Lattice Frameworks for Multisource and Bidirectional Data Flow Problems

STEPHEN P. MASTICOLA
Siemens Corporate Research
THOMAS J. MARLOWE
Seton Hall University
BARBARA G. RYDER
Rutgers University

Multisource data flow problems involve information which may enter nodes independently through different classes of edges. In some cases, dissimilar meet operations appear to be used for different types of nodes. These problems include *bidirectional* [MR79; DRZ92] and *flow sensitive* [Cal88; SL93] problems as well as many static analyses of concurrent programs with synchronization [CS88; DS91; MR93]. *K-tuple frameworks*, a subset of standard data flow frameworks [MR90; Hec77], provide a natural encoding for multisource problems using a single meet operator. Previously, the solution of these problems has been described as the fixed point of a set of data flow equations. Using our k-tuple representation, we can access the general results of standard data flow frameworks concerning convergence time and solution precision for these problems. We demonstrate this for the bidirectional component of partial redundancy suppression and two problems on the program summary graph. An interesting subclass of k-tuple frameworks, the *join-of-meets* frameworks, are useful for reachability problems, especially those stemming from analyses of explicitly parallel programs. We give results on function space properties for join-of-meets frameworks that indicate precise solutions for most of them will be difficult to obtain.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.4 [**Programming Languages**]: Processors; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages

Additional Key Words and Phrases: Data flow analysis, lattice frameworks

Authors' addresses:
Stephen P. Masticola, Siemens Corporate Research, 755 College Road East, Princeton, NJ 08540 USA. Phone: (609) 734-6594. Email: masticol@scr.siemens.com.
Thomas J. Marlowe, Department of Mathematics and Computer Science, Seton Hall University, South Orange, NJ 07079. Email: marlowe@cs.rutgers.edu.
Barbara G. Ryder, Department of Computer Science, Rutgers University, Hill Center, Busch Campus, Piscataway, NJ 08855. Email: ryder@cs.rutgers.edu.
This research was supported, in part, by National Science Foundation grants CCR90-23628 and CCR92-08632.
This work is relevant to data flow analysis research being conducted within the following ongoing projects within the Software Engineering department at Siemens Corporate Research:

—Program Analysis Tool (PAT) project;
—Fortran Maintenance Environments (FME) project;
—Program Maintenance Environments (PME) project.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1. INTRODUCTION

In complicated program analysis problems, static analysis information at a single statement s often arises independently from multiple sources. For instance, "live variable" information can, in an explicitly parallel program, reach a statement s through either control or synchronization (or both); even in a sequential program, information can reach both from a called procedure and from the statements following the procedure call. Moreover, the effect of this information can depend on the nature of its source or sources. We call such problems *multisource* problems. The main contribution of this paper is to provide a natural data flow framework encoding for multisource problems, called the k*-tupleframework.*

K-tuple frameworks provide a unifying approach for several important classes of problems. The set of multisource problems includes the *bidirectional* problems, in which the information depends upon both control predecessors and successors[1] of s. Many *flow sensitive* interprocedural problems likewise have multisource formulations. Perhaps most importantly, several analyses of explicitly parallel programs can be expressed in these frameworks. We expect problems fitting into the k-tuple framework will become more common as new analyses are designed for concurrent programs, and as more sophisticated and accurate analyses, both intra- and interprocedural, are developed for sequential programs.

Specific examples of multisource problems include:

Bidirectional problems. The "profitable to place" relations *PPOUT* and *PPIN* of Morel and Renvoise's *partial redundancy elimination* analysis [MR79; DRZ92]. These relations identify locations in the program where it is profitable to calculate the value of an expression before it is used, to minimize the total number of such calculations.

Concurrent program analyses. Program statements which must *always happen before* other program statements in concurrent programs. This is variously known as *SCPreserved* [CS88], $Before_{Sync}$, $After_{Sync}$, [DS91], and *B4* [MR93].

. Statement pairs (m, n), where m sets a semaphore, but cannot directly cause n to complete execution. This is the *SemKill* problem of [MR93].

. Statements at which a particular semaphore must be set. This is the *MustSet* problem of [MR93].

. Reaching definitions analysis in the presence of remote procedure calls between tasks [LC91; GS93].

Flow sensitive problems. The *Kill*, $\widehat{Kill}$, and *Use* problems on program summary graphs [Cal88]. In Section 4.2, we show that these problems, which Callahan expressed as data flow equations, can be formulated as a data flow framework. We also show that the corresponding function space is 1-semibounded, but not distributive, as Callahan had originally claimed.

[1] In this article, "predecessor" and "successor" mean immediate reachability between nodes, and "ancestor" and "descendant" refer to transitive reachability.

. Aliasing analysis in the presence of pointers [LR92]. For example, the *may-hold* (conditional may-alias) solution at a return node for a procedure call depends upon *may-hold* at the corresponding call and exit nodes, and is thus a multisource problem. We therefore expect that [LR92] can be reformulated as a k-tuple framework, using a different graph structure to represent the program.

. Sloman and Lake's analysis of constant propagation and liveness in the presence of aliasing [SL93]. The authors' original formulation is actually a lattice framework, though somewhat different from ours. Given this basis, though, their analysis should be easy to re-cast into our model.

Most of these problems had been formulated as problem-specific systems of data flow equations, rather than in explicit data flow frameworks. Algebraic and convergence properties were often either assumed, or inferred without rigorous proof from the forms of these equations, or from some implied underlying semantics not clearly related to the forms of the equations.

1.1 Contributions

The important contribution of this paper is a uniform encoding for these complex problems into a standard data flow framework [MR90; Hec77], enabling the use of previous complexity and precision characterizations. K-tuple frameworks also serve as a model for formalizing future analysis problems. We also define a useful subclass of k-tuple frameworks which seem natural to reachability calculations, *join-of-meets* frameworks (see Section 2.3); these were used in our earlier work on deadlock detection [MR91; MR93].

Our approach of using k-tuple frameworks has several significant benefits:

—By providing a unified formal algebraic model, and by requiring separate consideration of different types of flow, and then their consolidation, it supports reasoning about the semantic correctness of the data flow functions (the abstract semantics) in this type of problem.

—Within the model, the framework often allows convergence and precision properties of fixed-point computation for the data flow solution of a given multisource problem to be proven or disproven with standard techniques; these can then provide guarantees of performance, or identify situations requiring more careful analysis or re-engineering.

—Since it can naturally handle intra-process flow, communication, synchronization, task creation/termination etc. separately, it provides a natural template for statement of data flow problems for explicitly parallel languages.

—By providing a uniform framework, it may suggest situations in which data structures and solution techniques of one problem can be reused or modified to solve another, seemingly unrelated problem.

What k-tuple frameworks do not provide, however, is a new general and efficient algorithm for solving these complicated problems. Where problem-specific solution techniques have been developed, they are likely to be more efficient than any general data flow algorithm applied to the k-tuple formulation. However, the approach may be able to suggest new domains of applicability for some of these techniques. Even so, it should be practical to use general algorithms (after problem-specific optimization) on multisource problems for which no better algorithm exists; such algorithms were in fact used for the analyses of explicitly parallel programs in [Mas93].

1.2 Outline

To outline the presentation of our results, in Section 2, we summarize the standard data flow framework and describe k-tuple frameworks using an example, the multisource problem involved in *B4* analysis [MR93]. We also define *join-of-meets* frameworks, an interesting subset of k-tuple frameworks and show the interrelations between these lattice frameworks. Section 3 presents necessary and sufficient conditions for k-tuple frameworks to be monotone or distributive, and the result that only trivial join-of-meets frameworks are distributive. Section 4 shows how k-tuple frameworks can be used for the partial availability analysis of Morel and Renvoise [MR79] and the flow sensitive, interprocedural analyses of Callahan [Cal88]. Finally, we present our conclusions. Various theoretical results concerning precision and convergence of different standard frameworks are included in the Appendix.

2. DATA FLOW FRAMEWORKS

Normalizing the representations used in static analysis provides an opportunity for researchers to apply previous solution techniques to related problems that can be formulated in the same model and makes possible the use of previously published results on solution accuracy and rate of convergence of fixed point iteration. Our k-tuple and join-of-meets frameworks, explained in this section, are specializations of data flow frameworks in which the functions on certain edges always are combined in specific ways. We present our model of a k-tuple framework through a detailed example of a multisource data flow problem and discuss the subclass of *join-of-meets* frameworks.

2.1 Standard data flow frameworks

Standard *data flow frameworks* [MR90; Hec77] are used to represent problems solvable through fixed point iteration. A standard form of a data flow framework is a quadruple $D = (G, L, F, M)$, where:

—$G = (N, E)$ is a graph structure representing the program.

—$L = (S, \sqcap, \sqcup)$ is a lattice. Depending on the framework designer's intent, the elements in S may represent information that holds over all of G, or at its components (commonly, its nodes).[2]

The *meet operator* $\sqcap$ represents what can be concluded from the logical disjunction of two pieces of information in L. For example, suppose that the members of L represent information that holds at each of the nodes of G. Suppose further that:

—For the nodes n, p, and q, either p or q, but not necessarily both, must finish execution immediately before n starts, and;

[2]Throughout this paper, we assume that the set of data flow equations results from evaluation of flow at a single node (local flow functions), and that each should be naturally expressible in a data flow framework.

—The lattice element s_1 is known to hold at the conclusion of p, and s_2 holds at the conclusion of q.

The lattice element $s_1 \sqcap s_2$ therefore holds when n starts to execute. This means, "the information represented by either s_1 or s_2 must hold for all executions of n, but s_1 and s_2 do not necessarily both hold for all executions of n." $s_1 \sqcup s_2$ correspondingly means, "s_1 and s_2 both hold for all executions of n."

Similar semantics for the meet operator apply for frameworks in which the elements of L represent information that holds over the entire program.

We adopt the convention that the meet of an empty set of elements of S is $S_\top$, the top element of S. This avoids certain technical problems arising from the associativity of the meet operator. Usually, when we adopt these conventions on the meaning of $\sqcap$ and $\sqcup$ and where L represents node information, $S_\top$ also correctly represents the "anything can happen" situation. Upon careful consideration, this semantics usually correctly models a node with no incoming information.[3]

—$F \subseteq L \rightarrow L$ is a monotone function space, which contains the identity function ι, and is closed under meet and function composition.

—$M : 2^N \times 2^E \rightarrow F$ is a partial function which maps a component of G to a function representing how the information is transformed if that component is executed. Often M is defined only on singleton nodes or edges. In *forward* data flow problems, where L represents node information, the function captures the effect of the source node of an edge; in *backward* problems, it captures the effect of the target node.

2.2 K-tuple frameworks

In multisource problems, the edges in E may be members of different *edge classes* (e.g., control and synchronization edges). Each class of edges represents one particular type of interaction between nodes (e.g., control predecessor, synchronization, or the call point/procedure relationship). Thus, the partition of E into edge classes partitions the neighbors of each node n into nodes that interact with n in different ways. Information may be derived at n independently from each of these sets of edges, possibly in different ways, corresponding to the interaction of n's neighbors with n.

We will take as an example the problem of finding pairs of statements in an explicitly parallel program, such that one statement must always occur before the other [MR93; DS91; CS88]. Figure 1 shows a graph G representing a simple program with two tasks, `T1` and `T2`. These tasks synchronize using binary semaphores `sem1` and `sem2`, using Dijkstra's familiar P and V operators [Dij68]. The nodes `begin` and `end` represent the beginning and the end, respectively, of the program. In G, directed edges represent either control flow or synchronization, as noted. We can readily observe that node `b` always happens before `f`, and `h` happens before `d`, due to synchronization. Control flow within `T1` and `T2` imposes further partial orderings between statements.

[3] As an example, consider problems in which E represents control flow. Any non-entry node n with no control predecessors never executes, and is dead code. Therefore, anything is possible when n does execute. (The entry node is usually treated as a special case.) Another illustration of this is seen in the discussion of $PAVIN(y)$ in Section 4.1.1.

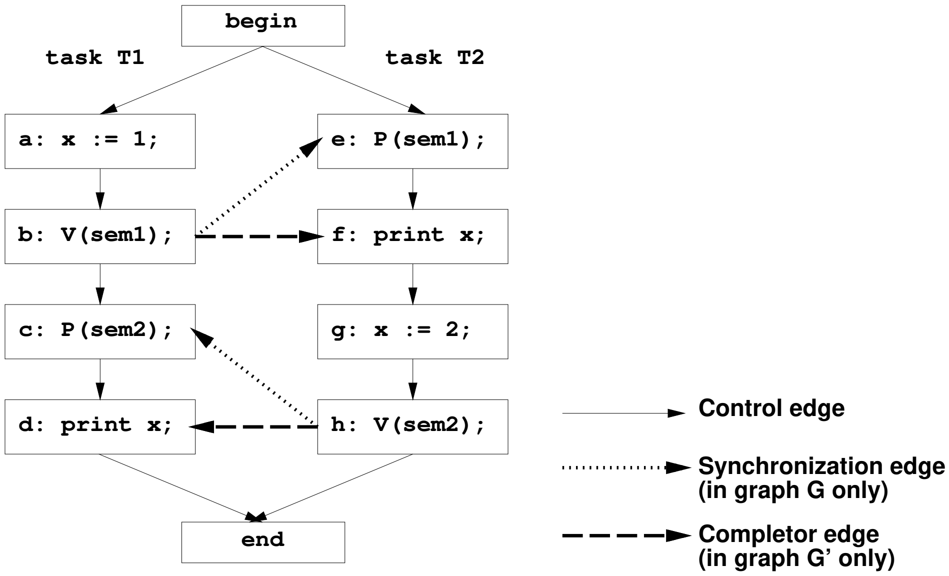

Fig. 1. Graph representation of a program with two synchronizing tasks.

2.2.1 *Graph representation.* A node m *must happen before* n if either (a) m happens before all control predecessors of n, or (b) m happens before all nodes that can enable n to start by synchronizing with m's control predecessors, thus allowing them to complete. Thus, we know that all instances of m happen before any instance of n (denoted $m \in B4(n)$ [MR93]) if either (a) m is either in the $B4$ set of all of n's control predecessors, or (b) m is in the $B4$ set of all the nodes that can synchronize with the most immediate control predecessors of n that wait on synchronization.

2.2.2 *The k-tuple framework.* If we insist on having S directly represent the information of interest and wish to retain this accuracy, we are forced to abandon the data flow framework formulation at the level of nodes and edges, in favor of a set of data flow equations. Although this approach has been the one most commonly used in the literature, it denies us access to the power of the theory developed for data flow frameworks. We might still be able to formulate the problem for the entire graph G as a cross-product data flow framework, in which elements of S are vectors with one element corresponding to the data flow information at each node [MR90; RP86].; however, the resulting framework is very cumbersome to analyze.

Rather than lose information or the ability to use proven general results, we adopt the following conventions for representing information in S, in a k-source problem.

—Information must always propagate across some explicit edge in G; each edge corresponds to one of the k sources of information. If G is not constructed in this manner, we produce a graph $G' = (N, E')$ from G, such that G' has all the necessary edge classes.
Note that G' may be a multigraph, if edges between the same pair of nodes represent different information propagation. We thus make E' a set of triples

(m, n, c), where m and n are nodes in N, and c is an integer representing the class of the edge, $1 \leq c \leq k$.
In graph G of our example (Figure 1), the synchronization edges do not directly represent the "synchronization enabling to complete" relation between nodes, and therefore do not represent information flow in the *B4* problem. To capture this effect directly in the graph structure, we transform G into a graph $G' = (N, E')$. E' contains all the control edges from E, plus *completor edges*, representing synchronization with each node's most immediate control ancestor nodes that represent P operations.[4] These classes of edges partition E' into two subsets of edges, corresponding to the two information sources in the problem. Therefore, for control edges, we let $c = 1$; for completor edges, $c = 2$.

—Given that there are k sources of information in the problem, we represent the information as a lattice of k-tuples. If the lattice $L = (S, \sqcap, \sqcup)$ represents the information of interest, we form a cross product lattice L^k. Each position in a k-tuple in S^k represents propagation from one source, across one edge class.
In the *B4* example, $k = 2$ and facts are sets of nodes, so we use a lattice of 2-tuples, $2^N \times 2^N$. Each element in the lattice is a pair (S_C, S_S). At a node n, the nodes of S_C and S_S are known to be in $B4(n)$ due to control and synchronization information, respectively.

—The edge function f_e for an edge e of type c is of the form,

$$\begin{aligned} f_e((s_1, s_2, ..., s_k)) &= T \\ &= (S_\top, S_\top ... g_e((s_1, s_2, ... s_k)), S_\top ... S_\top) \end{aligned}$$

where all positions of T except the c'th are $S_\top$, and the c'th position is a value $g_e(s_1, s_2, ... s_k)$. g_e is a function of the form $S^k \rightarrow S$, and is assigned to the edge e.
By observing this function assignment, we guarantee that information from each edge class will participate only in the meet for that edge class at the nodes in N; we thus maintain accuracy by meeting information from different sources separately.
In the *B4* example, for $S_C \subseteq N$ and $S_S \subseteq N$, $f_e((S_C, S_S)) = (S_C \cup S_S \cup Q_e, N)$ for control edges, and $f_e((S_C, S_S)) = (N, S_C \cup S_S \cup Q_e)$ for completor edges. Q_e serves as a "generate" set for the edge. For a control edge $(m, n, 1)$, Q_e is the set of nodes n' such that there is a control path from n' to n, but not vice versa; all instances of n' must occur before any instances of n, and thus, $n' \in B4(n)$. For a completor edge $(m, n, 2)$, Q_e similarly contains the set of nodes in m's task which must finish execution before n, assuming that m sets the semaphore that allows n to complete. Thus, Q_e contains at least the set of nodes m' in the task of m such that there is a control path from m' to m, but not vice versa.
If m has at most a single instance in any execution in the program, and if m enables n to complete, then m itself must also precede n. Thus, m is contained in Q_e for the completor edge $(m, n, 2)$, if and only if m is not contained in any control loops.
Table I shows the Q_e sets for the edge functions created for the example of Figure 1.

[4]Redundant completor edges are suppressed in the example.

Edge	Q_e
`(begin,a,1)`	`{begin}`
`(a,b,1)`	`{begin,a}`
`(b,c,1)`	`{begin,a,b}`
`(c,d,1)`	`{begin,a,b,c}`
`(d,end,1)`	`{begin,a,b,c,d}`
`(begin,e,1)`	`{begin}`
`(e,f,1)`	`{begin,e}`
`(f,g,1)`	`{begin,e,f}`
`(g,h,1)`	`{begin,e,f,g}`
`(h,end,1)`	`{begin,e,f,g,h}`
`(b,f,2)`	`{a,b}`
`(h,d,2)`	`{e,f,g,h}`

Table I. Q_e sets for edge functions for the example program of Figure 1.

—Occasionally, the graph G' must be such that some nodes cannot legitimately be the endpoints of any edges in some class c. If n is such a node, then position c in n's incoming k-tuples will always be $S_\top$. However, this has the undesirable semantics that we know *all possible information* at n.
We can conceptually fix this problem by creating dummy edges $e_d = (n, n, c)$ of type c that have n as both its endpoints. We assign the constant function $g_\perp$ to e_d, which always returns $S_\perp$ (the bottom element of S). Thus, f_{e_d} is a constant function which produces a k-tuple with $S_\perp$ in the c-th position and $S_\top$ in all others.[5] We thus avoid adding any spurious incoming information to n's value.
In the example of Figure 1, dummy completor edges would be added for all nodes shown that are not the targets of the completor edges shown. A dummy control edge (`begin`, `begin`, 1) is also created, since the `begin` node has no control predecessor.

The above conventions define the mapping function M for the data flow framework. The function space F should contain at least the closure of the set of possible edge functions under meet and function composition; in many instances, we can define F to be precisely this set. A precise characterization of F helps in proving tight convergence time and accuracy bounds for the framework, as we discuss further in Section 3.

Table II shows the fixed-point solution and *B4* sets for the example of Figure 1.

2.2.3 *Tuple width and problem size.* It is not always possible to specify a constant tuple width (k) for all instances of a given problem representable by k-tuple frameworks. The example of Figure 1 bears this out, through a minor defect in the solution of the *B4* problem as given.

In Table II, the *B4* solution for the `end` node omits nodes `c` and `d`, which must complete before the program ends. This defect occurs because we have used control edges to represent task initiation and completion, as well as flow of control, for the sake of a simple example. Properly, the *B4* information at `end` should be the join, rather than the meet, of the *B4* information coming from each task. If the program

[5] In practical fixed-point iteration, we would simply note that the c-th position of n's value is always $S_\perp$, rather than adding dummy edges.

Node	Fixed Point	*B4*
`begin`	$(\emptyset, \emptyset)$	$\emptyset$
`a`	({`begin`}, $\emptyset$)	{`begin`}
`b`	({`begin, a`}, $\emptyset$)	{`begin, a`}
`c`	({`begin, a, b`}, $\emptyset$)	{`begin, a, b`}
`d`	({`begin, a, b, c`}, {`begin, a, b, e, f, g, h`})	{`begin, a, b, c, e, f, g, h`}
`e`	({`begin`}, $\emptyset$)	{`begin`}
`f`	({`begin, e`}, {`begin, a, b`})	{`begin, a, b, e`}
`g`	({`begin, a, b, e, f`}, $\emptyset$)	{`begin, a, b, e, f`}
`h`	({`begin, a, b, e, f, g`}, $\emptyset$)	{`begin, a, b, e, f, g`}
`end`	({`begin, a, b, e, f, g, h`}, $\emptyset$)	{`begin, a, b, e, f, g, h`}

Table II. Fixed-point solution and *B4* sets for the example of Figure 1.

has a total of t tasks, we would then need at least $\max(t, 2)$ edge types, to represent the join of *B4* information at the `end` node.

This difficulty with the *B4* problem in the tasking model shown, can be easily resolved by treating the `end` node as a special case. However, the need for varying tuple width arises if we use a *fork-join* model of tasking, in which an arbitrary (though statically determined) number of threads can be created and terminated dynamically. Varying tuple widths are also needed to represent those *and-or* graph problems in which both the *and* and the *or* nodes may have an arbitrary number of incoming edges. In both cases, k may vary with the size of N, or with some other parameter of the problem size.

2.2.4 *Advantages of our formulation.* Many multisource data flow formulations lend themselves to intuitively simple, but difficult to analyze, systems of data flow equations. For example, consider the *Kill* data flow problem of [Cal88], in which call nodes have the logical *and* as their meet operator, and other types of nodes have logical *or.* [6] The lattices for the *and* nodes have *False* as the bottom element; the lattices for the *or* nodes have *True* as the bottom element. There is no single node meet operator. Therefore, if we insist on representing information at nodes as a single Boolean variable, no data flow lattice can represent the problem, which effectively denies us the use of the analytical tools already developed for these problems.

In a k-tuple framework, each edge function has the same properties in all of the node lattice spaces, since all these spaces are identical. Thus, the properties of the data flow problem (e.g., monotonicity, closure under meet and function composition, k-boundedness) can often be derived directly from those of its function space.

2.3 Join-of-meets problems

A *join-of-meets* data flow problem is a multisource problem in which some piece of information holds at a node if it holds for any of the k sources. For example, in the *B4* problem, m happens before n if it happens before either all completors or all

[6]It is possible to consider the *and* as building a summary function for the procedure call site, rather than as an alternative meet operator; however, deducing the underlying data flow framework from the equations given in a straightforward manner leads to the conclusions cited above.

control predecessors of n. These problems are quite common. *B4*, *SemKill*, and *MustSet* [MR93], the *Kill*, $\widehat{Kill}$, and *Use* problems on program summary graphs [CS88], and intertask reaching definitions problems [LC91; GS93] are all join-of-meets problems.

A k-tuple framework can represent join-of-meets problems by simply using the function g_e to perform the joins. The effect of the node and edge are simulated by applying a function h_e (with signature $S \to S$) to the conjunction of all s_i. g_e is defined in terms of h_e:

$$g_e(s_1, s_2, ... s_k) = h_e(s_1 \sqcup s_2 \sqcup ... \sqcup s_k).$$

This specialized definition of g_e defines the *join-of-meets framework* which obviously is a type of k-tuple framework. In Section 3, we examine solution efficiency and accuracy issues for join-of-meets frameworks.

2.3.1 *Relationships of framework models.* The interrelationships of the frameworks we have defined in this section are expressed in the inclusion relation pictured in Figure 2. There are other lattice formulations which use a single global function to express the data flow solution for the entire program. These are usually formulated as cross product frameworks, as mentioned above. This representation is no more powerful than the one we are using, but is more difficult to use to reason about properties.

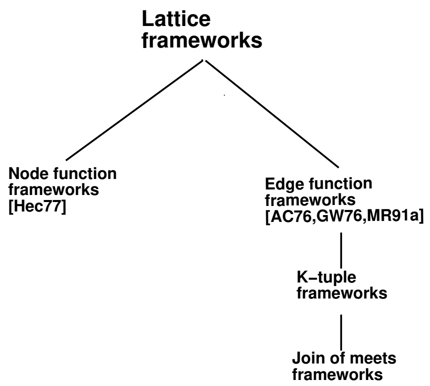

Fig. 2. Data flow frameworks and their interrelation.

3. PROPERTIES OF K-TUPLE FRAMEWORKS

In Section 2.2.2, we defined the edge functions f_e of k-tuple frameworks to depend on a function g_e, with signature $S^k \to S$. Here, we investigate what properties of g_e are necessary to assure the important properties of the k-tuple framework, (i.e., monotonicity and distributivity). We will examine join-of-meets problems as an important special case. In this section we will be using some standard results listed in the Appendix.

3.1 Monotonicity

The function spaces of data flow frameworks must be monotone, if they are to converge to a fixed point. Here we show a simplified way to prove that k-tuple and join-of-meets frameworks are monotone.

Lemma 8 in the Appendix can easily be made specific to k-tuple frameworks to show monotonicity and assure termination. To do this, we give sufficient conditions for the edge functions to be monotone:

LEMMA 1. *f_e is monotone iff, for all $s_i \in S, 1 \leq i \leq k$ and for all $s'_i \in S$ such that $s_i \sqsubseteq s'_i$, $g_e(s_1, s_2, ...s_k) \sqsubseteq g_e(s'_1, s'_2, ...s'_k)$.*

Proof: From the definitions of monotonicity and f_e. □

For join-of-meets frameworks, the proof of monotonicity is even simpler:

COROLLARY 1. *In a join-of-meets framework, f_e is monotone iff h_e is monotone.*

Proof:

$$g_e(s_1, s_2, ...s_k) = h_e(s_1 \sqcup s_2 \sqcup ... \sqcup s_k)$$

and

$$g_e(s'_1, s'_2, ...s'_k) = h_e(s'_1 \sqcup s'_2 \sqcup ... \sqcup s'_k).$$

We know that, since each $s_i \sqsubseteq s'_i$,

$$s_1 \sqcup s_2 \sqcup ... \sqcup s_k \sqsubseteq s'_1 \sqcup s'_2 \sqcup ... \sqcup s'_k.$$

The rest follows from the definition of monotonicity. □

3.2 Distributivity

Distributivity of the function space is of special importance for two reasons. First, if a data flow framework is distributive, its maximum fixed point (MFP) is equivalent to its meet-over-all-paths (MOP) solution; that is, an iterative solution method calculates an exact solution. Second, the Kam-Ullman rapidity result [KU76], which can provide useful upper bounds on convergence time, requires distributivity as a precondition. Therefore, we would like to show whether, and under what conditions, k-tuple and join-of-meets frameworks are distributive.

LEMMA 2. *f_e distributes over meet iff, for all $s_i, s'_i \in S, 1 \leq i \leq k$,*

$$g_e(s_1 \sqcap s'_1, s_2 \sqcap s'_2, ...s_k \sqcap s'_k) = g_e(s_1, s_2, ...s_k) \sqcap g_e(s'_1, s'_2, ...s'_k).$$

Proof: From the definitions of f_e and distributivity. □

Unfortunately, this is a very stringent requirement, and has bad implications for join-of-meets frameworks:

COROLLARY 2. *In a join-of-meets framework where $k > 1$, f_e distributes over meet iff h_e has a constant value.*

Proof: The "if" direction is trivial. For the "only if" direction, without loss of generality, consider a framework where $k = 2$. Suppose that there is some $s \in S$ such that $h_e(s) \neq h_e(S_\perp)$. Then:

$$f_e((s, S_\perp) \sqcap (S_\perp, s)) \;=\; f_e((S_\perp, S_\perp))$$

$$\begin{aligned}
&= (h_e(S_\perp \sqcup S_\perp), S_\top) \\
&= (h_e(S_\perp), S_\top).
\end{aligned}$$

$$\begin{aligned}
f_e((s, S_\perp)) \sqcap f_e((S_\perp, s)) &= (h_e(s \sqcup S_\perp), S_\top) \sqcap (h_e(S_\perp \sqcup s), S_\top) \\
&= (h_e(s), S_\top) \sqcap (h_e(s), S_\top) \\
&= (h_e(s), S_\top) \\
&\neq f_e((s, S_\perp) \sqcap (S_\perp, s)).
\end{aligned}$$

Thus, f_e is not distributive over meet. □

This implies that the Kam-Ullman rapidity result cannot be applied to any non-trivial k-tuple framework, since distributivity is needed to show rapidity. Although this result may disappoint those who are looking for quick convergence time, it may save some wasted effort.

3.3 Boundedness and convergence

It is important to be careful in formulating data flow problems to capture as many properties as possible of the function space. This is useful both to validate the correctness, precision and complexity of problems not previously expressed as a formal data flow framework and to help formulate new problems by directing researchers to reasonable solution procedures. Since join-of-meets problem are rarely, if ever, distributive, careful formulation may allow one to use other function space properties to aid in problem solution. Simpler complexity bounds can be obtained for problems defined on well-structured (i.e., reducible) flow graphs; for example, although reducibility is not required for the Kam-Ullman results on the complexity of fixed point iteration on distributive data flow frameworks [KU76], this graph structure simplifies the boundedness result obtained. The acyclicity of a flow graph can similarly simplify the convergence of fixed point iteration (as is the case in Section 4.2).

Since a boundedness condition for the set of edge functions is not sufficient to prove a similar boundedness condition for the framework (Appendix, Lemma 11), it is difficult to prove that a particular framework is, for example, either Graham-Wegman fast or Kam-Ullman rapid from the properties of the edge functions alone, without either (a) elaborating their closure under meet and function composition, or (b) showing that the boundedness properties follow from more specialized properties of the edge functions, and that those properties are preserved in the closure. Indeed, for the special case of non-trivial join-of-meets frameworks, we have shown that Kam-Ullman rapidity does not hold; we must use other properties to prove good time bounds. However, it may be possible to prove distributivity for frameworks built upon other classes of edge functions, using Lemma 2.

Section 4.1 shows a case where the closure can be easily elaborated. Examples of boundedness properties preserved by closure or of algebraic properties which are preserved and imply boundedness include *1-boundedness* [MR90] or *inflationary* *property* [Mas93] [7] and the *meet-with-constant* algebraic form. [8]

K-tuple frameworks do offer one important advantage over other approaches to formulation of complex data flow problems in proving boundedness properties. Unlike ad-hoc data flow equations, they allow the function space to be defined and reasoned about. As Corollary 2 shows, they also have an important advantage in *disproving* conditions such as Kam-Ullman rapidity.

4. EXAMPLE REFORMULATIONS OF TWO PUBLISHED DATA FLOW PROBLEMS

In this section, we reformulate two published data flow problems as multisource frameworks. We use these frameworks to prove theoretical properties about the data flow problems, refining the results of the original authors.

4.1 Morel and Renvoise

In this section, we show how to represent the bidirectional data flow problem specified by Morel and Renvoise [MR79] in a k-tuple lattice framework model. The *available expressions* problem [ASU86] finds statements s in the program which use an expression e, such that every path to s contains a previous use of e with no intervening definition of the variables in e. Such statements can use these previous values of e, rather than computing their own. Morel and Renvoise [MR79] extend this problem to *partial availability of expressions*, i.e., statements s for which at least one ancestor of s computes the value of e. Morel and Renvoise's goal in computing partial availability was to find program points at which it would be *profitable to place* computations of e, to minimize the number of times e is computed in program execution. The data flow equations for the profitability function, taken from [DRZ92], are given in Equations 1 through 8 below; the equations are formulated in terms of the following Boolean predicates:

—$AVIN(y)$: e is available at the top of y.

—$AVOUT(y)$: e is available at the bottom of y.

—$NONE(y)$: y does not modify any of the variables of e.

—$AVGEN(y)$: y uses the value of e and does not overwrite any of e's variables before it does.

—$PAVIN(y)$: e is partially available at the top of y.

—$PAVOUT(y)$: e is partially available at the bottom of y.

—$PPIN(y)$: It is profitable to place a computation of e at the bottom of y.

—$PPOUT(y)$: It is profitable to place a computation of e at the the top of all successors of y.

—$CB4(y)$: y uses e and does not redefine any of e's variables before it does. (y *sees* e *before* it changes e's variables.)

—$INSERT(y)$: The optimization inserts a computation of e at the bottom of y.

—$DELETE(y)$: The optimization deletes a computation of e at the bottom of y.

[7] *i.e.*, $\forall x : f(x) \geq x$

[8] *i.e.*, functions of the form $\{ f_c(x) = x \sqcap c \mid c \in L \}$ [Zad83].

$$AVIN(y) = \begin{cases} \mathit{False} & \text{if } y = \texttt{Entry} \\ \displaystyle\bigwedge_{x \in Pred(y)} \begin{array}{l} AVGEN(x) \\ \vee [AVIN(x) \wedge NONE(x)] \end{array} & \text{if } y \neq \texttt{Entry} \end{cases} \tag{1}$$

$$AVOUT(y) = AVGEN(y) \vee [AVIN(y) \wedge NONE(y)] \tag{2}$$

$$PAVIN(y) = \begin{cases} \mathit{False} & \text{if } y = \texttt{Entry} \\ \displaystyle\bigvee_{x \in Pred(y)} \left(\begin{array}{l} AVGEN(x) \\ \vee [PAVIN(x) \wedge NONE(x)] \end{array} \right) & \text{if } y \neq \texttt{Entry} \end{cases} \tag{3}$$

$$PAVOUT(y) = AVGEN(y) \vee [PAVIN(y) \wedge NONE(y)] \tag{4}$$

$$PPIN(y) = \begin{cases} \mathit{False} & \text{if } y = \texttt{Entry} \\ \left(\begin{array}{l} PAVIN(y) \wedge \\ \left(\begin{array}{l} CB4(y) \\ \vee [PPOUT(y) \wedge NONE(y)] \end{array} \right) \wedge \\ \displaystyle\bigwedge_{x \in Pred(y)} [PPOUT(x) \wedge AVOUT(x)] \end{array} \right) & \text{if } y \neq \texttt{Entry} \end{cases} \tag{5}$$

$$PPOUT(y) = \begin{cases} \mathit{False} & \text{if } y = \texttt{Exit} \\ \displaystyle\bigwedge_{z \in Succ(y)} PPIN(z) & \text{if } y \neq \texttt{Exit} \end{cases} \tag{6}$$

$$INSERT(y) \stackrel{\text{def}}{=} PPOUT(y) \wedge \neg AVOUT(y) \wedge \neg (PPIN(y) \wedge NONE(y)) \tag{7}$$

$$DELETE(y) \stackrel{\text{def}}{=} PPIN(y) \wedge CB4(y) \tag{8}$$

$AVIN$ and $PAVIN$ can be computed using conventional node-function or edge-function lattice frameworks. $AVOUT$, $PAVOUT$, $INSERT$, and $DELETE$ can be computed in closed form if their component variables are known. However, $PPIN$ and $PPOUT$ form an equation system that cannot be cast in the form of a single-source lattice framework. Instead, we form their lattice on a transformed control flow graph, using two edge types that represent forward and backward control edges, respectively.

Figure 3 shows the graph transform used. Each control flow edge $e = (x, y)$ is replaced by an edge $e_t = (x, y)$ representing forward data flow and an edge $e_i = (y, x)$ representing backward data flow.

To compute $PPIN(y)$, Equation (5) requires the intersection of the value $t_x = PPOUT(x) \wedge AVOUT(x)$ for all predecessors of x (as well as $PPOUT(y)$ and some values that remain constant during the solution of Equations (5) and (6)). Similarly, to compute $PPOUT(y)$, Equation (6) requires the intersection of $i_z = PPIN(z)$ for all successors z of y. $PPOUT(y)$ can be computed using Equation (6) and substituted into Equation (5) to obtain $PPIN(y)$.

The information kept at the nodes must be sufficient to allow the edge functions to produce these t and i values, given the other values that remain constant during the solution of Equations (5) and (6). The intersection of the t values of predecessors, and the intersection of the i values of successors, is sufficient for this edge function computation. Following generation and propagation of these values by the edge functions, the lattice meet operator should find the intersection separately for the predecessors and successors of y.

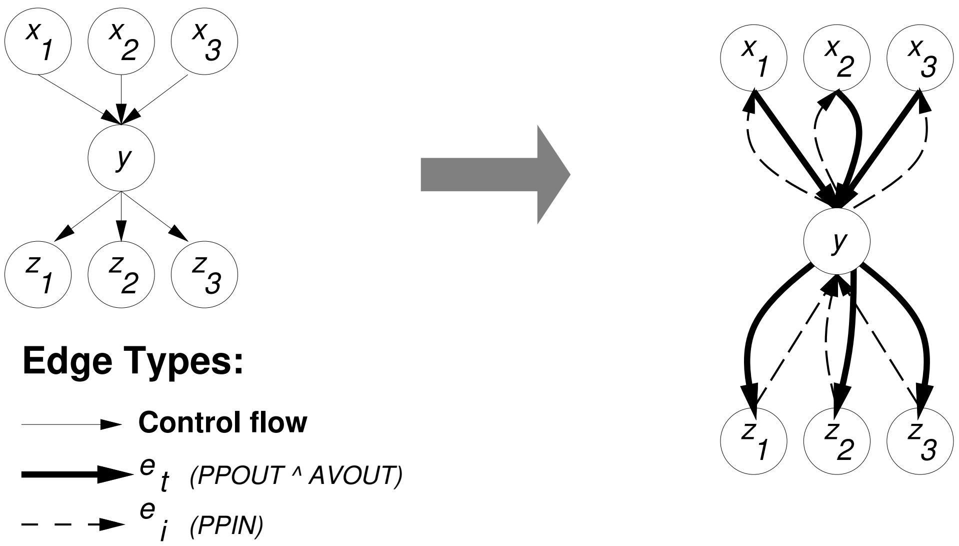

Fig. 3: Graph transform to formulate the *PPIN* and *PPOUT* data flow equations as a lattice framework.

Since two sources of information (from predecessors and successors) are used separately at y, we might think to formulate the problem using a lattice of pairs of pairs of bits, $((t_{pred}, i_{pred}), (t_{succ}, i_{succ}))$. We observe, though, that the resulting edge functions would not use i_{pred} or t_{succ}, so we need not include them in the lattice elements.

The lattice L_{PP} (shown in Figure 4) is, therefore, a lattice of pairs of bits (t, i). At a node y, t represents the intersection of $PPOUT(x) \wedge AVOUT(x)$ from y's control predecessors x, and i represents $PPIN(z)$ information from y's control successors z. Meet is the bit-wise $\wedge$ operator; top, bottom, and the ordering relation $\sqsubseteq$ are defined accordingly. $PPIN(y)$ and $PPOUT(y)$ can be computed from these values, and other values that remain constant during the iteration, using Equation (5) and Equation (6) respectively. The edge functions, in fact, compute $PPIN(y)$ and $PPOUT(y)$ given these values as inputs.

We define the mapping function $M_{PP}(e)$ to map an edge $e = (y, w)$ to an edge function f_e as follows:

$$f_e((t,i)) = \begin{cases} \left(\begin{matrix} [i \wedge AVOUT(y)]\,, \\ True \end{matrix} \right) & \begin{array}{l} \text{for } e = e_t, \text{ representing} \\ PPOUT(y) \wedge AVOUT(y), \\ \text{where } y \neq \texttt{Exit}, \end{array} \\ \\ \left(\begin{matrix} True, \\ \left[\begin{array}{l} PAVIN(y) \\ \wedge \left(\begin{array}{l} CB4(y) \\ \vee [i \wedge NONE(y)] \end{array} \right) \\ \wedge t \end{array} \right] \end{matrix} \right) & \begin{array}{l} \text{for } e = e_i, \\ \text{representing } PPIN(y) \\ \text{where } y \neq \texttt{Entry} \end{array} \end{cases} \tag{9}$$

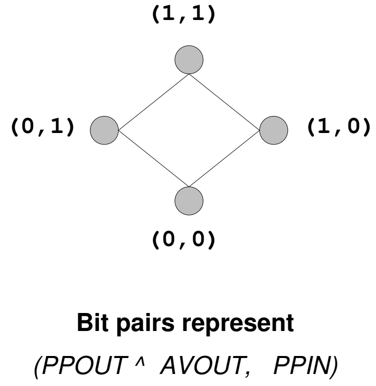

Fig. 4. The lattice L_{PP}.

For an edge $e = e_t = (\texttt{Exit}, w)$ representing $PPOUT(\texttt{Exit}) \wedge AVOUT(\texttt{Exit})$, $f_e((t,i)) = (\mathit{False}, \mathit{True})$. For $e = e_i = (\texttt{Entry}, w)$ representing $PPIN(\texttt{Entry})$, $f_e((t,i)) = (\mathit{True}, \mathit{False})$.

Lemma 3 shows that the desired solution is a fixed point of the lattice framework.

LEMMA 3. *Let G be a flow graph and G' be G as transformed by the mapping of Figure 3. Let $Pred_G(y)$ denote the predecessors of y in G, and $Succ_G(y)$ denote the successors of y in G. The assignment of the element $(\wedge_{x \in Pred_G(y)}[PPOUT(x) \wedge AVOUT(x)], \wedge_{z \in Succ_G(y)} PPIN(z))$ of L to each node y is a fixed point of the framework $(G', L_{PP}, F_{PP}, M_{PP})$.*

Proof: We assume that elements of L_{PP} are assigned to nodes as stated, and show that the assigned value is maintained for each node.

Let y be any node in G'; y is also in G. By assumption, each $x \in Pred_G(y)$, the value $(t_x, i_x) = (\wedge_{w \in Pred_G(x)}[PPOUT(w) \wedge AVOUT(w)], \wedge_{v \in Succ_G(x)} PPIN(v))$ is assigned to x. By Equation (6), $i_x = PPOUT(x)$.

Since $x \in Pred_G(y)$, an edge $e_{(x,y,t)} = (x,y) \in e_t$ exists in G'. (No edge $(x,y) \in e_i$ exists in G' unless x is also in $Succ_G(y)$.) By definition of M_{PP}, if $x \neq \texttt{Exit}$, then $e_{(x,y,t)}$ is assigned the function:

$$\begin{aligned} f_{e_{(x,y,t)}}(t_x, i_x) &= \begin{pmatrix} [i_x \wedge AVOUT(x)], \\ \mathit{True} \end{pmatrix} \\ &= \begin{pmatrix} [PPOUT(x) \wedge AVOUT(x)], \\ \mathit{True} \end{pmatrix} \end{aligned}$$

If $x = \texttt{Exit}$, then

$$\begin{aligned} f_{e_{(x,y,t)}}(t_x, i_x) &= \begin{pmatrix} \mathit{False}, \\ \mathit{True} \end{pmatrix} \\ &= \begin{pmatrix} [PPOUT(x) \wedge AVOUT(x)], \\ \mathit{True} \end{pmatrix}, \end{aligned}$$

since $PPOUT(\texttt{Exit}) = \mathit{False}$.

Similarly, for each $z \in Succ_G(y)$, the value $(t_z, i_z) = (\wedge_{r \in Pred_G(z)}[PPOUT(r) \wedge AVOUT(r)], \wedge_{s \in Succ_G(z)} PPIN(s))$ is assigned to z. By Equation (6) again, $i_z = PPOUT(z)$. Since $z \in Succ_G(y)$, an edge $e_{(z,y,i)} = (z, y) \in e_i$ exists in G'. By definition of M_{PP}, if $z \neq$ `ENTRY`, then $e_{(z,y,i)}$ is assigned the function:

$$
\begin{aligned}
f_{e_{(z,y,i)}}(t_z, i_z) &= \begin{pmatrix} \mathit{True}, \\ [PAVIN(z) \wedge (CB4(z) \vee [i_z \wedge NONE(z)]) \wedge t_z] \end{pmatrix} \\
&= \begin{pmatrix} \mathit{True}, \\ \begin{bmatrix} PAVIN(z) \wedge (CB4(z) \vee [PPOUT(z) \wedge NONE(z)]) \wedge \\ \bigwedge_{w \in Pred_G(z)} [PPOUT(w) \wedge AVOUT(w)] \end{bmatrix} \end{pmatrix} \\
&= \begin{pmatrix} \mathit{True}, \\ PPIN(z) \end{pmatrix}
\end{aligned}
$$

If $z =$ `ENTRY`, then

$$
\begin{aligned}
f_{e_{(z,y,t)}}(t_z, i_z) &= \begin{pmatrix} \mathit{True}, \\ \mathit{False} \end{pmatrix} \\
&= \begin{pmatrix} \mathit{True}, \\ PPIN(z) \end{pmatrix}
\end{aligned}
$$

since $PPIN($`ENTRY`$) = \mathit{False}$.

Thus, for node y, the meet of the edge function values assigned to the edges from predecessors of y in G' is $(\wedge_{x \in Pred_G(y)}[PPOUT(x) \wedge AVOUT(x)], \wedge_{z \in Succ_G(y)} PPIN(z))$. Thus, the assignment of this value to every node y is a fixed point of the framework. □

4.1.1 *Computing $INSERT(y)$ and $DELETE(y)$.* Equation (7) requires $PPOUT(y)$ and $PPIN(y)$ as inputs to compute $INSERT(y)$. At the fixed point, the value of i_y at node y is $PPOUT(y)$, and Equation (5) can use t_y as input to compute $PPIN(y)$. In fact, if y has any predecessors in G, then some edge function in e_i should have already computed $PPIN(y)$ at its last evaluation before the fixed point is reached. If y has no predecessors, then it is either unreachable, or is `ENTRY`; in either of these special cases, $PAVIN(y) = \mathit{False}$,[9] so $PPIN(y) = FALSE$. $DELETE(y)$ requires $PPIN(y)$, and can be computed similarly using Equation (8).

4.1.2 *Function space.* The edge functions are thus of the form:

$$
f_e((t,i)) = \begin{cases} \begin{pmatrix} \mathit{True}, \\ wt + yti \end{pmatrix} \text{ or} \\ \\ \begin{pmatrix} bi, \\ \mathit{True} \end{pmatrix} \end{cases} \tag{10}
$$

[9]The latter case comes from applying Equation (3) over an empty set of predecessors. In this separate problem, the meet operator is the logical or, so the top element of its lattice would be *False*.

where b, w, and y are constant. The closure under meet and function composition of these functions is the space F_{PP}, which includes exactly all functions f of the form,

$$f((t,i)) \;=\; \begin{pmatrix} at+bi+cti+d, \\ wt+xi+yti+z \end{pmatrix} \tag{11}$$

where a, c, d, x, and z are likewise constant. Equivalently, $f((t,i)) = (g_1(t,i), g_2(t,i))$, where each $g_j(t,i)$ is one of *True*, *False*, t, i, $t+i$, or ti. The identity function ι is a member of F_{PP}.

Lemma 4. *F_{PP} is monotone.*

Proof: Let $f \in F_{PP}$ be a function as defined above. (t,i) and (t',i') be elements of L_{PP} such that $(t,i) \sqsubseteq (t',i')$, i.e., $t' \Rightarrow t$ and $i' \Rightarrow i$. Therefore, $at'+bi'+ct'i'+d \Rightarrow at+bi+cti+d$ and $wt'+xi'+yt'i'+z \Rightarrow wt+xi+yti+z$. So $f((t,i)) \sqsubseteq f((t',i'))$. □

Lemma 5. *F_{PP} is not distributive.*

Proof: From Lemma 2 as applied to the component g_j functions. □

Lemma 6. *F_{PP} is not 2-bounded.*

Proof: Let $f \in F_{PP}$ be a function as defined above, such that $b = 1$ and all other coefficients are 0. $f^2((1,1)) = (0,0)$ and $(f \sqcap \iota)((1,1)) = (1,0)$. Therefore $f^2 \not\sqsupseteq f \sqcap \iota$. □

Lemma 7. *F_{PP} is 3-bounded.*

Proof: For a given function f and lattice element (t,i), the three functions $\iota((t,i))$, $f((t,i))$, and $f^2((t,i))$ can take on at most three distinct values. The meet of any three distinct values in the lattice is $\perp$ (see Figure 4). Therefore, it must be true that $f^3((t,i)) \sqsupseteq f^{[3]}((t,i))$. □

4.1.3 *Complexity results.* The height of the lattice L_{PP} is two; the time to either evaluate an edge function or to perform a meet is constant. Therefore, by Lemma 9, a worklist algorithm to find *PPIN* for every node, for a single variable, can achieve convergence in $\mathcal{O}(|N|^2 \log(|N|))$ time. Since the function space is not 1-semibounded, Lemma 10 is not applicable.

The analysis is carried out on the control flow graph of a program, in slightly transformed form. If the shape of the graph is constrained in any way, it may be possible to improve the above complexity result. For example, if the number of control edges is proportional to the number of nodes (as is usually the case in structured programs), the time bound decreases to $\mathcal{O}(|N|\log(|N|))$.

4.1.4 *Accuracy results.* Since the function space is not distributive, the maximum fixed point of L_{PP} (and of the equational formulations) is not necessarily equal to the meet-over-all-paths solution.

4.1.5 *Comments.* Although the authors in [DRZ92] also uses pairs of bits to represent information at a node, their lattice elements represent the pair $(PPIN, PPOUT)$ rather than our choice of $(PPIN, PPOUT \wedge AVOUT)$. This choice prohibits a simple encoding of their data flow equations as a lattice framework.

4.2 Program summary graphs

Callahan [Cal88] presents a *program summary graph* $G = (N, E)$ representing interprocedural control flow. The program summary graph has four types of nodes:

—*Entry nodes*, denoting a formal parameter at the start of a procedure invocation;

—*Exit nodes*, denoting a formal parameter at the end of a procedure invocation;

—*Call nodes*, denoting an actual parameter at the call of a procedure;

—*Return nodes*, denoting an actual parameter at the return of the procedure.

Edges in the set E are divided into two classes: *intraprocedural* edges based on local reaching information, and *interprocedural* edges denoting the binding of actuals to formals at the call of a procedure, and the binding of formals to actuals at the return. Figure 5 repeats the example program summary graph of Figure 1 in [Cal88].

```
PROGRAM MAIN
CALL SUBA(A,B)
PRINT *,A
STOP
END

SUBROUTINE SUBA(X,Y)
EXTERNAL SUBB
X = 4.0
IF (X .EQ. 5.0) K = SUBB(X,Y)
K = Y + K + X
RETURN
END

INTEGER FUNCTION SUBB(U,V)
SUBB = U + V
IF (U .GT. V) U = V
RETURN
END
```

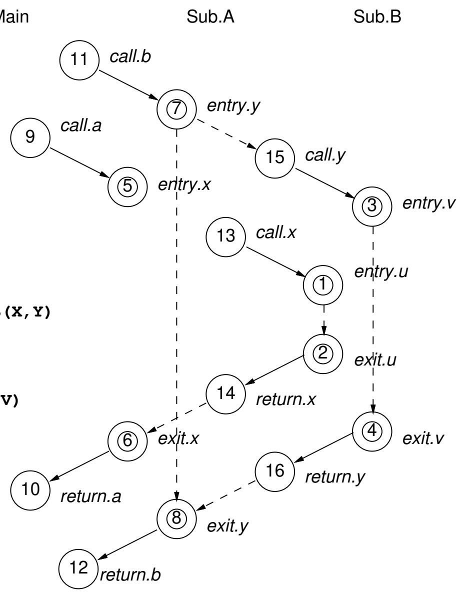

Fig. 5. Example program and program summary graph from Figure 1 of [Cal88].

Callahan also formulates several data flow problems on the program summary graph. These problems are expressed as iterative predicate calculus equations on some item of interest, (e.g., a reaching definition of a variable). Equivalently, the solution for all items of interest could be represented as a set.

4.2.1 *The Kill problem.* We will first consider Callahan's *Kill* problem: at a node x for a variable v, $Kill(x) = True$ if v must be modified by the procedure invocation represented at x. $Kill(x)$ is defined as follows in [Cal88]:

$$Kill(x) = \begin{cases} False & \text{if } x \text{ is an exit node;} \\ \bigwedge_{(x,y)\in E} Kill(y) & \text{if } x \text{ is either an entry or a return node;} \\ Kill(y) \vee Kill(z) & \text{if } x \text{ is a call node; } y \text{ is the corresponding return node and } z \text{ is the corresponding entry node.} \end{cases}$$

We observe that information is propagating to nodes from three sources: from successors in the set E of edges in the program summary graph, from return nodes to their corresponding call nodes, and from entry nodes to their corresponding call nodes. To explicitly represent this propagation, we modify the structure of the program summary graph slightly, to create edges along which this propagation occurs. Information propagates backwards along the edges of the resulting graph structure.

The resulting *Kill program summary graph* $G_{Kill} = (N, E \cup E')$ is shown in Figure 6. Note that the edges are now divided into two classes. The edge set E contains the edges of Callahan's original program summary graph; these edges are assigned an edge type $c = 1$. The edge set E' contains edges from call nodes to their corresponding return nodes; these edges have $c = 2$. E' also contains dummy edges corresponding to all edges in E other than those (shaded) edges leaving call nodes; this avoids driving $Kill(x)$ to *True* when x is other than a call node.[10]

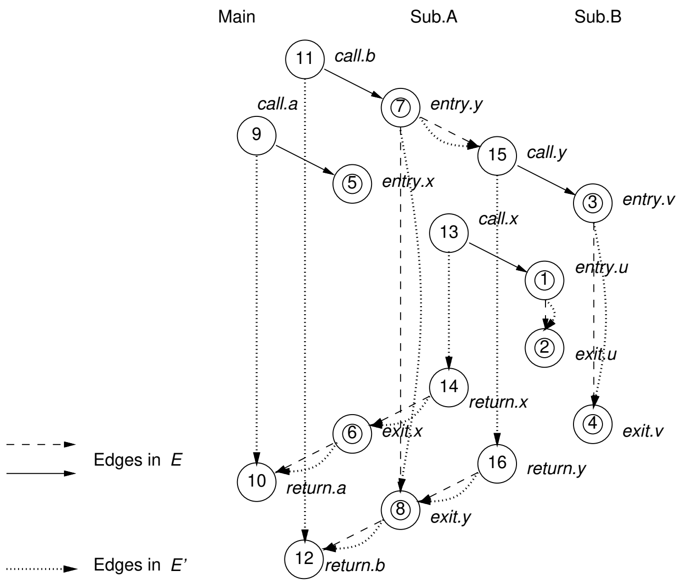

Fig. 6. *Kill* program summary graph.

[10] Alternately, we could simulate this behavior in the edge function set.

The lattice L_{Kill2} represents *Kill* information as 2-tuples of Boolean values (k, k'), where k and k' represent, respectively, *Kill* information entering from edges in sets E and E'. Meet in L_{Kill2} is element-wise *and*. Note that the lattice L_{Kill2} is thus the same as L_{PP} of Section 4.1.

We denote the value of L_{Kill2} at a node m as $\mathit{Kill2}(m)$, to distinguish it from Callahan's $\mathit{Kill}(m)$; for a node n, if $\mathit{Kill2}(m) = (k, k')$, then $\mathit{Kill}(m) = k \vee k'$. For a node m,

$$\mathit{Kill2}(m) = \bigsqcap_{\substack{e=(m,n,c),\\ e\in E\cup E'}} f_e(\mathit{Kill2}(n)).$$

Propagation of *Kill2* information is thus backward along the edges in both E and E'. We use the convention that the meet over an empty set of edges is the top element of the lattice, e.g., in L_{Kill2}, $(\mathit{True}, \mathit{True})$. Note that this produces the expected result at node 5 in Figure 6, i.e., that y is killed at node 5.

We define $M_{\mathit{Kill2}}(e)$ to map an edge $e = (m, n, c)$ to an edge function f_e as follows:

$$f_e((k,k')) = \begin{cases} (\mathit{False}, \mathit{False}) & \text{if } n \text{ is an exit node; (a)} \\ (k \vee k', \mathit{True}) & \text{if } n \text{ is not an exit node and } e \in E \\ & (c = 1)\text{; (b)} \\ (\mathit{True}, k \vee k') & \text{if } n \text{ not an exit node and } e \in E' \\ & (c = 2)\text{. (c)} \end{cases}$$

The function space F_{Kill2} must contains the closure over meet and function composition of the edge functions. These generate a space if functions f such that:

$$f((k,k')) \;=\; \begin{pmatrix} ak \vee abk' \vee d, \\ wxk \vee xk' \vee z \end{pmatrix} \tag{12}$$

where a, b, d, w, x, and z are constants. Equivalently, $f((k, k')) = (g_1(k, k'), g_2(k, k'))$, where $g_1(k, k')$ is one of *False*, *True*, k, or $k \vee k'$, and $g_2(k, k')$ is one of *False*, *True*, k', or $k \vee k'$.

Since it is contained in F_{PP} above, F_{Kill2} is monotone and 3-bounded. Lemma 2 can be applied to the g_j functions to show that F_{Kill2} is not distributive. To show that F_{Kill2} is not 2-bounded, let $f((k, k')) = (k \vee k', \mathit{True})$ and apply the value $(\mathit{True}, \mathit{False})$.

4.2.2 *Improving the boundedness property.* It is possible to improve the 3-boundedness result above to a 1-semiboundedness result. Doing so gives us the ability to use Tarjan's algorithm to get an $\mathcal{O}(\log n)$ better convergence time.

Using methodology similar to [Zad84], it is possible to eliminate constant-valued edge functions. This is done by initializing constant-valued nodes once and not changing their values again during the fixed-point iteration. The edges with constant edge functions, e.g. the edges $(8, 12)$ in E and E', are simply ignored.

Eliminating the edge functions of type (a) results in a function space of 1-bounded edge functions. The closure over meet and function composition of type (b) and (c) functions and the identity function ι is contained in the set of functions of the form:

$$f_e((k, k')) = \begin{pmatrix} k \vee ak' \vee b, \\ ak \vee k' \vee c \end{pmatrix}$$

where a, b, and c are constant. We denote this function space F^-_{Kill2}. F^-_{Kill2} has the same properties as F_{Kill2}, with the exception that it is 1-bounded.

4.2.3 *Complexity results.* The shape of the program summary graph is constrained in such a way that we can use a node-listing algorithm to get a tighter bound on convergence time than is possible using general iterative methods. Here, we first show the best results we can get using the worklist and Tarjan's algorithm, and then show how Callahan improves this result.

Since meet operations and edge function evaluations take constant time, and since the height of the lattice is constant, a worklist algorithm for this framework would require $\mathcal{O}(|N|^2 \log(|N|))$ time on a general graph. Tarjan's algorithm would terminate in $\mathcal{O}(|N|^2)$ time.

The *Kill* program summary graph, like Callahan's original program summary graph, is acyclic, and has $\mathcal{O}(|N|)$ edges. This property allows us to define a node listing order (i.e., topsort order) such that each edge only needs to be visited once. This gives a tighter upper bound on convergence time (also $\mathcal{O}(|N|)$) than a worklist algorithm, and demonstrates that knowledge of the graph's shape properties can be useful in tightening the upper bound.

In [Cal88], Callahan made the claim that his data-flow equation system was distributive. Since F_{Kill2} is not distributive, we can see clearly that this claim is incorrect.

4.2.4 *Accuracy results.* Since F_{Kill2} is not distributive, the maximum fixed point of L_{Kill2} is not guaranteed to be the same as its meet-over-all-paths solution.

4.2.5 *The* $\widehat{Kill}$ *problem.* $\widehat{Kill}$ is a similar problem, formulated for better efficiency. Parameters are checked in the $\widehat{Kill}$ problem only when they might possibly be modified. Callahan formulates $\widehat{Kill}$ as:

$$\widehat{Kill}(x) = \begin{cases} (v \notin MOD(p)) & \text{if } x \text{ is an exit node associated with variable } v \text{ in procedure } p; \\ \bigwedge_{(x,y)\in E} \widehat{Kill}(y) & \text{if } x \text{ is either an entry or a return node;} \\ \widehat{Kill}(y) \vee ((v \in MOD(p)) \wedge \widehat{Kill}(z)) & \text{if } x \text{ is a call node, } y \text{ is the corresponding return node, } z \text{ is the corresponding entry node, and } v \text{ is the variable associated with } z \text{ in procedure } p. \end{cases}$$

We use lattice L_{Kill2} and the graph $G' = (N, E \cup E')$ to represent the $\widehat{Kill}$ problem. Note, however, that the edges from call to entry nodes must be assigned different edge functions than other edges, because of the $((v \in MOD(p)) \wedge \widehat{Kill}(z))$ term.

In the mapping function $M_{\widehat{Kill}}$, we map an edge $e = (m, n)$ to an edge function

f_e as follows:

$$f_e((k, k')) = \begin{cases} (v \notin MOD(p), v \notin MOD(p)) & \text{if } m \text{ is an exit node;} \\ (k \vee k', \mathit{True}) & \text{if } m \text{ is not an exit node and } e \in E; \\ (\mathit{True}, (k' \vee k') \wedge (v \in MOD(p))) & \text{if } m \text{ not an exit node and } e \in E'. \end{cases}$$

Note that the terms $v \in MOD(p)$ and $v \notin MOD(p)$ are constant for any particular edge e. This implies two useful facts:

—If m is an exit node, then f_e has a constant value of either $(\mathit{True}, \mathit{True})$ or $(\mathit{False}, \mathit{False})$, depending on whether $v \in MOD(p)$.

—If m is not an exit node and $e \in E'$, then $f_e((k, k'))$ has a constant value of $(\mathit{True}, \mathit{False})$ where $v \in MOD(p)$, and a value of $(\mathit{True}, k' \vee k')$ otherwise.

Thus, we can again apply a graph transformation similar to that of [Zad84] to eliminate edge functions other than those which are monotone and inflationary. Under this formulation, the lattice framework $D_{\widehat{Kill}}$ is also 1-bounded.

The formulation for Callahan's *Use* problem is, in principle, quite similar to that for $\widehat{Kill}$. The *Kill* problem with set representation for globals can be formulated as a cross-product of the $\widehat{Kill}$ problem with one 2-tuple per global variable. For the latter, Callahan gives a bound of $\mathcal{O}(l \log(l))$ bit vector operations using Graham and Wegman's interval analysis [GW76] (where l is program length), and an $\mathcal{O}(l\alpha(l, |N|))$ bound given using Tarjan's analysis [Tar81a; Tar81b]. This bound is justified, but only if the type (a) edge functions are eliminated; otherwise, the problem is not 2-semibounded.

5. CONCLUSIONS

We have demonstrated that k-tuple frameworks for multisource data flow problems provide a uniform representation for a wide class of problems which previously had been formulated in a variety of *problem-specific* approaches. While analyses of explicitly parallel programs is likely the most important type of problem in this class, we have also given examples of bidirectional and interprocedural flow sensitive problems which fall into the class. Formulating these problems as k-tuple frameworks allows use of the standard complexity and precision results for data flow frameworks. By casting problems such as partial availability and program summary graph computations into our k-tuple frameworks, we have been able to characterize the precision and convergence time for fixed point interaction on these problems. Also, we have uncovered some errors in previous *problem-specific* analyses.

We have presented k-tuple frameworks as a useful conceptual tool for formulating new analysis problems, rather than as a solution tool. The join-of-meets subclass is especially useful in analyses dealing with such explicit parallelism; use (after optimization) of standard techniques with these frameworks may indeed provide the best solution techniques currently available for many analyses of explicitly parallel programs.

6. APPENDIX

6.1 Inferring framework properties from the function space

Many important properties of functions, such as monotonicity and distributivity, are preserved under function meet and composition. Thus, we may infer such properties for F (in any lattice), if we can prove them for the set of functions used as edge functions. For example:

LEMMA 8. *The following sets of functions on a lattice are closed under function composition and meet:*

—*Monotone functions*

—*Distributive functions*

—*Inflationary functions*

—*Meet-with-constant functions*

The intersections of any of these sets are also closed under function composition and meet. (Proofs of these results can be found in Lemmas 6,7,8 of [Mas93] as well as elsewhere.)

6.1.1 *Termination.* Since monotonicity on a fixed height lattice implies eventual termination of fixed point iteration, it is valuable to be able to deduce that the function space is monotone. Lemma 8 allows us to do this directly from edge functions, without having to tightly characterize the entire function space.

6.1.2 *Execution time bounds.* The height of the lattice, and the time to perform meet and function application operations, give at least a preliminary upper bound on the execution time of worklist algorithms and Tarjan's algorithm:

LEMMA 9. *If a lattice $L = (S, \sqcap, \sqcup)$ has finite height h, and is used in a monotone data flow framework $D = (G, L, F, M)$ where $G = (N, E)$, then the worklist iterative algorithm of [Hec77] applied to D where each node has an initial value of $S_\perp$ terminates after no more than $h|N|^2\lceil\log_2(|N|)\rceil$ meet operations and $h|N|^2$ edge function evaluations in the worst case. The cost to maintain the worklist during this time is $\mathcal{O}(h|N|\lceil\log_2(|N|)\rceil)$ in the worst case. This result also holds when each node has an initial value of $S_\top$.*

LEMMA 10. *Under the assumption that Tarjan's algorithm [Tar81a; Tar81b] terminates, a solution to a 1-semibounded data flow framework can be found in $\mathcal{O}(|N|^2(t_f + t_\sqcap))$, where t_f is the time for one edge function application and $t_\sqcap$ is the time for one meet operation.*

With better knowledge of the properties of the function space, we can sometimes apply other algorithms and achieve better time bounds.

6.2 Boundedness properties

6.2.1 *Graham-Wegman fastness and Kam-Ullman rapidity.* A monotone function space $F : L \rightarrow L$ is *2-bounded* iff, for all $f \in F$ and $x \in L$, $f^2(x) \sqsupseteq f(x) \sqcap x$. f is *1-semibounded* iff, for all $f \in F$, all $x, y \in L$, and all $r \geq 1$, $f^r(x) \sqsupseteq x \sqcap f(y)$. [MR90].

These boundedness properties are relevant to two important convergence time results. 2-boundedness is a sufficient condition for *Graham-Wegman fastness* [GW76];

1-semiboundedness and distributivity are sufficient for Kam-Ullman rapidity [KU76]. However, as the following lemma shows, we cannot necessarily characterize a function space as simply the closure of edge functions and prove Graham-Wegman fastness or Kam-Ullman rapidity.

LEMMA 11. *The sets of idempotent, $k+1$-bounded, or k-semibounded functions for $k \geq 1$ are not closed under function composition. Furthermore, these sets are not closed if they are restricted to be monotone or distributive, or both.*

From Lemma 11, we see that we must characterize the entire function space to use the Graham-Wegman method (unless we are lucky enough to be able to apply other theoretical tools, e.g., Lemma 8 on inflationary edge functions.)

REFERENCES

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.

D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, 1988.

E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages; NATO Advanced Study Institute*, pages 43–112. Academic Press, London, 1968.

D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, 1992.

E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 36–48, October 1991.

D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 159–168, San Diego, CA, May 1993.

S. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

J. B. Kam and J. D. Ullman. Global flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

D. L. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 21–35, Vancouver, BC, October 1991.

W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

S. P. Masticola. *Static detection of deadlocks in polynomial time.* PhD thesis, Rutgers University, New Brunswick, NJ, April 1993.

E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Informatica*, 28:121–163, 1990.

S. Masticola and B. G. Ryder. A model of ada programs for static deadlock detection. In *Proceedings of 1991 ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA*, pages 91–102, May 1991. published as ACM SIGPLAN Notices, vol 26, no 12, December 1991.

S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–138, May 1993.

B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

B. Sloman and T. Lake. Scalar renaming, rescoping and optimising liveness analysis in the presence of pointer induced aliasing and side-effecting expressions. In *Proceedings of the Fourth International Workshop on Compilers for Paralel Computers*, December 1993.

R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.

F. K. Zadeck. *Incremental Data Flow Analysis in a Structured Program Editor*. PhD thesis, Department of Mathematical Sciences, Rice University, 1983.

F. K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 132–143, June 1984.