Section 7 discusses the significance and applicability of the results presented in this article.

2. BIDIRECTIONAL DATA FLOWS: AN EXAMPLE

This section introduces the Morel and Renvoise Algorithm (MRA) [Morel and Renvoise 1979] for partial redundancy elimination which is used as a representative bidirectional problem throughout the article.

MRA unifies the traditional optimizations of code movement, common-subexpression elimination, and loop optimization. The importance of the MRA framework lies in the fact that unification of many classical optimizations reduces the size as well as the running time of an optimizer; a 35% reduction in the size and a 30% to 70% reduction in the execution cost has been reported in the literature [Morel and Renvoise 1979]. It has been implemented in at least two important production compilers (MIPS and PL.8) and has inspired several other unifications [Chow 1988; Dhamdhere 1988a; Joshi and Dhamdhere 1982a; 1982b].

The data flow properties and the data flow equations for MRA are given in Figure 1. Note that PPIN_i is the bit vector for node i which represents the property PPIN_i for *all expressions*, whereas PPIN_i^l is the bit representing the expression e_l.

Local property ANTLOC_i^l represents *local anticipability*, i.e., the existence of an upward exposed expression e_l in node i, while TRANSP_i^l reflects *transparency*, i.e., the absence of definition(s) of any operand(s) of e_l in the node. The global property of *anticipability* ($\text{ANTIN}_i^l/\text{ANTOUT}_i^l$) indicates whether expression e_l is very busy at the entry/exit of node i—a necessary and sufficient condition for the safety of placing an evaluation of e_l at the entry/exit of the node [Kennedy 1972]. Equations (1) and (2) do not use $\text{ANTIN}_i^l/\text{ANTOUT}_i^l$ properties explicitly; they are implied by $\text{PPIN}_i^l/\text{PPOUT}_i^l$ properties. The data flow property of *availability* ($\text{AVIN}_i^l/\text{AVOUT}_i^l$) is computed using the classical forward data flow problem [Aho et al. 1986]. The partial redundancy of an expression is represented by the *partial availability* of the expression (PAVIN_i^l) at the entry of node i. PPIN_i^l indicates the feasibility of placing an evaluation of e_l at the entry of i while PPOUT_i^l indicates the feasibility of placing it at the exit. Computations of an expression e_l are inserted at the exit of node i if $\text{INSERT}_i^l = \mathbf{T}$. REDUND_i^l indicates that the upward exposed occurrence of e_l in node i is redundant and may be deleted.

LOCAL DATA FLOW PROPERTIES :

ANTLOC_i^l	Node i contains a computation of e_l, not preceded by a definition of any of its operands.
COMP_i^l	Node i contains a computation of e_l, not followed by a definition of any of its operands.
TRANSP_i^l	Node i does not contain a definition of any operand of e_l.

GLOBAL DATA FLOW PROPERTIES :

$\text{AVIN}_i^l/\text{AVOUT}_i^l$	e_l is available at the entry/exit of node i.
$\text{PAVIN}_i^l/\text{PAVOUT}_i^l$	e_l is partially available at the entry/exit of node i.
$\text{ANTIN}_i^l/\text{ANTOUT}_i^l$	e_l is anticipated at the entry/exit of node i.
$\text{PPIN}_i^l/\text{PPOUT}_i^l$	Computation of e_l may be placed at the entry/exit of node i.
INSERT_i^l	Computation of e_l should be inserted at the exit of node i.
REDUND_i^l	First computation of e_l existing in node i is redundant.

DATA FLOW EQUATIONS :

$$\text{PPIN}_i = \text{PAVIN}_i \cdot (\text{ANTLOC}_i + \text{TRANSP}_i \cdot \text{PPOUT}_i) \cdot \prod_{j \in pred(i)} (\text{AVOUT}_j + \text{PPOUT}_j) \qquad (1)$$

$$\text{PPOUT}_i = \prod_{k \in succ(i)} (\text{PPIN}_k) \qquad (2)$$

$$\text{INSERT}_i = \text{PPOUT}_i \cdot \neg\text{AVOUT}_i \cdot (\neg\text{PPIN}_i + \neg\text{TRANSP}_i)$$

$$\text{REDUND}_i = \text{PPIN}_i \cdot \text{ANTLOC}_i$$

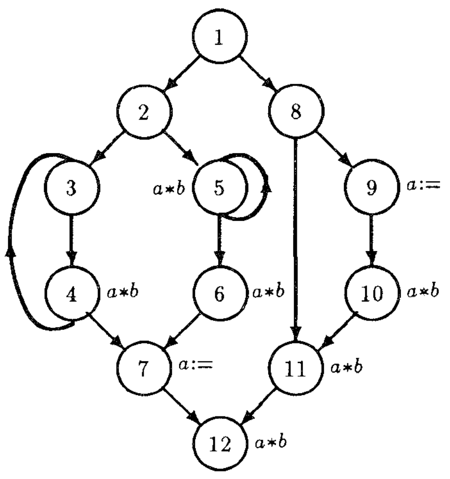
Fig. 1. The Morel-Renvoise algorithm.

The PPIN_i equation is slightly different from the original equation in MRA; the term $\text{ANTIN}_i \cdot (\text{PAVIN}_i + \neg\text{ANTLOC}_i \cdot \text{TRANSP}_i)$ in the original MRA equations is replaced by the term PAVIN_i to prohibit redundant hoisting when the expression is not partially available. The PAVIN_i term represents the *profitability* of hoisting in that there exists at least one possible execution path along which the expression is computed more than once. The other two terms in the PPIN_i equation represent the *feasibility* of hoisting.

Bidirectional dependencies of MRA arise as follows. *Redundancy* of an expression is based on the notion of availability of the expression which gives rise to forward data flow dependencies (reflected by the $\prod$ term in the PPIN_i equation). The *safety* of code movement is based on the notion of anticipability of the expression which introduces backward dependencies in the data flow problem (reflected by the $\prod$ term in the PPOUT_i equation).

Example 2.1. Consider the program flow graph in Figure 2. The partial redundancy elimination performed by MRA subsumes the following three optimizations:

—*Loop-Invariant Movement*: The computations of a $a * b$ in node 4 and node 5 are hoisted out of the loops and are inserted in node 2 (REDUND_4^l, REDUND_5^l, and INSERT_2^l are **T**).



Node	Transp	Antloc	Pavin	Avout	Ppin	Ppout	Insert	Redund
1	T	F	F	F	F	F	F	F
2	T	F	F	F	F	T	T	F
3	T	F	T	F	T	T	F	F
4	T	T	T	T	T	F	F	T
5	T	T	T	T	T	T	F	T
6	T	T	T	T	T	F	F	T
7	F	F	T	F	F	T	T	F
8	T	F	F	F	F	F	F	F
9	F	F	F	F	F	F	F	F
10	T	T	F	T	F	F	F	F
11	T	T	T	T	F	T	F	F
12	T	T	T	T	T	F	F	T

Fig. 2. Program flow graph and properties for Example 2.1.

—*Code Hoisting*: The partially redundant computation of $a * b$ in node 12 is hoisted to node 7. As a result of suppressing this partial redundancy, the path 1-8-11-12 would have only *one* computation of $a * b$; the unoptimized program has two.

—*Common-Subexpression Elimination*: The totally redundant computation of $a * b$ in node 6 is deleted as an instance of common-subexpression elimination.

Note that the partially redundant computation $a * b$ in node 11 is not suppressed since hoisting it to node 8 would be unsafe—the path 1-8-9 had no computation of $a * b$ in the original program.

Example 2.2. Bidirectional data flows have been used also in register assignment and strength reduction optimizations. Figure 3 presents the data flow equations of two such algorithms. The SPPIN/SPPOUT problem of LSIA

- BASIC LOAD STORE INSERTION ALGORITHM (LSIA) [Dhamdhere 1988b]

$$\begin{aligned} \mathrm{SPPIN}_i &= \prod_{j \in pred(i)} (\mathrm{SPPOUT}_j) \\ \mathrm{SPPOUT}_i &= \mathrm{DPANTOUT}_i \cdot (\mathrm{DCOMP}_i + \mathrm{DTRANSP}_i \cdot \mathrm{SPPIN}_i) \cdot \\ &\quad \prod_{k \in succ(i)} (\mathrm{DANTIN}_k + \mathrm{SPPIN}_k) \end{aligned}$$

- COMPOSITE HOISTING AND STRENGTH REDUCTION ALGORITHM (CHSA) [Joshi and Dhamdhere 1982a; Joshi and Dhamdhere 1982b]

$$\begin{aligned} \mathrm{NOCOMIN}_i &= \mathrm{CONSTA}_i \cdot \mathrm{NOCOMOUT}_i + \\ &\quad \sum_{j \in pred(i)} \mathrm{CONSTB}_i \cdot \mathrm{NOCOMOUT}_j \\ \mathrm{NOCOMOUT}_i &= \mathrm{CONSTC}_i + \mathrm{CONSTD}_i \cdot \mathrm{NOCOMIN}_i + \\ &\quad \sum_{k \in succ(i)} \mathrm{CONSTE}_i \cdot \mathrm{NOCOMIN}_k \end{aligned}$$

Fig. 3. Data flow equations of some other bidirectional problems.

performs sinking of STORE instructions using partial redundancy elimination techniques [Dhamdhere 1988b]. The NOCOMIN/NOCOMOUT problem of CHSA is used to inhibit the placement of an update computation following a high-strength computation [Joshi and Dhamdhere 1982a; 1982b].

3. NOTIONS FROM CLASSICAL DATA FLOW ANALYSIS

This section presents an overview of the classical theory of data flow analysis and compares various solution methods and their complexities. Our description is based mostly on Graham and Wegman [1976], Hecht [1977], and Marlowe and Ryder [1990]. A more detailed treatment can be found in Aho et al. [1986], Graham and Wegman [1976], Hecht [1977], Kam and Ullman [1977], Kildall [1973], Marlowe and Ryder [1990], and Rosen [1980]. The concluding part of this section motivates the need for a more general setting.

3.1 Preliminaries

A data flow framework is defined as a triple $\mathbf{D} = \langle \mathscr{L}, \sqcap, \mathscr{F} \rangle$ (Figure 4). Elements in $\mathscr{L}$ represent the information associated with the entry/exit of a basic block. $\sqcap$ is the set union or intersection operation which determines the way the global information is combined when it reaches a basic block. A function $f_i \in \mathscr{F}$ represents the effect on the information as it flows through basic block i.[2]

[2]Alternatively, the functions can be associated with in-edges (out-edges) of node i for forward (backward) flow problems.